修 士 学 位 論 文

Procedural content generation for tower defense games (タワーディフェンスゲームのための手続き型コンテンツ生成)

2021 年度

広域科学専攻　広域システム科学系

31-206830

許 悦銘

ABSTRACT

As the importance of Procedural Content Generation (PCG) for game development increases, researchers explore new avenues for generating high-quality content. One relatively new paradigm is Procedural Content Generation via Machine Learning (PCGML).

Many machine learning methods have been proved effective for PCG tasks, such as dungeons generation, puzzle generation, maze generation. Nevertheless, the effectiveness of PCGML for complicated game level generation remains unclear. For example, tower defense games require detailed level design and difficulty balancing to create an enjoyable player experience, and there are still few studies in this area.

This research focuses on adversarial reinforcement learning-based PCG, which successfully applied to truck generation for racing games. We developed a tower defense game simulator based on an existing commercial game's rules as an environment for reinforcement learning agents. We found suitable action spaces for agents playing tower defense games and explored methods to generate tower defense games levels using reinforcement learning.

Acknowledgements

Throughout the writing of this thesis, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Tetsuro Tanaka , whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would particularly like to acknowledge all members of Tanaka Lab. and Game Programming Seminar and for their wonderful collaboration and patient support.

Finally, I would like to thank my family for their wise counsel and sympathetic ear. You are always there for me.

Contents

Chapter 1

Introduction

Procedural Content Generation (PCG) is the process of algorithmically creating content. PCG can be employed to increase games' replay value and reduce the production cost and effort for the games industry[1]. Some forms of game content such as trees and landscapes have been generated procedurally for a long time.

This study explores the methods of using reinforcement learning to generate levels for *Tower Defense (TD)* games.

1.1 Motivation

On the one hand, players are always happy to get a variety of experiences from games, and their demands on the quality of games are becoming higher and higher. As a result, the amount of content in modern video games is becoming larger and larger. On the other hand, developers of ongoing games, such as mobile games and *Massively Multiplayer Online Role-Playing Games (MMORPGs)*, need to add new content to their games to keep players interested. Finally, with the popularity of the *Metaverse*, *User Generated Contents (UGC)* in games is starting to gain the attention of most game developers. PCG can be a tool for mass generation of a particular art asset in a game or an algorithmic model to assist designers or players in creating game assets.

However, many traditional PCG usually needs developers to do much hand-coding work. *Machine Learning (ML)* has achieved great success in content production, including generating audio, photo, and other content types across different domains. It stands to reason that machine learning would apply to games content generation. One relatively new paradigm of PCG called *Procedural Content Generation via Machine Learning (PCGML)*[2] has had enjoyed considerable popularity recently. With PCGML, the extra workload imposed on game developers by traditional PCG methods is expected to be mitigated.

We consider the design of game levels an iterative process of alternating design and testing. In this iterative process, designers and testers will get their work done better and better. *Reinforcement Learning (RL)* is a kind of machine learning, and video games are one of the most widely used fields. Therefore, we propose to use reinforcement learning agents to replace human designers and testers.

The *Tile-Based* games are one of the most popular testing grounds for level generation, in which levels are made up of different types of *Tiles*, including walls, roads, collectible items, and even enemies. Many previous works [3], [4] have made outstanding contributions to the level generation of tile-based games. However, there is little research on PCGML applied to more complicated games such as *Tower Defense (TD)* games. TD is a popular casual game genre that has proven to be a good testbed for AI and game research.[5]. TD games often require precise level design, making us think it is challenging and meaningful to study level generation in tower defense games.

1.2 Goals

The main goals for this study are the following:

- Develop an environment where reinforcement learning agents can be trained, i.e., a tower defense game simulator.
- Use reinforcement learning to train agents to play tower defense game levels.
- Use reinforcement learning to train agents to generate levels for tower defense games.

1.3 Outline

This thesis is organized as follows:

Chapter 2, Background: Introduces the background knowledge that the reader needs to know in order to understand this thesis.

Chapter 3, Related Works: Introduces s several works of high relevance to this study.

Chapter 4, Environment: Introduced the tower defense game simulator we developed.

Chapter 5, Proposed Methods: Introduces the method we use to generate levels for tower defense games.

Chapter 6, Experiments: Introduces the experiments we did to generate the levels for the tower defense game.

Chapter 7, Conclusions and Future Work: Introduces the findings of this study and the areas that still need work.

Chapter 2

Background

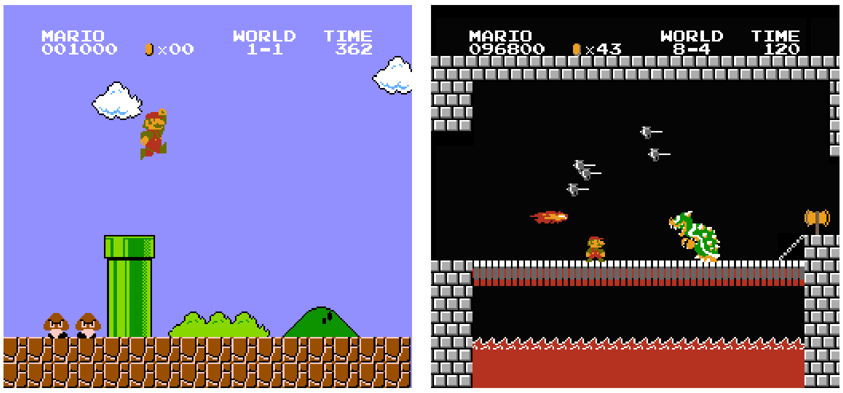
This chapter presents the necessary background information needed to understand this thesis. It starts with explaining levels and tiles and is followed by a description of tower defense games. Finally, it presents the definition of procedural content generation used in this thesis.

2.1 Level and Tile

This section uses *Super Mario Bros.*[1] as an example to illustrate two critical terms about games that will appear in this thesis, namely level and tile.

2.1.1 Level

A level is a section of a complete game in which the player usually needs to complete a specific goal to advance to the next level. A level is a combination of game elements. Different combinations result in different game flows and difficulties. Figure 2.1 shows different combinations of map elements and enemies can make up different levels in Super Mario Bros.



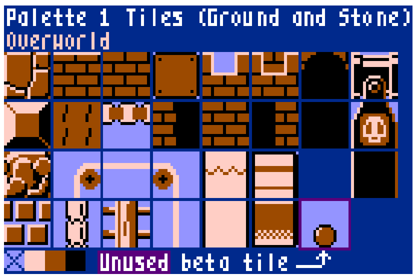
(a) World 1-1.   (b) World 8-4.

Figure 2.1: Examples on different levels of Super Mario Bros.

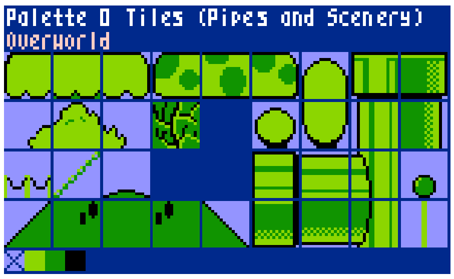
2.1.2 Tile

A tile is a minimum unit that can form a game level or a part of a game level. Games that are made up of tiles are called tile-based games, and the collection of all tiles used in a tile-based game is called *tilesheet*. Figure 2.2 shows a part of the tilesheet of Super Mario Bros.

[1]Super Mario Bros. is a platform game developed and published by Nintendo.



(a) Tiles of ground and stone.



(b) Tiles of pipes and scenery.

Figure 2.2: A part of tilesheet of Super Mario Bros.

2.2 Tower Defense

This section illustrates the basic gameplay of tower defense games and the properties and abilities of towers in games. As a popular game genre, tower defense games have many variants in which the various parts differ. Therefore, to make this study more effortless for the reader to understand, only the parts common to most tower defense games are presented in this section.

2.2.1 Basic gameplay elements

In order to understand the gameplay of tower defense games, it is necessary to have an understanding of the basic gameplay elements of tower defense games. The following are the four basic gameplay elements of tower defense games:

- The *resource* can be spent by players to build or upgrade towers, commonly in the form of money, deployment points, etc. The resource can usually be obtained by killing enemies in the level.
- The *defensive points* are places where players must prevent enemies from reaching.
- The *towers* are buildings that automatically attack enemies and generally cannot be changed in position after being built. Depending on the game theme, towers may appear in other forms, such as plants, operators, etc.
- The *wave* refers to a group of enemies. Game designers usually design the waves in a level carefully to give the player a suitable playing experience. The wave information usually includes the type of enemies, the number of enemies, and the time of their appearance.

2.2.2 Gameplay

Tower Defense (TD) is a subgenre of strategy game in which players build towers on the map that automatically attack to prevent enemies from reaching defense points. A complete tower defense game usually consists of multiple levels consisting of several waves. In each wave, many enemies appear from the map and move towards the defense point. A typical game flow for a level is as follows:

1. Depending on the level setting, players will be granted a set amount of resource and life count to begin with, which may be spent to place and improve towers.

2. At the beginning of the wave, enemies appear from the map and move towards the defense points. By killing enemies, the player may gain additional resources.

3. Players should wisely arrange the use of resources to eliminate as many enemies as possible.

4. The player's life count will be reduced if enemies arrive at the map's exit points. The players win when all enemies have been defeated and fail when the player's life count is zero.

2.2.3 Abilities of towers

As one of the core elements of tower defense games, game designers often provide players with towers with different abilities to deal with different enemies and situations. As a result, tower defense games are simple to play but often have a certain depth of strategy. The following are four essential attributes of towers:

- *Attack Power* is used to calculate the damage dealt when each attack is made on the enemy. Generally, the more times the player upgrades the tower, the higher the attack power will be.
- *Attack Range* is the distance at which a tower may attack an enemy. Generally, the more times the player upgrades the tower, the wider the attack range will be.
- *Attack Speed* is the frequency of the attack. Generally, the more times the player upgrades the tower, the faster the attack speed will be.
- *Attack Type* is the type of damage caused to the enemy. A tower may deal different damage to different types of enemies.

2.2.4 Attributes of towers

In addition to abilities, towers usually have some primary attributes. While playing a level, players can enhance these attributes of towers by upgrading the towers they have already built. Here are four common attributes:

- *Slow* means the movement speed of the attacked enemy will be reduced.
- *Damage over time (DoT)* can cause the affected enemy to take damage until the effect expires constantly.
- *Splash* can attack more than one enemy at a time.
- *Assist* means other towers next to this tower will be strengthened.

2.3 Procedural Content Generation

This section first explains the basic concept of Procedural Content Generation (PCG). Then it introduces *Search-Based Procedural Content Generation*, one of the most frequently used PCG methods, and finally briefly introduces *Procedural Content Generation via Machine Learning (PCGML)*, which is the most relevant to this study.

2.3.1 Basic concepts of Procedural Content Generation

Procedural Content Generation (PCG) refers to the algorithmic generation of game assets with limited or indirect input. In other words, PCG refers to computer software that can create game content independently or together with one or more human players or game designers.

Content here refers to almost everything in the game: levels, game rules, graphics, story, music, etc. However, note that the game engine and non-player characters (NPC AI) are not considered to be content in the definition [6] our use.

2.3.2 Search-Based Procedural Content Generation

Search-based PCG [7] is one of the generate-and-test algorithms. This algorithm consists of a generating mechanism and a test mechanism. After generating a candidate content instance, it is checked against a set of criteria. If the test fails, all or part of the candidate content is rejected and regenerated, and the process is repeated until the content meets the requirements.

In the test mechanism, the criteria are represented by a test function. This test function takes the candidate content as input, and the output is a vector or a real number. There are different names for such a test function, such as *fitness*, *evaluation*, and *utility function*. In this study we will use the name *fitness function* and call its output *fitness*.

Therefore, search-based PCG is an algorithmic model that allows the fitness of the generated content to be continuously improved to generate the desired content.

2.3.3 Procedural Content Generation via Machine Learning

Summerville et al. define Procedural Content Generation via Machine Learning (PCGML) as the generation of game content by models that have been trained on existing game content[2].

One of the predictable advantages of PCGML is its adaptability brought about by the learning process. If a traditional PCG is used in the game development process, it is difficult to avoid adjusting the PCG method when the game itself changes. While game development is often a long iterative process, the frequent occurrence of such adjustments of the PCG method can create an additional workload for the developer. This is contrary to part of the original purpose of using PCG. The adaptability of PCGML gives it the possibility of alleviating this problem.

Chapter 3

Related Works

This chapter first introduces the basic concepts related to reinforcement learning. Then it presents the related work on level generation for tower defense games. Finally, the application of reinforcement learning in PCG is introduced.

3.1 Reinforcement Learning

This section presents a basic description of reinforcement learning and explains some terms that will appear in this thesis. Since reinforcement learning is a complex concept, we will explain fundamental terms instead of the reinforcement learning algorithm.

Reinforcement Learning (RL) is one of three fundamental machine learning paradigms, alongside supervised learning and unsupervised learning. Reinforcement learning is a model for learning optimal policy by trial and error. Because video games provide the ideal environment for reinforcement learning, video games are one of the most widely used areas of reinforcement learning algorithms. There are five basic elements in reinforcement learning as follows:

- The *agent* is the subject that makes the action in the environment. In most cases, the agent is the player-controlled character.
- The *environment* is where the agent is located, such as a room and a level in games.
- The *state* is the information about the agent's environment, and the set of all possible states is called the *state space*. It is usually denoted as $s \in S$.
- The *action* is made by the agent, such as moving, attacking, and the set of all possible action is called the *action space*. It is usually denoted as $a \in A$.
- The *reward* is given to the agent according to a pre-determined rule after the agent makes an action. It is usually denoted as r.

On top of the above basic elements, we can describe the core elements of reinforcement learning. The *return* is the sum of all future rewards the agent can gain, starting with a certain state. It is usually denoted as G:

$$G \doteq \sum_{k=t}^{T} r_k, \tag{3.1}$$

where t is the current moment, and T is the final moment. The *policy* is a mapping from states to actions, indicating what the agent will do in a given state. The goal of reinforcement learning is to find a policy that maximizes the expected value of return.

3.2 Level Generation for Tower Defense games

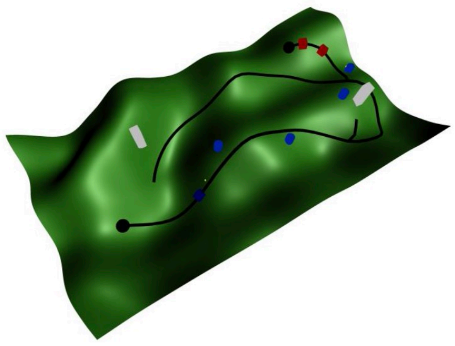
This section presents the work of Öhman and the work of Liu et al. These two works conducted level generation experiments using the rules of a self-created simple tower defense game and the rules of an existing tower defense game, respectively.

3.2.1 Öhman's work

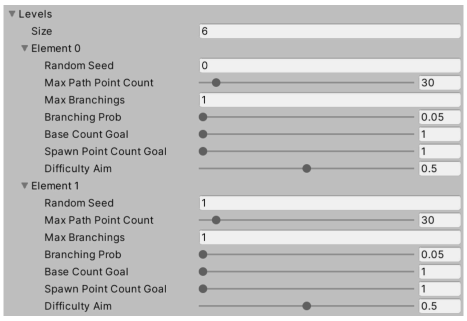
Öhman[8] proposed a method to generate levels for a 3D tower defense game implemented by himself (see Figure 3.1(a)), mainly using fitness function and Perlin noise.

Öhman divided the map generation into two phases: terrain and path generation. Terrain generation refers to the use of Perlin noise to generate the height map of the plane. After the terrain is generated, the path starts from a random point and expands it. Figure 3.1(b) shows that the style of the generated path can be controlled by a set of parameters, such as *Max Path Point Count*, *Max Branchings*, etc. Finally, the generated levels are evaluated using a predefined fitness function and discarded if the score is below a certain threshold.

The generation of wave information is divided into two parts: the number of waves and the number of enemies in the wave. The number of waves is calculated by an equation related to path density and level difficulty. The number of enemies is randomly generated between 1 and 5.



(a)



```
▼ Levels
    Size                    6
  ▼ Element 0
      Random Seed           0
      Max Path Point Count  30
      Max Branchings        1
      Branching Prob        0.05
      Base Count Goal       1
      Spawn Point Count Goal 1
      Difficulty Aim        0.5
  ▼ Element 1
      Random Seed           1
      Max Path Point Count  30
      Max Branchings        1
      Branching Prob        0.05
      Base Count Goal       1
      Spawn Point Count Goal 1
      Difficulty Aim        0.5
```

(b)

Figure 3.1: Screenshot from the implemented game (Left) and the parameters for path generation (Right). From J. Öhman, " Procedural generation of tower defense levels, " 2020.

3.2.2 Liu et al.'s work

Liu et al.[9] presented a framework that is capable of automatically generating levels with a similar style of existing content for tower defense games and conducted their experiments in a developed in-house tower defense game environment(see Figure 3.2).

Liu et al. considered that the tower defense level consists of four fundamental elements as follows:

- *Roads* refer to the combination of curves or straight lines in the map. Enemies will move along the roads.
- *Tower point* refers to the place where the player can build a tower.

- *Enemy sequence* refers to wave information.
- *Automated playability testing* ensures that the generated level is appropriately designed.

The method of road generation is drawn from the analysis of existing tower defense levels. An initial road segment is randomly generated near the map's center that extends toward the map edge. A segment will be generated near halfway toward the map edge if more branches are needed. Repeat this process until the road is completed.

The tower points are generated by a random search method. First, two to three points are placed near each road intersection, and then the rest of the points are randomly placed along the road and ensure that no two points are too close to each other. They also computed the road coverage distribution of the generated tower points and tower points in human-designed levels. Finally, the reasonable tower points are filtered using their Kullback-Leibler (KL) divergence.

The enemy sequence is generated by a genetic algorithm. To reduce the amount of computation required, they first extract a cluster of enemies from the original level. Then, the generation is done using these clusters rather than an individual enemy.

Automated playability testing was implemented using a reinforcement learning agent based on Monte Carlo search. They also calculated the difficulty for each moment while playing the level to get the difficulty of the whole level.

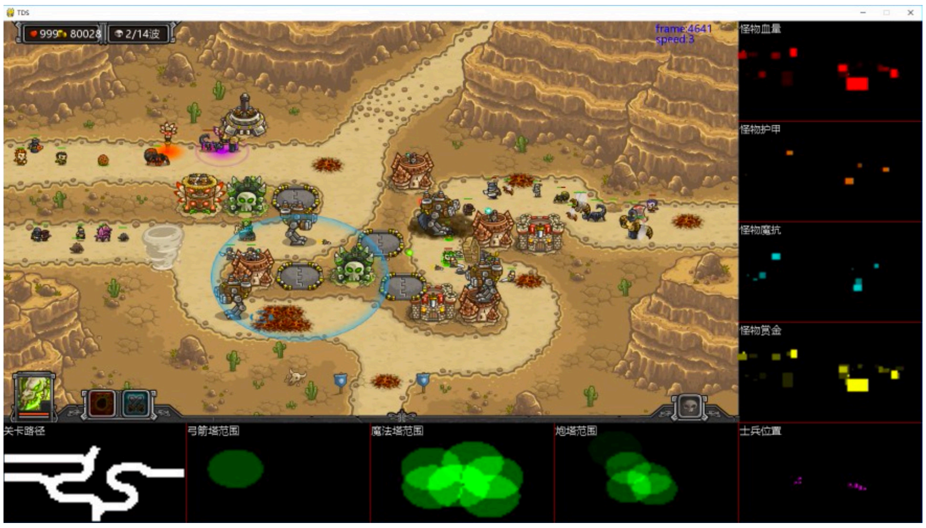


Figure 3.2: Screenshot from the implemented game. From S. Liu, L. Chaoran, L. Yue, M. Heng, H. Xiao, S. Yiming, W. Licong, C. Ze, G. Xianghao, L. Hengtong, D. Yu, and T. Qinting, " Automatic generation of tower defense levels using pcg, " in Proceedings of the 14th International Conference on the Foundations of Digital Games, FDG ' 19, (New York, NY, USA), Association for Computing Machinery, 2019.

3.3 Level Generation via reinforcement learning

This section presents two reinforcement learning-based level generation frameworks.

3.3.1 PCGRL

Khalifa et al.[4] proposed a framework for 2D tile-based game level generation using reinforcement learning, namely Procedural Content Generation via Reinforcement Learning (PCGRL). Figure 3.3 shows that there are three important modules in the PCGRL framework: Problem, Representation, and Change Percentage.

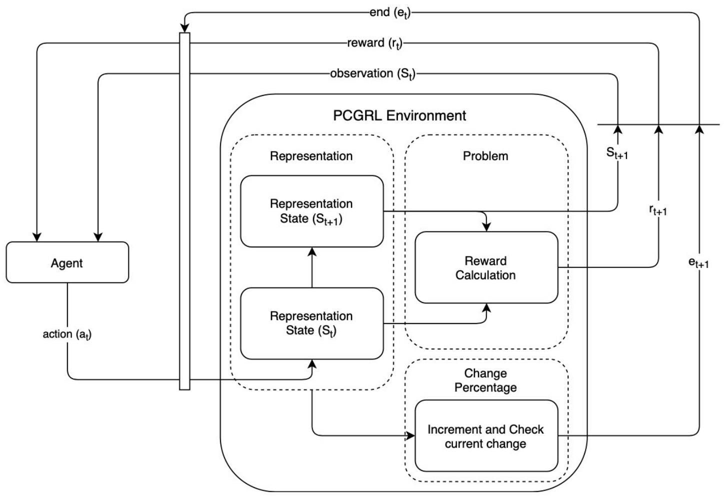


Figure 3.3: The system architecture for the PCGRL. From A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, " Pcgrl: Procedural con- tent generation via reinforcement learning, " 2020.

The *Problem* module provides all information related to the current generation task and has two main functions. The first function is to evaluate the change in the quality of the level after the agent takes action and gives a reward. The second function is to determine when to terminate the generation process.

The *Representation* module is mainly responsible for defining the agent's state space and action space. For simplicity, they represented the level as a 2D array of integers. The value in the array corresponds to the tile type (see Figure 3.4. In their work, the following three represents are defined:

- *Narrow* representation is the simplest way to represent a problem, with a minimal state and action spaces. In the generation process using narrow representation, the agent is given a position in the level at each step and is allowed to change the tile type of this position. Thus, its state space includes the level's current state, and its action space size is the number of tile types.
- *Turtle* representation is inspired by turtle graphics languages. In the generation process using the turtle representation, each step of the agent can move and change the tiles along the way. Hence, the state space contains the current state of the level and the agent's location, and the action and the action space includes the direction of movement and all tile types.
- *Wide* representation is the most complex way to represent the problem. At each step, the agent can select any position of the level and change the tile type. Thus, its state space contains the current state of the level, and the action space includes all positions and all tile types in the level.

1	1	1	1	1	1	1
1	4	0	0	0	0	1
1	0	3	1	3	0	1
1	1	0	1	0	1	1
1	0	2	0	3	0	1
1	4	0	0	0	4	1
1	1	1	1	1	1	1

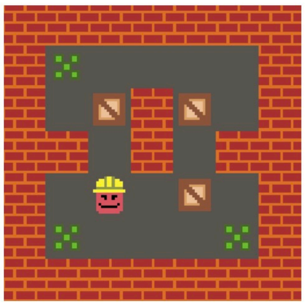


Figure 3.4: Game level as 2D integer array. From A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural con- tent generation via reinforcement learning," 2020.

The *Change percentage* is a vital hyperparameter that determines how many tiles the agent is allowed to change. Therefore, it also limits the length of episodes during the training process.

3.3.2 ARLPCG

Gisslén et al.[10] introduced a model based on adversarial reinforcement learning, which can improve the generalization of a reinforcement learning agent and generate levels for 3D games. It is called Adversarial Reinforcement Learning for Procedural Content Generation (ARLPCG).

The model of ARLPCG consists of two important parts: the *Generator* and the *Solver*, and both parts are reinforcement learning agents. Figure 3.5 shows the system architecture for the ARLPCG.

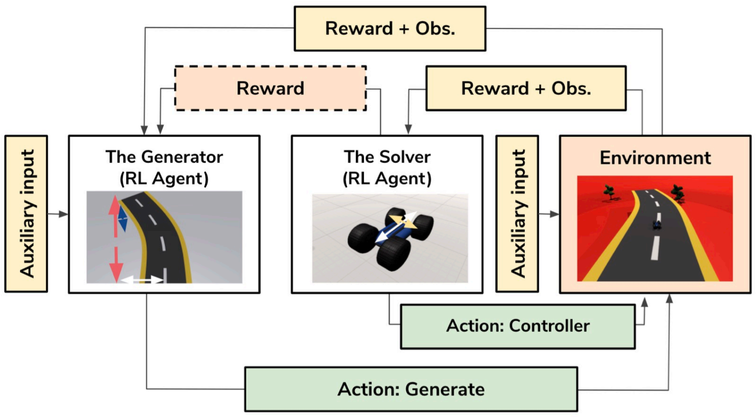


Figure 3.5: The system architecture for the ARLPCG. From L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, " Adversarial Reinforcement Learning for Procedural Content Generation, " 2021.

The generator is the agent responsible for generating the level. The generator in ARLPCG defines a new reward structure based on PCGRL: the generator's reward is partially derived from the solver, i.e., the solver's performance in the environment (whether it can clear the level) affects the generator's reward. The solver is the agent responsible for playing the level. The reason for using reinforcement learning agents instead of scripted agents here is to improve the generality of this model.

In Gisslén et al.'s work, the ARLPCG framework was used to generate levels for two games: a platform game and a racing game (see Figure 3.6). For the platform game, the levels consist of a starting point, an endpoint, and several platforms. For the racing game, the level consists of a start point, an endpoint, obstacles, and a track, where the track consists of several track segments interconnected, and the generator needs to ensure that obstacles do not obstruct the track. Gisslén et al. used a progressive approach to level generation for both games, i.e., the generator generates the level with an empty initial state and generates the level segment by segment.

However, the levels of tower defense games are more complex than those of the two games mentioned above and can not be simply segmented. Therefore, it is necessary to check whether the method based on the ARLPCG framework can be used to generate tower defense game levels. Our proposed method is based on the ARLPCG framework, and a detailed description of the method will be mentioned in Chapter 5.

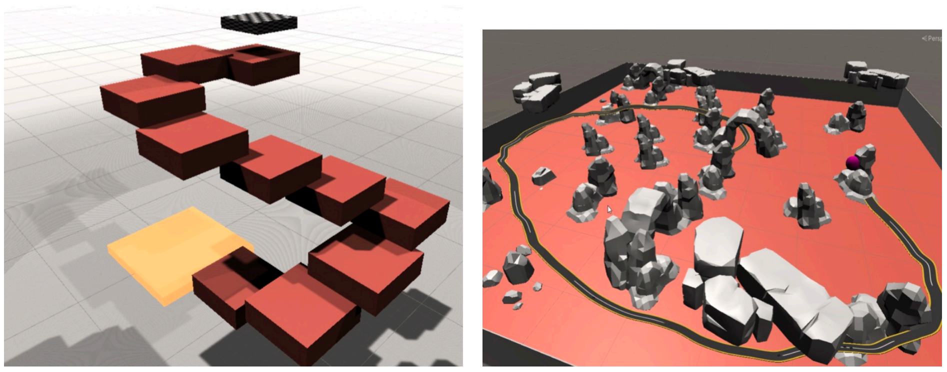

(a) The platform game. (b) The racing game.

Figure 3.6: The games used in Gisslén et al.'s work. From L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, "Adversarial Reinforcement Learning for Procedural Content Generation," 2021.

Chapter 4

Environment

We developed a tower defense game simulator based on the rules of *Arknights* in *Unity* as an environment in which reinforcement learning agents can be trained. This chapter first describes the basic information about Unity and Arknights and why we chose to use them. Then, it presents the details of the tower defense game simulator we developed.

4.1 Unity

This section introduces the concept of game engines and why we chose to use Unity to develop our tower defense game simulator.

Game engines are software that game designers can use to develop a video game or some interactive real-time graphics application. Game engines often include visual development tools and reusable components (graphics, sound, physics, and artificial intelligence) integrated with the development environment. Using game engines for development can save developers significant development costs, reduce development complexity, and shorten the time to market, all critical factors for the highly competitive game industry.

Game development teams generally develop non-public game engines. Still, there are also publicly available general-purpose game engines that all developers can use, and Unity is one of the most popular ones. Unity is a cross-platform 2D and 3D game engine developed by *Unity Technologies*[1] to develop cross-platform video games. Figure 4.1 shows the interface of Unity.

We chose to use Unity to develop our tower defense game simulator because Unity Technologies has developed a reinforcement learning toolkit for Unity, namely *The Unity Machine Learning Agents Toolkit (ML-Agents).* ML-Agents will be described in more detail in Chapter 6.

4.2 Arknights

This section introduces the basic information of Arknights and some differences in its rules from traditional tower defense games. It also describes the reasons why we chose Arknights as a testing ground.

Arknights is a tower defense mobile game developed by *Hypergryph* [2] (See Figure 4.3). It was released in China on 1 May 2019, in other countries on 16 January 2020. In order not to cause ambiguity, it is necessary first to introduce the differences between Arknights and traditional tower defense games:

[1]Unity Software Inc. is a video game software development company based in San Francisco.
[2]Hypergryph Network Technology Co. Ltd. is a mobile video game development company based in Shanghai, China.

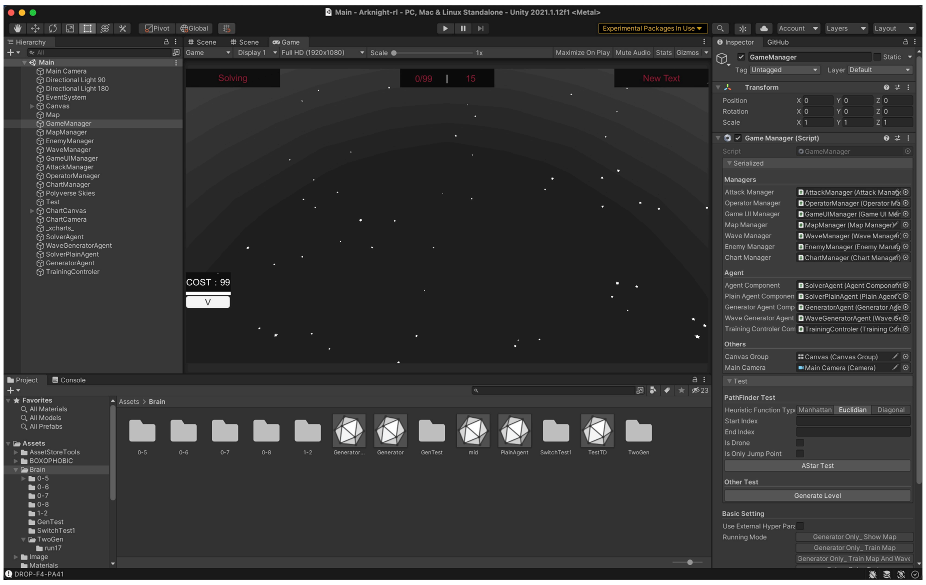
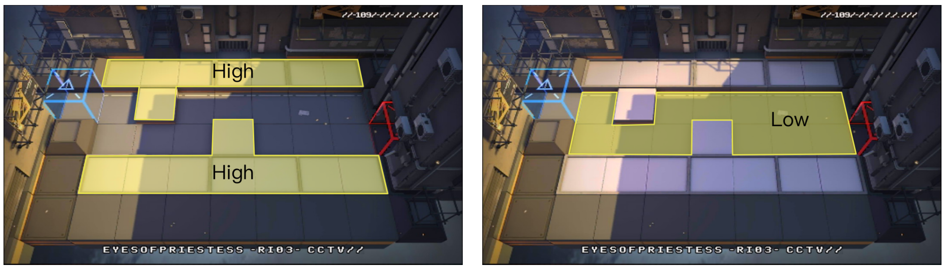


Figure 4.1: Screenshot of Unity game engine.

- While resources in traditional tower defense games are usually obtained by eliminating enemies, resources, which is called Deployment Points (DP) in Arknights, can be obtained by players in three ways:
  - DP regenerates at the rate of 1 DP per second.
  - Some towers can generate additional DP when they eliminate enemies.
  - Some towers can generate extra DP by activating skills.
- In traditional tower defense games, players can upgrade towers when they play the level to enhance their abilities, while there is no such system in Arknights. In Arknights, players can level up towers by using game props before starting the level like a role-playing game.
- The number of towers in Arknights is enormous. Therefore, unlike traditional tower defense games where players can build all kinds of towers while playing a level, they need to select up to 12 types of towers before playing a level. The player can only build these 12 selected towers when playing the level.
- The tiles in Arknights level map can be divided into two classes based on the type of towers that can be built, namely high tiles and low tiles (see Figure 4.2). In general, players can build melee towers on low tiles and ranged towers on high tiles. Besides, most enemies move on paths made up of low tiles.
- In Arknights, players can build towers in the path of enemies to block their progress, and the blocked enemies can attack the towers built by the player.

As mentioned in Chapter 3, PCG studies on tower defense games are generally conducted based on existing game rules or self-created simple game rules. We chose to base our level generation study on the rules of Arknights for three reasons:



(a) The high tiles. (b) The low tiles.

Figure 4.2: Screenshot of a level from Arknights.

- Arknights has a very active player community, and there is detailed information about levels and the numerical aspects of the game in the wiki[3]. This community and information facilitate the development of our tower defense game simulator.
- Arknights is a tower defense game with role-playing games' elements necessitating more intricate level design and difficulty balance to provide a satisfying player experience. It is very challenging and rewarding to study generating such levels.
- The average time to complete a level in Arknights is relatively short, so the reinforcement learning agent has less time to train, which facilitates the conduct of experiments.

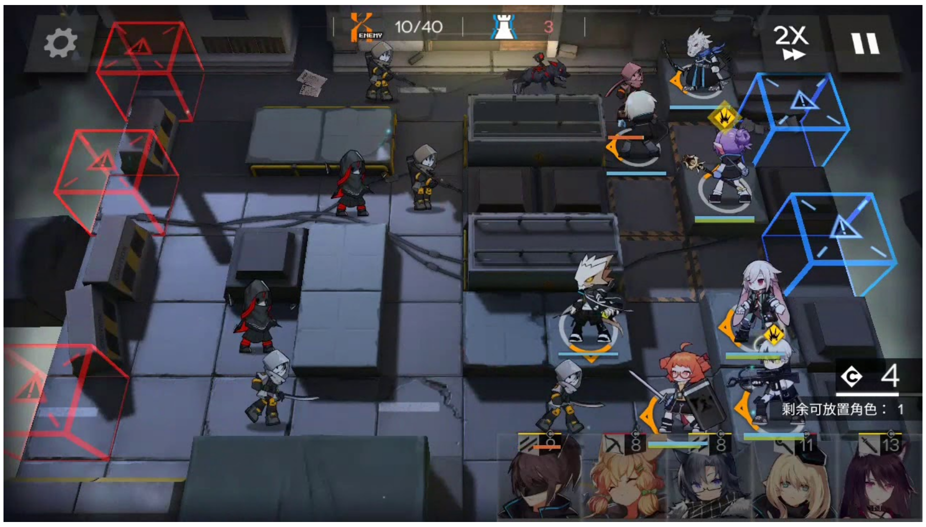


Figure 4.3: Screenshot of Arknights.

4.3 Tower Defense Game Simulator

Arknights is an ongoing game, so game content, including new enemies, new levels, new towers, are constantly being added to the game[4]. To ensure the boundaries of this research, only a portion of this content has been selected for implementation into the tower defense game simulator. This section introduces

[3]http://prts.wiki/ for the numerical aspects of Arknights and https://map.ark-nights.com/ for level information.

[4]The game data mentioned is as of 26 December 2021.

4.3.1 Tiles

We selected 8 of the 25 existing tiles to be implemented into the tower defense game simulator. These eight selected tiles are the fundamental components of Arknights' map. Table 4.1 shows the details of the eight types of tiles we selected.

Table 4.1: Tiles

Tile	Tile type	Description
⬛	Invalid	Padding of map
◼	Low	Component tile of ground path
⬜	High	Component tile of high platform
▨	Low Forbidden	Towers can not be placed on
▨	High Forbidden	Towers can not be placed on
🟥	Enemy Spawn Point	Enemies' spawn point
🟥	Flying Enemy Spawn Point	Flying enemies' spawn point
🟦	Defense Point	Players' territory

4.3.2 Towers

As mentioned in Section 4.2, Arknights has a large number of different types of towers, and the players need to select up to 12 types of towers as optional towers before playing each level. It should be noted that although the tower selection strategy is also an essential part of playing Arknights, this strategy is not part of this study. In this study, we will circumvent considering this strategy by fixing the use of 12 types of towers.

Arknights utilizes a *Gacha*[5] system for players to obtain the towers. We selected 12 types of towers from the existing 217 types of towers that players can get for free to implement and ensure that these 12 types of towers are sufficient to deal with most situations in the game.

Table 4.2 shows the basic information of the towers we chose, where *DP cost* refers to the resources the player needs to build the tower, *Position* refers to what type of tiles the players can build on the tower, and *Block* refers to the number of enemies the tower can block.

4.3.3 Enemies

We selected 10 of the 421 enemies to be implemented into our tower defense game simulator. These thirteen enemies contain the following classic types of enemies in tower defense games:

- Enemies with little HP but fast movement speed.
- Enemies that can fly but cannot attack the tower.
- Ranged enemies that can attack the tower from a distance.
- Enemies with high resistance to physical attacks from towers.
- Enemies with high resistance to magical attacks from towers.

[5]Gacha games, like loot boxes, entice players to spend in-game cash in exchange for a random virtual object. The in-game currency may be earned via gameplay or purchased with real-world money from the game's publisher.

Table 4.2: Basic information of the towers

Code	DP cost	Position	Block	Tower Type	Range	Attack Type
LT05	8	Low	1	Attacker	Melee	Physical
BS04	9	Low	2	Attacker	Melee	Physical
PA12	9	High	-	Attacker	Ranged	Physical
PA44	9	High	-	Attacker	Ranged	Physical
PA61	10	High	-	Attacker	Ranged	Magical
PA41	13	Low	1	Attacker	Melee	Physical
PA13	15	High	-	Healer	Ranged	-
PA43	15	High	-	Healer	Ranged	-
PA45	16	Low	3	Attacker	Melee	Physical
PA14	16	Low	3	Attacker	Melee	Physical
PA42	16	High	-	Attacker	Ranged	Magical
PA15	27	High	-	Attacker	Ranged	Magical

4.3.4 Levels

We selected 21 from 171 mainline levels to be implemented in our tower defense game simulator. Table 4.3 shows the basic information of these 21 levels, where Recommended Level refers to the recommended average level of the towers used by the player to play the level, as given by the game designers of Arknights; Enemies refers to the total number of enemies in the level; Life Count refers to the maximum number of enemies that can be tolerated to reach the defense point before the end of the level:

Table 4.3: Basic information of the levels

Code	Recommended Level	Enemies	Life Count
0-1	Level 1	11	20
0-2	Level 1	14	15
0-3	Level 1	23	15
0-4	Level 1	24	15
0-5	Level 1	24	15
0-6	Level 1	31	15
0-7	Level 1	20	15
0-8	Level 1	40	15
0-9	Level 1	64	15
0-10	Level 1	35	10
0-11	Level 1	37	10
1-1	Level 5	33	10
1-2	Level 5	41	10
1-3	Level 5	36	10
1-4	Level 5	39	10
1-5	Level 5	57	10
1-7	Level 10	41	10
1-9	Level 10	35	10
1-10	Level 10	11	10
2-1	Level 10	50	10
S2-1	Level 10	25	10

4.3.5 Visualization Tools

In addition to the above gameplay content, we also developed a visualization tool for the environment state based on XCharts[11]. With this tool, we can check in real-time through charts whether the state of the game environment is encoded as we expect it to be. Figure 4.4 shows visualization tools we developed.

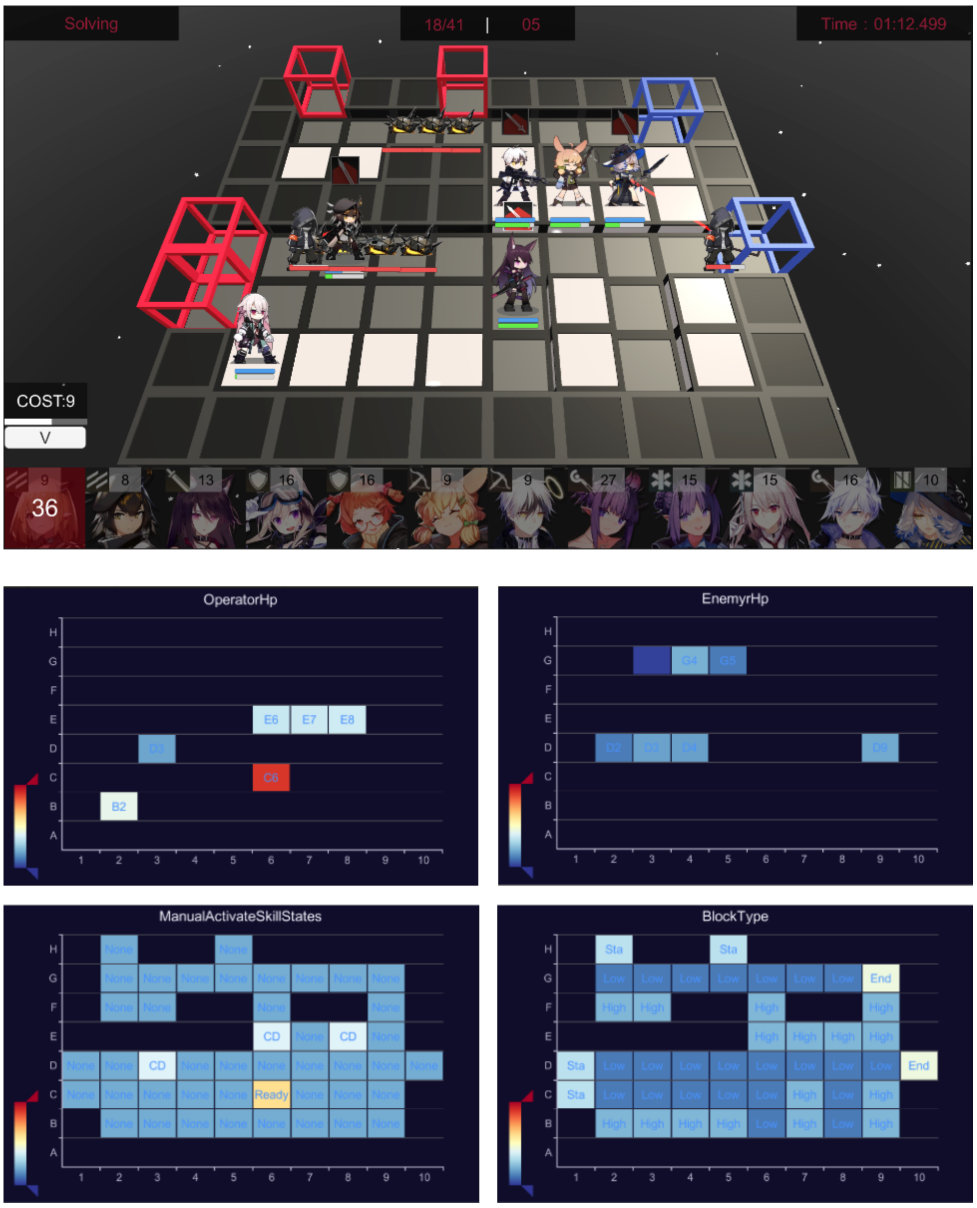


Figure 4.4: Screenshot of the tower defense game simulator (Up). Some charts generated by visualization tools (Down).

Chapter 5

Proposed Methods

The primary purpose of this study is to explore how to perform level generation for tower defense games through reinforcement learning. The method we use is based on the ARLPCG framework[10], where the solver agent and generator agent take action in the environment in turn and design rewards for the generator agent related to the performance of the solver agent.

However, unlike ARLPCG[10], we trained three agents in a tower defense game simulator: the *Solver* for playing levels, the *Map Generator* for generating maps, and the *Wave Generator* for generating wave information.

5.1 Training Process

Before going into the details of the individual parts, it is necessary to describe how we trained the three agents in the tower defense game simulator. The three agents are trained in an iterative way where other agents play with the current trained data when an agent is training. The following is our training process:

1. Read the map and wave information of a level implemented in the tower defense game simulator.
2. The map generator changes the map in the level.
3. The wave generator changes the wave information in the level.
4. Start the level, i.e., the enemies start appearing on the map according to the wave information.
5. The solver trains an episode in the environment, i.e., plays the level until it succeeds or fails.
6. Calculate the reward for the map generator and the wave generator based on the solver's playing result. The details of the calculation will be mentioned in the following section.
7. Check whether the episodes of the map generator and the wave generator are finished. The end conditions of generators' episodes will be mentioned in Section 5.4.4.
    (a) Yes: go to 1.
    (b) No: go to 2.

5.2 The Solver

This section presents how we trained the solver agent from three aspects: state, action, and reward.

We trained the solver through reinforcement learning with *PPO*[12]. We represent the tower defense games' environment states by a vector and a set of map size 2-dimensional tensors. We adjusted the *IMPALA ResNet* 's implement[13] by removing the max-pooling layers and used it to process the map size tensors. This is because using the max-pooling layer would cause unnecessary loss of state information.

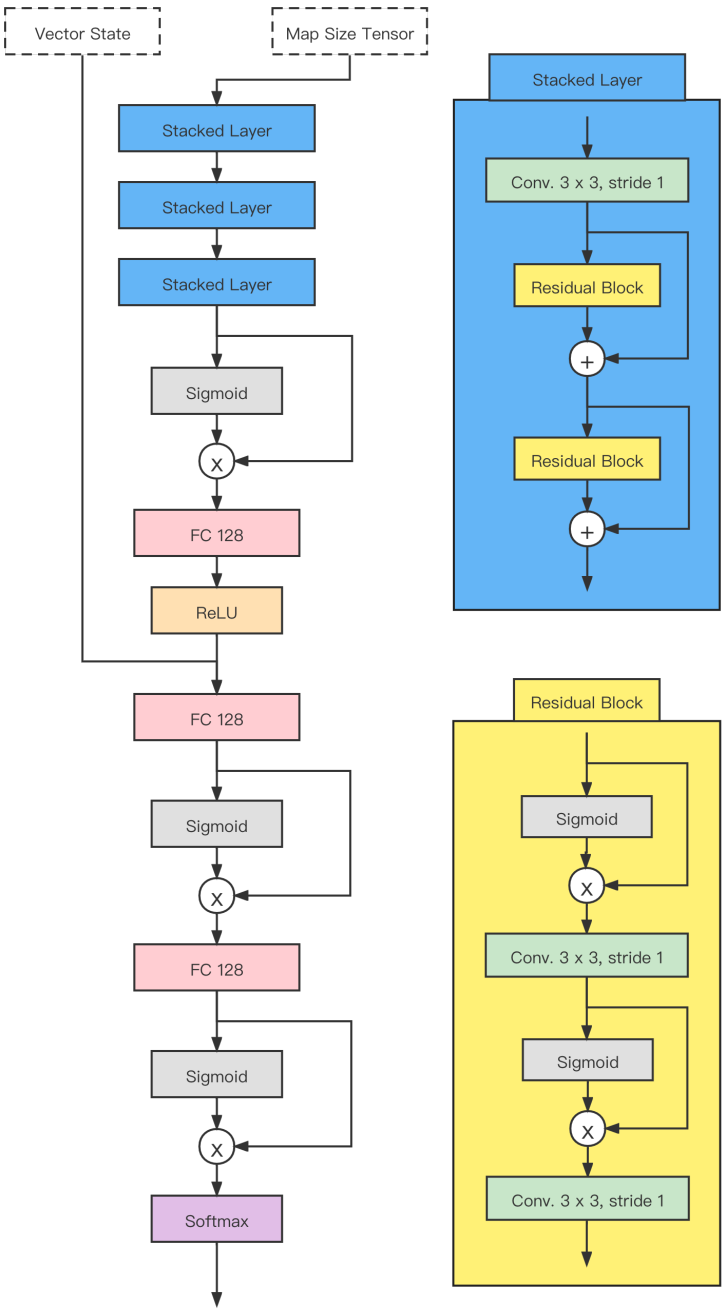


Figure 5.1: Neural network architecture.

5.2.1 State

We encoded the state of the environment based on what human players need to focus on when playing *Arknights*.

Figure 5.2 shows how the states are represented as map size tensors. The detail of the map size observations is presented in Table 5.1. The vector of states will be concatenated to flatten map size tensors before feeding to full-connected layers. Table 5.2 shows the detail of the vector.

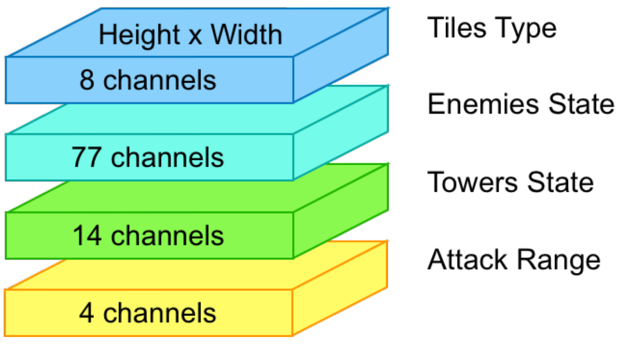


Figure 5.2: Illustration of how the states are represented

Table 5.1: The detail of map size tensors

State	Data Type	Channels
The type of tiles	one-hot	8
If in towers' attack range	one-hot	36
Number of flying enemies	one-hot	6
Number of normal enemies	one-hot	6
Number of ranger enemies	one-hot	6
Number of melee enemies	one-hot	6
Number of enemies who can not attack	one-hot	6
Number of enemies in the next wave	one-hot	6
Physical DPS of existing enemies	float	1
Magical DPS of existing enemies	float	1
Defense of existing enemies	float	1
Magical resistance of existing enemies	float	1
Current HP of existing enemies	float	1
Number of enemies towers can block	one-hot	6
Skill's state of towers	one-hot	5
Defense of existing towers	float	1
Magical resistance of existing towers	float	1
Current HP of existing towers	float	1
Melee DPS of existing towers	float	1
Ranged physical DPS of existing towers	float	1
Ranged magical DPS of existing towers	float	1
HPS of existing towers	float	1

Table 5.2: The vector of states

State	Data Type	Length
Current DP	integer	1
Player's current HP	integer	1
Towers' DP costs	integer	12
Towers' type (ranger or melee)	one-hot	24
Towers' type (attacker or healer)	one-hot	24
Does player have enough DP to place towers	one-hot	24

5.2.2 Action

In Liu et al.'s study[9], the agent's action is defined by simulating the operation of the mouse on the screen. However, with Arknights as a tile-based game, we can reduce the action space by simplifying the operations on the screen to operations on tiles. Theoretically, for each tile, the player can do the following actions:

- Build a tower.
- Destroy a tower on this tile.
- Activate the skill of a tower on this tile.

Obviously, among the above actions taken on tiles, some actions are not allowed by the game rules, such as activating the skill of a tower on a tile that has not yet built a tower. In this study, we call such actions that are not allowed by the game rules *Invalid Actions*, and on the contrary, we call actions that are allowed by the game rules *Valid Actions*. We define the following three types of action spaces according to how to deal with invalid actions:

- The agents with *Wide Action Space* have the largest action space containing valid and invalid actions. When the agent tries to do an invalid action, the action will not be executed, and the agent will be given a negative reward.
- The agents with *Narrow Action Space* have the smallest action space only containing valid actions.
- *Medium Action Space* contains all valid actions and some invalid actions. We get medium action space by removing the invalid action of building a tower without enough resources from the wide action space.

Through defining these three types of action spaces, we can experiment to find the most suitable action space. This experiment will be mentioned in Chapter 6.

5.2.3 Reward

Except for the negative reward that agents are given when they try to perform an invalid action, we designed the reward mainly according to the rules of tower defense games, i.e., we always want agents to destroy as many enemies as possible. Thus, agents receive a positive reward when the tower kills an enemy and a negative reward when the enemy reaches a defense point. We also set some additional rewards, such as a small positive reward for agents after each tower attack on the enemy. We used three reward settings in experiments, and these details will be mentioned in Chapter 6.

5.3 The Map Generator

This section describes the construction of the map generator in terms of state, action, and reward. The map generator uses wide representation from the PCGRL framework[4].

5.3.1 State

Like the solver, the map generator's observable state of the environment has two parts: information about the types of tiles on the map, i.e., the top layer of the Figure 5.2, an 8-channel two-dimensional tensor; the wave information of the level. Table 5.3 shows the status of one wave information, and the wave information of a level consists of multiple such information.

Table 5.3: The states of one wave information

State	Data Type	Length
Enemies' type	one-hot	11
Enemies' spawn point	one-hot	42
Enemies' target defense point	one-hot	21
Enemies' number	integer	1
First enemy's appearance time	float	1
Enemies' interval between the appearance	float	1

It should be noted that because we need to represent the empty wave information, the one-hot encoding length of the enemy type state is 11, although only ten types of enemies are implemented in the tower defense game simulator.

In this study, we do not discuss the case where the number of defense points is greater than 20. Therefore, adding the encoding indicating the empty wave sub-information, the one-hot encoding length of the defense point state is 21.

There are two types of enemy spawn points in our tower defense game simulator, used to spawn regular enemies and flying enemies. In reasonable wave information, enemies should be generated according to their type (regular or flying) at the corresponding spawn point. However, since there is a mismatch between the enemy type and the spawn point type during the generation of the wave information, we also need a one-hot code to represent this situation. Therefore, the length of the one-hot encoding for the enemy spawn point is 42.

5.3.2 Action

The map generator can change the type of tile at any location on the map. However, we have special treatment for the following two situations to ensure that the solver can play the level properly. When these situations occur, the action will not be executed, and the map generator will be given a negative reward.

- After the action is taken, the number of enemy spawn points or defense points on the map is 0 or greater than 20.
- After the action is taken, the enemy's movement path is blocked in the existing wave information.

5.3.3 Reward

The map generator's reward consists of two main parts: *Internal Reward* and *External Reward.* In general, we want to control the basic structure of the generated maps through internal reward and the difficulty of the maps through external reward. The map generator's reward is calculated as:

$$r \doteq \lambda \cdot r_{ext} + w_{int} \cdot r_{int}, \tag{5.1}$$

where r is the reward, r_{ext} and r_{int} are the external and internal rewards, respectively, and $w_{int} \in [0, 1]$ and λ are parameters. λ is the coefficient of the auxiliary input r_{ext}, and the positive or negative of this parameter determines the preference of the map generator for the solver's playing result, i.e., success or failure.

Internal Reward

To calculate the internal reward, we set a series of goals for the map generator before the training starts. When the map generator does an action, the tower defense game simulator will give a reward based on whether the action makes the current map state closer or further away from the goals. The goal is defined by a minimum and a maximum value, denoted as g_h and g_l. For example, if our goal for the generated map is to have 2 to 3 defense points, the minimum value is 2, and the maximum value is 3. In this study, we used calculating rewards used by Khalifa et al.[4] in generating game levels using reinforcement learning. r_{int} reward is calculated as:

$$r_{int} \doteq \begin{cases} p(v_{new}) + p(v_{old}) & (v_{new} < g_l \wedge v_{old} > g_h) \vee (v_{old} < g_l \wedge v_{new} > g_h) \\ p(v_{new}) - p(v_{old}) & \text{otherwise} \end{cases}, \tag{5.2}$$

where v_{old} is the value before the action and v_{new} is the value after the action (e.g., the number of defense points), and $p(v)$, which refers to penalty, is defined as:

$$p(v) \doteq \begin{cases} 0 & g_l < v < g_h \\ -\min(|(v - g_h)|, |(v - g_l)|) & \text{otherwise} \end{cases}. \tag{5.3}$$

In this study, the goals for the map include the following five aspects, where areas refer to the connected areas in the map where enemies can move (see Figure 5.3):

- The number of areas on the map.
- The number of spawn points on the map.
- The number of defense points on the map.
- Whether defense points and spawn points are on the edge of the map.
- Number of specific tiles.

External Reward

When the solver successfully clears a level, it will give the map generator a positive bonus. Otherwise, it will give a negative reward.

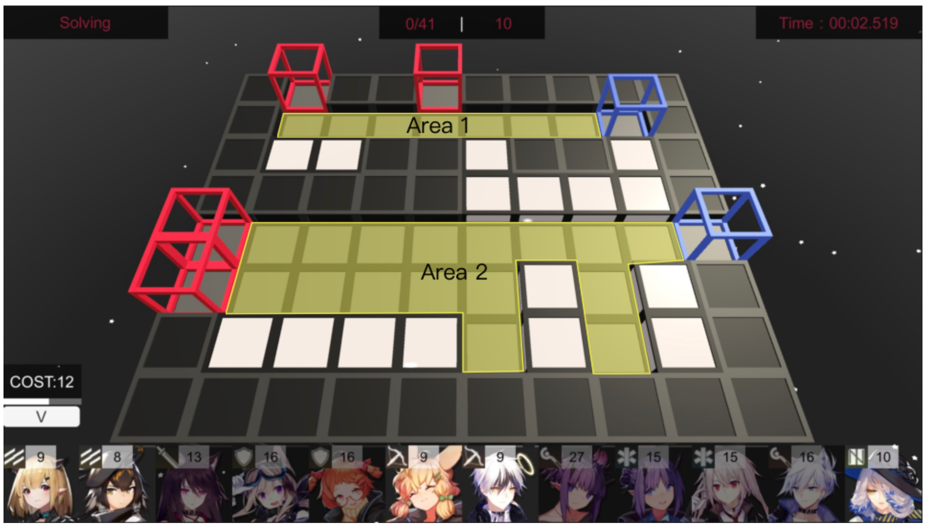


Figure 5.3: Screenshot of level 1-2 from tower defense simulator, which has two areas.

5.3.4 End Conditions of Episodes

An episode of the map generator can be terminated when any one of the following three situations occurs:

- The current number of steps is greater than or equal to the maximum number of steps.
- the current change percentage of the map is greater than or equal to the maximum change percentage.
- The goal has been met.

The maximum number of steps is denoted as S_{max}, and the maximum change percentage is denoted as C_{max}.

5.4 The Wave Generator

This section only briefly introduces the wave generator's state, action, and reward because the construction of the wave generator and the map generator is roughly the same.

5.4.1 State

The only information that the wave generator can observe about the state of the environment is the wave information mentioned in Section 5.2.1.

5.4.2 Action

The wave generator can take three types of actions, i.e., add, remove and modify wave information.

5.4.3 Reward

The wave generator's reward is set in much the same way as mentioned in Section. The difference is that when setting the internal reward, the goal of the wave generator contains the following two aspects.

- The level's length. In this study, we calculated the level's length as the time when the last enemy appeared on the map.
- The total number of enemies in the level.

5.4.4 End Conditions of Episodes

The end conditions of episodes of the wave generator are the same as the map generator.

Chapter 6

Experiments

6.1 Unity ML-Agents Toolkit

In this study, our reinforcement learning agents were trained using the *UnityML-Agents Toolkit (ML-Agents)*[14]. ML-Agents is an open-source project based on *PyTorch*[15]. Because *ML-Agents* is a plug-in for the game engine *Unity*, users can easily create an environment where agents can be trained. Therefore, *ML-Agents* has been used in research related to reinforcement learning[10][16]. This section presents the hyperparameters we use in training.

6.1.1 Hyperparameters

In machine learning, a hyperparameter is a parameter whose value is used to control the training process. In this section, we use the same nomenclature of hyperparameters as ML-Agents.

- *batch_size*: Number of experiences to collect before updating the policy model.
- *buffer_size*: Number of experiences in each iteration of gradient descent.
- *learning_rate*: Initial learning rate for gradient descent.
- *beta*: Strength of the entropy regularization, which makes the policy more random.
- *epsilon*: Influences how rapidly the policy can evolve during training.
- *lambd*: Regularization parameter used when calculating the generalized advantage estimate.
- *num_epoch*: Number of passes to make through the experience buffer when performing gradient descent optimization.
- *learning_rate_schedule*: Determines how learning rate changes over time. We only used *linear* in our experiments, which means that the learning rate will be decayed linearly, reaching 0 at max steps of training.
- *save_steps*: Number of trainer steps between snapshots.
- *team_change*: Number of trainer steps between switching the learning team.
- *swap_steps*: Number of *ghost steps* between swapping the opponent's policy with a different snapshot. A ghost step refers to a step taken by an agent that is following a fixed policy and not learning.

- *window*: Size of the sliding window of past snapshots from which the agent's opponents are sampled.

6.2 Experiment about the solver action space

Because of the limited number of prior studies on the application of reinforcement learning to tower defense games, we need to do some preliminary experiments to verify which of the action spaces mentioned in Section 5.2.2 is more effective for our tower defense game simulator. This section presents the experiment on solver agents' action space and the results. It should be noted that only solver agents were used for training in this section.

6.2.1 Preconditions

In this experiment, we used three reward settings: *Reward Setting A*, *Reward Settings B*, and *Reward Settings C*. Three reward settings are presented in Table 6.1, where N_e is the total number of enemies in the level, and N_l is the total number of life count in the level.

The detail of the environment is presented in Table 6.2. Table 6.3 shows the hyperparameters configuration in this experiment. We ensure the reproducibility and stability of the experiment by setting the random seed to a fixed value. In this experiment, we set the value of the random seed to 1.

Table 6.1: Reward Settings of the Solver

Reward Value \ Reward Setting Action	A	B	C
Take an action	-0.5	-0.5	-0.5
Take an invalid action	-1	-1	-1
Agents destroy a tower in 3s after building	-0.5	-0.5	-
Tower heals another tower	0.05	0.05	-
Tower attacks enemies	0.05	0.05	-
Tower eliminates a enemy	1	$10/N_e$	-
Tower is destroyed by enemies	-5	-5	-
Player's lives count is deducted	-10	$-10/N_l$	-
Eliminate all enemies	-	10	10
Lose all life count	-	-10	-10

Table 6.2: The detail of the environment

Name	Version
Python interpreter	3.8.8
PyTorch	1.7.1
CUDA	11.0
ml-agents	0.26.0
Unity	2021.1.1.12f1

Table 6.3: Hyperparameters Configuration

Hyperparameters	Value
batch_size	16
buffer_size	120
learning_rate	0.0003
beta	0.005
epsilon	0.2
lambd	0.99
num_epoch	3
learning_rate_schedule	linear

6.2.2 Results

We trained all solvers for 100000 steps. The learning curves of agents are presented in Figure 6.1 and Figure 6.2. Then, we let solvers make inferences for 100 episodes in the same environment where they are trained, and Figure 6.3 shows the results. Figure 6.4 and Figure 6.5 show histograms of the inference results for each level.

Because the life count given to the player at the beginning of each level is not fixed, in this experiment, we use the remaining life count as a percentage of the initial life count to express the result, i.e., when the value is 1, it means that the agents eliminated all the enemies; when the value is 0, it means that the agents lost all the life count.

6.2.3 Discussions

From the experimental results (see Figure 6.3), we can learn that regardless of which reward setting is used, the agents with medium action space have the best overall performance, the agents with wide action space have the second-best performance, and the agents with narrow action space have the worst performance[1].

We also learned that agents performed best when using reward setting A, second best when using reward setting C, and worst when using reward setting B. By looking at the learning curves in Figure 6.1 and Figure 6.2, we can see that the learning curve of agents using reward setting A is significantly more unstable than that of agents using reward setting C. We believe that this is because, in training, it is rare that all enemies are destroyed, or the player's life count is reduced to 0. This results in a slight difference in the cumulative reward for each episode, while the auxiliary reward in reward setting A increases the difference. Therefore, we believe that the auxiliary rewards encouraged agents to explore the action space to some extent, such that agents who used reward setting A performed better. Regarding the reason why agents with reward setting B performed the worst, on the one hand, we think it also comes from the fact that it is relatively rare to eliminate all enemies or to reduce the player's life count to 0, which causes both rewards to be ineffective. On the other hand, we believe that we reduced the reward for eliminating enemies and the penalty for reaching the defense point in reward setting B, which caused the agents to be insensitive to the auxiliary rewards for these two actions.

In this experiment, we used levels that random players can not clear but can

[1]In our previous experiments, randomly generated random seeds were used, which is inappropriate.[17].

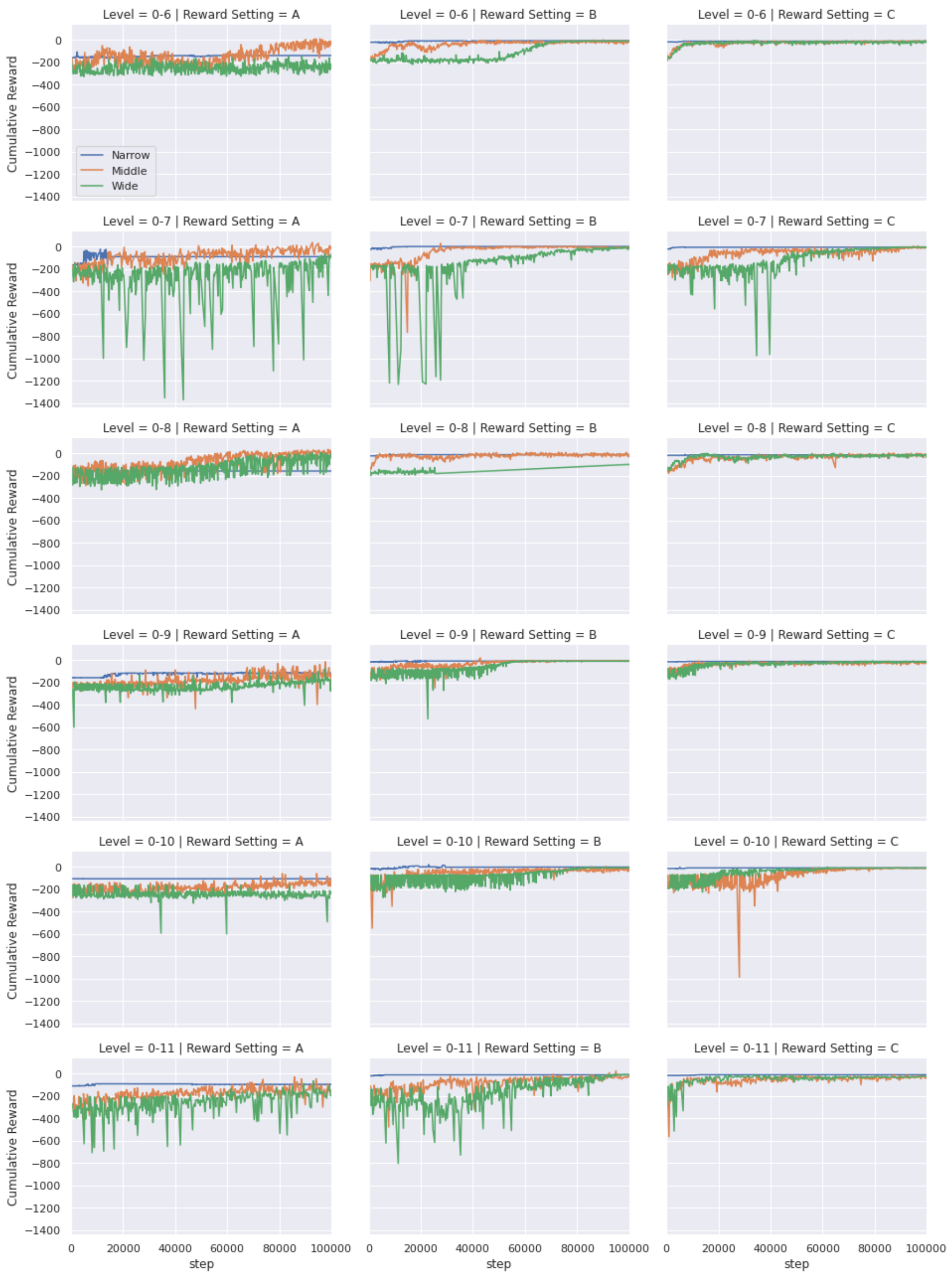


Figure 6.1: The learning curves of agents trained in level 0-6, 0-7, 0-8, 0-9, 0-10, 0-11.

be cleared by experienced players. For comparison, level 0-1 and level 0-2 can be cleared by random players because their initial life count is larger than the total number of enemies (see Table 4.3). From Figure 6.6, we can see that the success rate of agents using reward setting A and having medium action space is greater than 50%in 7 out of 11 levels. Therefore, we believe it is possible to use reinforcement learning to train agents to clear the levels of tower defense games.

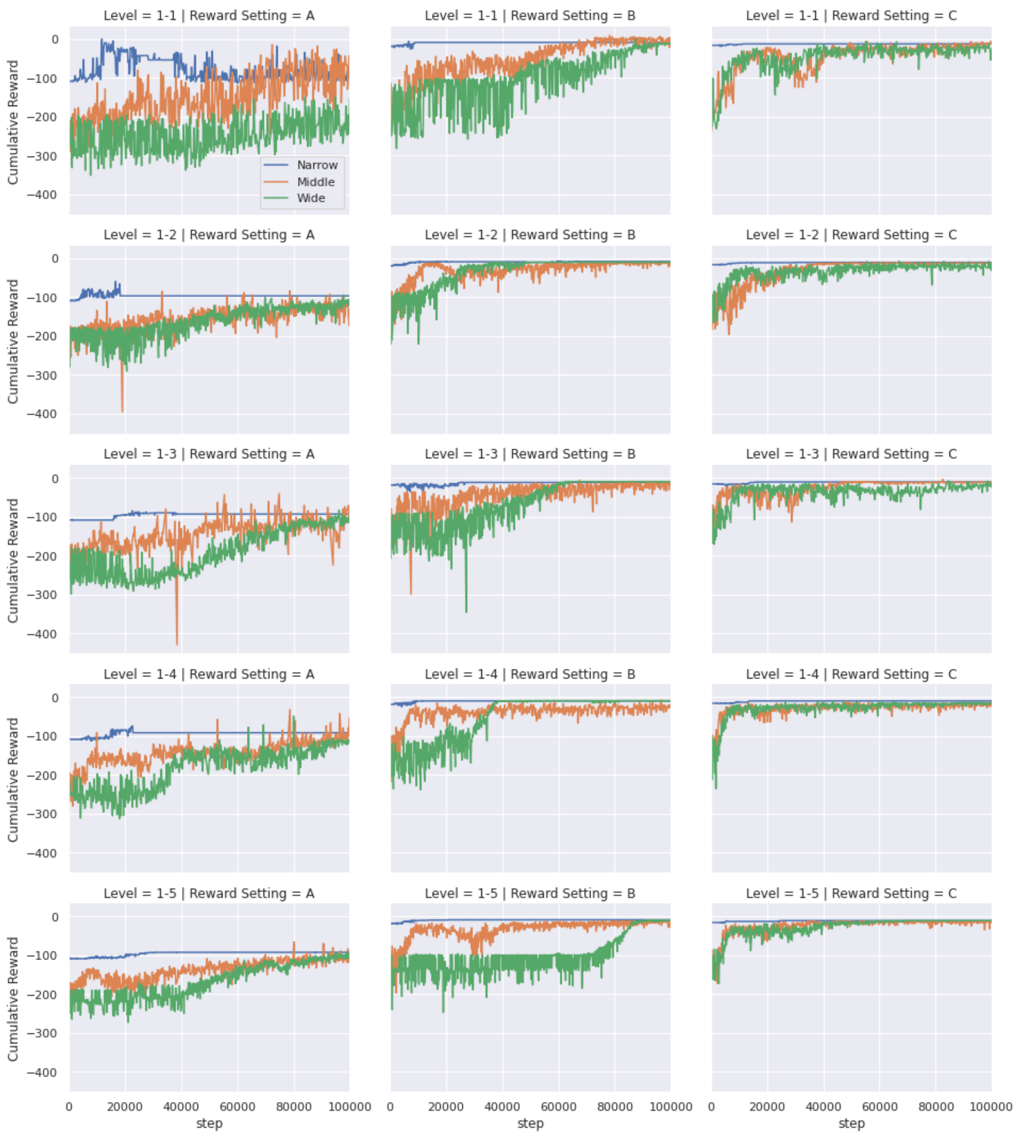


Figure 6.2: The learning curves of agents trained in level 1-1, 1-2, 1-3, 1-4, 1-5.

6.3 Experiments of Level Generation

In this experiment, we conducted level generation based on level 0-1. This means that at the end of each episode of generators, the game level adjusted by generators will be restored to the original level 0-1.

6.3.1 Preconditions

Based on the experiment results about the solver action space, we used the reward setting A (see Table 6.1) for the solver. Table 6.6 shows the parameters related to the calculation of the reward and end conditions of episodes for generators. In this experiment, we define the external reward as 1 when the solver passes the level and -1 when it fails.

As mentioned in Section 5.3.3, to calculate the internal rewards of generators, we need to set goals for generators before training starts. We set goals for the map generator and the wave generator based on our understanding of tower defense game levels. For example, we set a goal that the number of tiles where towers

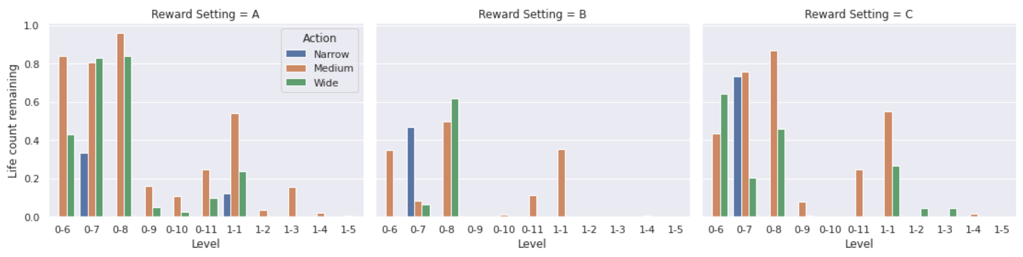


Figure 6.3: The remaining life count of agents after playing levels with different reward settings and action space.

can not be placed on (i.e., the low or high forbidden tiles) in the level's map should not exceed ten percent of the total number of tiles. The reason for setting such a goal is that if there are too many forbidden tiles, the player's freedom will be significantly reduced, which will make the level too difficult or not interesting enough. Besides, the goals we set were all based on a vision we had for generators: generators can adapt easy levels reasonably to be relatively difficult ones.

The goals of the map generator and the wave generator are presented in Table 6.4 and Table 6.5, respectively. In addition to the goals presented in the table, we also set some special goals. Although these goals are difficult to describe in the table, it is still possible to calculate the rewards by the method we mentioned in Chapter 5. We set the following three special goals:

- All defense points and spawn points are on the edge of the map.
- The wave information consists only of enemies with codes 17 and 01.
- There is no wrong wave information, where the wrong refers to the wrong type or non-existence of the spawn point the non-existence of the target defense point.

The detail of the environment is presented in Table 6.2, which is the same as the second experiment about the solver action space. Table 6.7 shows the hyperparameters configuration of the solver and generators. In this experiment, we set the value of the random seed to 1.

Table 6.4: Goals of Map Generator

Target	Min Value	Max Value
Number of areas	1	1
Number of spawn points	3	3
Number of defense points	2	2
Percentage of low forbidden tiles	0%	10%
Percentage of high forbidden tiles	0%	10%

Table 6.5: Goals of Wave Generator

Target	Min Value	Max Value
Level ’ s length	60s	120s
Number of enemies	60	100

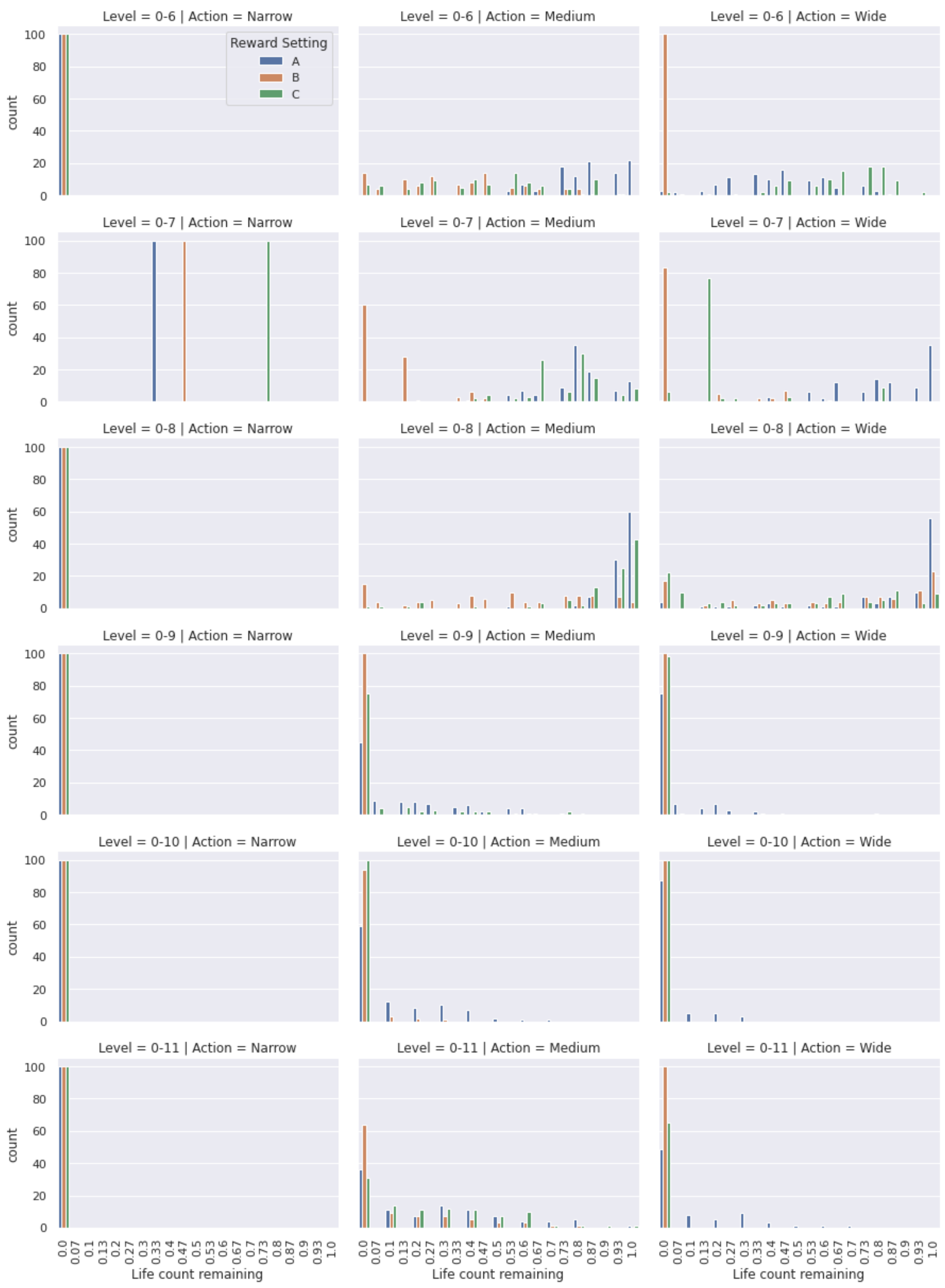


Figure 6.4: Histograms of the remaining life count for level 0-6, 0-7, 0-8, 0-9, 0-10, 0-11.

6.3.2 Results

We trained the solver 4000 episodes according to the training process defined in Section 5.1, which means that both the map generator and the wave generator were run for 4000 steps. Then, we let the agents inference in the order they were trained and let generators generate levels at the end of episodes. We generated ten levels based on level 0-1, i.e., ten sets of maps and wave information, respectively. These ten levels were generated at the end of the 48th, 95th, 141st, 189th, 238th, 282nd, 335th, 384th, 435th, and 477th episode of solver agents.

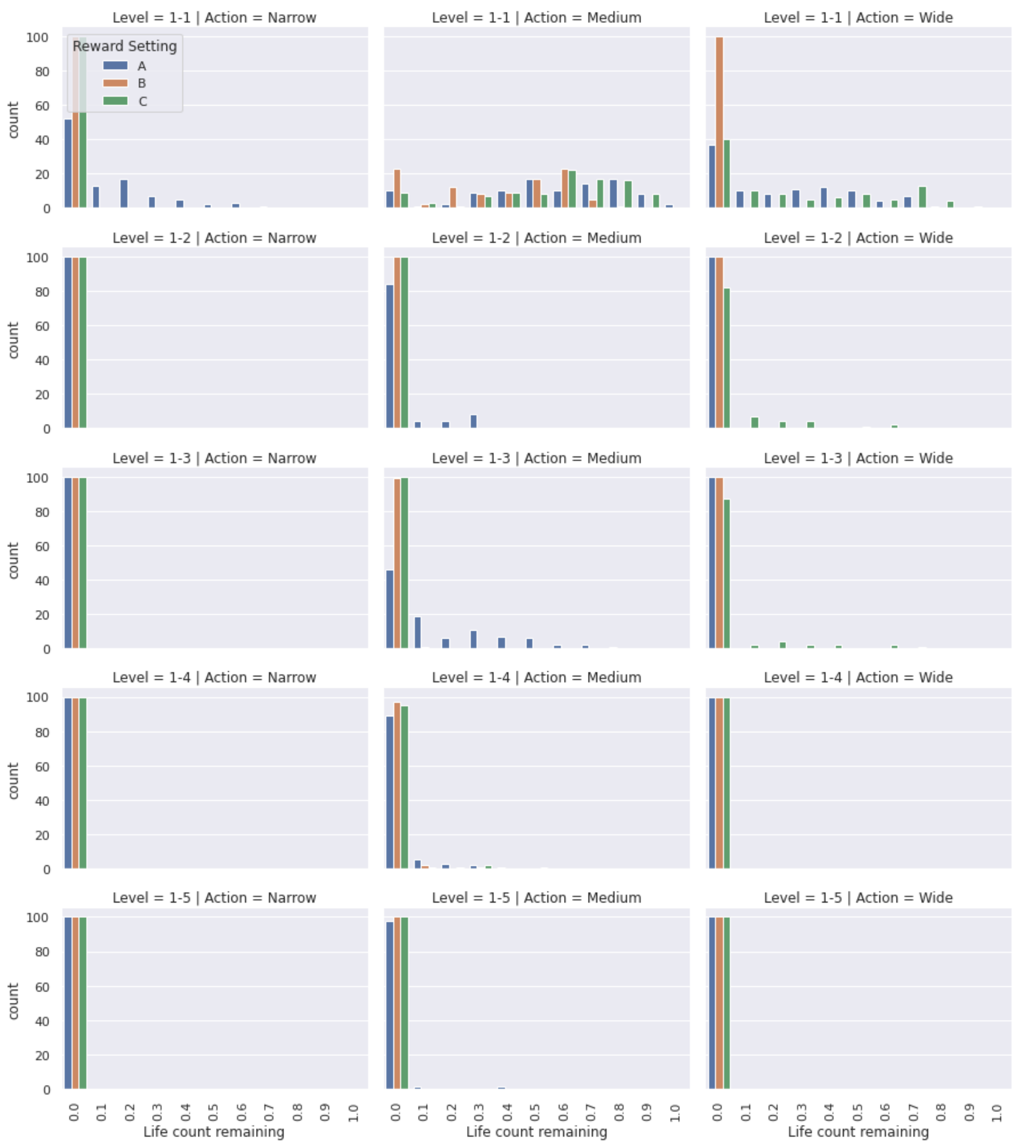


Figure 6.5: Histograms of the remaining life count for level 1-1, 1-2, 1-3, 1-4, 1-5.

Figure 6.7 shows the original maps of levels 0-1 and the map from the generated maps. All of the generated maps are presented in Figure 6.8.

Figure 6.9 shows the statistics of the generated levels and the original levels. The distribution of the number of enemies in all generated levels and the distribution of the number of enemies in the original levels are presented in Figure 6.10.

Table 6.6: Parameters of Generator

Parameters	Value
λ	-0.45
w_{int}	0.9
S_{max}	100
C_{max}	60%

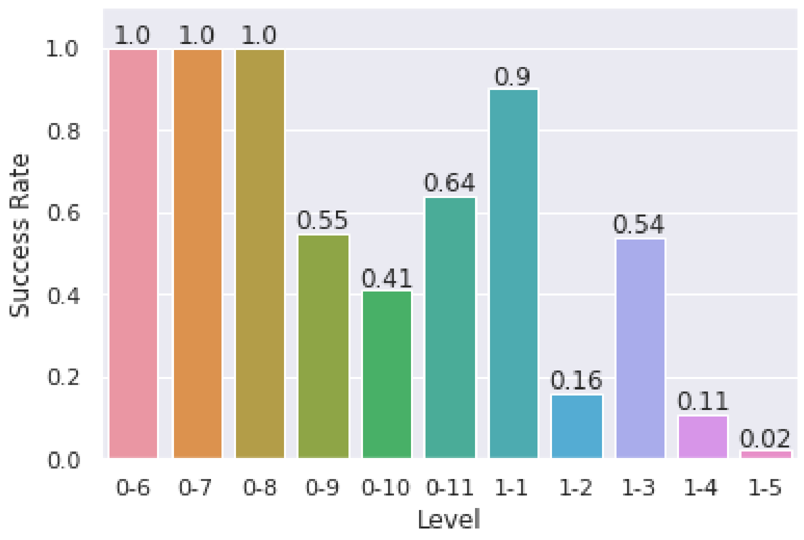


Figure 6.6: The success rate of agents using the reward setting A and the medium action space.

Table 6.7: Hyperparameters Configurations

Value \ Type of Agent / Hyperparameters	Solver	Generator
batch_size	16	10
buffer_size	120	100
learning_rate	0.0003	0.0003
beta	0.005	0.0005
epsilon	0.2	0.2
lambd	0.99	0.99
num_epoch	3	3
learning_rate_schedule	linear	linear
save_steps	5000	25
team_change	20000	75
swap_steps	400	10
window	10	10

6.3.3 Discussions

As can be seen from Table 4.3, the original level 0-1 player can clear unconditionally because the player's initial life count is greater than the total number of enemies. At the same time, we encourage the level generator to generate levels that make solver agents vulnerable to failure by setting λ to a negative value. As a result, we found that the map generator tends to replace tiles in the map where towers could be placed with tiles where towers could not be placed, while the wave generator had a trend of increasing the number of enemies. By doing these, the generators adjust levels that the solver can unconditionally clear to those that the solver cannot clear.

Regarding the distribution of the number of enemies in the wave information,

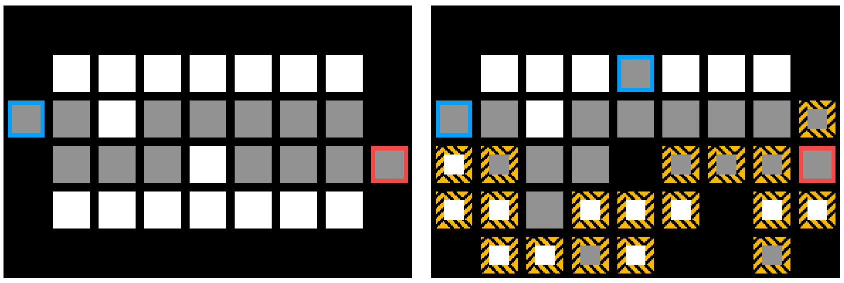

(a) (b)

Figure 6.7: The original maps of levels 0-1 (Left) and a generated map (Right).

we can learn from Figure 6.9 and Figure 6.10 that wave generator prefers to increase the difficulty of levels by increasing the total HP of enemies in all waves instead of the average DPS. Also, the wave generator seems to be more likely to generate waves with enemies coded A9. Enemies coded A9 can attack towers from a distance. We think this may be because the wave generator can quickly destroy player-built towers by increasing the number of such enemies to get more enemies to the defense point.

Also, as shown in Figure 6.7, generators changed many tiles at the bottom of the map to tiles that do not allow players to build towers. We think this can be seen as the map generator's exploration of increasing the level's difficulty at the map design aspect.

Finally, we checked all the generated levels, and we found that among the ten sets of maps and wave information generated, there were no maps and wave information that reached the goals we set in Table 6.4 and Table 6.5. By observing the generation results, on the one hand, we think the insufficient training of the map generator is the main reason for not generating maps that meet the goals. On the other hand, we found that the number of enemies in the waves generated by the wave generator was usually less than the number of enemies we set. We believe this is caused by the solver not being able to clear the level when the number of monsters has not reached the goal.

We chose a level from the generated levels as an example to illustrate how it does not meet our goals. Figure 6.11 shows the map of the generated level. As shown in Figure 6.12, the generated map only meets our requirement for the number of defense points, i.e., two defense points. The map has one enemy spawn point, and this number is less than our goal. Because the total number of tiles in this map is 54, the number of low forbidden tiles and high forbidden tiles are both greater than our goal. There are three areas on the map, and this number is also greater than our goal. One of the two defense points is not at the edge of the map, which also does not meet our goal.

Figure 6.13 shows the wave information of the generated level, where the meaning of the data names in the header row is as follows:

- *Code*: The code of the enemy.
- *Number*: The number of enemies.
- *Interval*: The interval of each enemy appearance time when the number of enemies is greater than 1.
- *StartTime*: The appearance time of the first enemy.

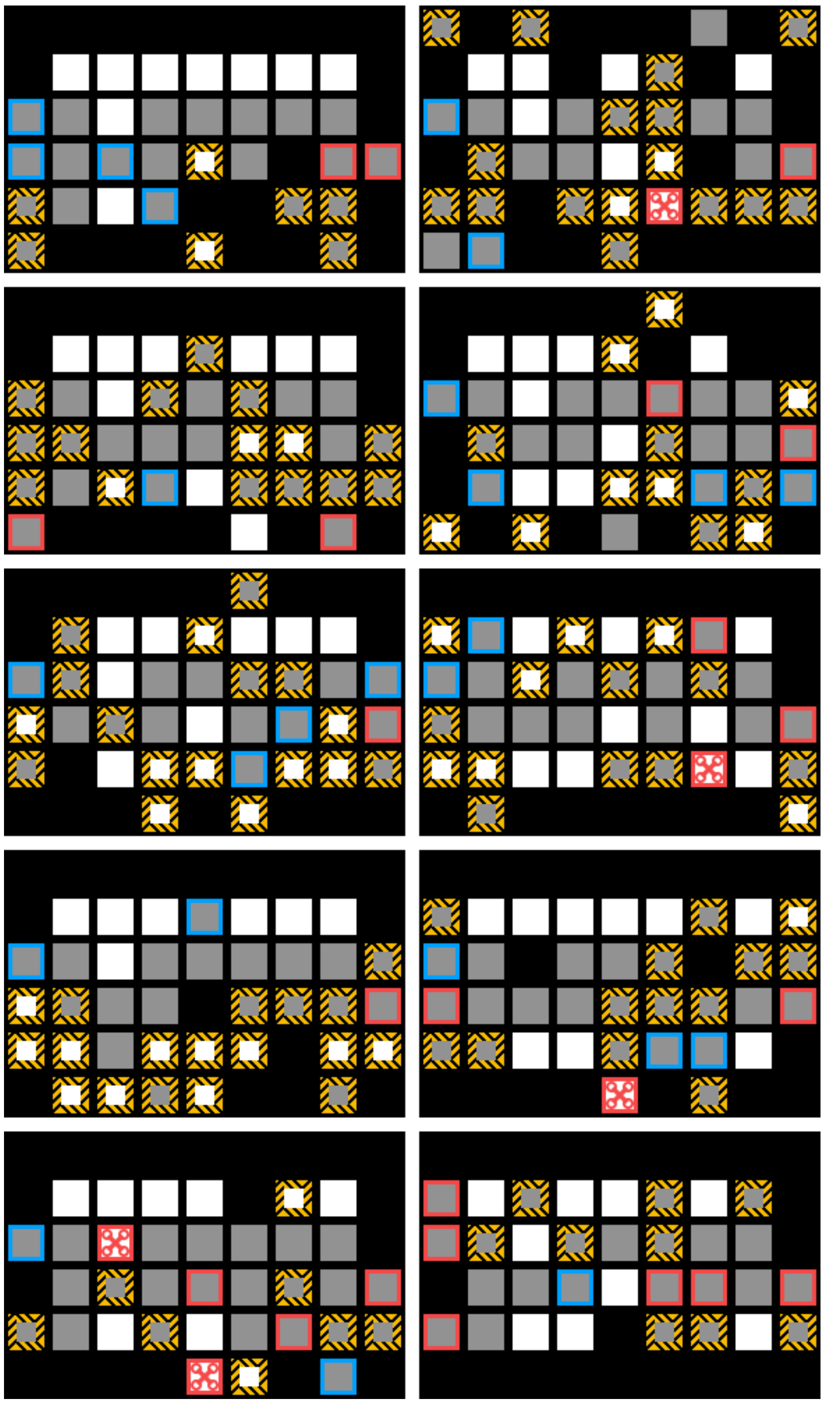


Figure 6.8: The ten generated maps.

- *Start*: The location of the enemy spawn point on the map. This location consists of a letter and a number. Figure 6.11 shows how the letter and number indicate the map location.
- *End*: The location of the target defense point on the map.
- *IsWrong*: Whether the wave information is wrong.
- *LastEnemyStartTime*: The appearance time of the last enemy.

As shown in Figure 6.13, the last enemy's appearance time exceeds our goal for the length of the level. Also, the total number of enemies in this generated

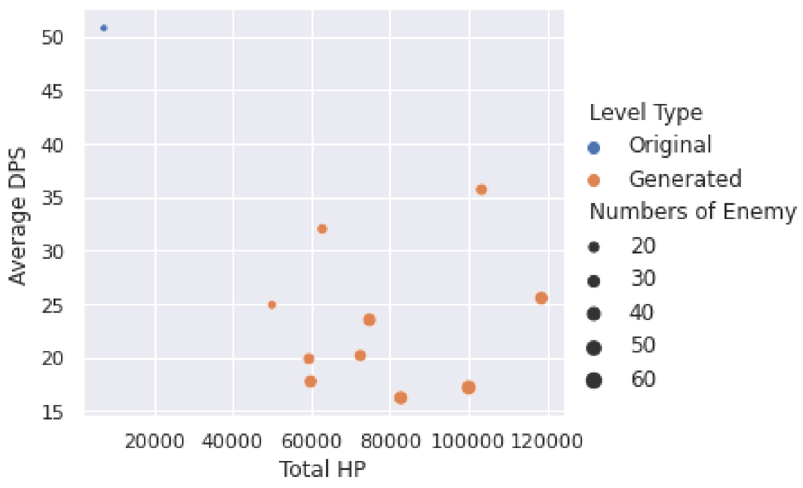
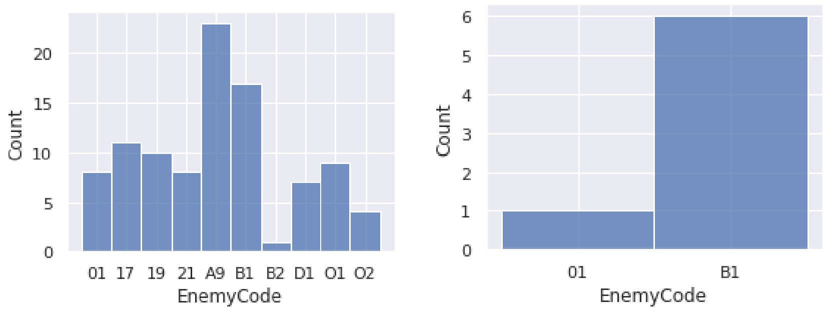


Figure 6.9: Statistics of the generated wave information and the original wave information.



(a) (b)

Figure 6.10: The distribution of the number of enemies in all generated levels (Left) and the distribution of the number of enemies in the original levels (Right).

wave information is 55, which is less than our target. Also, the wave information contains enemies other than the codes 17 and 01 and has the wrong wave information.

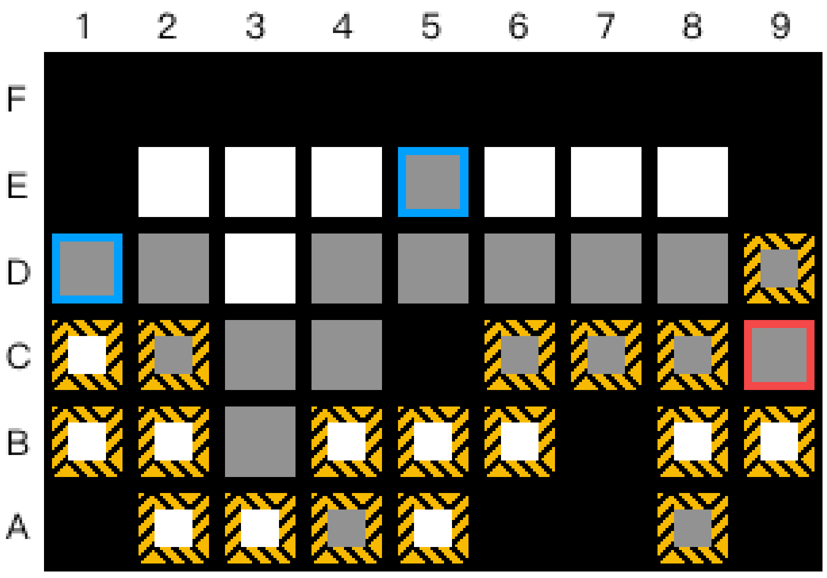
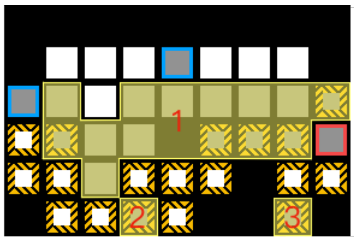
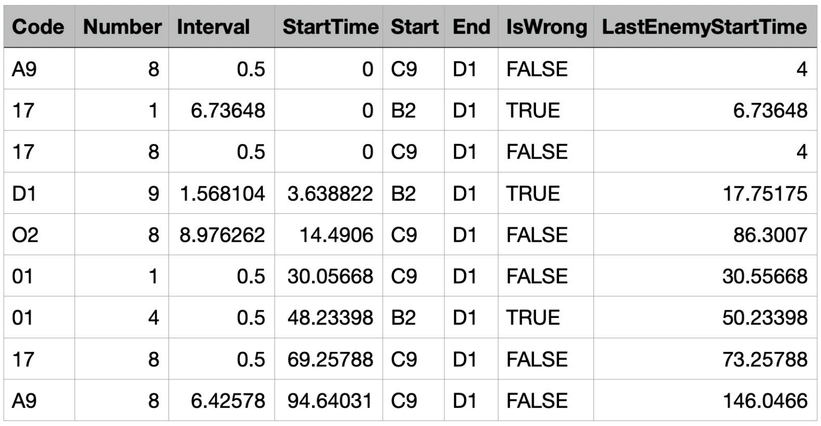


Figure 6.11: The map of a generated level.



Tile	Tile type	Number
▨	Low Forbidden	7
▧	High Forbidden	11
🟥	Enemy Spawn point	1
🟦	Defense point	2

Figure 6.12: The ares (Left) and the number of tiles (Right) of the generated map (Left).



Code	Number	Interval	StartTime	Start	End	IsWrong	LastEnemyStartTime
A9	8	0.5	0	C9	D1	FALSE	4
17	1	6.73648	0	B2	D1	TRUE	6.73648
17	8	0.5	0	C9	D1	FALSE	4
D1	9	1.568104	3.638822	B2	D1	TRUE	17.75175
O2	8	8.976262	14.4906	C9	D1	FALSE	86.3007
01	1	0.5	30.05668	C9	D1	FALSE	30.55668
01	4	0.5	48.23398	B2	D1	TRUE	50.23398
17	8	0.5	69.25788	C9	D1	FALSE	73.25788
A9	8	6.42578	94.64031	C9	D1	FALSE	146.0466

Figure 6.13: The wave information of a generated level.

Chapter 7

Conclusions and Future Work

We developed a tower defense game simulator based on the rules of an existing commercial game. We implemented some content based on the original game into the tower defense game simulator. Not only agents but also human players can play this tower defense game simulator, making it possible to test generated levels with human players in the future. We plan to iterate the tower defense game simulator we developed into an open-source project in the future. We first need to remove existing non-original art assets in the tower defense game simulator that would raise copyright issues. Then, we also need to improve the ease of use of this project in terms of game content extensions and the configuration of agents. Finally, we also need to write documentation to facilitate user participation in the open-source project. We believe that such an open-source project could be convenient for many researchers.

Because there is limited research related to the application of reinforcement learning in tower defense games, we first need to explore how to train agents that can successfully clear tower defense game levels through reinforcement learning. To do this, we defined three different sizes of action spaces (narrow, medium, and wide) and trained them in 11 levels using three different reward settings. A conclusion we can draw from the results of the experiment is that we can train an agent capable of clearing levels of Arknights with PPO and ResNet. Besides, we found that the solver agent with the medium action space, which we defined in Section 5.2.2 tends to have the highest success probability. At the same time, we also found a trend that the success probability of agents decreases as the difficulty of the game level increases. We believe that the results of the current stage of research can inform future work, especially regarding the application of reinforcement learning in tower defense games.

We have also conducted experiments on tower defense game level generation. Although we have not been able to generate levels that meet our goals at this stage of the experiments, our analysis of the trend of generators to make modifications in map and wave information leads us to believe that it is still possible to generate tower defense game levels using the ARLPCG-based approach.

Through the level generation experiments, we also identified a need for improvement in the training process we are using now. In the current training process, the map generator and the wave generator only take one action, while the solver is trained for a whole episode. This difference in the frequency of taking actions leads to a situation where the solver is no longer learning, but the map generator and wave generator are not yet fully trained. In the future, it is possible to introduce experiments in which generators can modify multiple aspects of the game level at once, e.g., the map generator modifies the types of two tiles at once.

Also, only experiments with negative λ values were conducted in this study, and in the future, we plan to conduct experiments with positive λ values. Further, we plan to verify whether the practice of connecting the reward function to the network as an auxiliary input in the work of Gisslén et al.[10] is valid for generating tower defense game levels.

The method we are using to calculate internal rewards is the one proposed by Khalifa et al.[4]. However, we note that there is still space for improvement in calculating rewards. In this study, the reward calculation when the value is outside the goal range before and after the action is taken is not unified (see Equation 5.2). We plan to unify it and conduct experiments.

In addition, We believe that the agents with the ability to clear tower defense game levels are vital for generating tower defense game levels. Such agents can be used as part of a generative model and for automatic testing of game levels. In this study, the purpose of the experiments on the solver is to find a relatively proper action space and reward setting for the solver in the level generation experiments, so we only made a few modifications to the existing neural network architecture and conducted a few experiments. In the future, we would like to improve the capability of reinforcement learning solver agents by improving the neural network architecture, hyperparameter tuning, and reward shaping. After successfully generating levels, we can introduce tests with human players to verify whether our generated levels can be used in game development.

References

[1] A. M. Connor, T. J. Greig, and J. Kruse, “Evaluating the impact of procedurally generated content on game immersion,” *The Computer Games Journal*, vol. 6, no. 4, pp. 209–225, 2017.

[2] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, “Procedural content generation via machine learning (pcgml),” 2018.

[3] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, “Bootstrapping conditional gans for video game level generation,” 2019.

[4] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, “Pcgrl: Procedural content generation via reinforcement learning,” 2020.

[5] P. Avery, J. Togelius, E. Alistar, and R. P. Van Leeuwen, “Computational intelligence and tower defence games,” in *2011 IEEE Congress of Evolutionary Computation (CEC)*, pp. 1084–1091, IEEE, 2011.

[6] J. Togelius, N. Shaker, and M. J. Nelson, “Introduction,” in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research* (N. Shaker, J. Togelius, and M. J. Nelson, eds.), pp. 1–15, Springer, 2016.

[7] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, “Search-based procedural content generation: A taxonomy and survey,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[8] J. Öhman, “Procedural generation of tower defense levels,” 2020.

[9] S. Liu, L. Chaoran, L. Yue, M. Heng, H. Xiao, S. Yiming, W. Licong, C. Ze, G. Xianghao, L. Hengtong, D. Yu, and T. Qinting, “Automatic generation of tower defense levels using pcg,” in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG ’19, (New York, NY, USA), Association for Computing Machinery, 2019.

[10] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, “Adversarial reinforcement learning for procedural content generation,” 2021.

[11] monitor1394, “unity-ugui-xcharts.” https://github.com/monitor1394/unity-ugui-XCharts.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.

[13] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, “Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures,” in *International Conference on Machine Learning*, pp. 1407–1416, PMLR, 2018.

[14] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," 2020.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[16] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, *et al.*, "Open-ended learning leads to generally capable agents," *arXiv preprint arXiv:2107.12808*, 2021.

[17] Y. Xu and T. Tanaka, "Procedural content generation for tower defense games:a preliminary experiment with reinforcement learning," in *Proceedings of Game Programming Workshop 2021*, vol. 2021, pp. 93–97, nov 2021.