**Implementation**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
1. Overview**

Implementation

involved the construction of multiple interweaved parts to create a reuseable reinforcement learning pipeline. To facilitate the training and testing of procedurally generated content on a model’s ability to generalise, a code structure for generating solve-able levels, that are reproduceable for further testing.  
Alongside this, a custom environment for gymnasium is needed to define the training environment and how models interact with it.  
Once these are in place, models can be built with Keras and trained within the environment. Focus was put on two different styles of model, being Deep Q learning and Policy Gradient.   
This lead to the creation of a model construction and training pipeline that allowed to quick creation and tuning of hyperparameters for continued testing of models across the environment as it grew in complexity.

**2. Core Features**

* **2.1 PCG**
  + **Map generation (algorithms, cycle removal, graphs of map building and parameter effects?)**

The procedural content generation pipeline was a critical element to tackle early on, to ensure consistency across

training, and enable reuseable testing environments to gather informed results when tuning hyperparameters and testing model structures and modifications.  
It was designed to use a node based network structure, to allow for the use of graph pathfinding algorithms, both for model use in the future and for the designing of maps.

Levels are generated by first creating a ‘base’ maze-like level that looks a little like a lattice, every second row, every second node is a path node, while all

other nodes are walls.

Using Kruskal’s algorithm a ‘perfect’ maze is built as a foundation for the rest of the level design process.  
A perfect maze is one featuring no cycles, essentially a path describes a minimum spanning tree across a graph.  
By slightly modifying the algorithm, levels are created where nodes also serve as walls, and a further algorithm is used to create cycles within the perfect maze, creating imperfect mazes that are customise-able in their nature.  
By tuning the algorithm parameters, levels can be generated in different ways, adding cycles in different manners leading to large open spaces, or smaller pockets of space. The process can be tuned to reach desired levels of cycle addition, and how biased towards wall removal removing neighbouring walls too.  
The process can also scale a level of hazards introduced into map generation, and features the ability to spawn enemies, keys to lock the exit until gathered, and of course the exit itself.  
By way of seed, the entire process is reproduceable given the same parameters and seed are used.  
Because of this, most training took place on a singular seed, attempting to replicate a medium difficulty environment.

A black and white background with white hashtags

AI-generated content may be incorrect.

***Figure 5 –*** *Pre Kruskal level*

A black and white background with white hashtags

Description automatically generated

***Figure 6 –*** *Post-Kruskal’s, ‘perfect’ maze, seed=79*

A black and white background with white hashtags

Description automatically generated

***Figure 11a*** *- ‘pockets’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1, ‘open’ tiles = 26,   
open ratio = 0.21*

A black and white background with white hashtags

AI-generated content may be incorrect.

***Figure 11b*** *-‘Manhattan dist.’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1,‘open’ tiles = 32,   
open ratio = 0.26*

A black and white background with white hashtags

Description automatically generated

***Figure 11c*** *-‘Sigmoid dist.’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1,‘open’ tiles = 52,   
open ratio = 0.43*

A diagram of a company

Description automatically generated

***Figure 13 –*** *Testing plan*

The PCG System timeline was completed, unfortunately the actual training of model timeline was not fully realised.

* **2.2 Environment**
  + **Creation**

**To get started with the reinforcement learning training we need to define an environ ment for the models to interact with using the Ferrama Foundation’s gymnasium library, we can do just this.**

**The core of the environment is providing an observation space and well defined functions On how the environment the reinforcement learning agent model is in reacts and interacts with the agent. This really makes up the foundation of the whole training pipeline, with a step function that assesses the agent’s action its impact on the environment and the reward signal that flows back to the agent. There must also be a well defined reset function that keeps this environment fresh for each new training episode without letting any Leakage of prior information into future training episodes and overall keeping the process running smoothly.**

**Alongside this comes rendering which allows for a visual pie game window to watch the training process unfold and better assess from a human perspective any potential pitfalls in behaviour and generally allowing more insight.**

**The environment that has been defined for this project allows for a constructor that can take a variety of arguments that will tailor make the environment for a particular training run. It allows for different observation spaces and reward shapings as well as defining how we’d like to render the training process if at all And stores references to the map currently being used in training the agent and all relevant information that may be needed.**

* + **Observation Spaces and Reward Settings**

**The observation space and reward settings are incredibly important for reinforcement learning process. In this environment the observation spaces available kept growing as testing continued, allowing for many different variations, whether that be coordinates for important locations, map states encoded either via one hot encoding or scalar interpretation, or numerical features such as agent health or using a star to input path length to the model.**

**Insert observation space figure here**

**Different models responded to different observation spaces, and it provided a very interesting outlook for testing How input features can greatly influence training.**

**Equally as important is how you shape the reward signal for a model, And this environment allows for a custom reward settings object to be posted, which defines reward amounts for all of the possible interactions with the environment and agent can have.**

**These rewards range from high spas rewards such as reaching the exit or the agent dying when touching hazards too often, but also more dense piecemeal rewards such as moving towards the goal or away from it giving both positive and negative smaller rewards, or punishments for being stuck in one place and running into walls.**

**Insert reward settings figure here**

* + **Step / Reset**

**The step function is arguably the most important function within the environment. It takes the proposed action of the model and assesses the outcome on this interaction with the environment, relaying back a reward signal, along with another observation of the environment as it now stands post action, and whether or not the training episode has finished either due to success, failure, or truncation due to time.**

**The reset function he’s also very important and is there to cleanly set the environment back into a state ready for another training loop with no leakage of information from any previous training episode, with all moving parts ready for a training loop.**

**Insert step and reset function figure here**

* + **PCG Capability**

**As the goal of this project was to assess procedural content generation when used within a training loop to improve generalisation, the environment needs to have some sort of capability to provide this.**

**Levels are created in the environment using a map options object, which defines a series of hyperparameters for the procedural content generation system to create a map of the desired settings and most importantly seated for reproducibility. The environment when initialised will take a map options object and itself create the map based on these options, there is another function that takes this created map and sets it to the map currently being used by the environment. From this point all other logic relates to this map, be it the step function or reset function or rendering.**

**Using this given a list of seeds, it is possible for the environment to simply change map at any point in time, and thus a training loop can be defined and performed over many maps using a singular environment.**

**Insert map creation figure here**

* + **Rendering**
    - **Heatmaps**

**While rendering is not necessary for the training loop, it does provide a useful insight into the training process and allows for human analysis of training.**

**I aimed to further this capability by not just providing a render function that allowed for rendering during training, but rather to render the history of training, allowing the rewatch of an entire training segment or simply parts of it. Further to this I developed a addendum that allowed for instead of rendering a training history episode by episode, rather it could deliver and save a heat map which gives a insightful and speedy overview of an agent’s behaviour across an entire training loop.**

**Insert heat map rendering function here**

* **2.3 Models**
  + **PG Testing, basic map**
  + **PG Testing, seed 1**
  + **DQN, FDQN, DBDQN seed 1**
  + **A2C seed 1**
  + **PG PCG**
  + **A2C PCG**
  + **PG seed 1 Post Train PCG TEST**
  + **A2C seed 1 Post Train PCG Test**
  + **PG PCG Post Train PCG TEST**
  + **A2C PCG Post Train PCG Test**
* **2.4 Testing Pipeline**
  + **Customiseable, efficient testing and logging pipeline for graphs, models, histories and allowing for quick copy and paste hyperparameter tuning**
  + **Now that there was many moving parts such as the environment PCG system, model building pipeline, training loop functions Analysis functions, a pipeline developed which allowed for ease of consistent reproducible and easily tunable testing. The process begins with defining an environment and instantiating it with the desired reward shaping and map options. Then using model configuration classes models can easily be structured with a desired layers neurons, hyperparameters, and descriptive information for future data saving and any sort of analysis and charting functions. Alongside this training configuration classes allow for a tailored training loop specific to each model, covering training parameters such as learning rate, exploration policies, the extent to which those exploration policies will decay or not, which particular type of model we are working with to provide the right training loop, how many iterations to run for, how many episodes per iteration, and much more.**

**Insert training configuration, model configuration, test and train pipeline figure here**

**Then comes a build and train multiple models function. This takes both the training configuration model configurations and a reference to the environment and builds the model in the desired manner and completes the training process given all the specified settings. This function returns a history of training as well as the trained models themselves for saving and future use. The training history object is built during the training process and accounts not just for reward history, but also agent locations throughout each episode, which enables the environments history rendering functionality.**

**Once training is complete the plot reward history function allows for an easy plotting of rewards across the training loop for each iteration, or in the case where multiple episodes happen per iteration an average across those episodes. This was a primary tool in assessing the success of each test and any modifications to a model and its training parameters, reward shaping, or observation space.**

**Insert plot reward history**

**3. Technical Challenges**

The process was not without difficulties, and it look a good amount of time to build a model that was able to converge on a solution for a very basic environment, before moving on to a more complex one.  
Partly due to my own inexperience in the setting and how certain parameters might be tuned, and also errors during implementation.

- time difficulties with running models

- it can be hard to discern what is going wrong, is it model tuning, observation space, reward shaping?

One of the biggest lessons coming out of this, is that honing in on what is troubling a reinforcement learning model can be a tricky process. So many factors are at play whether it be reward shaping, observation space, training hyperparameters or model structure, or even the training algorithms themselves, and the many adjustments and improvements that can be made to vastly affect a model’s learning.

A significant amount of time was spent changing parameters, messing with observations and features, adding, or tuning reward signals, and what felt like sometimes blindly trying to fit all of these puzzle pieces together into something that would work. It is often said that a lot of what goes on behind machine learning is dark arts and this definitely affirmed that for me. That being said the science of it, the search through hyperparameters and seeing the impacts that each change could have inspired me to keep going and assured me that with experience the process could be a smoother one.

It took quite some time to reach a model that converged at all on simply a basic environment, even longest still on a complex environment. Deep Q Learning did not converge no matter what was tried.

Given the nature of reinforcement learning coming with a large amount of computation, there was time constraints with simply how long it would take to run each model. At first the most basic iteration of the map and an incredibly simple policy gradient model did not take too long to run, a span of 10 minutes or less. However models like double DQN, or training on hundreds of different seeds, It could quite easily take many hours.

Some of this is due to errors in implementation that sometimes led to a lack of efficiency and increased computation time, and indeed there’s probably many ways the code could still be improved.

**4. Results**

* + **PG Testing, basic map**
  + **PG Testing, seed 1**
  + **DQN, FDQN, DBDQN seed 1**
  + **A2C seed 1**
  + **PG PCG**
  + **A2C PCG**
  + **PG seed 1 Post Train PCG TEST**
  + **A2C seed 1 Post Train PCG Test**
  + **PG PCG Post Train PCG TEST**
  + **A2C PCG Post Train PCG Test**

**The process of building the models from scratch it started instead of from a place of rough guidelines tutorials and exploration to try and figure out what was going to be the best starting point for my environment a lot of grid Weld examples out there used really small grids whereas mine was starting off with a 15 by 15 grid testing and building started from first creating the procedural content generation system then moving on to uh the environment which I used to create a simple setup that generated a bare bones basic map with the agent started in one corner and the exit was located in an opposite corner from top left to bottom right.**

**Models were built and tested on this bare bones map to get a first idea of the model structure and rough hyper parameters needed to get us very simple policy gradient model converging on the most basic implementation of my grid world.**

**From their exploration ensued trying out various depths and sizes of layers learning rates exploration policy’s from epsilon greedy to implementing Boltzmann exploration experimenting with the temperature decay rates pencil. Press enter once an initial model was found to converge I had a better understanding of what approach to take with a more complex environment from there the environment was developed further to allow for the procedural content generation system to provide customised maps complete with hazards and varying complexities of wolves. This was not the full extent of the procedural content generation system or my plans for the environment but this was the next step up in difficulty from that bare bones map. This is where a majority of the project lay in trying to explore hyperparameters and get models to converge on this more complex setting. This proved to be a much more difficult task then simply converging in a bare bones environment. Indeed deep Q learning never truly converged in this environment with my testing whether it be using straight deep Q networks or deep U networks with fixed targets or double deep queue networks. Policy gradient models however did converge after extensive testing and hyperparameter searching.**

**During this testing process it was not just model structure or hyperparameter tuning but also side by side developing new features to add to the training configuration or model configuration allowing for Allowing for various techniques like learning rate scheduling or experience replay.**

**Alongside this was also the development of the environment and the observation space that could be fed to the models as well as how rewards were delivered and allowing for more reward shaping. This can be seen with the reward settings class here.**

**Insert reward settings class figure here**

**Insert training config model config figure here**

**The process of tuning every enforcement learning model has been an enlightening one and often it’s very hard to know exactly what is either holding your model back or driving improvements is it your reward shaping and the signal that’s being given back from the actions it takes? Is it the hyperarameters you set for your training loop the learning rate how fast its exploration policy decays exactly how it decides to explore? Is it the depth you’ve allowed the model and its ability to fit or over fit the environment? Or is the input and what you’ve allowed the model to see giving everything it needs to or too much?**

**With hundreds of models developed and tested, hyperparameters tuned, model structures tried, observation spaces developed, reward shapes fed- I did manage to bring some models to converging on this more complex environment.**

**By this point given how long the tests take I found myself running out of time to explore everything fully, and found that deep queue networks not converging was holding me back from attempting the rest of the project.**

**At this point it was determined the best possible route was to continue with the original goal of the project and put the procedural content generation system to work by testing what converged models I did have on a set of 100 unseen seeds.**

**Up until this point every model had been trained on seed one for consistency through training. At this point I chose to test the performance of an already trained model on 100 unseen levels. Seeds 101 to 200 we used for the post training runs.**

**As expected these baseline models trained on one environment only over fit massively but they did manage to score well on some levels.**

**Insert post train figures here**

**It can be seen from the figures above that even with extensive overfitting the models still were able to perform on a few levels this surprisingly did not correlate extensively with sheer similarity to the original seed one. There was the slightest correlation with With the distance that the door had to the original location of the door in seed one where the model was trained.**

**5. Further Work**

**- DQN working**

**-Heirarchical Model**

**-Further testing with better models and CNN specifically**

**- I had CNN as a goal to add after converging with simpler networks, but the truth is that the DQN struggled despite a lot of attempts to improve it, and CNN may have helped with giving spatial structure meaning and leading to a good model**

**- Overfitting on coordinates and perhaps full map view, maybe better with moving vision grid and features that change based on surroundings and relation to exit for generalisation**

**-Curriculum learning using PCG system to tune levels during multiple level process, see if it leads to better rewards.**

A diagram of a diagram

Description automatically generated

***Figure 1 –*** *Training baseline*

**6. Training with PCG:**

A diagram of a company

Description automatically generated

***Figure 2 –*** *Training with PCG pipeline*

**7. PCG:**

A white paper with black text

Description automatically generated

***Figure 3 –*** *Kruskal’s Algorithm (from [3])*

The algorithm changes are based on a paper [3] where they add steps to take a generated perfect maze via Kruskal’s, but add further connections between nodes, based on a degree of cycle bias, and further horizontal and vertical bias as shown in figure 4.

This idea of varying levels of ‘bias’ in the addition of cycles will prove useful in being able to scale the difficulty in the generation of levels, from practically empty for lowest degree of difficulty, to a ‘perfect’ maze, for the highest level of difficulty.

A math equations on a white background

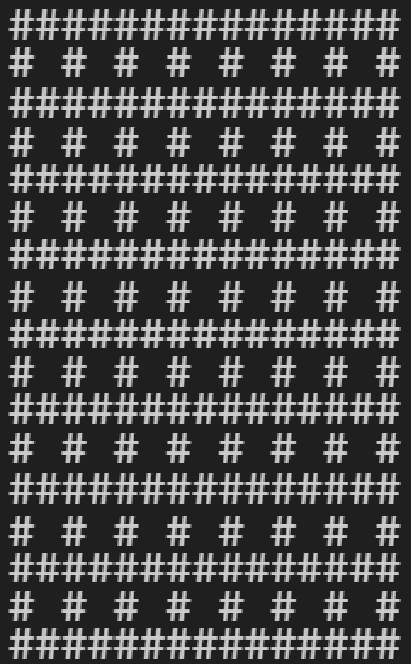
Description automatically generatedA white paper with black text and numbers

Description automatically generated

***Figure 4 –*** *‘non-perfect’ maze generation (from [3])*

The algorithm used in this project utilises the core of the pseudocode and cycles bias while ignoring horizontal and vertical bias. It is modified further to provide methods of varying how the levels are formed, with a focus on creating open areas within training levels to different degrees.

It is also important to note, that the generation of mazes differs for this project, as the paths are not just the connections between nodes, with walls being inferred by lack of connection between nodes. Rather, every path or wall is a tile within a fifteen by fifteen tensor where every outside tile is a wall. The set of nodes used for Kruskal’s are the set of every second tile within this grid, separated by wall tiles. Edges then, are the wall tiles separating them [Figure 5].



***Figure 5 –*** *Pre Kruskal level*

A black and white background with white hashtags

Description automatically generated

***Figure 6 –*** *Post-Kruskal’s, ‘perfect’ maze, seed=79*

*A screen shot of a computer program

Description automatically generated*

***Figure 7 –*** *UnionFind for Kruskal’s*

Connections are made using a UnionFind system for efficiency [Figure 7], and once a perfect map is generated [Figure 6], the next step is cycle addition.

The challenge with this system is the fact we can only considered a hash-like pattern of tiles, leaving on central wall always between a set of 4 nodes, which differs from the methods described in ‘Non-perfect maze generation using Kruskal algorithm’ [3] and usual use cases.

For cycle addition then, all remaining walls must be considered in the set for removal, lest the training level end up with a pattern of untouched walls evenly spaced apart – providing a homogenous and uninteresting display of generation.

**7.1 Cycle addition / wall removal**

There are three methods outlined for adding cycles that prioritise adjacent walls for further removal, the algorithm can provide a higher decree of ‘clustered’ removal, and create pockets of open space, rather than a fragmented maze with many small walls.  
all three methods inherit the method of cycle bias from ‘Non-perfect maze generation using Kruskal algorithm’ [3], in that a number of cycles to be added is computed from the floor of the number of remaining walls, multiplied by the cycle bias [Figure 8].



***Figure 8 –*** *Calculation for number of cycles to add*

The method for the creation of levels also features a seed parameter, which allows for reproducibility of results for testing and evaluation.

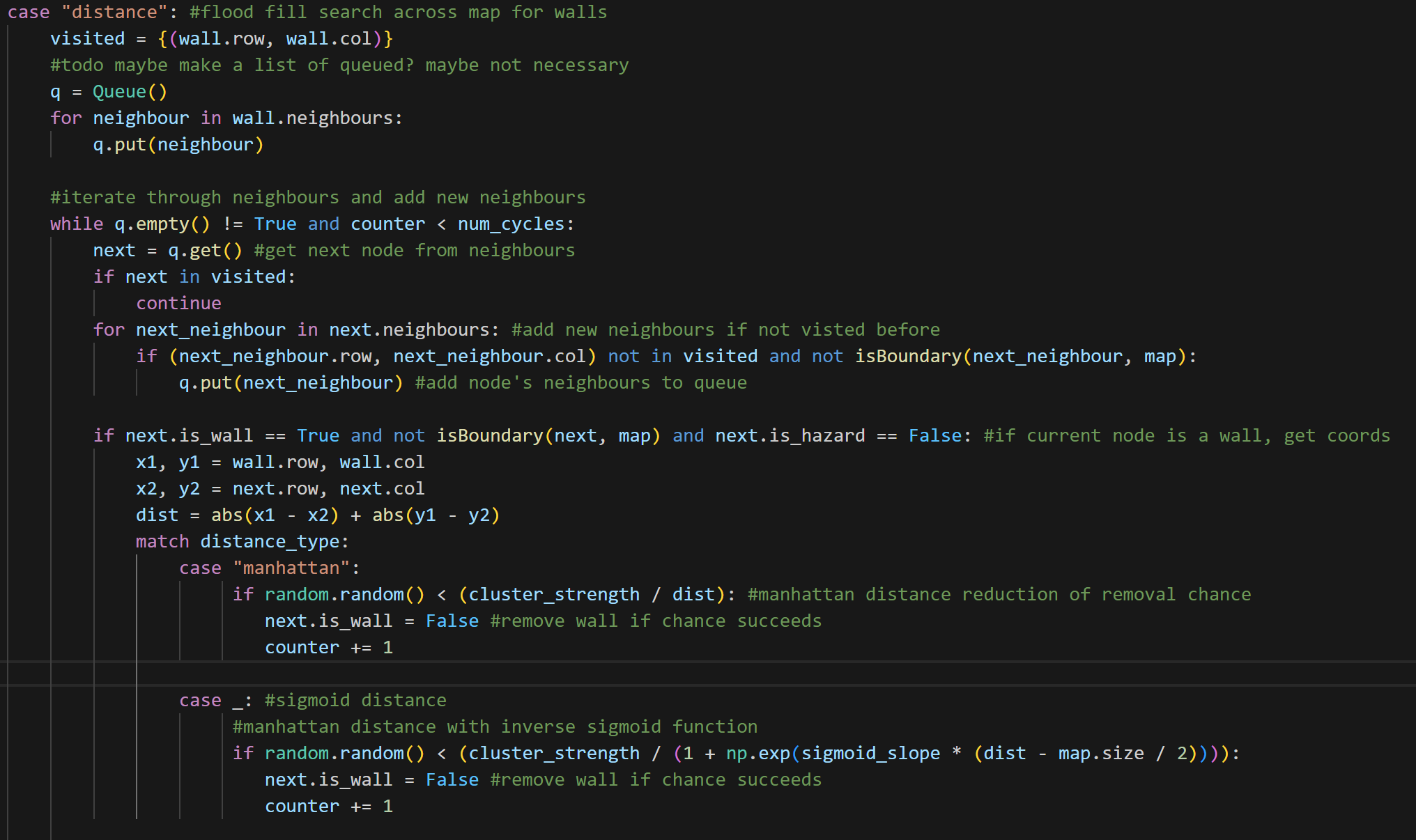
***Pocket:***

When using ‘pocket’ [Figure 9] as the removal method, the algorithm will select a random wall from the remaining walls in the generated perfect maze, and remove it.   
Then, all walls within a five by five area centred on the removed wall are added to a list.  
Each one has a roll of chance against a ‘cluster strength’ value, as an input to the algorithm, determining whether it too is removed.  
This method focuses on creating numerous localised pockets of removal within the training level. [Figure 11a]

A computer screen with text

Description automatically generated

***Figure 9 –*** *Pocket method of cycle addition*

******

***Figure 10 –*** *Distance methods of cycle addition*

***Manhattan / Sigmoid Distance:***

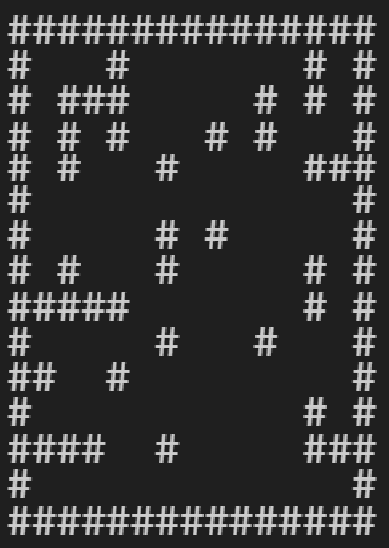
These two methods are based on the flood fill algorithm. Firstly a random wall is selected from the remaining walls. Then a flood fill begins from that wall, queuing all its neighbours to be visited. As each node is visited, it’s neighbours are also queued to be visited, excluding any that have already been queued and thus ‘visited’.  
As each node is visited, there is a percentage based chance to make it a path if it is a wall. This is based on the cluster strength input divided by either the Manhattan distance, or a weighted sigmoidal curve of the distance from the original tile.

The outcome is generally larger pockets of open space, spread location while surrounded by tighter traditional maze paths. By nature of the function, the sigmoid distance tends to stay strong locally, eventually falling off as distance grows. The Manhattan distance weakens faster, earlier, tending to lead to more spread wall removal, rather than one big open area.  
In the following examples of generation, a hash symbol denotes a wall, while open space is a path tile. Generation is all completed using the same seed for the random number generator involved in chance rolls and path shuffling for the algorithm. Cycle bias is set to 0.5, which will remove 50% of remaining walls after perfect maze generation. Cluster strength is set to 1, which is the strongest possible setting for favouring each clustering techniques ‘dice’ rolls. [Figure 10, 11b, 11c]

A black and white background with white hashtags

Description automatically generated

***Figure 11a*** *- ‘pockets’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1, ‘open’ tiles = 26,   
open ratio = 0.21*



***Figure 11b*** *-‘Manhattan dist.’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1,‘open’ tiles = 32,   
open ratio = 0.26*

A black and white background with white hashtags

Description automatically generated

***Figure 11c*** *-‘Sigmoid dist.’  
Params: seed = 79, cycle bias = 0.5,   
cluster strength = 1,‘open’ tiles = 52,   
open ratio = 0.43*

Following all of these methods, the degree of ‘openness’ can be measured within a level, by counting the amount of non-wall tiles that have no immediate (up, down, left and right) neighbour tiles that are walls. This can be viewed as an integer count, or a ratio based on the total possible ‘open’ tiles. The maximum total is the sum of 11x11 in the case of a 15x15 map, as boundary tiles and tiles next to the boundary are excluded from the count. Therefore the ratio is calculated as: open tiles / (121).

This allows for the evaluation of the expected ‘difficulty’ of the layouts within the levels, for use within the training process.

**7.2 Entity Generation:**

The next step in generating training levels is generating entities, such as minor objectives, an exit door, the key to open said door, enemies, the player, and hazards.

Hazards function as walls, but with the added property of ‘is\_hazard’. They will function to hurt the agent if it tries to move into them, causing negative reward for it’s learning process.

It is imperative that these hazards do not block pathways so that the level becomes unsolvable. To that end, there were two possible pathways to solving this algorithmically. One solution is to wait until cycles have been added, then look to add hazards back in, Checks can be performed to make sure once each hazard is added, that it has not cut off sections of the map and created inaccessible areas.  
A simpler solution, and one that is ultimately being used for the project, is to instead step in after perfect maze generation, but before cycles are added.  
At this point, the paths present are a perfectly solvable maze, and the set of remaining walls is still large, presenting many possible positions for hazard placement.  
Walls can be randomly selected and turned into hazardous tiles, and then disqualified from the cycle removal process. [Figure 12]  
This avoid extra computational costs in checking traversal possibility after each hazard is placed, and also allows for hazards to exist within open spaces.  
The drawback of this method is that hazards will only ever exist in places where initially walls were present – meaning that there is a checkerboard pattern of path tiles that will always be paths. It is possible that the model may learn this pattern.

A black and white rectangular object with white lines

Description automatically generated with medium confidence

***Figure 12*** *-‘Sigmoid dist.’ + Hazards   
  
Params: Hazard level = 5, Seed = 80, cycle bias = 0.5,   
cluster strength = 1,‘open’ tiles = 50,   
open ratio = 0.41*

*(X represents hazardous tile)*

The rest of the entities, can be placed at will on open path tiles, as they do not represent something impassable by the agent. This process simply involves finding all existing path tiles, and placing entities at random, while removing those path tiles from the list of remaining path tiles available for placement for future entities.

This can be extended to include difficulty enhancements, such as minimum distance between door and key, prioritising placement near hazards or enemies, and of course the number of enemies.

**8. Testing Plan**

Testing [Figure 13] begins with the initial baseline phase, where 2 models will be trained upon a singular level, and tested upon an unseen level.  
The training hyperparameters will be tuned to allow for moderate success on the training environment first, before evaluating performance on the test set as a ground truth. The testing process here focuses on whether agents are implemented correctly, and are able to learn from training material.

Next implementing PCG into the training process, models will then be tested with increasing numbers of diverse training levels.

Finally adding in the enhancement techniques such as curricula learning, and convolutional layers, models will be tested for performance again.

The PCG system will be tested for efficacy in generation, ensuring that it can generate diverse, solvable levels, which contain all necessary entities.   
Test functions will be written to ensure that every level passes checks to ensure validity.  
Finally it is imperative that the train and test set of levels are distinct, so checks must be made to ensure this.

REFERENCES:

**[1]** ‘Gymnasium Documentation’. Accessed: Jan. 12, 2025. [Online]. Available: <https://gymnasium.farama.org/introduction/create_custom_env.html>

**[2]** R. Niel and M. A. Wiering, ‘Hierarchical Reinforcement Learning for Playing a Dynamic Dungeon Crawler Game’, in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, Bangalore, India: IEEE, Nov. 2018, pp. 1159–1166. doi: [10.1109/SSCI.2018.8628914](https://doi.org/10.1109/SSCI.2018.8628914).

**[3]** M. Ihsan, D. Suhaimi, M. Ramli, S. M. Yuni, and I. Maulidi, ‘Non-perfect maze generation using Kruskal algorithm’, *J. Nat.*, vol. 21, no. 1, Feb. 2021, doi: [10.24815/jn.v21i1.18840](https://doi.org/10.24815/jn.v21i1.18840).