Design Specifications

---

1. Overview

This project focuses on enhancing the generalisation capabilities of Reinforcement Learning (RL) Agents. Specifically the aim is to utilise Procedural Content Generation (PCG) to create varied environments for the training process, alongside other techniques such as - convolutional network layers, regularisation, environmental stochasticity and curriculum learning – to train RL agents that perform in diverse and unseen training 'levels'.
The project will evaluate these techniques using two different RL models.

- Deep Q-Learning
- Policy Gradient

These models provide both an online and offline style of reinforcement model, and will be trained and tested within a grid-based dungeon-crawler-like game environment with a discrete action and state space.

2. Functional Requirements

2.1 Training Environment and PCG Integration

- **Procedural Content Generation (PCG):**

Procedural content generation will be used as the primary method aiming to minimise the overfitting of agents to a limited set of training environments, and improve agent's capacity to generalise.

The goal is to algorithmically generate a wide range of diverse training 'levels' for the agents, based on Kruskal's algorithm, with modifications to create environments suitable for the needs of the project.

The PCG process must allow for variation in environmental difficulty (layout complexity, number of obstacles/enemies) based on input parameters, supporting simplified curriculum learning.

2.2 Model Implementation and Training

- **Deep Q-Learning Model (DQN):**

A functional Deep Q-Learning Model must be implemented and integrated with the custom gymnasium environment, such that it can learn from the proto-game training environment.

- **Policy Gradient Model (PG):**

Similarly, a functional implementation of a Policy Gradient Model is needed, such that it can interact with the proto-game environment, using the custom gymnasium environment, and learn optimal policies through its interactions.

- **Techniques:**

Additional techniques aimed at enhancing the generalisation capability of agents must be integrated into the existing implementations of models and the training workflow.

This means:

- Altering the training process to allow for simplified curricula learning using modified training sets for difficulty scaling.
- Altering the models to use regularisation or convolutional network layers to extract features from the spatial environment.
- Adding in a degree of environmental stochasticity through the exploration of the state and action space.

2.3 Evaluation and Benchmarking

- **Performance Metrics:**

Average Time Steps: Measure the average number of time steps taken by agents to complete levels.

Percentage of Levels Solved: Quantify the success rate of a model against a set of test levels, calculating the percentage of total solved levels.

Final Score: Qualify the overall success of the model based on a total game score, including factors like speed of completion, objectives gained, damage taken, etc.

- **Comparative Analysis:**

In order to evaluate the impact of the Procedural Content Generation, and added generalisation enhancing techniques, the project must compare the performance of agents against a ground truth, or baseline, where all additional techniques are absent. This means that each model will have no training outside of a single training level, and will be assessed on it's ability to perform on an unseen environment, standalone.
As well as this baseline comparison, the project will also compare the performance of both model's against each other, from baseline to the end.

3. Software environment:

The project will be undertaken using Jupyter Notebooks, written in Python, utilising the powerful, industry standard packages of tensorflow, numpy, and gymnasium (formerly OpenAI Gym).
The basis consists of subclassing the gymnasium Env class, to make a custom gymnasium environment [1].
This class defines the action space, observation space, the step function, initialisation process, and reset methods, amongst other processes like rendering the environment visually.

The action space is discrete, featuring basic movement actions in up, down, left and right, and simple ranged combat mechanics where the agent may fire in a direction and hit everything in the path.

The observation space will also be discrete, with features extracted for the model to utilise in understanding it's environment, such as shortest paths to objectives, nearby enemies and direction, and the agent's position as well as the tile grid.

4. Game environment:

The proto-game environment itself is modelled after dungeon-crawler games, in the vein of Gauntlet, also used as inspiration for similar projects [2].

In the case of this project, this involves minor obstacles, directional movement, and objectives such as keys to unlock doors, and enemies to combat.

There will be several facets to the success of a level. Firstly, an exit, which first the agent must find a key for. Secondly, reward objectives that increase score for the level, but are not necessary for completion. Lastly, survival and combat, avoiding hazards and enemies will be critical, as the agent can 'die' and fail a level this way.
There will be a maximum amount of allotted timesteps per level, with each action taking one time-step. In this way an agent can also fail by taking too long.

5. Training without PCG – baseline:

Before truly being able to evaluate the performance of Reinforcement Learning agents in the proto-game environment, and the effectiveness of any techniques to enhance their generalisation capabilities, the project must first establish a ground truth.
This means that there must be a baseline performance for the two agents, that can be compared to as additional training techniques are employed.

To do this, both models will be trained on one level, seeded for the same layout, and given a reasonable and equal amount of epochs to train an optimal policy.

The difference in performance between this training level, and a second unseen test level will be noted, to see the drop in performance. This is the baseline for all future evaluation.

This test level will be kept separate from all train sets, for future evaluation after further generalisation techniques.

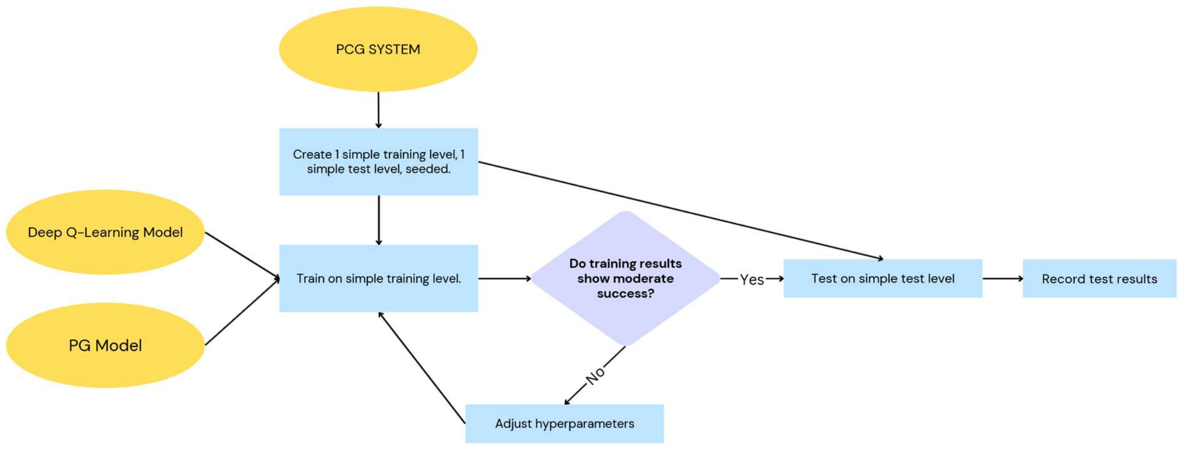

PCG SYSTEM
Create 1 simple training level, 1 simple test level, seeded.
Deep Q-Learning Model
PG Model
Train on simple training level.
Do training results show moderate success?
Yes
No
Test on simple test level
Record test results
Adjust hyperparameters

***Figure 1** – Training baseline*

6. Training with PCG:

Once a ground truth is defined, the training process to enhance generalisation can then begin, with models training on many unseen environments.

To do this, the project will utilise a ‘level’ generation pipeline that can algorithmically create many training environments for the models to train in. This pipeline allows for the scaling of difficulty, and agents will initially train on a large set of ‘moderate’ difficulty training environments. The amount of iterations will be varied, to evaluate how training on varied amounts of unseen environments affects performance. The amounts will be 500, 1000, 2000, 4000, 8000.
Every training iteration will take place on a new unseen training environment.
Training levels will be generated prior to training, and each ‘level’ will be seeded, and checked for uniqueness. This is to ensure no levels are present in both the test and train set.

This is for evaluation of agents with nothing but PCG as the technique for evaluation. After this, the training workflow can include a scaling of difficulty as the agents begin to see consistent results on easier levels, implementing a simplified form of curriculum learning, as well as modifying the model architectures to include convolutional layers, regularisation between layers, and adding a degree of environmental stochasticity through Boltzmann Exploration [2] in the actions of the agents.

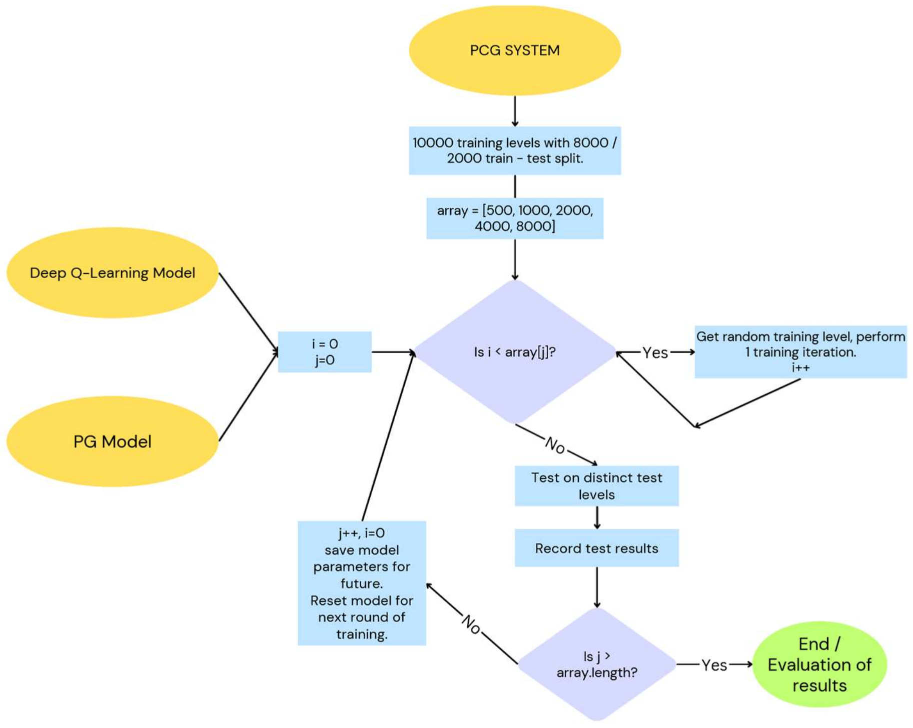
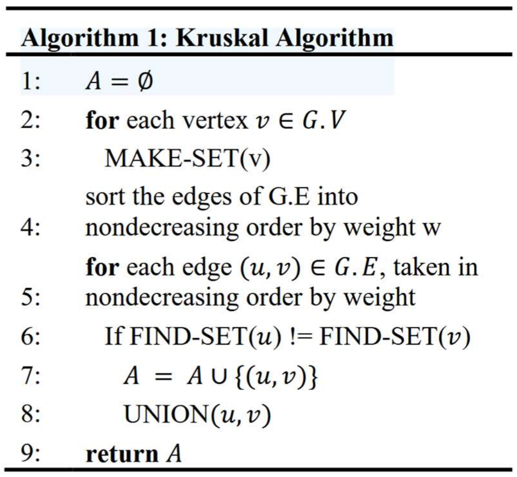

PCG SYSTEM
10000 training levels with 8000 / 2000 train - test split.
array = [500, 1000, 2000, 4000, 8000]
Deep Q-Learning Model
PG Model
i = 0
j=0
Is i < array[j]?
Yes
Get random training level, perform 1 training iteration.
i++
No
Test on distinct test levels
Record test results
j++, i=0
save model parameters for future.
Reset model for next round of training.
No
Is j > array.length?
Yes
End / Evaluation of results

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

7. PCG:

A system for procedurally generating training levels is necessary. This will be done algorithmically, with a basis in Kruskal's Algorithm.
Kruskal's algorithm through slight tweaks, can be made to generate random perfect mazes rather than minimum spanning trees. Instead of connecting the lowest weight edge, you randomise the edges to be joined.
A perfect maze is one that features no cycles, just as the minimum spanning tree.



Algorithm 1: Kruskal Algorithm

```
1:  $A = \emptyset$
2:  for each vertex $v \in G.V$
3:    MAKE-SET(v)
4:  sort the edges of G.E into
    nondecreasing order by weight w
5:  for each edge $(u, v) \in G.E$, taken in
    nondecreasing order by weight
6:    If FIND-SET($u$) != FIND-SET($v$)
7:      $A = A \cup \{(u, v)\}$
8:      UNION($u, v$)
9:  return $A$
```

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

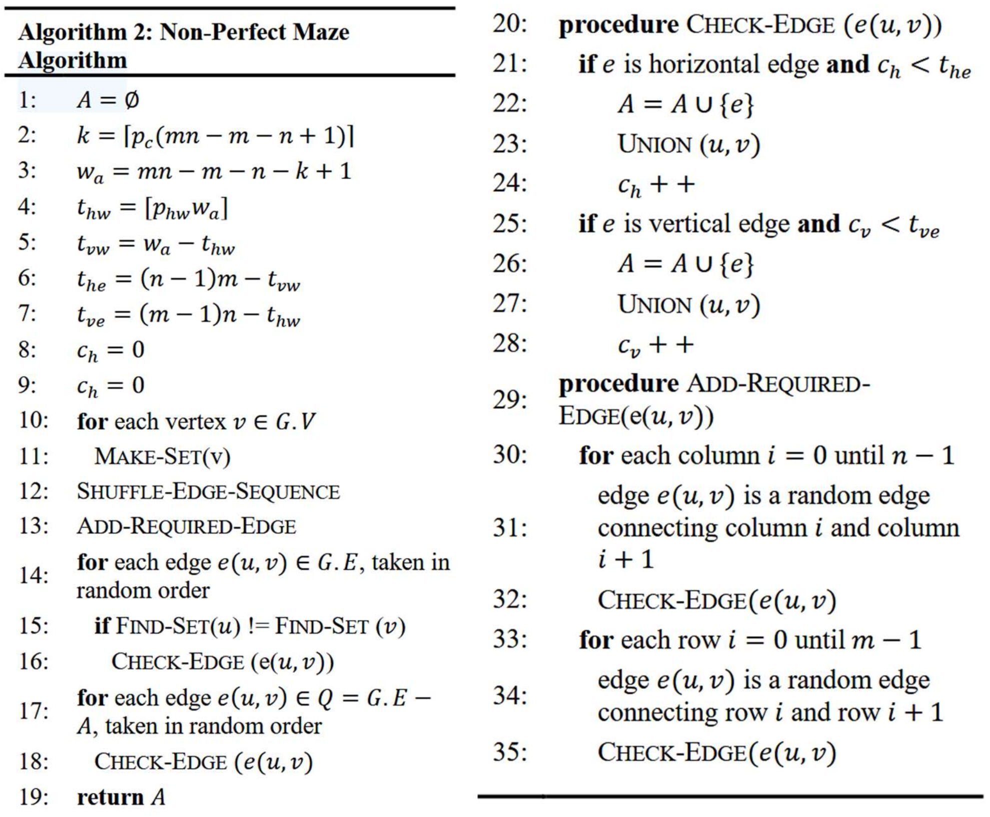
Using these tweaks, it is possible to generate many permutations of 'perfect' mazes, which would provide many training 'levels' for the agent.

However, a ‘perfect’ maze is quite a challenging environment for a model to begin learning in, especially once the dimensions grow beyond single digits.

Therefore, it makes sense to find a way to provide somewhat easier training ‘levels’ for the agent. This project will achieve this by further modifying Kruskal’s algorithm to add cycles to a perfect maze, thus making ‘non-perfect’ mazes.

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.

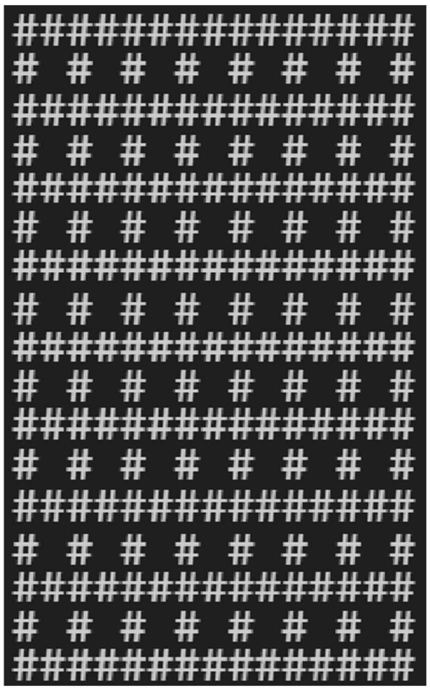


```
Algorithm 2: Non-Perfect Maze Algorithm
$A = \emptyset$
$k = \lceil p_c(mn - m - n + 1) \rceil$
$w_a = mn - m - n - k + 1$
$t_{hw} = \lceil p_{hw} w_a \rceil$
$t_{vw} = w_a - t_{hw}$
$t_{he} = (n-1)m - t_{vw}$
$t_{ve} = (m-1)n - t_{hw}$
$c_h = 0$
$c_h = 0$
for each vertex $v \in G.V$
  MAKE-SET(v)
SHUFFLE-EDGE-SEQUENCE
ADD-REQUIRED-EDGE
for each edge $e(u,v) \in G.E$, taken in random order
  if FIND-SET($u$) != FIND-SET ($v$)
    CHECK-EDGE (e($u, v$))
for each edge $e(u,v) \in Q = G.E - A$, taken in random order
  CHECK-EDGE ($e(u,v)$)
return $A$
procedure CHECK-EDGE ($e(u,v)$)
  if $e$ is horizontal edge and $c_h < t_{he}$
    $A = A \cup \{e\}$
    UNION ($u, v$)
    $c_h{+}{+}$
  if $e$ is vertical edge and $c_v < t_{ve}$
    $A = A \cup \{e\}$
    UNION ($u, v$)
    $c_v{+}{+}$
procedure ADD-REQUIRED-EDGE(e($u, v$))
  for each column $i = 0$ until $n - 1$
    edge $e(u,v)$ is a random edge connecting column $i$ and column $i + 1$
    CHECK-EDGE($e(u,v)$)
  for each row $i = 0$ until $m - 1$
    edge $e(u,v)$ is a random edge connecting row $i$ and row $i + 1$
    CHECK-EDGE($e(u,v)$)
```

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

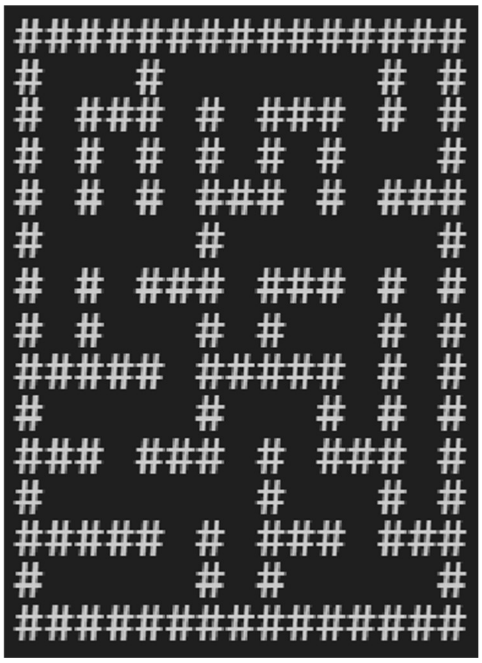
The algorithm used in this project utilises the core of the pseudocode and cycles bias while ignoring horizontal and vertical bias. It is modified further to provide methods of varying how the levels are formed, with a focus on creating open areas within training levels to different degrees.

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].



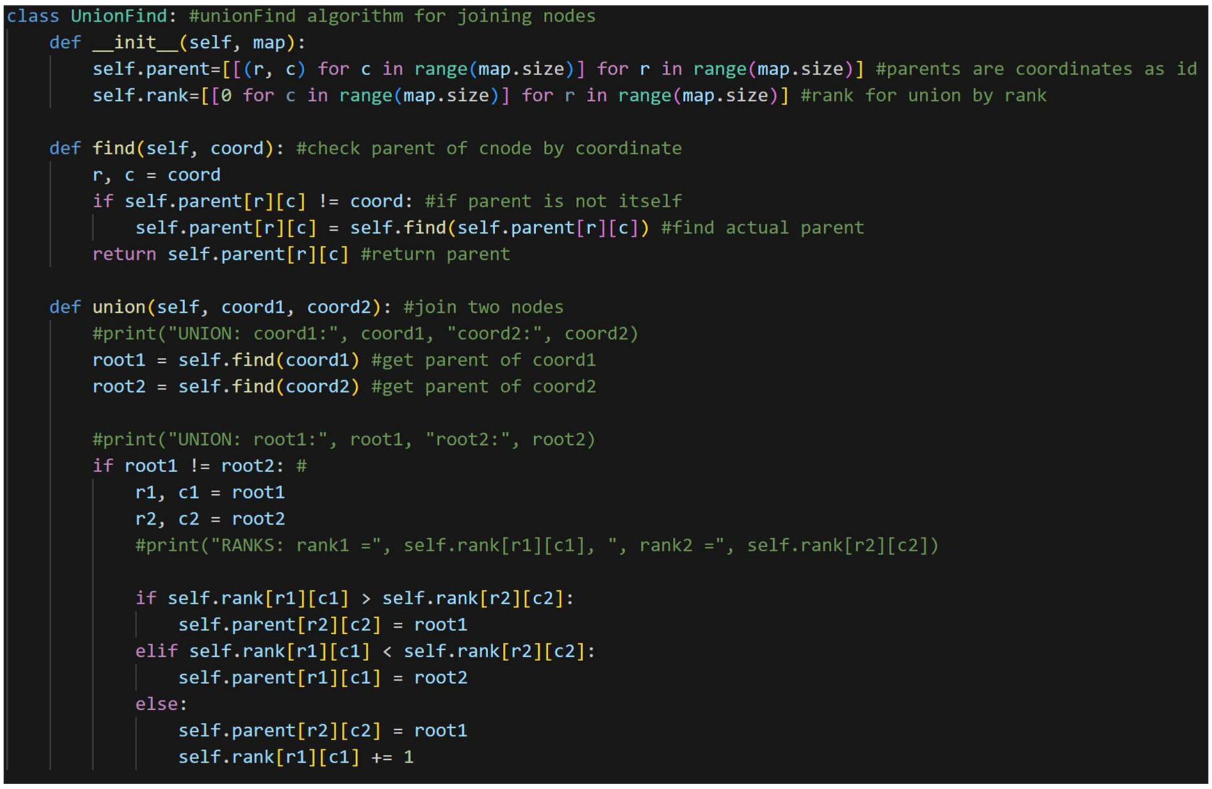
```
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
```

***Figure 5** – Pre Kruskal level*



```
###############
#   #       # #
# ### # ### # #
# # # # # #   #
# # # ### # ###
#     #       #
# # ### ### # #
# #   # #   # #
##### ##### # #
#     #   # # #
### ### # ### #
#       #   # #
##### # ### ###
#     # #     #
###############
```

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



```python
class UnionFind: #unionFind algorithm for joining nodes
    def __init__(self, map):
        self.parent=[[(r, c) for c in range(map.size)] for r in range(map.size)] #parents are coordinates as id
        self.rank=[[0 for c in range(map.size)] for r in range(map.size)] #rank for union by rank

    def find(self, coord): #check parent of cnode by coordinate
        r, c = coord
        if self.parent[r][c] != coord: #if parent is not itself
            self.parent[r][c] = self.find(self.parent[r][c]) #find actual parent
        return self.parent[r][c] #return parent

    def union(self, coord1, coord2): #join two nodes
        #print("UNION: coord1:", coord1, "coord2:", coord2)
        root1 = self.find(coord1) #get parent of coord1
        root2 = self.find(coord2) #get parent of coord2

        #print("UNION: root1:", root1, "root2:", root2)
        if root1 != root2: #
            r1, c1 = root1
            r2, c2 = root2
            #print("RANKS: rank1 =", self.rank[r1][c1], ", rank2 =", self.rank[r2][c2])

            if self.rank[r1][c1] > self.rank[r2][c2]:
                self.parent[r2][c2] = root1
            elif self.rank[r1][c1] < self.rank[r2][c2]:
                self.parent[r1][c1] = root2
            else:
                self.parent[r2][c2] = root1
                self.rank[r1][c1] += 1
```

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

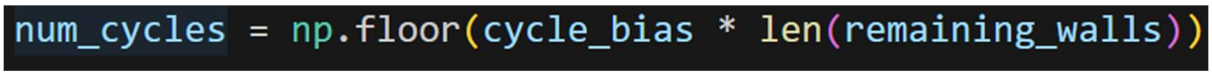
Connections are made using a UnionFind system for efficiency [Figure 7], and once a perfect map is generated [Figure 6], the next step is cycle addition.

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].



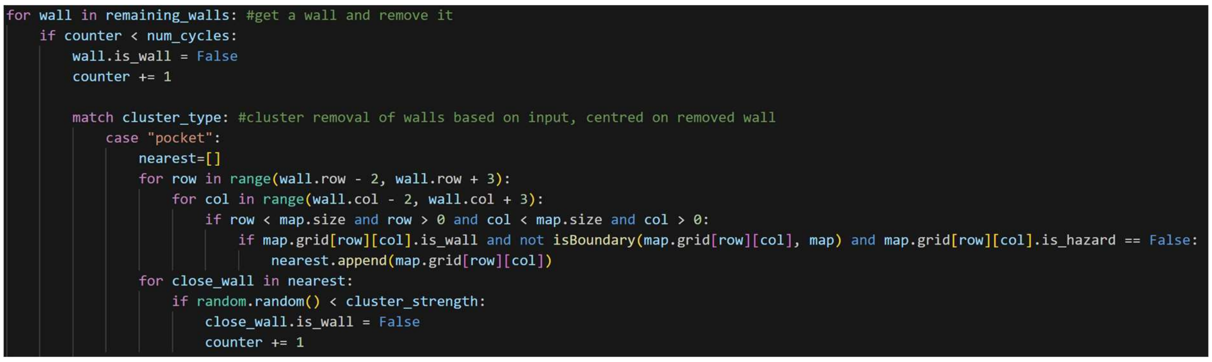
```
num_cycles = np.floor(cycle_bias * len(remaining_walls))
```

***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.

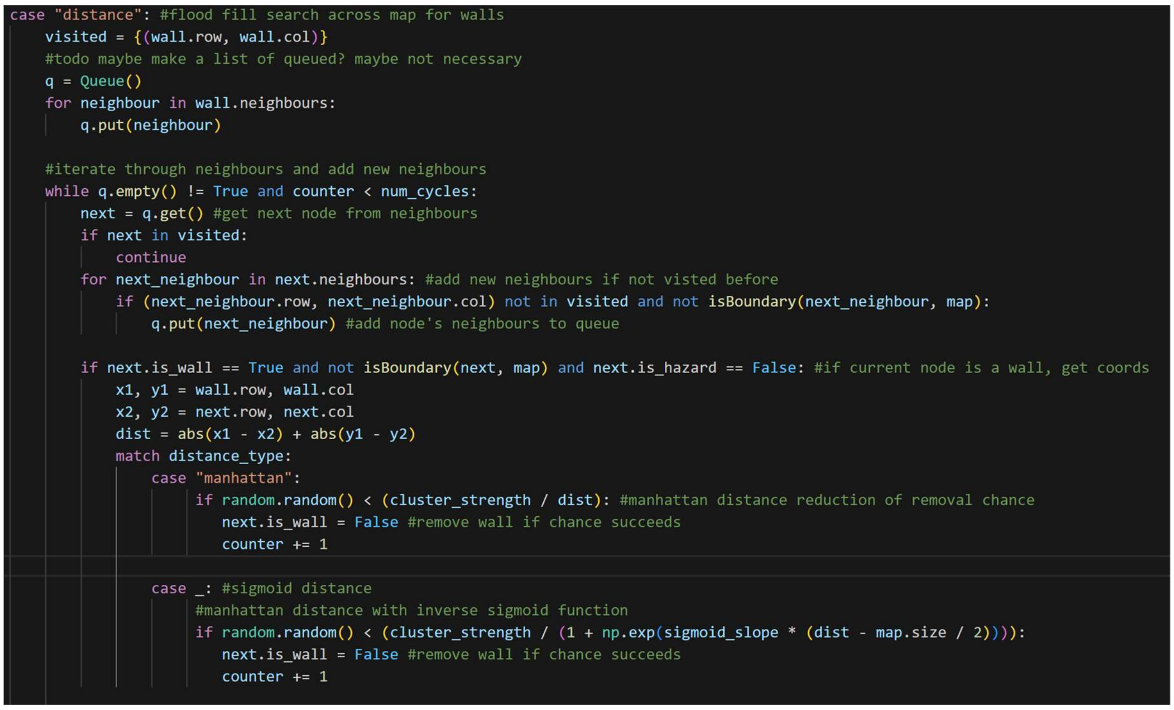
Pocket:

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]



```
for wall in remaining_walls: #get a wall and remove it
    if counter < num_cycles:
        wall.is_wall = False
        counter += 1

        match cluster_type: #cluster removal of walls based on input, centred on removed wall
            case "pocket":
                nearest=[]
                for row in range(wall.row - 2, wall.row + 3):
                    for col in range(wall.col - 2, wall.col + 3):
                        if row < map.size and row > 0 and col < map.size and col > 0:
                            if map.grid[row][col].is_wall and not isBoundary(map.grid[row][col], map) and map.grid[row][col].is_hazard == False:
                                nearest.append(map.grid[row][col])
                for close_wall in nearest:
                    if random.random() < cluster_strength:
                        close_wall.is_wall = False
                        counter += 1
```

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



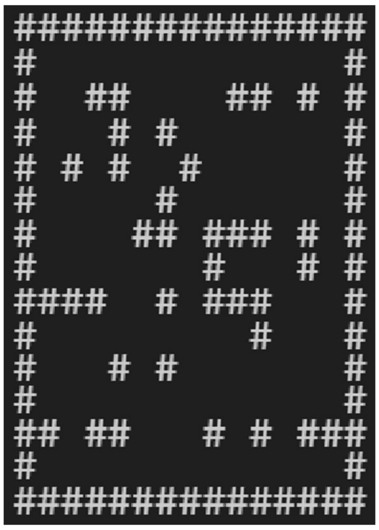
```
case "distance": #flood fill search across map for walls
    visited = {(wall.row, wall.col)}
    #todo maybe make a list of queued? maybe not necessary
    q = Queue()
    for neighbour in wall.neighbours:
        q.put(neighbour)

    #iterate through neighbours and add new neighbours
    while q.empty() != True and counter < num_cycles:
        next = q.get() #get next node from neighbours
        if next in visited:
            continue
        for next_neighbour in next.neighbours: #add new neighbours if not visted before
            if (next_neighbour.row, next_neighbour.col) not in visited and not isBoundary(next_neighbour, map):
                q.put(next_neighbour) #add node's neighbours to queue

        if next.is_wall == True and not isBoundary(next, map) and next.is_hazard == False: #if current node is a wall, get coords
            x1, y1 = wall.row, wall.col
            x2, y2 = next.row, next.col
            dist = abs(x1 - x2) + abs(y1 - y2)
            match distance_type:
                case "manhattan":
                     if random.random() < (cluster_strength / dist): #manhattan distance reduction of removal chance
                        next.is_wall = False #remove wall if chance succeeds
                        counter += 1

                case _: #sigmoid distance
                     #manhattan distance with inverse sigmoid function
                     if random.random() < (cluster_strength / (1 + np.exp(sigmoid_slope * (dist - map.size / 2)))):
                        next.is_wall = False #remove wall if chance succeeds
                        counter += 1
```

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

Manhattan / Sigmoid Distance:

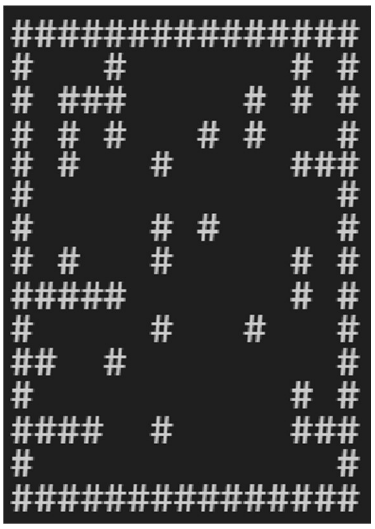
These two methods are based on the flood fill algorithm. Firstly a random wall is selected from the remaining walls. Then a flood fill begins from that wall, queuing all its neighbours to be visited. As each node is visited, it's neighbours are also queued to be visited, excluding any that have already 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]



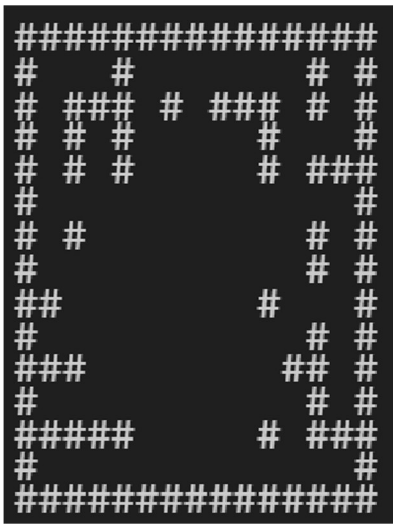
```
###############
#             #
#  ##    ## # #
#   # #       #
# # #  #      #
#     #       #
#    ## ### # #
#       #   # #
####  # ###   #
#         #   #
#   # #       #
#             #
## ##   # # ###
#             #
###############
```

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



```
###############
#   #       # #
# ### # ### # #
# # #     #   #
# # #     # ###
#             #
# #         # #
#           # #
##        #   #
#           # #
###        ## #
#           # #
#####     # ###
#             #
###############
```

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).

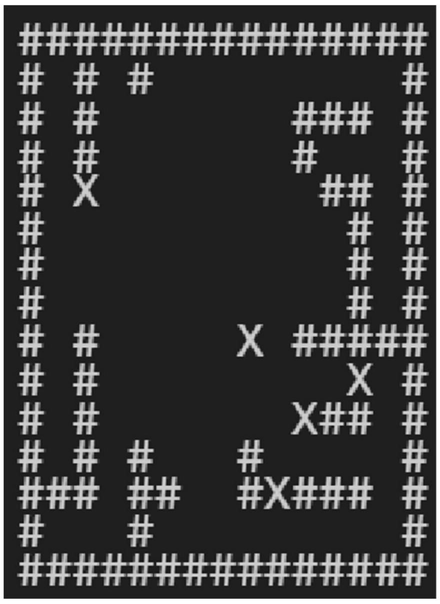
This allows for the evaluation of the expected 'difficulty' of the layouts within the levels, for use within the training process.

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.



```
###############
# # #         #
# #       ### #
# #       #   #
# X        ## #
#           # #
#           # #
#           # #
# #     X #####
# #         X #
# #       X## #
# # #   #     #
### ##  #X### #
#   #         #
###############
```

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.

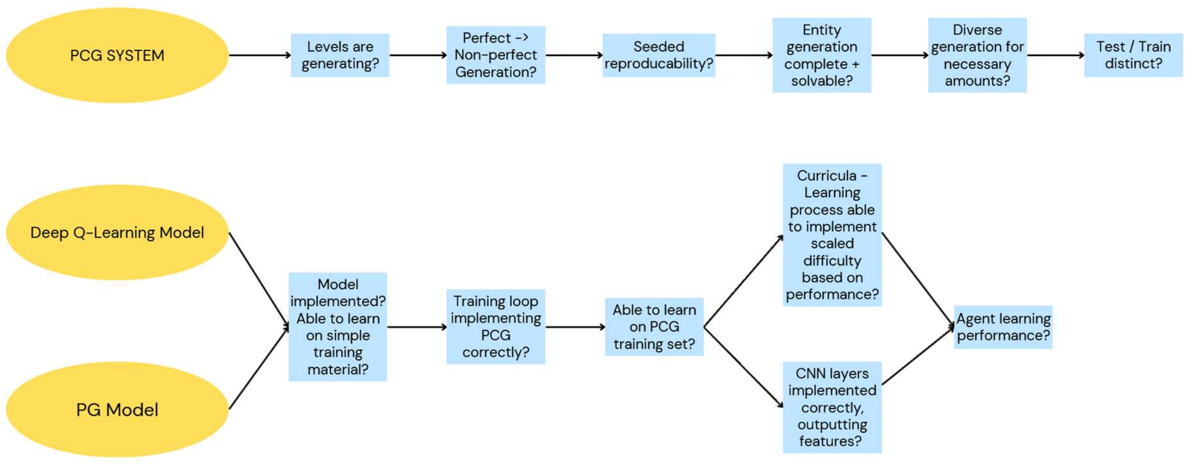

PCG SYSTEM
Levels are generating?
Perfect -> Non-perfect Generation?
Seeded reproducability?
Entity generation complete + solvable?
Diverse generation for necessary amounts?
Test / Train distinct?
Deep Q-Learning Model
PG Model
Model implemented? Able to learn on simple training material?
Training loop implementing PCG correctly?
Able to learn on PCG training set?
Curricula - Learning process able to implement scaled difficulty based on performance?
CNN layers implemented correctly, outputting features?
Agent learning performance?

Figure 13 *– Testing plan*

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.

[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.