Assignment 3: Modern Pac-Man: PCG, natural game interaction and RL Course: Modern Game AI Algorithms, Master CS, Leiden University

Ivan Horokhovskyi[1] **Andreas Paraskeva**[1] **Pavlos Zakkas**[1]

Abstract

In this assignment, we aim to explore different player-game interactions, in the traditional game *Pac-Man*. Alternative means of engagement with the game include, voice and vision controls, in order to enhance the player experience whilst also providing accessibility for people with disabilities. Additionally, **P**rocedural **C**ontent **G**eneration (PCG) is applied to generate new *Pac-Man* mazes with the usage of **G**enetic **A**lgorithms (GA). Finally, experimentation with **D**eep **R**einforcement **L**earning (DRL) techniques was engaged, for the training of an agent that plays the game.

1. Introduction

For the purposes of this assignment we have been requested to apply knowledge acquired through the course of "Modern Game AI Algorithms", complemented with external material and engage in a project of our choice.

The game of our choice is *Pac-Man*, since it provides a good balance between simplicity (having adequate branching factor) and variability of the game due to its non-trivial nature. The ability to provide extensions to such a popular game, which can enhance the player experience as well as possibly increase the replayability is intriguing.

It is important to note that the environment selected for this task was developed by UC Berkeley (Berkeley), and has been used for assessment of students for the implementation of model-based and model-free reinforcement learning algorithms. The project seemed to be outdated, since it's development was in Python 2, but and adaptation to Python 3 is available on Github (Spacco).

Traditionally, interaction with this game has been achieved through the usage of an arcade console, and at a later stage with console controllers or keyboard and mouse. These are undoubtedly convenient methods of input that lots of users have become accustomed to, however the possibility of increasing the means of engagement can be beneficial both for the lifespan of the game as well as the end user experience and target demographic. Users that have some sort of disability can now interact with the game. This is why it was decided to experiment with audio and vision inputs, and translate these to actions in the game. The simplicity of the game, due to the small action space of just four potential moves, allows these inputs to retain a player experience with relatively low learning curve whilst being seemingly natural.

The next goal for this project was the implementation of PCG, in order to expand on the number of levels available, and provide alternating difficulty, through random generation of *Pac-Man* mazes of varying sizes. This was achieved through the usage of GA, which would converge to an individual, containing the maze, that satisfies the different restriction and characteristics of a *Pac-Man* maze.

Finally, even though the game itself possesses a small action space, the winning state is hard to be achieved. Therefore we decided to experiment with DRL approaches that aim to solve the game, through the training of an agent. Different methods for a reward-based approach and heuristics have been implemented in an attempt to create said agent.

2. Pac-Man environment

2.1. Game environment

Pac-Man (PAC-MAN) is considered an action maze video game. The player essentially controls this eponymous character and tries to traverse a maze, with the aim to eat all the dots placed in the path. The goal is essentially to eat all of the food (dots), whilst avoiding coming in contact with the ghosts. There are at most four ghosts, Blinky (red), Pinky (pink), Inky (cyan), and Clyde (orange) which pursue *Pac-Man*. Consumption of an energizer allows the character to pursue and eat the ghosts that are in this transitive state, in an attempt to boost the score. Each time frame essentially reduces the accumulated score, whilst the consumption of food increases this.

[1]Leiden Institute of Advanced Computer Science, The Netherlands, The Netherlands.

On a superficial level, the game itself seems simple, but there are many aspects of that lead to its non-trivial nature and stochasticity. The ghost agents (which are automated and behave based on their move legalities) are developed in the Berkeley environment. Interacting and controlling the character utilizes the arrow keys of the keyboard, which are mapped to the respective moves (up, down, left, right).

An example of the environment and the initial state of the game is depicted in Figure 1, however the maze used can vary according to the selection by the user, which will consequently affect the starting positions of *Pac-Man* as well as the ghosts. The scoring will also be affected by the size of the Maze and the amount of food and energizers present.

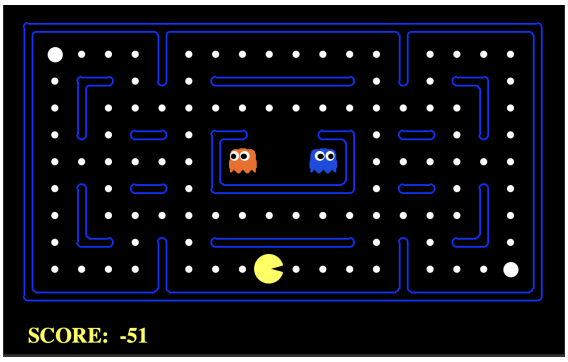


Figure 1. An example of a valid state of the *Pac-Man* environment.

2.2. Bots environment

The Berkeley implementation included two simple agents for *Pac-Man* which made the comprehension of the implementation easier. These agents included a *Left-turn agent*, which as suggested by the name will turn left at every available opportunity, i.e. when turning left is a legal action, and a *Greedy Agent*, which selects the best possible action to be taken at each state, investigating possible states of depth one. In the case that an optimal action (providing maximal reward) from the current state is not identified, random action selection is performed. These are very simple agents and have many flaws which we aim to eliminate with the creation of a DRL agent and consequently more advanced methods of action selection.

3. Background

In this section we will explore previous research or implementations, as well as the theoretical background related to our project and the respective aspects explored.

3.1. Player Experience Modeling

The 'effectiveness' or 'success' of a game is partly determined by the idea itself. A very important aspect is the replayability of a game and its ability to reccurently encourage the player to revisit the game and/or promote extended sessions. An overview of the different aspects of game related to **A**rtificial **I**ntelligence (AI) and its relation to game development is depicted in Figure 2.

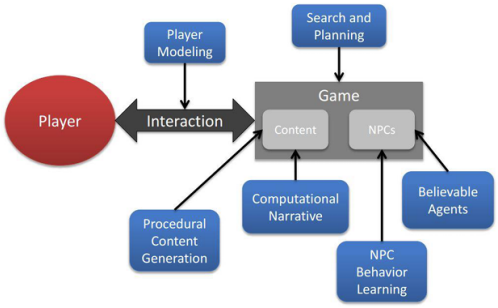


Figure 2. The panorama of AI research viewed from a player-game interaction perspective. (Yannakakis & Togelius, 2015)

Studies revolving around the theory of flow (Csikszentmihalyi, 1990), showed that there is an ideal threshold between the skill of a person and the difficulty of the task which can affect their flow, otherwise known as their ability to focus on the task at hand. This is also related to gaming, since the difficulty of the game itself as well as means of interaction (i.e. input) can affect the flow of the player. These means of input are often referred to as experience modeling. The game of *Pac-Man* is relatively simple in terms of controls, since there are only 4 possible directions that the player can perform. On the other hand, the difficulty of *Pac-Man* derives from the complexity of the maze and the movements of the ghost agents that are trying to eliminate the *Pac-Man* agent.

Based on the above, the learning curve for controlling the character through a keyboard, or controller is very minimal, whereas obtaining the ability to avoid ghost agents and consume all the food blocks of the maze requires more effort. As a result, providing a more natural form of engagement which does not require external devices handled by the player can enhance the experience. The challenge of this approach is to keep the control of the character relatively easy in order not to hinder the user with additional complexity, but let them focus on the actual difficulty of the game.

Providing methods of input such as voice (O'Donovan et al., 2009) and vision (Park et al., 2006) allows the player to interact with the game in a natural way, whilst not adding considerable effort in terms of getting used to the controls. This is mainly attributed to the small action space. The conclusion that both voice and vision controls have an in-

significant learning curve has been extracted through our own experience as well as a small-sized test group.

Another factor which substantially influenced our decision in providing more means of interaction with this game was the provision of accessibility for this game, and potentially allow users with disabilities to play *Pac-Man*.

3.2. Procedural Content Generation

In computing, generation of new data is essential, and ways to create/generate such data without the manual interference or effort of a human is very desirable. PCG has gained popularity with the exponential growth of AI as a field of study, since this can also be applied to games (see Figure 2). The combination of human-generated assets and algorithmic methodologies can allow exploitation of varying cases arising based on **R**andom **N**umber **G**eneration (RNG), in order to generate new content.

Such a task is definitely not trivial, especially with highly complex games that need to follow certain restrictions and rules in order to retain validity and believability. A game like *Pac-Man*, despite its longevity and popularity, needs to provide new material to sustain its high engagement and players' interest.

For the prospect of generating new mazes for this game, extensive analysis of existing mazes and extraction of rules that define a valid maze was necessary. In the following list we can observe some of the extracted constraints and how these have been taken into account:

Map Size The original game consists of 28×36 tiles. Even though this is the size of the original mazes, it was decided that alternative sizes should also provide an enhanced player experience that can be designed for different target groups. For example, smaller mazes might be more appealing for a smaller challenge and new players, but at the same time can be appealing for players seeking high scores and competitions.

Path Size The paths that *Pac-Man* and the ghost can traverse through, should be restricted to a thickness of 1 tile.

Wall Turns The turns formulated through the placement of walls should not form a sharp turn (i.e. not exclusively joined in the diagonal formation)

Dead ends Paths should not lead to dead ends.

Wall shapes The walls should create shapes of a cross, l-shaped, t-shaped or straight lines of length 3. The existence of a rectangular wall would also be valid but should be limited in terms of occurrence.

3.3. Genetic Algorithms

A GA is an adaptive search method that relies on processes and techniques which are based on aspects of biological evolution. There are three main processes which have been used for the implementation of a genetic algorithm: *mating selection*, *crossover* and *mutation*. These traditional GA steps though can have various implementations which would consequently alter the effectiveness of our optimization problem greatly. Hence, experimentation with different versions for these steps and hyperparameter tuning, were essential for concluding on our optimal GA setup based on our findings.

As aforementioned, some of these operators were explored more thoroughly and experimentation with different implementation as well as research was essential. Post research of applied knowledge of GA approaches, assession of potential implementation of GA in the project was formulated. Since the validity of a maze can be evaluated through the extracted constraints, it was deemed appropriate to have a population of individuals which consisted of the maze of *Pac-Man*. Any violation of the restriction can lead to an increase in the fitness, hence approaching this as a minimization problem is feasible.

Implementation details, and explanation of the algorithmic approach will be overviewed in Section 4.3.

3.4. Deep Reinforcement Learning

In this section we will talk about intellectual agents and our motivation behind picking deep reinforcement learning. For *Pac-Man* agents, we considered the following approaches:

- Heuristics based
  - Mini-Max
  - Alpha-Beta pruning
- Monte Carlo Tree Search
- Data based
  - Probabilistic Models
  - Classic Machine Learning
  - Supervised Deep Learning
  - Reinforcement Learning

The simplest and most intuitive approach would be to use heuristics based method. Implementing this method only requires a state search function (for instance BFS) and a state evaluation function (heuristics). The heuristics function can be the score of the game, but also the position evaluation can be added to avoid too greedy solutions. However, *Pac-Man* can be theoretically played very long (or even infinitely);

the computational resources are limited and long-running episodes would lead to a great reduction is score (score negation per time-frame). This is why the depth of the search is usually limited. Nevertheless, in such cases, we might spend our computational time on non-promising branches and that is why several modifications like Mini-Max algorithm or Alpha-Beta pruning could be added. Mini-max algorithm provides an optimal move to a player assuming that opponent (in our case ghost) is also playing optimally. Alpha-Beta pruning optimizes the search by not expanding non-promising branches. These techniques are pretty simple, but probably they can solve the *Pac-Man* game. However, if branching factor was high or reward gain was too sparse these algorithms cease to perform well. Also, our goal was to implement a more complicated and generic agent, since creating heuristics requires in-depth domain knowledge. Specifically, it needs manual and custom heuristics metrics, which can in turn slow down the agent learning procedure because it needs additional time to evaluate each state. Furthermore, it is very inflexible to apply changes when a more complicated logic needs to be implemented.

The next approach we considered is **M**onte **C**arlo **T**ree **S**earch, which partially solves the problems of heuristic approaches and does not require human knowledge about the game. Experimentation with this algorithmic approach in a previous assignment, as well as basic understanding of its concepts lead us to the conclusion that it cannot solve a problem of a high branching factor.

Regarding data based approaches, the main problem is probably the amount and quality of the dataset. These types of agents need quite some time for training, but when playing the game (evaluating) the computational complexity is low. A simple probabilistic model would most likely work worse than a **M**achine **L**earning (ML) one, while classic ML algorithms are not very good in working with high-dimensional data. Thus, we switched our focus to deep learning approaches. For supervised learning a labelled dataset is required, which seems to be a problem in our case, since from a single state there can be multiple strategies that the agent could follow, and not one absolute correct label. In addition, a model can overfit or collected game states can be skewed which will lead to poor generalization of the agent. Because of the aforementioned issues it is not trivial to get a proper dataset. **R**einforcement **L**earning (RL) can potentially formulate data on its own, learn how to tackle new challenges would not require any human expertise on the domain. The drawback though is that an RL approach requires a lot of time and computational resources while training.

4. Implementations

4.1. Vision assisted experience modeling

Following the intuition explained in Section 3.1, we decided to enhance further the user experience by enabling vision assisted interaction. In this mode, the user can move the *Pac-Man* agent through hand gestures. Depending on the movement of the hand, we are going to decide which of the actions *Left*, *Right*, *Up* and *Down*, should be applied to the agent.

4.1.1. Recording hand movements

In order to record the movements of the user's hand we experimented with OpenCV and mediapipe, which can be integrated to our game in order to provide the position of specific points of the hand, as they are identified through the user's camera. By taking into account these points, we can easily calculate the mean point of the user's hand. Then, by storing the history of the last 20 mean points with a time interval equal to 500ms, it is possible to determine how the hand position changes over time and understand in which direction it moves. To achieve that, we calculate the difference between pairs of consecutive points and average them across each dimension. Then by checking if the mean difference between consecutive points, either in the vertical or the horizontal axis, is larger than a *move threshold*, we can ignore slight hand movements and keep only those that correspond to actual gestures. This *move threshold* was set to 4 pixels.

As a result, if the hand position changes from left to right and exceeds the *move threshold*, we will apply the *Right* action to the agent. Moreover, since there are usually fluctuations in both vertical and horizontal axes we need to determine which change is larger. For example, when the user tries to move their hand from left to right, there might be a slight movement from up to down as well. Thus, we need to ignore this slight movement and apply only the *Right* action. For that purpose, we keep either the horizontal or the vertical movement based on which of those had a higher mean difference across the history of the recorded mean points.

The above method was a descent initial approach, but it could not provide a playable interaction. Specifically, the user had to perform each move exactly at the time when the corresponding action was legal. Otherwise, the action was considered invalid and thus it was not applied.

4.1.2. Planning next move

In order to solve the issue of requiring from the user to perform each action exactly at the time that it is valid, we decided to allow for planning the next move beforehand. For instance, if *Pac-Man* moves towards right and the user

wants to go up at the next cross section, they can perform the related upwards hand movement beforehand and the agent will apply it when it is legal for the first time, which corresponds to the aforementioned cross section.

As a consequence, the agent does not discard a received action if it is not applicable. Instead, the agent keeps it in memory as pending and executes it when it is valid. Meanwhile, if a new action is given by the user, the memory of the last recorded action is updated and the previous pending action is replaced with the new one. In this way, the game becomes much more playable, since the user is able not only to plan the next move but also undo it if needed.

4.1.3. Resetting hand to the center

After implementing the aforementioned solution, another problem was revealed. To illustrate, when the user performs an action to the *Left*, their hand will end up on the left side of the screen. Then when they return their hand to the center area of the screen in order to perform the next actions, they should make unavoidably a hand movement towards the right. This movement though results in performing a *Right* action without being the intend of the user.

To avoid this issue, we need to allow the user to reset their hand back to the center area of the screen by ignoring the related hand movement. As a result, when a movement is determined, we consider it as valid if the last position of the hand is close to the center area of the screen. The *center area* is defined as a square window that deviates 10% from the central point of the screen.

The implementation of this feature made the control of agent smooth allowing the user to easily perform the desired actions. The intuition behind this idea can be related to the design of a joystick controller, in which the user takes an action towards a direction and then brings the controller back to the center before taking the next action.

4.1.4. Adjusting to player's distance

Finally, despite the fact that after using the aforementioned method the game became as playable as using the keyboard, we observed a last issue. The user's distance from the camera was affecting the interpretation of hand movements, since the applied *move threshold*, described in Section 4.1.1 was not accurate for every position of the user.

For this reason, we decided to adapt our implementation and replace this constant threshold with a dynamic threshold that depends on the depth (i.e. the distance of the user from the camera). Thus, we recorded the Z-axis values of the hand positions and based on the mean value we scaled the *move threshold* accordingly so as to allow a smaller threshold when the user stands far from the screen. In this way, the user will be able to apply an action by moving naturally their hand independently of the distance they have from the camera. To formulate this scaling factor we measured that the maximum Z-axis value recorded when the user is very close to the screen is 0.4, whereas the minimum value is approximately 0.0001. By using these values, we can scale the move threshold of 4 pixels by the factor $\frac{z_{mean}-0.0001}{0.4}$, which results in lower move threshold values when distance is higher.

4.2. Voice assisted experience modeling

Another approach to enhance user experience is the voice assisted interaction. Specifically, the user will be able to control the character by using voice commands that will be executed by the agent.

We explored different speech to text APIs that could be integrated to our environment in order to transform voice commands to text in real time. Then by using the transcribed text, it will be possible to identify which command was given by the user. The allowed voice commands are *left*, *right*, *up* and *down*, which correspond to the possible moves of the *Pac-Man* agent.

4.2.1. SpeechRecognition: Google Speech API

Firstly, we experimented with the Google Speech API of the speech recognition package (Uberi). The transcription provided by this approach was accurate, but it lacked real-time processing. Specifically, even though the delay was relatively small, there were cases where more than half or one second was needed to transcribe the speech command to text. For our application, this delay has a significantly negative impact on user experience, since it does not allow the user to perform the desired actions on time, resulting in inefficient control of the agent. Even after trying to calibrate the parameters of the speech recognizer (i.e. the minimum audio energy to consider for recording, the seconds of non-speaking audio before a phrase is considered complete), it was not possible to achieve the desired interaction.

4.2.2. PocketSphinx speech recognition

As a second approach, we experimented with PocketSphinx (Bambocher). Since, this API is supposed to transcribe speech in real-time, it seemed to be a promising solution for our problem. Indeed, the interpretation of voice commands was fast enough, but the transcription was not accurate. In order to solve this issue, we tried to change the voice commands to *west*, *east*, *north* and *south* respectively in order to avoid using the *up* command which has a very small length and can be easily misinterpreted. Also, we calculated the Levenshtein distance of the transcribed text from the aforementioned commands and when the distance was less than 1 or 2, we applied the respective command. Nevertheless, none of these approaches was able to solve

the issue of inaccurate speech to text transformations of the pocketsphinx API.

4.2.3. AssemblyAI Speech-to-Text API

As a last approach, we decided to use the Speech-to-Text service provided by AssemblyAI (Assemblyai), since it offers real-time transcription with high accuracy. This method met our expectations, as it was able to transcribe accurately the voice commands with relatively low latency. Also, by slightly decreasing the frame speed of the game, we managed to provide a pleasant user experience, where the user could give voice commands that were executed on time. Moreover, as in the vision assisted interaction (see Section 4.1.2), the user is able to give beforehand a command that will be applied when it is applicable for the first time. Thus, it is possible to plan the next move. Besides this feature though, we decided to enhance further the interaction by allowing the user to give two consecutive actions with one voice command. In the next section, we present the details of this feature.

4.2.4. Planning two moves ahead

In order to enable the user to plan the next two consecutive actions, we implemented a voice agent which is able to store a stack of two actions. Specifically, the user can give a combination of actions with a single voice command such as *up and right*. Then the agent keeps those two actions in memory and performs them sequentially when they are applicable. Thus in our example, with a single voice command, the agent will go *Up* when this action is possible, and then it will go *Right* when applicable as well.

The impact of this feature was quite significant for the voice assisted interaction, because in cases where the user wanted to perform two successive actions, the controlling difficulty increased. For instance, in case of a 'U-turn' (around the wall), the user knows beforehand which actions should be done, but if the commands are given one by one, the user should act quite fast when giving the second command in order not to miss the 'U-turn'. On the other hand, when the user is able to plan the next two moves, the control of the agent becomes smoother and more accurate as presented in Section 6.2.

4.3. PCG with GA

The PCG for the generation of different *Pac-Man* mazes has been implemented with the usage of GA. In the sections that follow we provide a brief overview of the individual structures as well as the algorithmic approach followed in order to generate said mazes.

4.3.1. Population

The number of individuals in each generation is denoted as μ, while the number of offspring that are generated during each iteration is represented by λ.

Each individual consists of half of the maze, which can follow a size of $n \times m$, where n and m are multiples of 3. Each individual is initialized through stochastic selection from the 3×3 building blocks shown in Figure 3. The blocks shown in this figure, were then rotated by $[90°, 180°, 270°]$ and flipped across the vertical axis in order to create all possible combinations of valid blocks. These were 35 discrete blocks in total. It should also be noted that the next generation of each population is selected in a deterministic way, in which the best μ individuals are selected either out of the λ offspring (μ, λ) or out of the μ parents and λ offspring $(\mu + \lambda)$. Both selection methods have been tested to conclude that the best one was deemed to be $+$-selection.

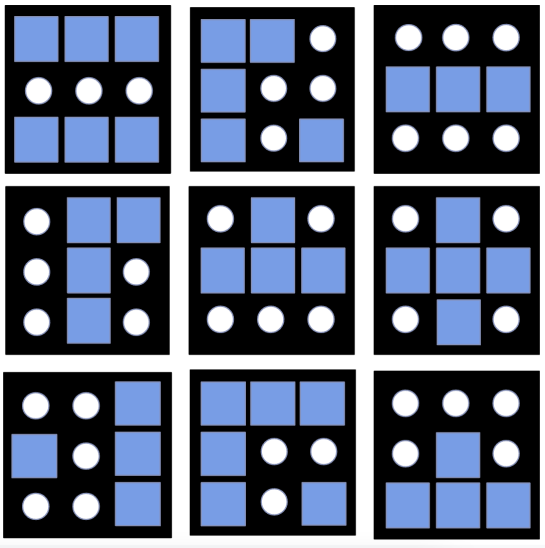


Figure 3. The 3×3 building blocks used for the individuals.

4.3.2. Fitness

For the evaluation of the individual we have used a number of different 2×2, 3×3, 3×2, 2×3 and 4×1 blocks, which contain patterns that would violate the constraints of a *Pac-Man* maze. Similarly to the methodology for acquiring all the possible unique building blocks, we have created a list of the invalid blocks and rotated and flipped them to get all possible combinations. This logic is applied to 2×2 and 3×3 invalid blocks (see Appendix A) but for the 4×1 which essentially is just four walls in a row, we only needed to transpose it in order to check both column-wise and row-wise for the existence of said pattern. Regarding the 3×2 and 2×3 invalid blocks, we did not need to perform any transpositions since all versions are presented in 15, and

they are just four in total.

4.3.3. Algorithmic Approach

Our algorithmic approach is based on the following sequence of iterative and repeating processes of evolution.

1. Mating Selection:

   Proportional Selection: We used fitness proportionate selection, the fitness is associated with each respective individual and the probability of selection will be correlated to its fitness. To further improve the probability of more fitted individuals (parents) to be selected for the mating process, we used scaling. This essentially will boost the probability of selecting parents with higher than average fitness and those with lower than average fitness will be less likely to be selected. The formula for the probability assignment is as follows:

   $$p_i' = \frac{f_i - c}{S_f - c * \mu}$$

   where $S_f = \sum_{j=1}^{\mu} f_j$ and $c =$ the minimal fitness value of the selection pool.

   Hyperparameters:

   (a) factor: determines how many children will be produced, based on the initial population size. #of children = factor $*$ population size

   Tournament Selection: This consists of two main steps that are repeated for each parent selection. Firstly, we select a random subset of the population (of size k, determined by our hyperparameter). Following, we will select the best fitted individuals out of the subset, based on their fitness values. *Hyperparameters:*

   (a) k: is the number of randomly selected individuals.

   (b) factor: determines how many children will be produced, based on the initial population size.

2. Crossover: After the selection of pairs from the population of parents is finalized, then we proceed with the crossover.

   Uniform crossover: The two offspring are initially identical copies of the two parents. We then proceed to traverse one by one their building blocks which form the maze, and for each one a probability is used to determine whether the currently visited block is switched with the one of the other offspring. *Hyperparameters:*

   (a) pc: the probability of the crossover taking place. typically assigned to 0.5 for fairness.

3. Mutation:
   For mutation, we introduce the concept of block fitness alongside the individual fitness. This is in an attempt to prioritize blocks which have the most violations. More in depth explanation is provided later in this section. *Hyperparameters:*

   (a) pm: the partial constant used to determine probability of the mutation taking place.

   (b) s_{pm}: small value to avoid zero probability of mutation. Tuned to optimize performance.

An algorithmic overview for the general procedure followed can be depicted in the algorithm pseudocode 1.

Mutation and Block Fitness

In an attempt to optimize the mutation phase, we introduce block fitness. As stated earlier, when an individual is evaluated with the set of invalid blocks, we increment its fitness by 1 every time that a violation occurs, during out iterative check of whole maze (half-maze). However a violation can occur in more than one of the building blocks of the individual. Some examples can be observed in Figure 4, where the red outline indicates the area that the violation was identified in. Therefore, whenever a violation is identified, the fitness that will be added to the individual is equally divided in all of the building blocks that overlap in the violation.

With this technique, during the mutation phase, we can multiply the division of the individual block fitness and individual fitness by the chance of mutation, in order to create proportional mutation probabilities, where the block with the most violations (i.e. higher block fitness) will have a higher chance of being mutated. Since we also desire a probabilistic mutation even for more fit blocks the formula also includes a parameter (s_{pm}) which is added to avoid zero probability of mutation for a block. We generate a random value between zero and one and if this is smaller than our retrieved calculation we perform mutation. The finalized formula is as follows:

$$p_m * \frac{individual.block\_fitness + s_{pm}}{individual.fitness}$$

4.4. DRL Agents

The RL field describes the use of the active interactions of an agent with its environment to learn the optimal behavior or achieve a specific goal in that environment. RL problems in general can be formulated as a **M**arkov **D**ecision **P**rocess, a tuple $< S, A, T, R, \gamma >$ where S is the set of states s in the environment, A the set of allowed actions a, $T : S \times A \times S' \rightarrow [0, 1]$ is the transition function, $R : S \times A \times S' \rightarrow R$ is the reward function and γ is the discount factor (Plaat, 2022).

Algorithm 1: Overview of the Genetic Algorithm used for the PCG of *Pac-Man* mazes.

Input : Population size μ
Offspring size λ
Next generation selection [',', '+']
Crossover ['Uniform']
Mutation type ['Block Fitness Proportional']

Termination : The algorithm terminates when: - the target value has been reached

p ← Population(μ);
parent_evals ← Evalutation(f, p);
while *fitness_opt > 0* **do**
    p' ← MatingSelection(p);
    p" ← Crossover(p') ; // Probabilistic creation of offspring with parents' blocks
    offspring_evals ← Evaluation(f, p");
    p"' ← Mutation(p");
    offspring_evals ← Evaluation(f, p"');
    /* Perform '+' selection */
    p += p"';
    evals ← parent_evals + offspring_evals;
    p, parent_evals ← Selection(p, λ, evals) ; // Get λ best individuals and their fitnesses
end
final_grid ← best_individual.mirror_pad() ; // Mirror across vertical axis. Pad with food and outer wall.

Algorithm 2: Actor-Critic Algorithm

Input : a differentiable policy π_θ, learning rate η, update frequency f, exploration rate σ, discount fraction γ and annealing rate ϵ, bootstrap depth n.

Initialize the actor and critic NNs with random weights θ and ϕ.
Set **actor loss** and **critic loss** to deques with size f;
for *trajectory* ***in*** *budget* **do**
    Sample trace $h = \{s_0, a_0, r_0, ..., s_{n+1}\}$ with π_θ.;
    $R, A \leftarrow 0$;
    $L_{\text{actor}} \leftarrow 0$;
    for $t \in n, ..., 1, 0$ **do**
        $R \leftarrow \gamma^n \cdot V_\phi(s_{t+n}) + \sum_{k=0}^{n-1} \gamma^k \cdot r_{t+k}$;
        $A \leftarrow R - V_\phi(s_t)$;
        $L_{\text{actor}} \stackrel{+}{=} A \cdot \log \pi_\theta(a_t|s_t)$;
    end
    Append L_{actor} to **actor loss**.;
    $L_{\text{critic}} \leftarrow \text{MSE}(R(s_t, a_t), V_\phi(s_t))$;
    Append L_{critic} to **critic loss**.;
    if *trajectory* $\% f = 0$ **then**
        $L_{\text{actor}} \leftarrow$ mean(**actor loss**);
        Update the actor NN with L_{actor}.;
        $L_{\text{critic}} \leftarrow$ mean(**critic loss**);
        Update the critic NN with L_{critic}.;
        $\sigma \leftarrow \epsilon \cdot \sigma$;
    end
end

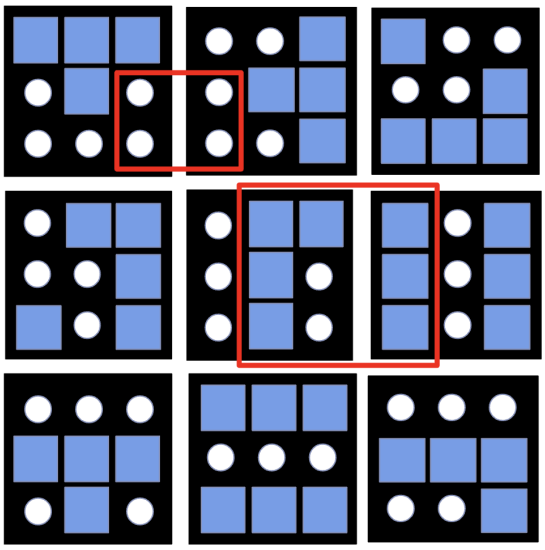


Figure 4. Evaluation of individual and block fitness example. The red outline indicates the area that the (sample of) violations was identified in. Visualized can also be found some of the building blocks of the individual.

We seek an optimal policy for the agent to follow, defined as $\pi^* : S \rightarrow A$ that maximizes the expected discounted sum of the future rewards $Q(s, a) = E_\pi[\sum_i \gamma^i r_{t+i+1} | s_t = s, s_t = a]$.

Simpler Deep RL problems that do not have continuous action spaces can be solved with value-based approaches where a **N**eural **N**etwork (NN) is trained to learn the value-function that maps the state-action pairs to their respective Q-value. On the other hand, policy-based methods, which can be applied in both continuous and discrete action spaces, aim to learn directly a stochastic policy, which introduces natural exploration of states. We decided to use the Actor Critic (AC) modification, which is expected to converge faster than value-based Q-learning, as it offers a good balance between bias and variance by using a value head along with the policy head (Sutton & Barto, 2018). The pseudocode is presented in Algorithm box 2.

To make our agent work we needed to adjust our *Pac-Man* environment. The model perceives the game state as a 3D matrix: $width \times height \times cell$ value which is a one hot encoded vector of 5 values. Specifically, each value represents the positions of the 5 different entities of our game, including *Pac-Man*, ghosts, walls, food, and energizers. The absence of all those entities means that the block is empty. The aforementioned input representation is also visible in the input state shown in Figure 5.

During the first iteration, we used a single random ghost agent to play against our *Pac-Man* agent during training and we used the score of the game to identify the reward of each

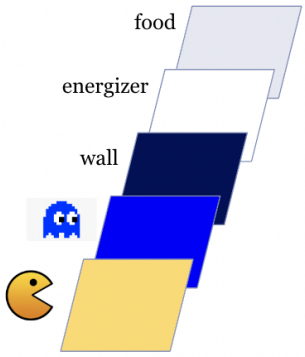


Figure 5. State representation of *Pac-Man* environment

action. To be more specific, if the agent is at state s and performs an action, the game environment moves to a next state s', by moving the *Pac-Man* agent and the ghost agent accordingly. The score difference of the two states can be used as the reward returned by the environment for action a.

This vanilla method did not yield the expected results, as the agent could not usually perform legal actions and was eaten by the ghost very soon, resulting in very short episodes that were unable to train the agent. Based on this observation, we decided to remove the ghost agent in order to let *Pac-Man* solve the maze without any threat. Despite this, it failed to perform only legal actions.

To deal with this issue, we performed manual adjustment of the default reward mechanism, which was previously developed solely from the game score. At first, we penalized illegal actions significantly. The impact of this approach was quite positive since the agent learnt the game rules and started performing legal moves in most of the cases. However, it could not escape specific states, since after consuming the first 'food' particles, it entered a looping motion of backward and forward moves in a specific section of the maze. This behaviour is indicative of the agent being stuck in a local minima. Thus, we decided to modify the interaction with the environment by adjusting the rewards further.

In order to create such hand-crafted rewards, we experimented with the following events. The positive reward events are: win, eating food or energizer, eating ghost, while the negative rewards are: illegal move, performing a step without eating food, step backwards, game loss. Different weight values were assigned to each of these events, and the final action reward was the summation of all the rewards of each specific event. Thus, if the agent performs a move that eats the ghost but does not eat any food, we sum up the positive reward of eating the ghost and the negative reward of doing a step without consuming food. Of course, in this case, the weight of the reward of eating a ghost should

be quite higher than the one of not eating a 'food' particle (which also aligns with the scoring mechanism of the game). Moreover, in most of the cases, there is no point for an agent to make a step backwards and this is why we penalize the agent for this action slightly more, in an attempt to restrict backward moves to the absolute necessary cases. This was the main approach we used in order to avoid local minima and train our agent explore different parts of the maze without being stuck in specific areas whilst performing a looping motion.

Furthermore, we tried to increase exploration using a Soft Actor Critic (SAC) (Haarnoja et al., 2018) agent, which aims to avoid the collapse of the policy distribution. Specifically, despite the fact that a policy-based method includes exploration by nature as it learns a stochastic policy, it is a common approach to enable the exploration of more states further, in case a problem is not solvable with the default Actor Critic method. As a result, SAC introduced the entropy regularization method, which uses an additional penalty term to the loss function, which is the entropy of the actions distribution predicted by the policy model. In this way, SAC enforces entropy to stay larger, and prevents policy from being too narrow and selecting the same actions back to back.

Finally, different network architectures were considered. The first one is 2D convolutions, in which we start with 5 channels in the beginning, which represent each cell of the state. The motivation behind this network is that we have an image-like input with which **C**onvolutional **N**eural **N**etworks (CNN) and **Vi**sion **T**ransformers (ViT) (Dosovitskiy et al., 2020) are considered as state of the art. However, training a ViT is not a trivial task especially for RL where a lot of hyperparameters need to be tuned. The next architectural approach was formulated based on the assumption that we should not mix the state cells with CNN channels, as they would share the same NN weights. This approach does not seem to be intuitive and this is why we needed to add another dimension and end up using 3D convolutions. Another possible extension would be utilizing **L**ong-**S**hort-**T**erm-**M**emory (LSTM) to take advantage of the history of states. Further explanation is provided in Section 7.2.

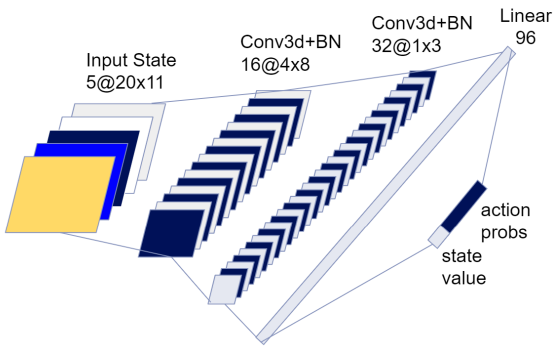


Figure 6. Policy based, actor-aritic RL agent network architecture. Two 3D convolutions with batch normalization. Activation functions ReLU and Softmax on final layer.

5. Tuning

The free parameters that have been introduced by the algorithms explained in the previous section arose the need of performing thoughtful hyperparameter optimization. This section summarizes the experiments conducted to achieve optimal results.

5.1. Tuning: Vision assisted experience modeling

For vision assisted interaction, we had to experiment with various configuration in order to end up with a playable solution.

At first, we tried different time intervals between consecutive position recordings of the user's hand. Gathering many hand positions within a small interval requires a large amount of points to be stored in agent's memory in order to be able to identify a possible movement. On the other hand, sampling hand positions more rarely may result in missing important information related to the user's movements. Thus, sampling a hand recording every 500ms and keeping a history of the last 20 hand positions achieved the best results compared to other configurations.

Secondly, one of the most important parameters was the *move threshold* described in Section 4.1. Large threshold values required larger hand movements from the user while small threshold values resulted in undesired actions due to high sensitivity. It was observed though, that a 4 pixels *move threshold* is an optimal value for a user that stands within a mid-range distance from the screen. Nevertheless, applying the same threshold independently of the user's distance was not optimal for all cases, and that is why we needed to multiply the default 4 pixels threshold with the scaling factor $\frac{z_{mean}-z_min}{z_max}$, where $z_{min} = 0.0001$ and $z_{max} = 0.4$ as measured in Z-axis.

Last but not least, the window of the *center area* was

also tuned in order to allow the user return their hand to the center and then perform their next action. This area is a square window where each vertex lies at a specific distance from the center point of the screen. This distance is defined as a percentage of the screen's resolution. For large values, the user should perform larger hand movements to escape the *central area* and take an action, but for smaller values, slight movements could result in unwanted actions, when resetting the hand to the center. After experimentation, a distance from the center that is equal to the 10% of the screen's resolution seems optimal for our case to avoid the aforementioned issues.

5.2. Tuning: Voice assisted experience modeling

As mentioned in Section 4.2, after experimenting with different speech-to-text services, we ended up using AssemblyAI service which did not require any tuning process since it met our expectations.

5.3. Tuning: PCG with GA

Since there are no concrete ways to evaluate the produced matrix, aside from validity based on the constraints that we introduced in Section 3.2, our only way to evaluate the performance of each run was to repeat 5 runs with the current settings under evaluation. Following this, we averaged the number of generations needed to generate the maze, and this was used as one of the metrics for performance evaluation of each hyperparameter setting. Another important metric was the average time needed to generate a maze, since the time for each generation is dependent on many of our hyperparameters.

Next Generation Selection

The relevant hyperparameters or tunable methodologies have been explained in the Section 4.3. Initially one of the most important choice for a GA approach would be the decision on the type of selection applied for the new generation of individuals. As aforementioned the next generation of each population is selected in a deterministic way, in which the best μ individuals are selected either out of the λ offsprings (μ, λ) or out of the μ parents and λ offsprings $(\mu + \lambda)$. After experimentation with both methods, it was concluded that $(\mu + \lambda)$ has yielded better performance and more consistent convergence within a reasonable number of generations.

Population Size

The population size was also a crucial hyperparameter since a value that was too small would hinder the ability to converge to a solution, whilst a higher than optimal setting would lead to more time needed for each generation. Through our testing the optimal value was deemed to be 8 individuals but this is very sensitive and can change with the combination of the other hyperparameter settings. However we do not expect it deviate greatly from this identified optimal.

Mating Selection

Our mating selection phase could utilize two different methods. During our experimentation we determined that both methods were determined to be equally suitable. These included proportional selection and tournament selection. Additionally these methods both include the hyperparameter referred to as $factor$, which determined the number of children that would be produced based on the population size. We concluded that a factor of 1.0, meaning that we produce the same number of offspring as the initial population size. Having a smaller factor was not favorable, whilst a higher than 1.0 factor did not seem reasonable so no experimentation was performed. The tournament selection method introduced the k hyperparameter which is the number of randomly selected individuals for this specific method. A value of $2 - 3$ was concluded to be optimal since higher values lead to a decrease in relative performance.

Uniform Crossover

The tunable hyperparameter relevant to this was p_c. Taking into consideration the usage of $+$-selection, thus keeping the previous generation of parents in the pool of selection, it will be favorable to have a relatively higher chance of altering the offspring. Thus we experimented with values for p_c between 0.5 and 1.0, where 1.0 leads to the crossover happening everytime. We concluded that an optimal setting based on our search space would be 0.7.

Mutation

Due to the customized approach for mutation and the need for block fitness as well as individual fitness, the typical value of p_m would not suffice. We definitely need to take this into consideration since it will determine the probability of a mutation taking, proportionally to the block fitness. Thus a higher than typical value was necessary. Through our experimentation with a vast number of values, ranging from 0.05 to 0.7 we concluded to the value of 0.5 as a good performing setting. Additionally, we introduce the hyperparameter of s_{p_m} which is a relatively small value to be added to each block fitness to avoid zero probability of mutating, even good blocks. Through careful consideration and limited testing in values, the value of 0.05 is reasonable to provide a small probability of mutating well fitted blocks in the individuals.

It is worth noting, that our concluded values for the optimal settings were retrieved through our search space which is no way compensating for all the possible combinations of these settings. The concluded settings are performing well but further tuning will possibly result in a higher performance (with the aforementioned metrics considered).

5.4. Tuning: RL *Pac-Man* agent

Given the nature of NNs, **h**yper**p**arameter **o**ptimization (HPO) has substantial impact on the performance of ma-

chine learning and deep learning algorithms, especially on RL domain, given the amount of additional hyperparameters that are required to define the training and validation of the RL agent. This leads to an inherently large search space that renders it difficult to fully explore such a big number of configurations. For this problem, three main HPO methods were considered namely: grid search, random search and Bayesian optimization. Random search is more efficient and explores better than grid search, and theoretically converges to the optimal solution. Bayesian optimization, cannot be parallelized efficiently and possesses additional hyperparameters (such as the acquisition function and kernel of the Gaussian processor), which need to be tuned separately. Random search is a good balance in simplicity and efficiency so it was picked for this problem.

Hyperparameters can be split in NN and RL categories:

- Neural Network hyperparameters
  - network type (see 4.4)
  - number of convolutional layers
  - number of fully connected layers
  - usage of batch norm regularization
  - usage of (max) pooling
  - convolution layers hyperparameters
  - activation function
  - weights precision (can speed up training process, but make the performance a bit lower)
  - network optimizer
  - network loss function
- Reinforcement Learning hyperparameters
  - gamma
  - step size
  - use of baseline substraction
  - entropy regularization (SAC)
  - budget
  - rewards list

For hyperparameter tuning we created an automated framework which uses a configuration file (in JSON format) with hyperparameters. In the case of continuous hyperparameters, these are accompanied by distribution and ranges. For categorical parameters, we provide a list of values to be tested. The program samples a predefined amount of hyperparameter sets which are used for the training and evaluation processes. To make the results more statistically significant, each experiment needs to be executed several times, which is also defined in the configuration file. Last but not least, the program tries to efficiently utilize the resources of available, by running many processes concurrently.

6. Results

Our experimentation phase was essential in order to evaluate our implementation methods explained in Section 4. Different sections required different degree and type of experimental procedure.

6.1. Results: Vision assisted experience modeling

After following the implementation described in Section 4.1, it is possible to record hand movements as shown in Figure 7 and eventually control *Pac-Man* actions. For further interpretation of the results, a video (https://youtu.be/w2EI2vXkWhot) was recorded demonstrating the whole user experience when using the vision assisted interaction.

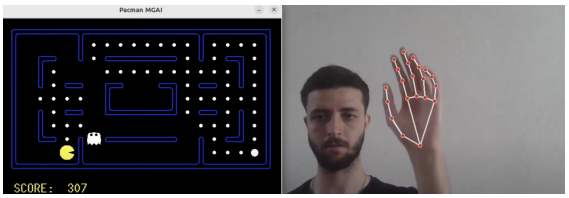


Figure 7. Vision assisted agent

6.2. Results: Voice assisted experience modeling

As described in Section 4.2, the voice assisted interaction was implemented, where the user can give voice commands to the *Pac-Man* agent and even perform two consecutive actions with a single command, as shown in the detected text in Figure 8. A demonstration of such a game is presented in the following recorded video: https://youtu.be/nB9Nk-UfIis

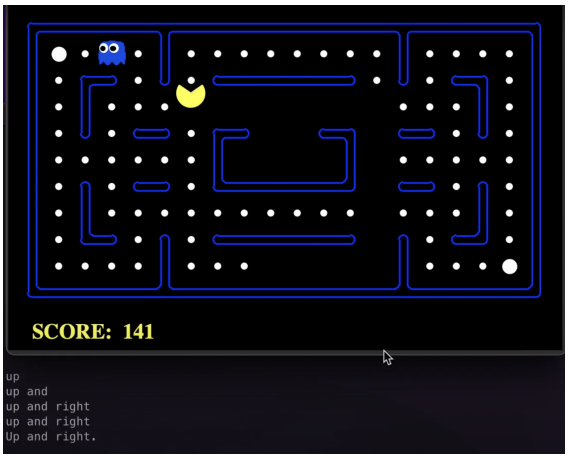
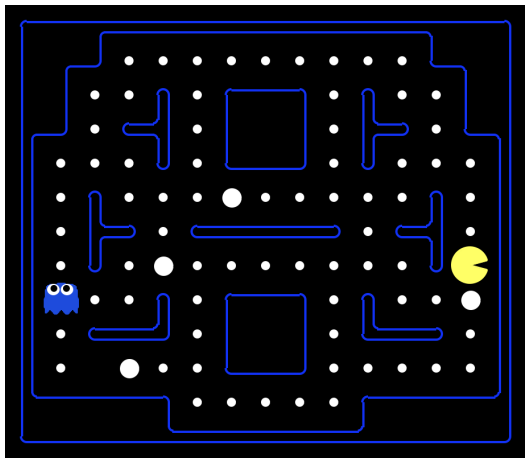


Figure 8. Voice assisted agent

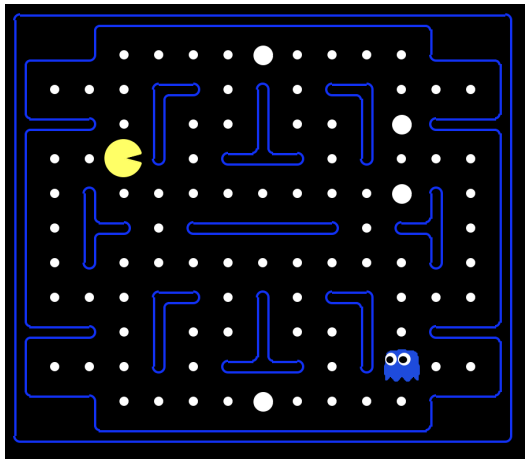
6.3. Results: PCG with GA

In an attempt to evaluate as well visualize the results of the PCG, we used the Berkeley *Pac-Man* project with the generated layouts. It was important to test different sizes of mazes, in order to ensure that our restrictions are met throughout, as well as manage to create different randomly generated mazes. These different sizes can provide further replayability as well as varying degrees of challenge to improve the flow of the game.

Some of examples of the retrieved mazes are shown in Figures 9, 10, 11.



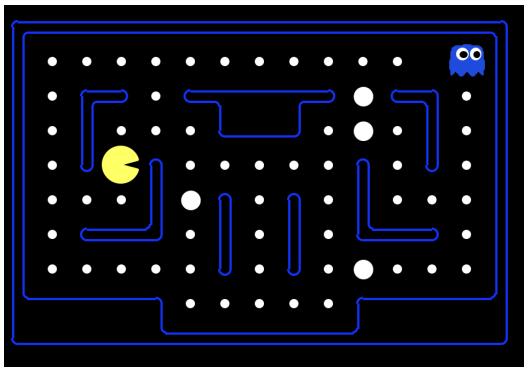
(a) An example of 13×15 generated maze



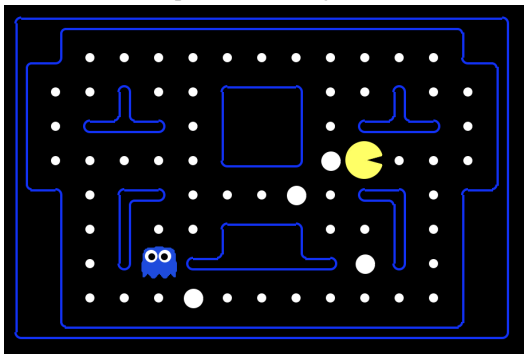
(b) An example of 13×15 generated maze

Figure 9. Some of the generated mazes are shown to present the validity and varying sizes capabilities of the algorithm

Closer inspection of these mazes leads to the conclusions that the generation of the mazes with the GA approaches is successful and satisfies the constraints that were listed earlier. However, some minor problems can arise when mirroring the maze. There is a possibility of creating some shapes that are not within the valid ones. Restrictions re-



(a) An example of 10×15 generated maze



(b) An example of 10×15 generated maze

Figure 10. Some of the generated mazes of size 10×15 are shown here

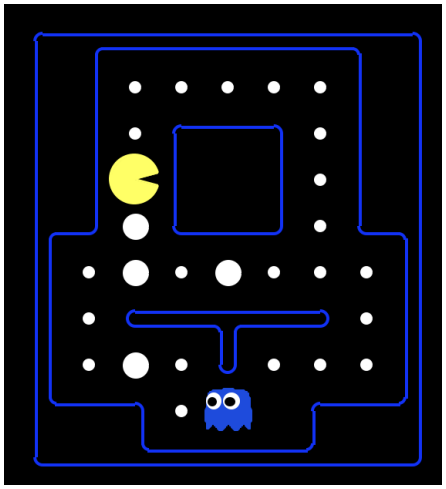


Figure 11. An example of 10×9 generated maze.

garding the path and dead ends are fully met, which would render the possible existence of a shape not in the original ones not of huge importance since it does not affect the playability of the game. It should be noted however as a violation and further investigation for potential solutions can be performed. Initial attempts indicate that this is not such a trivial task since we need to also retain the satisfaction of other constraints and altering the shape will lead to a violation of another rule, such as the single-width path.

6.4. Results: RL *Pac-Man* agent

When starting the training process it takes a while to train an agent to avoid violation of the game rules and learn legal moves. After that, the agent starts to not only do legal moves but also search for food. As already mentioned in Section 4.4, another problem we faced was stucking in local optima, as the agent learned to collect food at a limited part of the maze, and afterwards started to wander around aimlessly without exploring the rest parts of the maze.

Despite the fact that we managed to train the agent to make legal moves and explore a part of the maze, it was not possible yet to solve the game. Nevertheless, it was obvious from our experimentation that with further hyperparameter optimization and training the agent will be able to reach convergence and solve the maze using the implemented poicy-based RL approaches. Due to limited time and lack of computational resources though, the next steps needed to train a sufficient agent were not made yet, but they are proposed as future work in Section 7.2.

7. Conclusions and Future Work

7.1. Conclusions

To conclude, the additional player experience models provided replayability as well as more natural ways to interact with the game. Our small test group unanimously said that the addition of new ways to interact with this classic game has been a fun experience and that the learning curve for both vision and audio assisted controls has been smooth and minimal. Therefore, these have not hindered their experience, and on the contrary it allowed the game to retain replayability. Furthermore, even though we were unable to test the implementations with a group of people with disabilities, we strongly believe that these two modes of interactions will allow the target demographic to expand and provide means of interactions to those who previously lacked it.

Based on our search space, the suggested Speech-to-Text service for the voice game interaction is the Live transciption service from AssemblyAI, whereas for the vision assisted implementation the retrieved optimal settings, when using the mediapipe and OpenCV modules, are the following:

time interval between recorded moves : 500ms

move threshold : $scaling\_factor \times 4$ pixels, where $scaling\_factor = \frac{z_{mean} - 0.0001}{0.4}$

central area : square window defined by a distance of 10% of resolution's width or height from the center point

In regards to the PCG with GA, we can conclude that overall the performance is good since generated mazes seem to be valid. However there are some cases where due to the mirroring some undesired shapes may occur. Our current efforts to fix these cases have not been successful since it is not such a trivial task to make changes whilst retaining the previous constraints satisfaction. However, aside from the issue that arose through mirroring, which might not necessarily manifest itself with most of the generated mazes, all of the constraints are fully met and the believability of the authenticity of the generated mazes is more than satisfactory.

The identified optimal settings for the PCG are as follows:

Next Generation selection $+$-selection.

Population Size and Offspring Size 8 for both.

Mating Selection Both tournament and proportional selection were deemed equally favorable. Factor of 1.0 and $k = [2, 3]$.

Crossover Uniform crossover with $p_c = 0.7$.

Mutation $p_m = 0.5$ and $s_{p_m} = 0.05$.

Current efforts to tune the actor-critic model for the *Pac-Man* environment were unsatisfactory but displayed a learning process even though convergence to a wining state was not achieved. We believe that the current model would be satisfactory for solving this environment, however further tuning will be necessary. In an attempt to perform a more extensive search for optimal hyperparameter settings, we have developed a framework to be used for experimenting with large search spaces. The respective hyperparameter settings and retrieved scores can be accessed and evaluated. The model which lead to the optimal settings will also be accessible so retraining would not be necessary. Due to lack of resources and time, this large-scale training and hyperparameter tuning was not feasible but it should be useful for any future efforts to tune this model and train a successful agent.

7.2. Future Work

The player experience models developed seem to be achieving our initial goals and expectations, thus it would be interesting to explore further this area. Typically *Pac-Man* is a single-player game, but it would be interesting to

take advantage of these alternative and natural methods of input in order to allow the game to be played by two players simultaneously. One player could be controlling the ghost agent whilst the other one the eponymous character. This can be achieved by taking advantage of both vision and audio controls at the same time. This idea could be interesting and it can provide more game related content beyond the classical game.

For the PCG with GA method to generate *Pac-Man* mazes, it would be advised to investigate potential solutions to the problem that could arise through mirroring. Additional tuning of the current hyperparameters can also further optimize the implementation. Currently multi-threading has also been implemented, but just for the creation of mazes in parallel. It would be a good idea to parallelize the maze creation by running each individual's generational procedure in parallel. Furthermore, different **E**volutionary **S**trategies can also be experimented with for the maze creation, or additional mating selection, crossover and mutation strategies can be added.

Finally, as aforementioned, for the RL agent, it would be adivsed to perform extensive tuning and the tool created for exhaustive search of parameters might be useful for the search of optimal settings. Once the model is tuned, leading to a successful agent using Actor-critc, we can expand to other policy-based approaches or implement further methods that would optimize the current approach. Another idea for extending further the DRL implementation would be capturing the last N frames, thus keeping track of the recent history of states, instead of keeping only the last one. The benefit of such approach is that the network can track in which directions ghosts are moving and extrapolate the ghosts' trajectory. To implement such network behavior, a convolutional LSTM network can be used. Different policy optimization such as **P**roximal **P**olicy **O**ptimization with **C**ovariance **M**atrix **A**daptation (PPO-CMA) can also be explored. An interesting way that could lead to a more generic agent (that could solve multiple mazes) would be to expand the training procedure, by allowing the agent to interact with more than one environmental structures. This can be achieved by utilizing our PCG with GA model and using multiple generated mazes, of varying sizes, for the training of the agent.

References

Assemblyai. Assemblyai speech-to-text api: Automatic speech recognition. URL https://www.assemblyai.com/.

Bambocher, P. Python interface to cmu sphinxbase and pocketsphinx libraries. URL https://github.com/bambocher/pocketsphinx-python.

Berkeley, U. Berkeley: The pac-man projects. URL http://ai.berkeley.edu/project_overview.html.

Csikszentmihalyi, M. *Flow: The Psychology of Optimal Experience*. 01 1990.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

O'Donovan, J., Ward, J., Hodgins, S., and Sundstedt, V. Rabbit run: Gaze and voice based game interaction. In *Eurographics Ireland Workshop, December*, 2009.

PAC-MAN. History the official site for pac-man - video games amp; more. URL https://www.pacman.com/en/history/.

Park, H. S., Jung, D. J., and Kim, H. J. Vision-based game interface using human gesture. In *Pacific-Rim Symposium on Image and Video Technology*, pp. 662–671. Springer, 2006.

Plaat, A. Deep reinforcement learning, a textbook, 2022. URL https://arxiv.org/abs/2201.02135.

Spacco, J. Jspacco/pac3man: Python3 port of berkeley pacman. URL https://github.com/jspacco/pac3man.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Uberi. Speech recognition module for python, supporting several engines and apis, online and offline. URL https://github.com/Uberi/speech_recognition.

Yannakakis, G. N. and Togelius, J. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7 (4):317–335, 2015. doi: 10.1109/TCIAIG.2014.2339221.

A. Invalid Blocks

This section includes the set of invalid blocks used for evaluating the individuals.

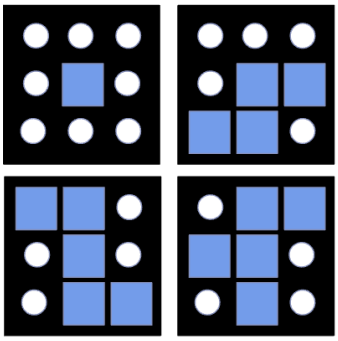


Figure 12. The 3×3 invalid blocks.

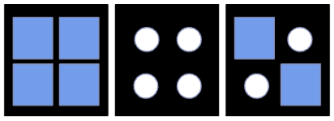


Figure 13. The 2×2 invalid blocks.

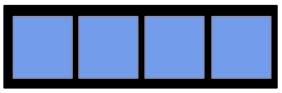


Figure 14. The 4×1 invalid blocks.

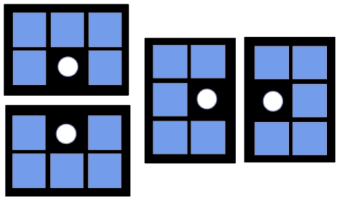


Figure 15. The 2×3 invalid block on the left hand side and 3×2 on the right.