Learning to Use Money through Reinforcement Learning

John Flynn

Abstract—**Money, in its various forms, has played a pivotal role in shaping civilisations throughout human history. By facilitating cooperation among strangers, currencies have enabled monumental advancements in trade, settlement, and migration that surpassed the limitations of barter systems and other early exchange mechanisms. But despite its benefits and ubiquitous nature, it remains a mystery how humans learned to use these mediums of exchange in the first place. This project explores the origins of money through reinforcement learning, chosen for its resemblance to human learning processes. Two multi-agent Q-learning models were designed, developed, and experimented on, drawing from recent research into safe swapping behaviours. The first model, or the "swapping model", examines whether a population of agents can naturally learn the behaviours necessary to perform safe swaps with strangers in episodic meetings. The second model, or the "token model", builds on this by investigating learned agent behaviours when inherently worthless yet persistent tokens are introduced in a continuous stream of meetings. With the versatility to represent a diverse range of swapping scenarios and interactions, these models provide valuable insights into the motivations behind money's use and the fundamental requirements for a population to adopt such behaviours.**

I. Introduction

A. Motivation

HUMAN beings have been using money in various forms for approximately 40,000 years [1]. Every day, monetary systems demonstrate their power to foster cooperation between strangers, enabling the exchange of goods and services, settlement, migration, and more. With cooperation as one of humankind's most beneficial traits [2], it only makes sense that money has been with us for a long time. However, this gives money an enigmatic origin that has largely been left up to speculation [1].

Popular theories suggest money emerged to address challenges with bartering [3]. Bartering requires a simultaneous coincidence of wants between parties, and scales poorly when determining exchange rates among a growing array of products. In contrast, money is a persistent and standardised medium of exchange, simplifying debt and pricing. While these explanations of money are reasonable, they assume individuals already recognise they can trade it for something of value, which does not reflect reality. Money is intrinsically worthless, and monetary exchanges between strangers are fraught with risk. Furthermore, these theories are susceptible to hindsight bias, as the widely known benefits of money today may inadvertently influence the interpretation of historical motivations, potentially conflating the origins of money with its positive outcomes [4].

Artificial intelligence offers a fresh perspective on the origins of money. While traditional theories focus on historical and economic factors, the avenue of reinforcement learning has the potential to simulate the creation of a monetary system through a population of agents. As reinforcement learning closely resembles how humans learn through experience [5], this novel approach could provide insights into the emergent behaviours that led to the establishment and attribution of value to money. Can agents learn to use money through reinforcement learning? And if so, what can we learn from this?

It is important to note that this project is not concerned with model optimisation, and thus typical benchmarks are not appropriate. Rather, it explores whether reinforcement learners can naturally arrive at behaviours that amount to using money, the fundamental requirements of this, and any theoretical insights that can be gleaned. Regarding sustainability, this project primarily overlaps with the United Nation's 4th and 7th sustainability goals [6]; "Quality Education" and "Affordable and Clean Energy". This project promotes quality education through attempting to deepen the general understanding of money and its origins, potentially contributing to several fields of research, including artificial intelligence, economics, cognitive science, and anthropology. In terms of energy efficiency, the software in this project was designed with simplicity in mind, utilising a lightweight variant of Q-learning as its reinforcement learning algorithm, keeping the model sizes to a minimum, and taking advantage of efficient software libraries for computation such as NumPy. This keeps energy use low, with the additional benefit of minimising agent training times.

B. Solution

The problem of getting a population of agents to learn money-like behaviours can be represented as a multi-agent reinforcement learning (MARL) control problem [7]. This involves agents interacting with each other in a sequence of repeated games or "meetings", where agents perform actions within a shared environment or "Markov Decision Process" (MDP) to maximise their own cumulative rewards [8]. Using a reinforcement learning algorithm such as Q-learning, each agent should independently learn the expected rewards of certain actions, and therefore an optimal strategy or "policy". Ultimately, these strategies should resemble the use of money.

To achieve this goal, two reinforcement learning models were designed, implemented, and experimented on. The environment and training process of each model extends the

This project was supervised by Marcus Frean and Stephen Marsland.

ideas in “Holds enable one-shot reciprocal exchange” by M. Frean and S. Marsland [9], which demonstrates through value iteration that safe swapping is possible between strangers. However, these new models address certain limitations in their work, which is detailed further in Section 2.

The first model, referred to as the “swapping model”, examines whether a population of agents can naturally learn the behaviours necessary to perform safe swaps with strangers in episodic meetings, in a system akin to bartering.

- Each episode, the population is randomly divided into pairs, and each agent pair undergoes a “meeting”. In a meeting, one agent is hungry but starts off holding a drink, while the other is thirsty but starts off holding food. This creates a coincidence of wants. Agents take turns toggling holds on each item and each other, until the agents exit with the items they are holding. If an item is held by both agents, or an agent is being held, neither can exit. Each agent is rewarded for how well their obtained items meet their needs, but punished (with a negative reward) the longer they stay in the meeting.
- This creates a mixed-sum game [10], where the optimal outcome for either agent is to exit with both items, but the optimal strategy is to cooperate and achieve a swap as quickly as possible. This serves as a prerequisite to monetary exchanges by demonstrating that safe swaps between strangers can be learnt through experience, given the ability to hold. In this context, a “hold” is any means of preventing the other agent from leaving with or using items, although it can be thought of as a physical grasp for example.

The second model, referred to as the “token model”, builds on the first by investigating learned agent behaviours when intrinsically worthless yet persistent tokens are introduced in a continuous stream of meetings.

- Each meeting, one agent is hungry, while the other is not hungry but starts off holding food. Leaving a meeting hungry and without food incurs a massive pain (a negative reward) due to starvation. However, either agent may also start a meeting with a token. A token is a holdable item just like food, but instead of giving a reward upon exit, the agent will bring it into their next meeting. One token each is given to a random 50% of the population at the start of training, and tokens are then free to flow throughout the population like an economy.
- This creates a scenario where barter is impossible; food is the only item of value, and there is never a coincidence of wants. Gift-giving is also a suboptimal strategy as agents have no way of identifying each other. Therefore, this model provides the perfect opportunity for agents to use tokens as money, simply due to its property of persistence.

Both models use an extension of the Q-learning algorithm designed specifically for this project, referred to as “variable order Q-learning”, or VOQ. VOQ extends the trajectory of the Bellman optimality equation [11] to dynamically adjust the number of state transitions considered when updating Q-values. This borrows ideas from temporal difference (TD) learning [12] and allows agents to learn from passive state transitions. Passive state transitions, where an agent’s state is changed by a different agent’s actions, arise due to the use of MARL in shared environments, and must be learned from for agents to adapt to each other’s actions. Each agent stores their own Q-values in a Q-table, and the epsilon-greedy policy is used to balance exploration of the environment with exploitation of learned behaviours [13].

Additionally, both models are equipped with features for aiding experimentation and analysis. A diverse suite of parameters can be modified in code or loaded from a JSON file to customise the MDP, the agent population, and the training process. Agent Q-tables can also be saved and loaded from a JSON file alongside parameters, allowing the training process to be seamlessly paused and resumed, or inspected in a human-readable format. Finally, the models produce a variety of visualisations for investigating learned agent behaviours. This includes Q-table plots for examining a single agent’s preferred actions, “swap dance” graphs displaying what actions two agents would perform across the span of a meeting, and cosine distance plots for measuring the consistency of agent behaviours across the population.

C. *Key Findings*

The ultimate goal of this project was to determine whether reinforcement learning is capable of learning safe swapping behaviours and money-like behaviours, extending the work in “Holds enable one-shot reciprocal exchange” by M. Frean and S. Marsland [9]. Therefore, the models were evaluated by their effectiveness in determining whether the target behaviours can be learned or not. These involved analysing visualisations of the learned behaviours of each model across a range of population sizes. Shown in Figures 8 and 9, each run produced a swap dance graph and cosine distance plot. These were used to quickly determine if two agents had learned the target behaviours, and whether the rest of the population had converged to the same or similar behaviours. The population size in each visualisation is 2, 4, and 8, from top to bottom.

In Figure 8, the swap dance graphs across all population sizes exhibit identical sequences of actions, showcasing successful and symmetric swaps between agents. Additionally, a consistent downward trend in the cosine distance can be seen for all population sizes during training, signifying convergence toward similar swapping behaviours. Notably, larger populations demonstrated a slightly smoother downward trend, suggesting increased stability and scalability of these swapping behaviours. This evidence strongly supports the conclusion that the swapping model can learn safe swapping behaviours, and therefore that safe swapping behaviours can be learnt through reinforcement learning.

Figure 9 shows successful token-food swaps with mostly symmetric action sequences in the swap dance graphs for population sizes 4 and 8. However, a population size of 2 fails to achieve a token-food swap, displaying seemingly random actions without clear cooperation. The cosine distance plot for the population of 2 ends with an upwards trend, indicating divergence at the end of the training process. Despite this,

there are some instances of small cosine distances, suggesting sporadic convergence. On the other hand, the cosine distance plots for population sizes 4 and 8 show similar stable trends to each other, both gradually converging after the bulk of exploration is done. The cosine distance plot is the most stable with population size 8, implying potential scalability and increased stability of the token model with larger populations. This evidence affirms that the token model is capable of learning money-like behaviours, and therefore that money-like behaviours can be learnt through reinforcement learning.

II. Literature Review

To ensure the reinforcement learning models being produced are unique and advantageous, a literature review has been conducted on the value iteration model this project is extending. This review discusses the model's approach and outlines its limitations regarding the purpose of this project.

In the paper "Holds enable one-shot reciprocal exchange" [9], M. Frean and S. Marsland created a value iteration model that trains two agents to optimally navigate a competitive Markov decision process (MDP). An MDP represents a decision-making problem containing states, actions, state transition probabilities, and rewards [8]. In this case, the MDP models the combined states of two agents and two items, where agents can toggle holds on items or each other, pass, or exit with the items they are holding if not contested. Additionally, the agents value the items differently and start off holding the item they value least. This means that although the best outcome for an agent is to exit with both items, the best outcome overall is for them to achieve a swap. On the other hand, value iteration is a dynamic programming algorithm used to solve MDPs, proven to converge to an optimal solution given enough iterations [14]. Therefore, the value iteration model is able to prove when safe swapping is possible and when it is not.

In terms of developing insights into the initial motivations behind the use of money, this existing model has severe limitations. Firstly, value iteration as an algorithm requires full knowledge of the MDP from the start [14]. This means any agent using value iteration already has omniscience regarding the immediate outcomes of every possible decision. As this requirement is impossible for humans, behaviours observed in humans cannot be accurately modelled or compared using value iteration alone. Secondly, the closest example to mercantile exchange the model can produce is having a money-like item with the same perceived value to both agents. However, for safe swapping to remain the optimal behaviour, this value cannot be zero (unless the agent holding the other item wants to give it away for nothing). As safe swapping requires an immediate reward for receiving money, the model is incapable of representing it as an inherently worthless token as it is in reality.

To address these limitations, both of the project's models utilise reinforcement learning, and the second model implements truly worthless tokens. Through reinforcement learning, the models naturally learn from interactions with the environment and adapt their behaviour based on feedback, similar to humans. This allows for a more realistic representation of the decision-making process concerning the use of money. In the second model, the introduction of truly worthless tokens better captures the essence of money as an inherently valueless medium of exchange. By not giving the tokens an intrinsic value, the model can explore the motivations behind the use of money without the bias of immediate rewards or preconceived notions of value.

III. Tools and Methodology

A. Programming Language

There are several popular programming languages available for reinforcement learning. For this project, it was important that I selected a language in which I had sufficient experience to reduce the learning curve and start prototyping quickly. Additionally, the selected language needed to have libraries for easy numerical handling and processing, along with the ability to display results in the form of various graphs. I considered Java, C++, and Python for this project, but ultimately went with Python.

Python is a high-level programming language that offers similar features to Java and C++, like extensive library support and object-oriented programming [15]. However, its relative simplicity, readability, and fast prototyping capabilities were key advantages that proved essential to achieve the scope of this project in the given time [16]. Python's short, easy-to-understand syntax and use of dynamic typing allowed for more time spent on functionality and experimentation instead of fixing syntax. Additionally, Python supports various libraries that are popular for manipulating numerical data and generating various types of graphs, such as NumPy [17] and MatPlotLib [18]. These libraries are known for their ease of use and efficiency, and were invaluable for implementing the learning algorithm and visualising results. Furthermore, my prior experience with developing machine learning models in Python minimised the personal learning curve. For these reasons, Python was the most suitable programming language for this project when compared to Java and C++.

B. Software Libraries

As mentioned in the previous subsection, Python offers several popular software libraries that simplify and optimise the implementation and analysis of reinforcement learning algorithms. Two libraries in particular were used extensively throughout the project: NumPy and MatPlotLib.

NumPy [17], short for Numerical Python, provides a powerful multidimensional array object, the NDArray or N-Dimensional Array, that enables efficient and flexible operations on large data sets. It includes a diverse set of mathematical functions for array manipulation, linear algebra, and statistical analysis. Almost every component of each model uses NDArrays or NumPy operations to achieve otherwise tedious or error-prone tasks. These tasks included initialising arrays with arbitrary dimensions and size, determining the maximum or minimum value of a multi-dimensional array, or chaining functions to perform complex operations like normalisation of

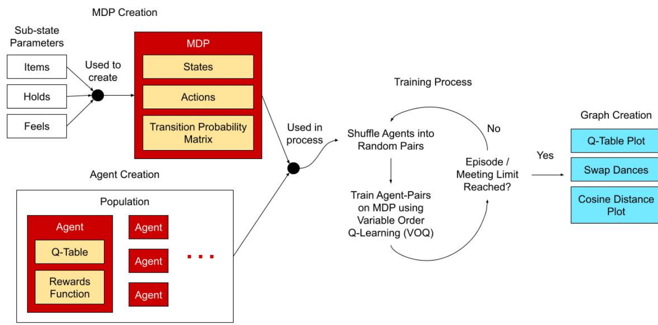

Figure 1. Base Model Flow Diagram. Shows the high-level design of both models. Execution generally flows from left to right. Object types are organised by type; white is collections, red is classes/instances, yellow is class fields/methods, and blue is graphs.

an entire Q-table. Ultimately, this made these operations much more concise, efficient, and easier to implement.

MatPlotLib [18], on the other hand, is a versatile data visualisation library with user-friendly functions for creating a wide variety of plots and charts. It is used in the graphing component of the models to plot Q-tables, state diagrams, state transition graphs, and cosine difference scatter plots. Due to its incredible flexibility, it is the only visualisation library required to produce every plot except swap dances, meeting almost every need. Some complex features used from MatPlotLib included manually-positioned, styled text labels on state transition graphs, colour bars next to Q-tables, manually-resized elements, and the saving of diagrams to different file formats like PDF and PNG. This heavily streamlined the iterative design process, as plots took very little time to implement, update, and display, allowing results to be viewed very quickly.

C. Development Methodology

This project explored a novel approach combining reinforcement learning, swapping behaviours, and money, which made it prone to unforeseen design problems. Therefore, its development methodology needed to accommodate frequent design changes and feedback from supervisors. I considered two popular methodologies; waterfall and agile. The waterfall methodology follows a strict linear approach with phases that must be completed sequentially and little room for change. On the other hand, the agile methodology promotes flexibility, incorporating short development cycles with continuous feedback and adaptive planning. Due to the project's evolving nature and the factors discussed above, the agile methodology was chosen as the most suitable option. This was expressed through weekly iterations and meetings with the project's supervisors, and proved essential for quickly handling the many design problems we faced. These design problems are discussed in Section 4.

IV. Design

The two models share a very similar design in most aspects. Their architecture, as shown in Figure 1, is composed of four main components: MDP (Markov decision process) creation, agent creation, the training process, and graph creation. These are each responsible for different tasks, but flow on from each other to train a population of agents in a curated environment and visualise their learned behaviours. Ultimately, this design aims to demonstrate whether a population of agents is capable of learning to safely swap items or exhibit money-like behaviours through reinforcement learning, given certain parameter values.

Firstly, MDP creation is responsible for defining the states, actions, and transition probability matrix of the environment. These components are derived from sub-state parameters defining the items available, the holds that can be established, and how agents can feel (hungry agents get a larger reward from food, etc.). A state is defined by how an agent feels, what items exist in their current meeting, what they are holding, and what the other agent is holding. Actions include toggling holds on the other agent or any item in the meeting, passing, or exiting the meeting. The transition probability matrix represents the possible state transitions of the environment, dictating which actions are possible in each state and what the resulting state would be when that action is taken.

Next, the agent creation component is responsible for creating the population of agents. An agent is a single reinforcement learner that has its own Q-table and rewards function, which act as the agent's brain and source of pleasure/pain respectively. The Q-table is created at the same time as the agent, and is used for determining what action the agent will take in each state. The rewards function calculates the immediate reward of a certain state transition.

The training process component is responsible for training the population of agents over a number of meetings using variable order Q-learning (VOQ). Each meeting, agents are randomly sorted into pairs and take turns performing actions to navigate the MDP. The immediate and future rewards of these actions are then used to update the agent's Q-table. When working correctly, agents should learn to prefer actions that maximise their long-term rewards.

Finally, the graph creation component constructs plots that visualise the results of the training process. Each plot displays the learned behaviours of agents from a different perspective, with a trade-off between the number of agents shown and the level of detail. Q-Table plots show individual agent behaviours in each state, swap dance graphs show two agents' behaviours across a sequence of a states, and cosine distance plots show the similarity between all agents' behaviours over time.

A. Design Problems

While designing the two models, some of the initial ideas resulted in practical issues that impeded agent learning, or conceptual issues that made the models less applicable to the real life phenomenons they were trying to recreate. Of these problems, the most significant one was getting Q-learning to work in a MARL context. This issue ultimately resulted in the design of variable order Q-learning (VOQ), which is now used in both models.

The swapping model initially used a plain Q-learning algorithm, where the temporal difference target of a state-pair would be the immediate reward of the action, plus the

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

Figure 2. Q-learning equation.

discounted Q-value of the best action in the next state. The full Q-learning equation is shown in Figure 2.

Q-learning was chosen for its general effectiveness in reinforcement learning problems, and its simplicity which would help streamline the debugging process. The idea was that if there was a bug, it would be easy to verify that the bug was not located in the Q-learning algorithm due to its straightforward and transparent implementation. Ironically, using plain Q-learning itself resulted in a major bug.

In both models, every "meeting" involves two agents interacting with each other. The environment is shared between both agents, and therefore, the actions of one agent can change the state of the other agent. This introduces two concepts:

- An agent can change state without performing an action. I.e., a "passive state transition."
- An agent's actions can influence future actions of the other agent.

In order to learn these two concepts, agents must learn from passive state transitions, despite not performing an action. Plain Q-learning is incapable of this, as agents only learn *when they themselves are performing an action*. This creates discontinuities or gaps in the propagation of Q-values whenever a passive state transition occurs, and as a result, Q-values only consider rewards up until the other agent acts. In other words, with plain Q-learning, agents do not learn from the effects of other agents' actions, and thus are unable to learn any behaviours that require the other agent to perform certain actions. This makes deliberate cooperation impossible, preventing agents from learning swapping behaviours.

I experimented with two different approaches to get agents to learn from passive state transitions.

Firstly, I implemented "Pseudo-Passing". Passing is an action that keeps an agent in the same state, allowing them to wait for the other agent to perform an action. Therefore, a passive state transition could be interpreted as one agent passing while the other performs an action. I.e., a fake/pseudo pass. This approach gave passive state transitions an action, allowing Q-learning to learn these state changes and fully propagate Q-values throughout each agent's Q-table. This enabled cooperation and worked for the most part, but introduced a major flaw: pseudo-passing and regular passing shared the same Q-values, inflating the expected value of regular passing. Even when an agent could perform other more beneficial actions, they would often prefer to pass. This resulted in agents perpetually passing in some states instead of exiting with their desired item, because they falsely learned that waiting in the same state is the same as changing state due to another agent's actions. In other words, with pseudo-passing, each agent valued regular passing as if it forced the other agent to act, leading to sub-optimal behaviours where agents pass at inappropriate times.

Secondly, I took inspiration from temporal difference (TD) learning, and attributed passive state transitions to the agent's last action and the state they were in when they performed it. This essentially blames an agent's last decision for all the following actions performed by the other agent. This approach was effective at ensuring agents considered the consequences of their actions, and its design evolved into a new learning algorithm referred to as variable order Q-learning, or VOQ. VOQ extends the Bellman optimality equation for Q-values in a similar way to higher-order TD learning, allowing VOQ to perform a single Q-value update from an arbitrary number of state transitions. The full VOQ equation is shown in Figure 3.

When updating a Q-value, VOQ considers all rewards the agent receives until its next action, along with the estimated value of the state where it can take that next action. These are discounted appropriately, and the equation accounts for any number of passive state transitions in between actions. In other words, with VOQ, an agent learns that their last action was responsible for all the following state transitions, and they need to pick a different action next time if they want the other agent to respond differently. This teaches agents the full consequences of their actions, making them fully capable of learning cooperative behaviours. Figure 4 gives a visual overview of how each learning algorithm updates Q-values from the perspective of one agent.

B. Sustainability Considerations

In terms of sustainability, the design of the two models focuses primarily on environmental and technical considerations. This is through design decisions that minimise energy usage and future maintenance requirements.

The models prioritise energy efficiency through a combination of lightweight design and optimisation strategies. Both models employ a variant of Q-learning, a computationally lightweight algorithm compared to more complex alternatives like deep Q-networks and actor-critic methods. The design emphasises minimising the size of various components, including individual states, state space, transition probability matrix, agent Q-tables, and the number of actions. For instance, the swapping model contains 129 states, and the token model has 673, demonstrating a conscious effort to maintain small state spaces while retaining the ability to recreate complex social interactions. Additionally, the use of the NumPy library optimises numerical operations involving agent Q-tables and transition probability matrices, contributing to lower energy usage and shorter training times.

To reduce future maintenance requirements, the models prioritise code reuse. Key components, such as the MDP class, Agent class, graphing functions, persistence functions, and utility functions, are shared across both models. This approach minimises the need for redundant fixes and enhances code clarity. The entire code-base, including classes, methods, and

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\sum_{i=0}^{n} \left(\gamma^i r_{t+i} \right) + \gamma^{n+1} \max_a Q(s_{t+n+1}, a) - Q(s_t, a_t) \right)$$

Figure 3. Variable Order Q-learning equation, where 'n' is the "order", or the number of passive state transitions since the agent's last action at time-step 't'.

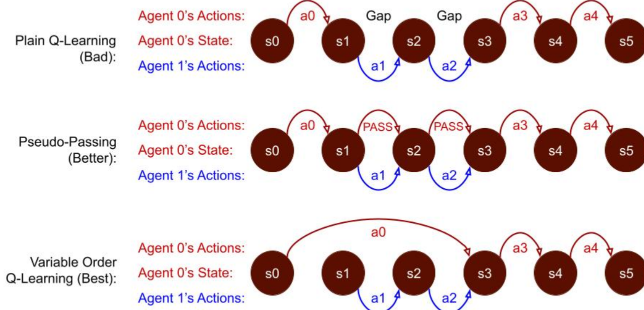

Figure 4. Overview of Q-value updates for each attempted learning algorithm, from the perspective of agent 0 (red). Actions by agent 0 (red) create active state transitions, and actions by agent 1 (blue) create passive state transitions. Arrows point from the state that had its Q-value updated (paired with the action under the arrow) to the state that provides the estimated future value of the update. 'Gap' indicates a state transition where Q-values failed to propagate backwards.

functions, is thoroughly documented, lowering the knowledge barrier for future maintainers. Finally, the software's dynamic design facilitates customisation through a diverse set of parameters for the MDP, population, and training process. This flexibility ensures ease of adaptation to new ideas, reducing the maintenance burden for future users.

V. Implementation

Like with their designs, both models have very similar implementations. Parameters defined by the user are used to create an MDP and a population of agents. Variable order Q-learning (VOQ) is then utilised to train the agents as they explore the shared MDP over a number of episodes. The results of this training are then plotted using three plots: a Q-table plot, a swap dance graph, and a cosine distance plot. Firstly, the resulting Q-table plot shows that an agent is able to learn optimal behaviours for individual states. Secondly, the swap dance graph shows that a pair of agents are able to learn optimal behaviours across the span of a whole meeting. Finally, the cosine distance plot shows that the entire population of agents has learned similar optimal behaviours across their entire Q-tables by the end of the training process.

The models were developed in Python using classes and methods in a structure that follows the design described in the design section. Extra functions were also created for persistence and general utility operations. Overall, the code can be split into five parts: MDP creation, agent creation, the training process, graph creation, and utility functions. All code can be found in the "src/" folder of the project's repository, with "src/main.py" controlling overall execution of the swapping model, and "src/main_token_continuous.py" controlling overall execution of the token model.

A. MDP Creation

MDP creation starts with a set of customisable parameters defining the possible states each agent can be in. These parameters include what items can be brought into each interaction, the number of holds each agent is capable of, and the various "feels" or preferences that agents can have. These parameters are then passed into the MDP class, which determines the full list of possible states, starting states, actions, and transitions between states. States are represented as binary strings (e.g., [0, 1, 1, 0, 0, 1, 1, 0, 1]), and possible state transitions are stored as values within a transition probability matrix. This is all stored within the MDP object, representing the "world" of the agents, which can then be passed around in later functions. The MDP class is stored in "src/mdp.py".

B. Agent Creation

Agent creation defines the population of agents to be used in the training process. The size of the population is set, and a corresponding number of instances of the Agent class are created. Each agent instance contains its own Q-table, which acts as the agent's brain for deciding which action is best in a given state, a rewards matrix, which defines the value of leaving with certain items with a given "feels" value, and a rewards function, which calculates the immediate reward experienced by the agent for any state transition. The Agent class is stored in "src/agent.py".

C. Training Process

The training process uses variable order Q-learning (VOQ) to train each agent over a specific number of episodes or meetings. An episode or meeting begins with randomly pairing up all agents within the population. Each pair is then given a random set of compatible starting states from the MDP, and the agents randomly take turns making actions until an agent exits or the maximum number of steps is reached. Actions are selected using the epsilon greedy policy. Each action changes the states of both agents, gives them rewards, and consequently updates values in their Q-tables such that better actions are made in following episodes. This process also comes with other adjustable parameters, including an epsilon value (the likelihood of picking a random action), learning rate (scales each Q-value change), and gamma value (scales the importance of possible future states when updating Q-values, aka the discount factor). Linear, exponential, and logarithmic functions have been implemented for decreasing epsilon and the learning rate with each episode. The linear function is currently in use as it has been found to provide a good balance between exploration and exploitation of the MDP.

In terms of differences between the swapping model and token model, states are completely reset between episodes in

the swapping model. With the token model, states are also reset between meetings, aside from an agent's stored tokens. Tokens are persistent, and at the end of each meeting, each agent will increment their "stored token count" by the number of tokens they exited with. Upon starting a new meeting, an agent will always start with one token if their stored token count is one or more, and their stored token count will be decremented. All training-related code can be found in "src/training.py" for the swapping model, and "src/training_token_continuous.py" for the token model.

D. Graph Creation

Graph creation starts with generating a unique run ID and creating the directory "results/⟨run_id⟩/plots/" for the swapping model, or "runs/⟨run_id⟩/results/⟨most_recent_meeting_id⟩/plots" for the token model, to house the graphs of the current run. The Q-tables of the agent population are then processed to produce three plots: a Q-table plot, a swap dance graph, and a cosine distance scatter plot. All plots are created using the MatPlotLib Python library, and additionally the GraphViz Python library [19] for swap dances, then saved as PDFs or PNGs. All graphing-related code can be found in "src/graphing.py".

For the Q-table plot, as shown in Figure 5, an random agent's Q-table is divided by its item preferences or "feels" (e.g., one column for when it is hungry, the other for when it is thirsty). Each sub-Q-table is then plotted as an image where each pixel's (x, y) coordinates correspond to its (action, state), and the colour represents its value. This plot is especially useful for analysing an agent's behaviours within a single state, as it is made explicitly clear which actions have higher Q-values, and therefore which actions the agent prefers. It is also invaluable for checking if their training was insufficient or faulty, as this would cause it to prefer sub-optimal actions.

For the swap dance graph, as shown in Figure 6, two random agents are selected and a breadth-first search is used to find each agent's preferred state transitions from a particular starting state. This creates a graph that branches out from the starting state, showing each agent's preferred actions and its resulting state transitions until they exit, pass, or enter a state they have already visited. The swap dance demonstrates the behaviours of both agents over an entire episode or meeting, showing their effectiveness at planning ahead, influencing each other, and cooperating. Like the Q-table plot, it can also detect insufficient or faulty training. However, it does so from a different perspective, focusing on decision sequences instead of individual decisions. This allows the model to be evaluated from two distinct dimensions.

Finally, for the cosine distance plot, as shown in Figure 7, every possible pairing of agents has their Q-tables compared with each other to calculate their cosine distance. These cosine distances are then plotted, with the episode or meeting number of the Q-table on the x-axis. Cosine distance measures the angle between two vectors, which in this context represents how different the values in two Q-tables are. Therefore, the cosine distance plot demonstrates how agent behaviours have become more similar (lower values) or different (higher

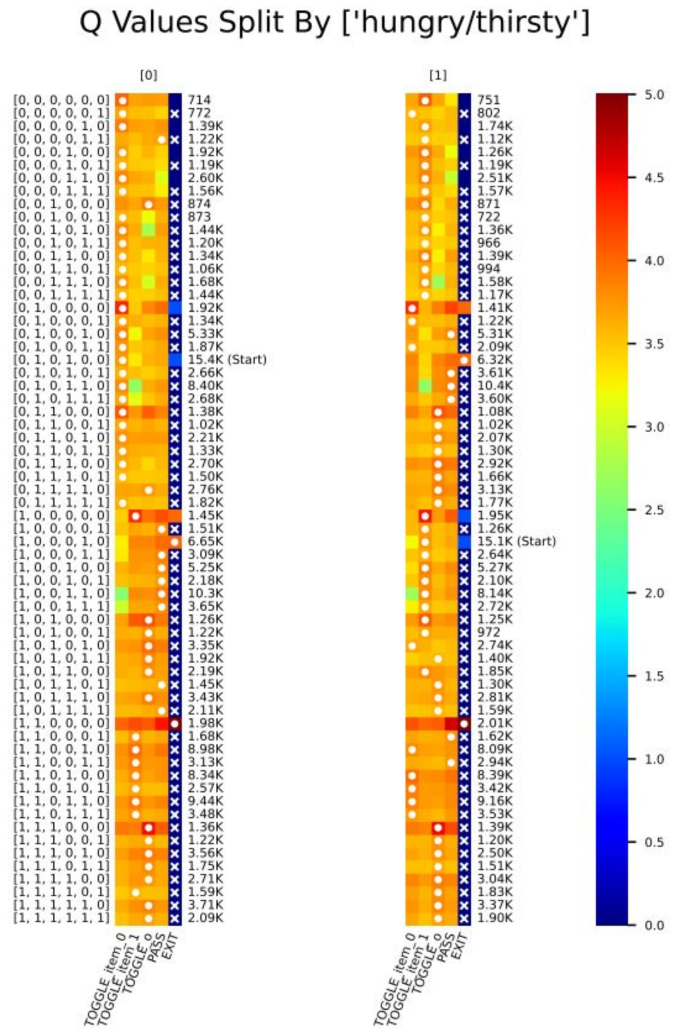

Figure 5. An agent's learned Q-table, split into two columns of hungry states vs thirsty states. Each cell represents a Q-value, with its numerical value visually represented by a colour. Colours can be interpreted by referring to the colour bar on the right. For each cell, the x coordinate represents the action, while the y coordinate represents the state. The values on the right of each row show how many times that state's Q-values have been updated. Cells with crosses describe impossible actions, and cells with white circles are the preferred action/s when in that state.

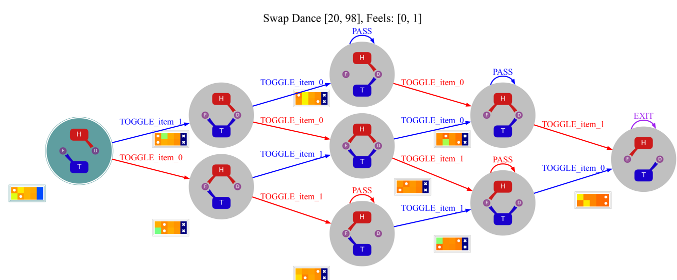

Figure 6. The swap dance of two agents after training on the swapping model. Each node depicts a state, while the edges/arrows represent the preferred action that the agent of that colour would take when in a given state. The top agent (red) is hungry, as shown by the 'H' label, and the bottom agent (blue) is thirsty, as shown by the 'T' label. In between them is two items, food on the left ('F'), and drink on the right ('D'). The box underneath each state shows a row from each agent's Q-table that corresponds to their current state. As shown by final state on the right where they both exit with their desired item, the agents have successfully learned how to swap items.

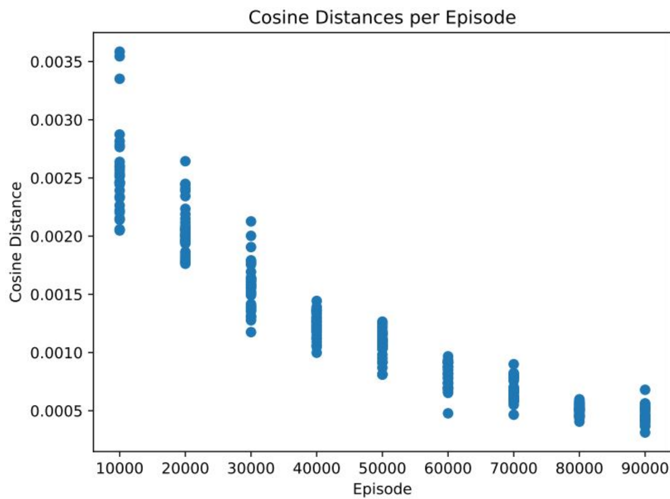

Figure 7. The Cosine Distances of the population after training on the swapping model for 100,000 episodes with a population size of 8. Having 8 agents creates 36 unique agent pairings. The cosine distance of each pairing is plotted using the agents' current Q-tables at every 10,000 episode interval. As the plot shows a clear trend downwards, agent behaviours across the population are becoming more similar over time.

values) over time. This allows the model to be evaluated from one final dimension: the consistency of agent behaviours across the entire population.

VI. Evaluation

The ultimate goal of this project was to determine whether reinforcement learning is capable of learning safe swapping behaviours and money-like behaviours, extending the work in "Holds enable one-shot reciprocal exchange" by M. Frean and S. Marsland. Therefore, the models should be evaluated on their effectiveness in determining whether the target behaviours can be learned or not. Typical performance benchmarks would be inappropriate for this. Instead, learned behaviours from each model must be analysed to form these conclusions.

Learned agent behaviours throughout a population can be most effectively understood through viewing a corresponding swap dance graph and cosine distance plot. Swap dances display the preferred actions of two agents within a meeting, giving a high-level overview of their learned behaviours that is easy to follow. This makes it simple to tell whether the target behaviours were learned or not at a glance. On the other hand, cosine distance plots show how similar learned behaviours are throughout the population at different points during training. Used together, one can quickly determine if two agents have learned the target behaviours, and whether the rest of the population has converged to the same or similar behaviours.

Swap dances and cosine distance plots for the swapping model and token model are shown in Figure 8 and Figure 9 respectively. Both figures show plots for runs with population sizes of 2, 4, and 8, ordered from top to bottom.

As shown in Figure 8, the swap dances at all population sizes are identical, demonstrating successful swaps with completely symmetric agent behaviours. Furthermore, all the population sizes achieve a clear downwards cosine distance trend during the training process, indicating that the rest of the population has converged to these same or similar clean swapping behaviours. If the cosine distance trend is affected by population size in any way, a larger population appears to make convergence more stable. This is indicated by the slightly smoother downwards trend when the population size is 8 as compared to the other two. This suggests that these swapping behaviours are scalable and can be achieved by even larger population sizes. From this evidence, it is clear that the swapping model is capable of learning safe swapping behaviours, and therefore they can be learned through reinforcement learning.

As displayed in Figure 9, the swap dances for population sizes 4 and 8 demonstrate successful swaps of a token for food. These two swap dances are also mostly symmetric, with agents mirroring many action sequences. However, a population size of 2 fails to converge on a swap, with its swap dance showcasing seemingly random action sequences without clear cooperative behaviours. Additionally, the cosine distance plot for a population of 2 appears very chaotic when compared to the other cosine distance plots, with no discernible trend for the first three points, and a concerning upwards trend for the last three points that suggests its behaviours have diverged from those of the other agent. However, it is worth noting that the cosine distance plot shows a downwards trend in the middle like the other population sizes, and there are three points with a very small cosine distance, possibly implying that the agents learned similar money-like behaviours a couple of times, but kept diverging due to instability. On the other hand, the cosine distance plots of population sizes 4 and 8 show very similar trends. This includes a period of divergence for the first third of the training process, probably due to high epsilon values and thus exploration, before converging throughout the remainder of the training as epsilon falls. This trend appears to be more stable with a population size of 8, as it has no gaps in between points for the same meeting. This could suggest that, like the swapping model, the token model might be scalable to larger population sizes, or even more stable as the population grows. From these observations, it is evident that the token model is capable of learning money-like behaviours, and therefore they too can be learned through reinforcement learning.

VII. Conclusions and Future Work

In this project, two reinforcement learning models, referred to as the swapping model and token model, were designed, implemented, and experimented on to determine whether reinforcement learning could learning safe swapping and money-like behaviours in restrictive general-sum games, composing the episodes or meetings. Through the analysis of visualisations from various runs, it can be safely concluded that both the swapping model and token model are able to achieve their desired agent behaviours.

However, the models are not free from limitations, with plenty of room for improvement regarding their implementations.

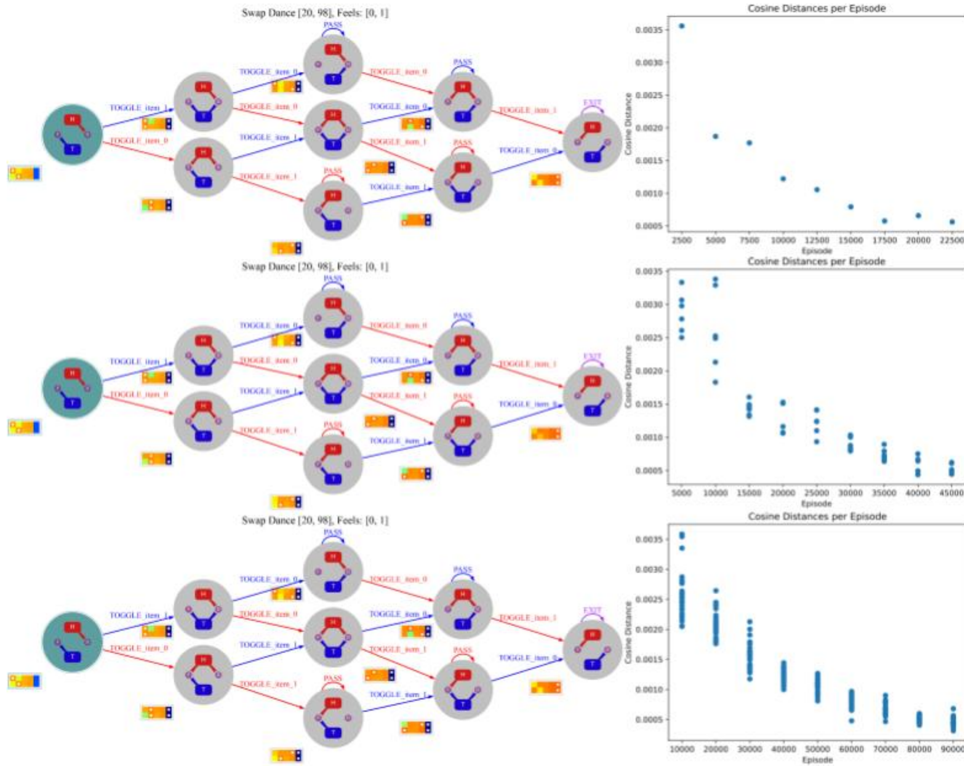

Figure 8. Swapping model results across population sizes 2, 4, and 8, from top to bottom. Trained for 25k, 50k, and 100k episodes respectively. At every population size, agents have learned safe swapping behaviours and execute them in a clean, symmetric fashion. These behaviours are consistent throughout each population.

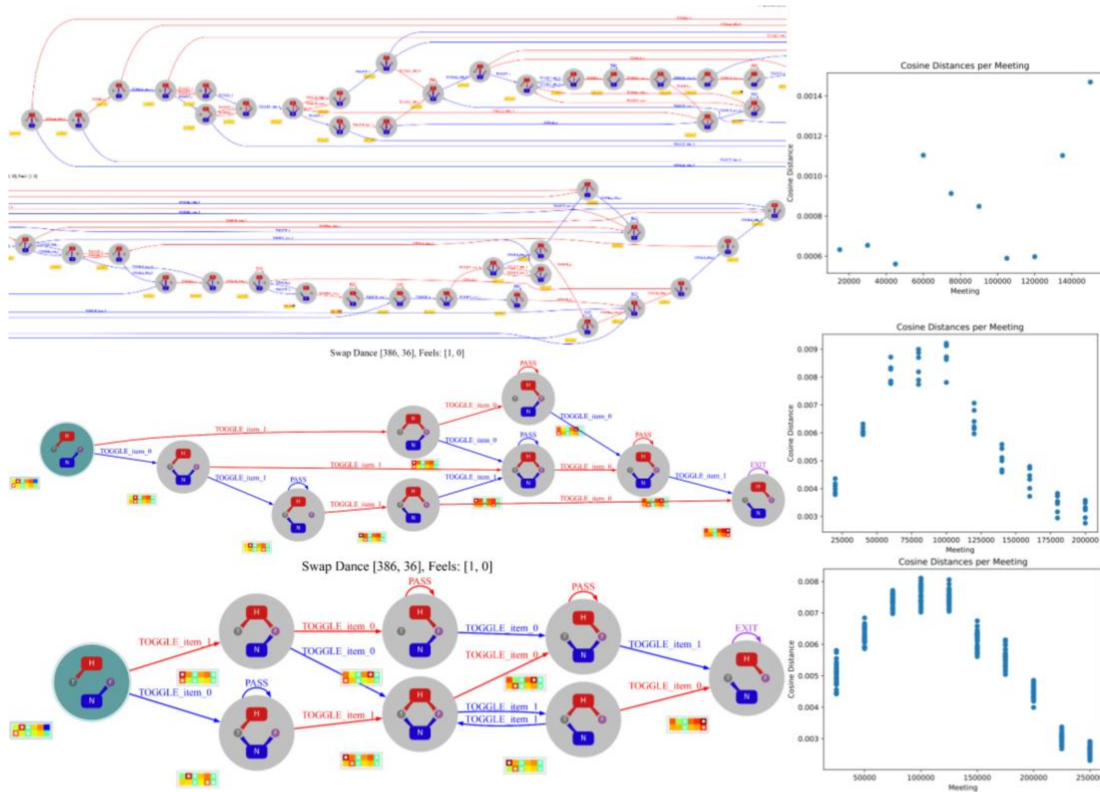

Figure 9. Token model results across population sizes 2, 4, and 8, from top to bottom. Trained for 100k, 150k, and 200k meetings respectively. The models of population sizes 4 and 8 successfully converged to money-like behaviours, swapping tokens for food, while 2 failed to achieve a swap. These behaviours appear more stable at larger population sizes. Note that the top swap dance is split in half due to its width.

Firstly, the majority of the code-base is written once and reused for both models. This includes all classes and functions for MDP creation, agent creation, graphing, persistence, and utility operations. However, this does not extend to "main.py" and "main_token_continuous.py", or "training.py" and "training_token_continuous.py", which contain plenty of duplicate functionality between them for handling each model's initialisation and training process. For example, both training files contain the exact same function implementing the epsilon greedy policy. Like the rest of the code-base, these files should have their functionality merged together to remove redundancy and make it more concise and maintainable. Although this is not a functional limitation, it is important for supporting future maintainers and achieving sustainability goals regarding long-lasting code.

Secondly, manually setting parameter values is currently a very unintuitive process. The models allow parameters to be loaded from a saved run, loaded from a JSON file, or manually set in the code. However, specifying the path of the JSON file must also be done in the code, so there is practically no way of customising the models without navigating the code-base. This would be a cumbersome process for any future users or maintainers, and there should at least be a way of setting parameters through command-line arguments when running the program, either through specifying a file path to load values from or entering which values to set directly. Along with this being more user-friendly, it would also allow a greater degree of automation, promoting the use of automated scripts to execute the models many times with different parameters. Overall, this would make the models more user-friendly and versatile.

Finally, although the models already benefit from plenty of optimisations from efficient libraries like NumPy, they could be significantly more optimised through multi-threading. Every pair of agents learns independently from the rest of the population each meeting or episode. Due to this natural partitioning of the training process, multi-threading could significantly speed up training if implemented properly. This could save future users plenty of time, and make the user-experience of the model feel better overall.

In terms of future work, all the limitations and potential improvements listed above could greatly improve the experience of future users and maintainers. However, a key idea that builds on the results of this project is investigating how easily bartering behaviours can transition to money-like behaviours. In this project, agents learned both how to safely swap and how to use money from scratch. However, this process does not fully reflect human history. Instead of learning from money-like behaviours from scratch, humans already had knowledge of how to barter, and could then use that swapping knowledge when learning how to use primitive forms of money. Therefore, it could be worth testing this idea through reinforcement learning to see if that process could be recreated. In particular, one approach could be achieving safe swaps with food and drink, then gradually swapping out drinks for persistent tokens, and observing whether agents are able to smoothly transition from one behaviour to another. This could further contribute to research regarding money and social behaviours.

Acknowledgments

I would like to thank Marcus Frean and Stephen Marsland for their tremendous support as supervisors of this project, pitching ideas and navigating the many design problems we encountered along the way. I would also like to thank Victoria University of Wellington for the use of their hardware to run these models, and the rest of humanity for running on inherently worthless, persistent tokens.

References

[1] C. Kusimba, "When – and why – did people first start using money?," The Conversation, Feb. 28, 2019. https://theconversation.com/when-and-why-did-people-first-start-using-money-78887
[2] D. Despain, "Early Humans Used Brain Power, Innovation and Teamwork to Dominate the Planet," Scientific American, Feb. 27, 2010. https://www.scientificamerican.com/article/humans-brain-power-origins/
[3] N. Szabo, "Shelling Out: The Origins of Money — Satoshi Nakamoto Institute," nakamotoinstitute.org. https://nakamotoinstitute.org/shelling-out/
[4] Wikipedia Contributors, "Hindsight bias," Wikipedia, Sep. 24, 2019. https://en.wikipedia.org/wiki/Hindsight_bias
[5] "Reinforcement Learning: An Introduction — MIT Press eBooks — IEEE Xplore," ieeexplore.ieee.org. https://ieeexplore.ieee.org/book/6267343
[6] United Nations, "The 17 sustainable development goals," United Nations, 2023. https://sdgs.un.org/goals
[7] Wikipedia Contributors, "Multi-agent reinforcement learning," Wikipedia, Jun. 09, 2023. https://en.wikipedia.org/wiki/Multi-agent_reinforcement_learning
[8] Wikipedia Contributors, "Markov decision process," Wikipedia, May 20, 2019. https://en.wikipedia.org/wiki/Markov_decision_process
[9] M. Frean and S. Marsland, "Holds enable one-shot reciprocal exchange," Proceedings of the Royal Society B: Biological Sciences, vol. 289, no. 1980, Aug. 2022, doi: https://doi.org/10.1098/rspb.2022.0723.
[10] "Non-Zero-Sum Games," cs.stanford.edu. https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/nonzero.html
[11] "Bellman equation," Wikipedia, Apr. 25, 2021. https://en.wikipedia.org/wiki/Bellman_equation
[12] "Temporal difference learning," Wikipedia, Jul. 12, 2021. https://en.wikipedia.org/wiki/Temporal_difference_learning
[13] "Papers with Code - Epsilon Greedy Exploration Explained," paperswithcode.com. https://paperswithcode.com/method/epsilon-greedy-exploration
[14] "What is value iteration in reinforcement learning?," Google LaMDA, Apr. 21, 2023. https://lambdagoogle.com/ai-faq/what-is-value-iteration-in-reinforcement-learning/
[15] Python, "Welcome to Python.org," Python.org, May 29, 2019. https://www.python.org/
[16] GeeksforGeeks, "Python Language Advantages and Applications - GeeksforGeeks," GeeksforGeeks, Oct. 23, 2017. https://www.geeksforgeeks.org/python-language-advantages-applications/
[17] Numpy, "NumPy," Numpy.org, 2009. https://numpy.org/
[18] Matplotlib, "Matplotlib: Python plotting — Matplotlib 3.1.1 documentation," Matplotlib.org, 2012. https://matplotlib.org/
[19] S. Bank, "graphviz: Simple Python interface for Graphviz," PyPI. https://pypi.org/project/graphviz/