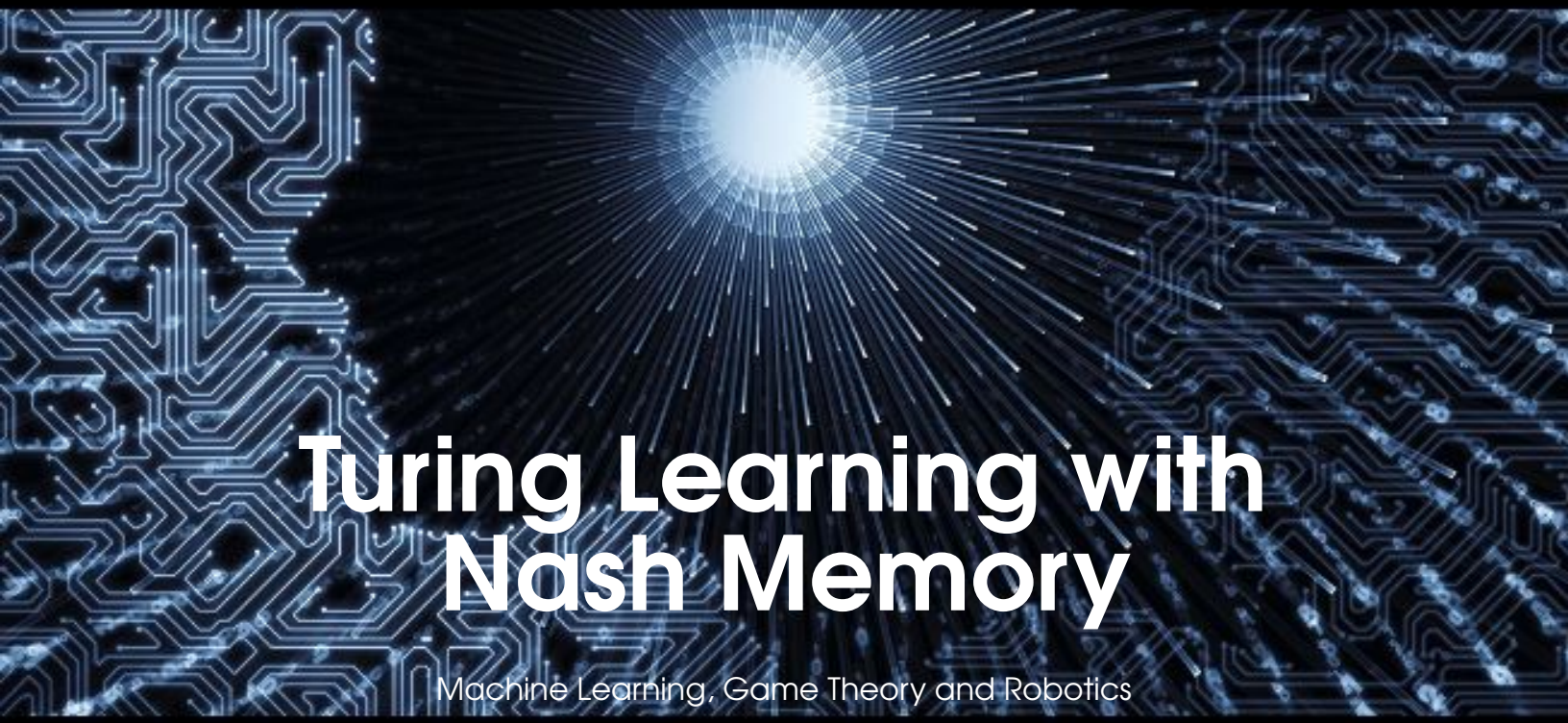


Turing Learning with Nash Memory

Machine Learning, Game Theory and Robotics

Shuai Wang

Master of Science Thesis

University of Amsterdam

Local Supervisor: Dr. Leen Torenvliet (University of Amsterdam, NL)

Co-Supervisor: Dr. Frans Oliehoek (University of Liverpool, UK)

External Advisor: Dr. Roderich Groß (University of Sheffield, UK)

ABSTRACT

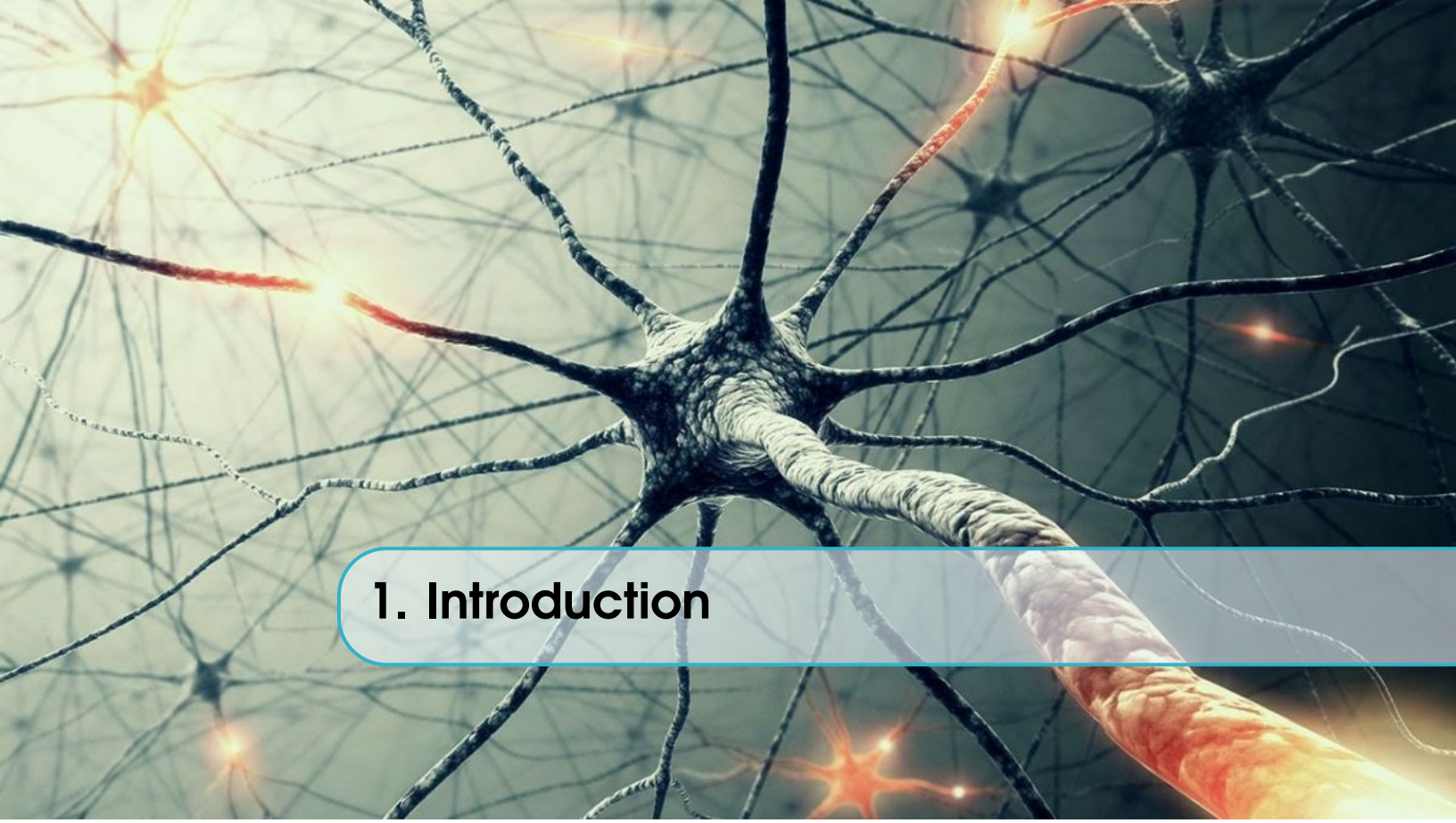
Turing Learning is a method for the reverse engineering of agent behaviors. This approach was inspired by the Turing test where a machine can pass if its behaviour is indistinguishable from that of a human. Nash memory is a memory mechanism for coevolution. It guarantees monotonicity in convergence. This thesis explores the integration of such memory mechanism with Turing Learning for faster learning of agent behaviors. We employ the Enki robot simulation platform and learns the aggregation behavior of epuck robots. Our experiments indicate that using Nash memory can reduce the computation time by 35.4% and results in faster convergence for the aggregation game. In addition, we present *TuringLearner*, the first Turing Learning platform.
Keywords: Turing Learning, Nash Memory, Game Theory, Multi-agent System.

COMMITTEE MEMBER:
DR. KATRIN SCHULZ (CHAIR)
PROF. DR. FRANK VAN HARMELEN
DR. PETER VAN EMDE BOAS
DR. PETER BLOEM
YFKE DULEK MSC

MASTER THESIS, UNIVERSITY OF AMSTERDAM

Amsterdam, August 2017

Contents



1. Introduction

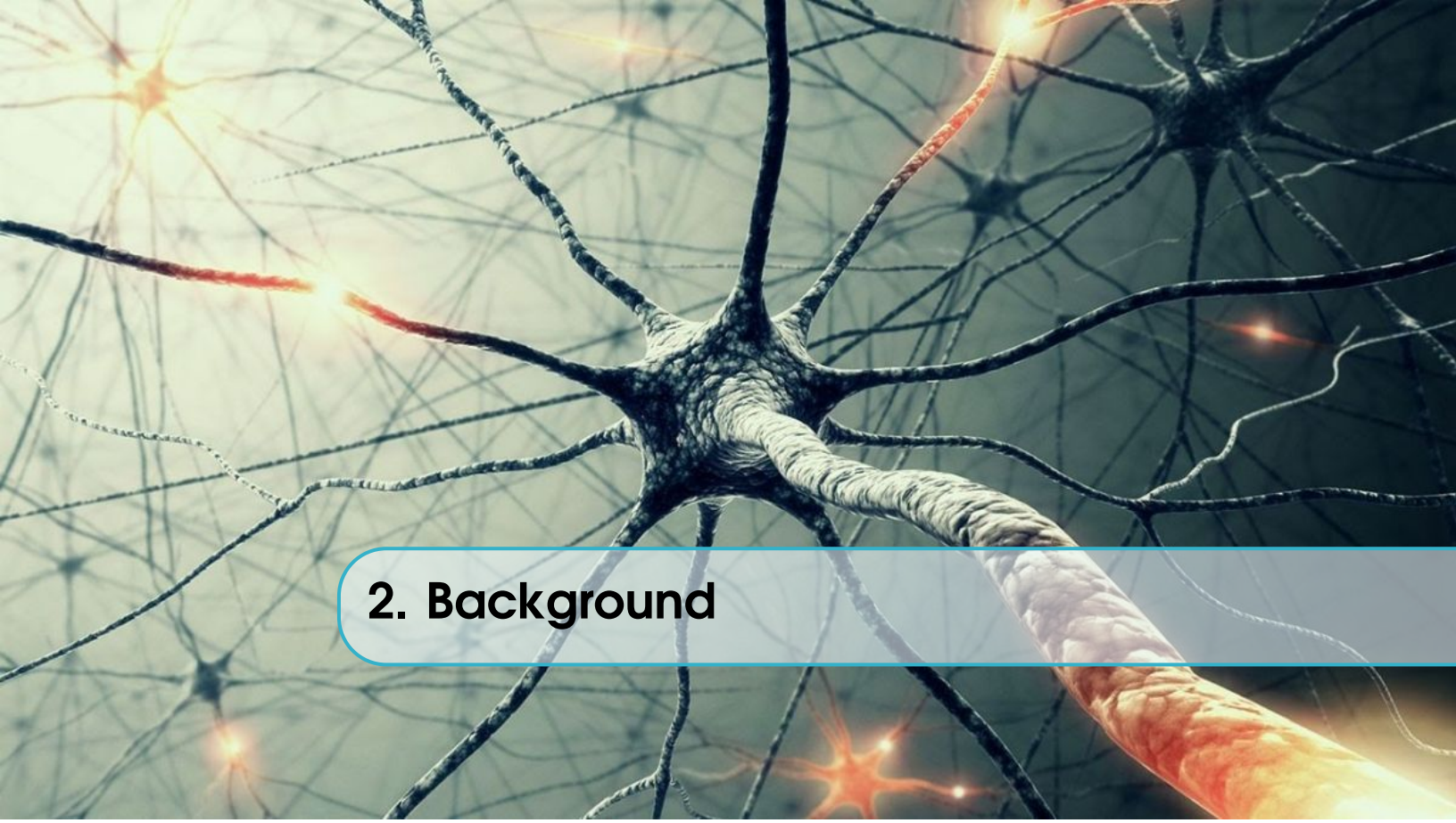
Turing Learning is a method for the reverse engineering of agent behaviors [20]. This approach was inspired by the Turing test where a machine can pass if its behaviour is indistinguishable from that of human. Here we consider a human as a natural/real agent and a machine as an artificial agent/replica. The behaviour of each agent follows their internal model respectively. In this test, an examiner is to tell two kinds of agents apart. The Turing Learning method follows a co-evolutionary approach where each iteration consists of many experiments like that of Turing test. On the one hand, this co-evolutionary approach optimizes internal models that lead to behavior similar as that of the individuals under investigation. On the other hand, classifiers evolve to be more sensitive to the difference. Each iteration of Turing Learning can be considered a game between models and classifiers with the result of experiments as payoff. This co-evolutionary approach has been reported to have better accuracy than classical methods with predefined metrics [20, 19].

Nash Memory is a memory mechanism for coevolution named after the concept of Nash equilibrium in Game Theory [8]. The two players form a normal-form game and the matrix entries are the payoff values. It updates its memory at each iteration according the Nash Equilibrium of the game corresponding to the payoff matrices. The initial design was only for symmetric zero-sum games. Parallel Nash Memory is an extension of Nash Memory and handles also asymmetric games [25].

This thesis extends Parallel Nash memory and introduces Universal Nash Memory (UNM). We study the integration of UNM with Turing Learning for efficient reverse engineering of agent behaviors. More specifically, we investigate the balance of strategies in UNM and study how UNM impact Turing Learning when facing changes and uncertainty in interaction. The current design of Turing Learning employs a simple coevolutionary scheme. The existing implementations of Turing Learning are hard-coded proof-of-concept test cases. We presents *TuringLearner*, the first Turing Learning platform together with test cases of Intransitive Number Game (ING), Percentage Number Game (PNG) and Real Number Game (RNG). Finally, we implement the aggregation game together with its two variants, Backwards Walking Game (BWG) and Safe Walking Game (SWG). More

specifically, we employ the Enki robot simulation platform and learns the aggregation behavior of epuck robots. Our experiments indicate that this new approach reduces the computation time to 35% and results in faster convergence for the aggregation game. The project is an international project co-supervised by Dr. Leen Torenvliet and Dr. Frans Oliehoek with Dr. Roderich Groß as external advisor.

This Thesis is organized as follows. Chapter 2 gives a general introduction to game theory, neural networks and evolutionary computing. Chapter 3 provides the foundations of evolutionary computing and introduces Turing Learning. Chapter 4 introduces Nash Memory and Parallel Nash Memory and combines Turing Learning with this memory mechanism. Following that are the design, implementation and evaluation details in Chapter 5. Finally, Chapter 6 further discusses the algorithms and presents further evaluation and proposes future work.



2. Background

2.1 Introduction to Game Theory

2.1.1 Players and Games

The solutions and strategies of individuals in interactive processes have long been of great interest to many. The interactive individuals are named *players* and participate in games. For a two-player game, each player P_i has a set of *actions* A_i available, $i \in \{1,2\}$. Players are decision makers at each step. At the end of the interaction, each player obtains a *payoff*. This computation is also known as the utility function of the actions players take. In this thesis, we consider only players that are selfish and non-cooperative. In other words, players try to maximize their pay-off regardless of that of the opponents. Note that a game can be defined as an interactive process between some actual individuals/agents (bidders in a flower market for example), as well as abstract entities.

The *strategy* s of a player P_i is the reaction to (its knowledge of) the *state*. The reaction can be a deterministic action (a *pure strategy*) or a set of actions with some probability distribution (a *mixed strategy*). If the agent cannot fully observe the current state of the game, the game exhibits state uncertainty. We say the player has partial or imperfect information and the game is a partial information game, otherwise, a perfect information game. *Game Theory* is the study of the strategy, interaction, knowledge, and related problems in such setting [3]. Games can be further classified according to properties such as:

Symmetric/Asymmetric Game A game is symmetric when the payoff of a specific strategy is independent from its player and the order. The Prisoner's Dilemma is a canonical example of symmetric game. Each player can cooperate or defect. The payoff matrix is as in Table 2.1. When both players cooperate, they both get punished a little and get a payoff of -1. When they both defect, they obtain a payoff of -5. When they play different strategies, the one who defect is free from punishment (payoff 0) while the one who cooperate gets the maximum punishment and the payoff value is -10. Using matrices M_1 and M_2 as a representation of

the game, we see that for any i and j, $M_1[i,j] = M_2[j,i]$. Formally, we define a game to be symmetric if $M_1 = M_2^T$ (the transpose of M_2). This thesis deals with asymmetric non-zero-sum games in normal form.

P_1 / P_2	Cooperate	Defect
Cooperate	-1, -1	-10, 0
Defect	0, -10	-5, -5

Table 2.1: The payoff matrix of the Prisoner's Dilemma game.

$$M_1 = \begin{bmatrix} -1 & -10 \\ 0 & -5 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} -1 & 0 \\ -10 & -5 \end{bmatrix}$$

Transitive/Intransitive To better explain with examples, we restrict the definition to symmetric games. A pure strategy s is preferred over another one s' (denoted $s > s'$) if the pay-off $E(s,s') > 0$, E is the evaluation function. A game is transitive if for all pure strategies s, s' and s'' , we have $s > s'$ and $s' > s''$ implies $s > s''$. Otherwise, the game is intransitive. A well-known intransitive game is the Rock-Paper-Scissors game (RPS game), which is also the smallest intransitive game.

P_1 / P_2	Rock	Paper	Scissors
Rock	0, 0	-1, 1	1, -1
Paper	1, -1	0, 0	-1, 1
Scissors	-1, 1	1, -1	0, 0

Table 2.2: The payoff matrix of the Rock-Paper-Scissors game.

Zero-sum/Non-zero-sum/Constant-sum A constant-sum game is a game where the sum of the pay-off of agents is a constant c. That is, $M_1[i][j] + M_2[i][j] = c$ for all i and j. A zero-sum game is where this constant $c = 0$. Clearly, the Prisoner's Dilemma game is a non-zero-sum game while the Rock-Paper-Scissors game is a zero-sum game. For simplicity, we normally denote such a game as one matrix representing the payoff of the row player P_1. For the Rock-Paper-Scissors game, we have M_{rps}. In this thesis, we deal with mostly non-zero-sum games.

$$M_{rps} = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}$$

2.1.2 Strategies and Equilibria

As we defined above, players choose strategies and get some payoff in games. A *pure strategy* or a *deterministic strategy* gives one action as the reaction to the state. While a *mixed strategy* of an agent

is a set of actions associated with their corresponding probability. A pure strategy is a special case of a mixed strategy where there is only one action with probability 1. In this thesis, both strategies in this thesis are uniformly denoted as s for simplicity.

A two-player game has a pure or mixed strategy *Nash equilibrium* in case both players have no better payoff by changing their strategies. Formally, for two players P_1 and P_2 and their strategies s_1 and s_2, (s_1, s_2) is in a Nash Equilibrium if for any strategies s_1' and s_2', $E(s_1, s_2) \geq E(s_1, s_2')$ and $E(s_2, s_1) \geq E(s_2', s_1)$. Not all games have a pure strategy Nash Equilibrium. However, all games have at least one mixed strategy Nash Equilibrium. For example, the Rock-Paper-Scissors game does not have any pure Nash equilibrium but has a mixed strategy Nash equilibrium where the three strategies are played uniformly.

For two players with mixed strategies S_1 and S_2 respectively in a zero-sum game, we have the Min-Max Theorem to help us find the Nash Equilibrium (i.e., solve the game) [3]. The theorem states that the pair of strategy (s_1', s_2') is a Nash Equilibrium where $s_1' = argmax_{s_1 \in S_1} \min_{s_2 \in S_2} E(s_1, s_2)$ and $s_2' = argmax_{s_2 \in S_2} \min_{s_1 \in S_1} E(s_2, s_1)$. Formally:

Theorem 2.1.1 — The Min-Max Theorem.

$$\max_{s_1 \in S_1} \min_{s_2 \in S_2} E(s_1, s_2) = -1 * \max_{s_2 \in S_2} \min_{s_1 \in S_1} E(s_2, s_1) = \min_{s_1 \in S_1} \max_{s_2 \in S_2} E(s_1, s_2)$$

In matrix form, let $x \in X$ and $y \in Y$ be the (probability distribution of) mixed strategies and M be the payoff matrix. The matrix form is as follows with v as the optimal payoff:

Theorem 2.1.2 — The Min-Max Theorem (matrix form).

$$\max_{x \in X} \min_{y \in Y} x^T M y = \min_{y \in Y} \max_{x \in X} x^T M y = v$$

From the description above, a mixed strategy of Nash Equilibrium shows the best a player can do to obtain optimal payoff. Intuitively, if a pure strategy beats such a mixed strategy, it is better than the "joint force" of the actions involved.

2.2 Computing Nash Equilibria

2.2.1 Introduction to Linear Programming

A *linear programming* (LP) problem maximizes (or minimizes) a value with respect to some linear constraints. For example, we are to maximize $v = 5 * x_1 + 3 * x_2$ regarding two constraints: $x_1 + x_2 = 1$ and $x_1 - x_2 \leq 3$. We can first substitute all x_1 by $1 - x_2$ and get $v = 5 - 2 * x_2$ and $x_2 \geq -1$. That is, the maximum of v is 7 when $x_2 = -1$ and $x_1 = 2$. Real cases can be much more complicated with hundreds of variables and constraints. Thus, specialized solvers were developed and employed to solve real-life linear programming problems.

In this project, we take advantage of existing tools. The `PuLP` library [23] is an open source Python package that allows mathematical programs to be described in the Python programming language. It is easy to interact with and fast in finding solutions [23].

2.2.2 Game Solving with LP

Next we take advantage of the Min-Max Theorem and convert the searching of a Nash Equilibrium into a linear programming problem. In this section, we introduce this approach step by step by solving the Rock-Paper-Scissors. The payoff matrix M_{rps} corresponds to the row player as follows. We introduce a vector of probability corresponding to the three actions of the column player: Rock, Paper and Scissors: $x = (x_1, x_2, x_3) \in X$ (similarly, for the row player, we have $y = (y_1, y_2, y_3) \in Y$).

$$M_{rps} = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}$$

The payoff for the row player is therefore:

$$v = \max_{x \in X} \min_{y \in Y} x M_{rps}^T y$$

. That is, the row player maximizes the value

$$v = \min_{y} M_{rps}^T y$$

- When the row player chooses Rock: $v = (0 * y_1) + (-1 * y_2) + (1 * y_3)$
- When the row player chooses Paper: $v = (1 * y_1) + (0 * y_2) + (-1 * y_3)$
- When the row player chooses Scissors: $v = (-1 * y_1) + (1 * y_2) + (0 * y_3)$

Now that the column player maximizes its payoff $-v$ (minimize v), the player would take the smallest of all three:

$$v \leq -y_2 + y_3$$

$$v \leq y_1 - y_3$$

$$v \leq -y_1 + y_2$$

We also know that $y_1 + y_2 + y_3 = 1$ and each of them is non-negative ($y_1 \geq 0$, $y_2 \geq 0$ and $y_3 \geq 0$).

Note that this would give us the half of the solution, namely the vector $y = [1/3, 1/3, 1/3]$. We can use similar approach to compute $x = [1/3, 1/3, 1/3]$. That is, there is no pure Nash Equilibrium. The best strategy is to uniformly play Rock, Paper, Scissors. An implementation (with Nash Memory) is open source at `https://github.com/airobert/RockPaperScissors`. A more detailed explanation of the code is included in Section A.3.

Now that we have some experience working using PuLP to solve games, we can generalize this approach to zero-sum games of any size. The following is a description of the procedure to find the equilibrium taking advantage of LP solvers.

Data: A pay-off Matrix M
Result: Two mixed strategies
initialise a list of variables $x = [x_1, x_2, \ldots, x_n]$;
initialise a LP solver L;
Set the objective of L to maximise v;
for *each column* c **do**
  add constraint $v \leq x_1 * M[1][c] + x_2 * M[2][c] + \ldots + x_n * M[n][c]$ to L
end
add constraint $\sum_{i=1}^{n} x_i = 1$ to L;
add constraint $x_i \geq 0$ for each $x_i \in x$ to L;
decode x from the result of L;
In similar approach, get that of the row player.

Algorithm 1: A general procedure of zero-sum game solving with LP.

2.2.3 Asymmetric Games

While symmetric games involve two players with access to the same set of actions, biological interactions, even between members of the same species, are almost always asymmetric due to access to resources, differences in size, or past interactions [21]. Soon after the introduction of game theory to biology, the study of asymmetric games has drawn attention of many [11]. Conflicts with built-in asymmetries like those between male and female, parent and offspring, prey and predator and the corresponding dynamics have been studied [28, 11, 21]. There are also a wide range of such games outside of biology: chess, poker, backgammon, etc. While asymmetric games are hard at first glance, it is possible to construct a new compound game consisting of two copies of the original asymmetric games for some asymmetric games [25]. A strategy in this game is a pair consisting of the strategies of both players in the original game: $t = (s_1, s_2)$. The payoff of $t' = (s'_1, s'_2)$ is defined as $E(t,t') = E(s_1, s'_1) + E(s_2, s'_2)$. In other words, the set of strategies is the set of Cartesian product of the set of P_1 and P_2. However, the flexibility with which the new mixed strategy is constructed is constrained: it is not possible to put weight on a particular first player strategy s_1 without putting the same weight on the corresponding second player strategy s_2 [25].

An example asymmetric game is the battle of the sexes (BoS) where a couple agreed to meet in the evening for some activities. The man prefers to watch a football match, while the woman would rather enjoy an opera. Both enjoy the company of the other. The payoff matrices are defined as follows:

$$M_{man} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$$

$$M_{woman} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

Observe that the game has two pure strategy Nash equilibria, one where both go to the opera and another where both go to the football game. There is a third (mixed strategy) Nash equilibria where the man goes to the football with probability 60% while the woman goes to the opera with probability 60%.

2.2.4 Bimatrix Games

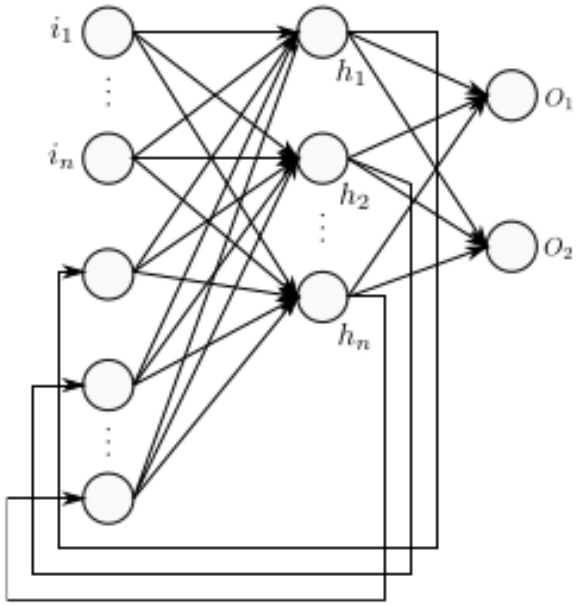
A more general type of game is the *bimatrix game* with one payoff matrix for each player. Among the the most famous algorithms, the Lemke-Howson Algorithm [26] can efficiently compute Nash equilibria for some problems. While the algorithm is efficient in practice, the number of pivot operations may need to be exponential in the number of pure strategies in the game in the worst case. It has been proved that it is **PSPACE-complete** to find any of the solutions [12]. Some more recent attempts shows that obtaining approximate Nash equilibria can be done in polynomial time [7, 26].

2.3 Neural Network

2.3.1 Introduction to Neural Network

Artificial neural networks (ANNs), or neural networks (NNs) in short, are computational models used in machine learning, computer vision, robotics and other research disciplines [24]. A network is a large collection of connected artificial neurons. Neurons are connected in layers: the signals travel from the first (input), through those in the middle (hidden), to the last (output) layer. The purpose of hidden neurons is to learn non-linear combinations of the original inputs, which is known as *feature extraction*. Depending on the structure and use, neural networks are further classified to different types. A feedforward neural network (a.k.a multi-layer perceptron) is a series of logistic regression models stacked on top of each other. The final layer is either the logistic regression or a linear regression model. Specifically, convolutional neural networks are widely used in image recognition, text processing and computer vision. One of the most well-known algorithm is the backpropogation algorithm [24]. In contrast, a recurrent neural network defines a nonlinear dynamical system where connections between units form a directed cycle. Such networks can use their internal memory to process arbitrary sequences of inputs. It is currently the best approach for problems like language modelling [24]. Simple examples include Elman networks and Jordan networks. Gradient descent is the most well-known algorithm for training such networks. While most models for simple learning tasks share a two-layer architecture, more complicated tasks require neural networks of many levels.

Figure 2.1: The structure of Elman neural network [19].



Deep learning is a recent trend in machine learning that builds neural networks of many levels. As a consequence, typically such networks have thousands to millions of parameters [24]. Despite that large neural networks can be hard to train, the past decades witnessed neural networks widely applied to tasks in robotics and swarm agents such as decision making, computer vision, classification, path planning and intelligent control of an autonomous robot, etc [18, 22, 20, 16].

2.3.2 Elman Neural Network

Figure 2.1 shows the structure of a recurrent Elman neural network [6]. This neural network consists of i input neurons, h hidden neurons, and two output neurons (O_1 and O_2). O_1 is fed back into the input and controls the stimulus. O_2 is used for making a judgement. In addition, a biased neuron with a constant input of 1.0 is connected to each neuron of the hidden and output layers. Such network has $(i+1)h+h2+(h+1)$ parameters. The activation function used in the hidden and the output neurons is the logistic *sigmoid function*:

$$sigmoid(x) = 1/(1+e^{-x}), \forall x \in \mathbb{R}$$

A neural network can serve as a controller for systems like rigid robot arm [18], autonomous vehicles and flight control [22], path planning [16], etc. Take the path planning problem as an example, autonomous robots should move safely in partially structured environment under the

control of a neural network to form a collision-free path [16]. This task may involve two (or more) neural networks. The first neural network is used to determine the "free" space using ultrasound range finder data while the second avoids the nearest obstacles and searches for a safe direction for the next robot section [16]. Neural networks are also widely used for system identification and pattern recognition [20, 19, 22]. A classifier observes the motion of an robot for a fixed time interval and outputs a boolean judgment if it has certain properties in behavior [20].

2.4 Introduction to Evolutionary Computing

2.4.1 Evolution

Darwin's theory of evolution offers an explanation of the origins of biological diversity and its underlying mechanisms. Natural selection favours those individuals that better adapt to the environment for reproduction and therefore accumulate beneficial genes [5]. Genes are the functional units of inheritance encoding phenotypic characteristics while genome stands for the complete genetic information of an individual [5]. The genome of an offspring maybe impacted by crossing-over, mutation, etc.

2.4.2 Evolutionary Computing

As the name suggests, Evolutionary Computing (EC) draws inspiration from the process of natural evolution. The idea of using Darwinian principles to automated problem solving dates back to the 1940s. The idea was first proposed by Turing in 1948 as genetical or evolutionary search. The subject embraced huge development since 1990s and many different variants of evolutionary algorithms were proposed [5]. The ideas underlying are similar: given a population of individuals within an environment with limited resources, competition for those resources causes natural selection (survival of the fittest).

Data: An initialised $\lambda + \mu$ population with random candidate solutions
Result: A pair of solutions
Evaluate each candidate;
while *Termination Condition is not satisfied* **do**
    select λ parents ;
    recombine pairs of parents and obtain μ offspring with mutation;
    evaluate new candidates;
    put together λ parents and μ offspring for the next generation;
end
return the best fit individual

Algorithm 2: The $\lambda + \mu$ algorithm.

Evolutionary Computation (EC) offers a powerful abstraction of Darwinian evolutionary models with application from conceptual models to technical problems [33]. The best-known class of EAs are arguably *genetic algorithms* (GAs) [33]. Most GAs represent individuals using a binary encoding of some kind and perform crossover and mutation during reproduction. Another type of EAs are *evolutionary strategies*. This approach is often employed for problems with a representation of real numbers. Evolutionary strategies first initialize a set of candidate genomes (i.e. solutions) at random and then simulate the competition with an abstract fitness measure. On the basis of such fitness values, some optimal candidates are chosen as *parents* to reproduce the next generation. This is done by applying recombination and/or mutation to them. *Recombination* is an operator that combines features of two or more selected candidates (i.e. parents). During reproduction, *mutation* (as an operator to each gene) may happen, which results in a partially different candidate. This process creates a set of new candidates, namely, the *offspring*. The algorithm terminates when some given terminating condition is reached. An example is the $\lambda + \mu$ algorithm (Algorithm 2) where the λ parents are selected to reproduce μ offspring. Notice that although those with better fitness are more likely to be selected, this probabilistic algorithm may not choose to let it reproduce, which is consistent with nature but may not be good for an optimization problem.

Common bisexual reproduction processes consist of two operations: recombination and mutation. A gene $g = [g_1, g_2, \ldots, g_n] \in G$ is associated with a sequence of mutation strengths (factors) $\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_n] \in \Sigma$. When producing with another gene $g' = [g'_1, g'_2, \ldots, g'_n] \in G$ with $\sigma' = [\sigma'_1, \sigma'_2, \ldots, \sigma'_n] \in \Sigma$ to form g'' and σ'', the recombination process is a choice between the parents' genes of even chance:

$$g''_i := g_i \text{ OR } g'_i, i = 1, \ldots, n; \tag{2.1}$$
$$\sigma''_i := \sigma_i \text{ OR } \sigma'_i, i = 1, \ldots, n \tag{2.2}$$

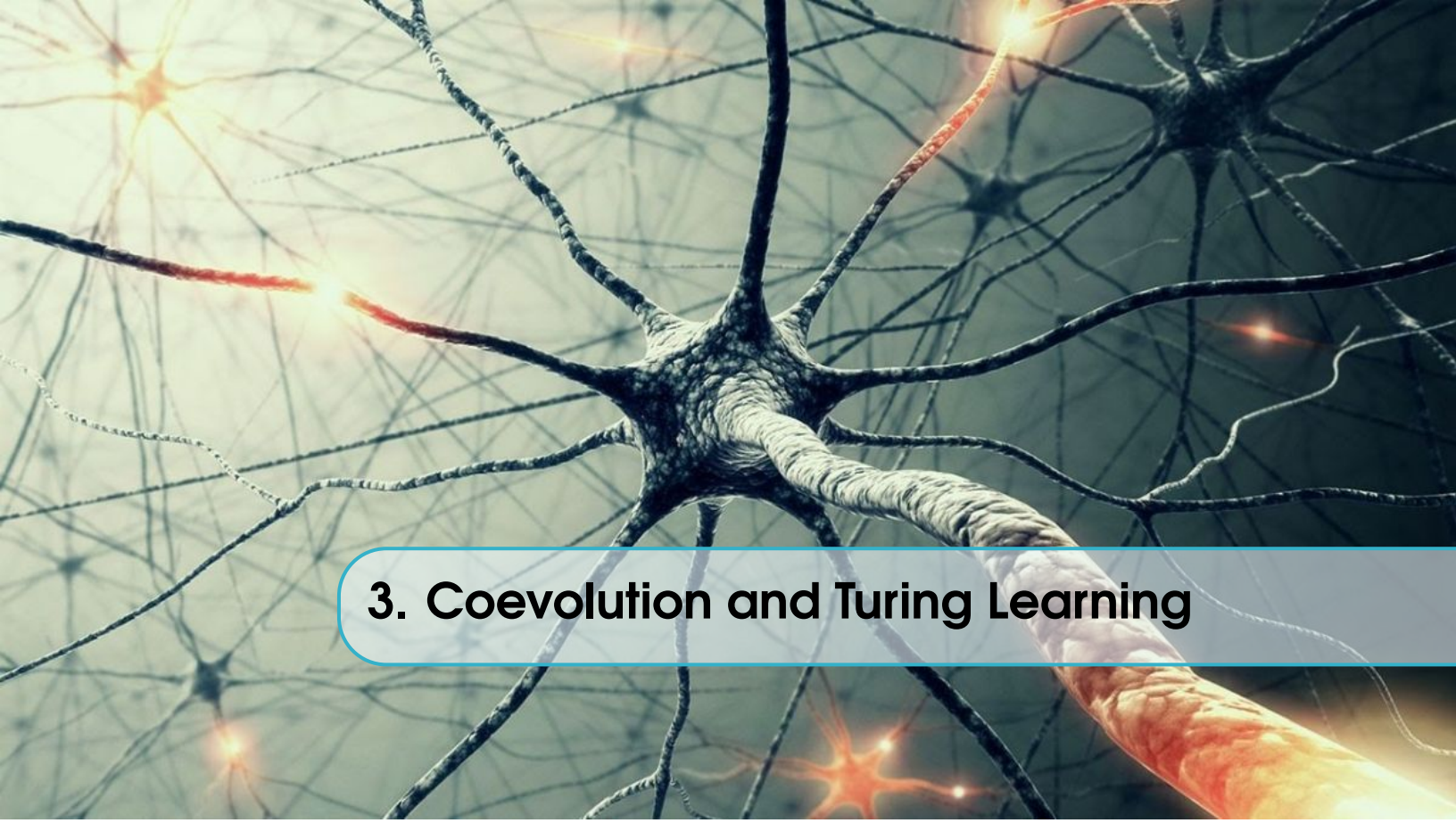
Then g' is mutated. For each generation, we first obtain a number r generated from the standard normal distribution (with mean of 0 and standard deviation as 1; see Appendix A.2 for implementation details). For each gene, in similar approach, we generate a number r_i.

$$\sigma''_i := \sigma''_i * exp(\tau' * r + \tau * r_i), i = 1, \ldots, n; \tag{2.3}$$
$$g''_i := g''_i + \sigma''_i * r_i, i = 1, \ldots, n \tag{2.4}$$

In the formula above, $\tau' = 1/2\sqrt{2 * N}$ and $\tau = 1/2\sqrt{2\sqrt{N}}$ with N being the size of the population. See Appendix A.2 for a detailed example.

EAs frequently demonstrate adaptive capability when applied to difficult problems where search

spaces are so large that they make traditional methods less tenable [33, 5]. This fascinating nature and often surprising successes of EC have drawn researchers to the field.



3. Coevolution and Turing Learning

3.1 Introduction to Coevolutionary Computation

3.1.1 Coevolution and Coevolutionary Computation

Evolutionary algorithms (EAs) are heuristic methods for solving computationally difficult problems inspired by Darwinian evolution. This approach is typical for a variety of problems from optimization to scheduling. However, in many problems in nature, the fitness of an individual does not only depends on the environment, but also on the behavior of the others in the environment. The existence of other individuals may change or have an influence on the attention, cognition, reproduction and other social activities [27]. For example, some flowering plants and honeybees have a symbiotic relationship: flowering plants depend on an outside source to spread their genes through pollination while bees receive nectar, which they convert into honey to feed their queen bee.

For certain problems in computer science, entities may change on many aspects. The corresponding search domain is defined by the Cartesian product of sub-spaces. To find a solution in such large domain can be expensive. An approach is to divide such problems into smaller sub-problems for faster searching of partial candidate solutions and then form a complete solution. The need for coevolutionary methods is therefore in demand. Recall that in Section 2.2.3 we introduced asymmetric games where the players of a game are different. To better understand the dynamics, we discuss coevolution in nature as well as in computer science in this section. To keep the glossary consistent in this thesis, we refer to an individual in evolutionary process as a *strategy*, which consists

of many genes. Abstract *players* manage strategies. In this thesis, we only deal with two-player coevolutionary processes. Coevolutionary processes have more complexity [33]. To better illustrate the concept of coevolution, we provide a simple abstract sequential cooperative coevolutionary algorithm below [33].

Data: Initialize a population with random candidate strategies for each P_1 and P_2
Result: Best strategies from both population
Evaluate each candidate;
while *Termination Condition is not satisfied* **do**
  select parents and reproduce the offspring ;
  select collaborators from the population;
  evaluate offspring with collaborators;
  select survivors from the new population;
end
return the best fit strategies in each population

Algorithm 3: A simple abstract cooperative coevolutionary algorithm.

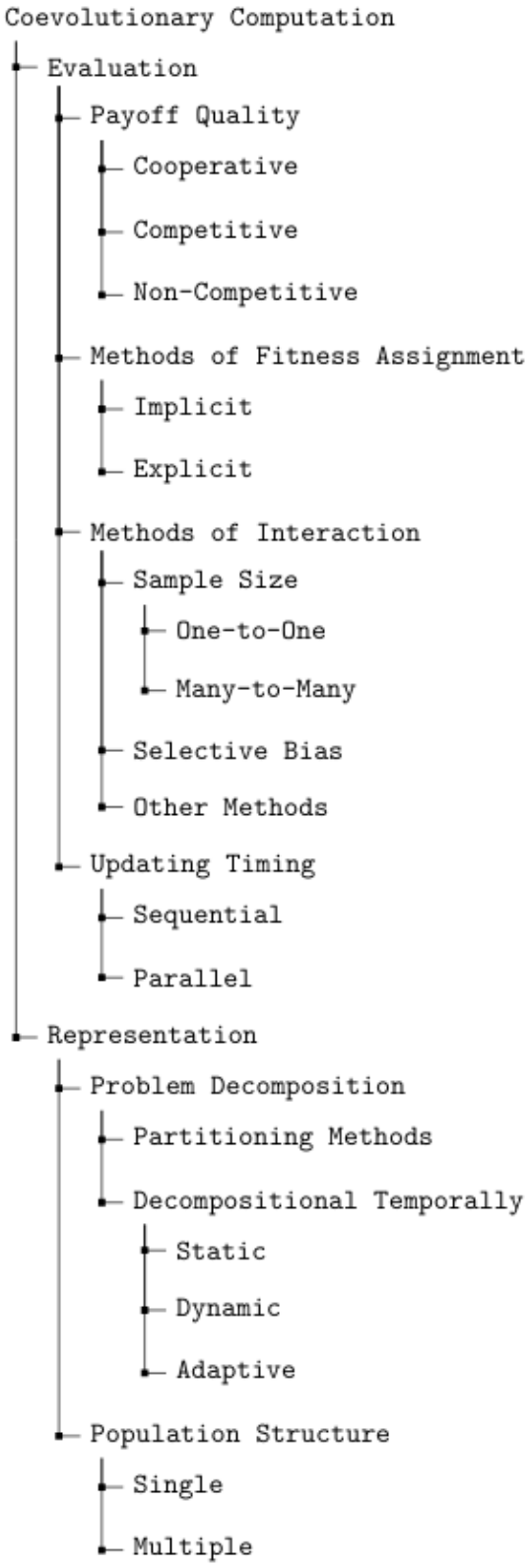
3.1.2 Coevolutionary Architectures

Coevolutionary Computation (CoEC or CEC) is the study of *Coevolutionary Algorithms* (CoEAs or CEAs) where the fitness of a strategy depends on the strategy of the other player and thus differs from traditional evolutionary methods [33, 28]. An *objective measure* evaluates an individual regardless the effect of the others, aside from scaling or normalization effects. Otherwise, it is a *subjective measure*. A measurement is *internal* if it influences the course of evolution in some way, otherwise, it is *external*. A CoEA employes a subjective internal measure for fitness assessment [33]. Depending on the relationship of the two players in an coevolutionary process, CoEAs can be further classified as *cooperative* CoEAs and *competitive* CoEAs. It is cooperative when the increase of fitness of one population of a player in the new generation leads to positive impact on the fitness of the other. Otherwise, it is a competitive CoEA. Most work in the field of CoEA studies competitive coevolution [33, 28]. In most nature-inspired computation, updating timing is sequential and the decomposition of problems are dynamic [28].

In simple coevolutionary process, a strategy usually correspond to a single object, while in some cases, it can be a group of similar/identical objects. For instance, when a strategy is a distribution of some property, the corresponding implementation may be a group of individuals following such distribution. The interactions may be one-to-one between two objects or that of a selection of some objects regarding a certain bias [33]. This thesis studies only one-to-one interactions. As a result, a payoff is assigned to each strategy. In many algorithms, the fitness is simply the sum of the payoff of a strategy against that of the opponent. Section 4.2 gives a different view on this and provides an implicit fitness evaluation. The selection of samples in evaluation is to be in Chapter 6.

Coevolutionary algorithms have an advantage of "understanding" complex structures without detailed examination and explanation of all the aspects explicitly [33]. However, this approach has

Figure 3.1: Coevolutionary Architecture Analysis [33].



its drawbacks. For instance, the dynamics is often harder to capture and the subjective fitness may not be accurate enough. One of the problems is the so-called *loss of gradient* where the strategy of one player converges making the other player lose "guidance" while coevolving. Another problem is "cyclic behavoir" where one player re-discover the same set of strategies after a few seemingly "better and better" coevolutionary generations. To further study this property of CoEAs, I present (a modified version of) the Intransitive Number (ING) Game as described in [8] to examine the properties of intransitive relations. To better understand the properties of the impact of entities during coevolution, I designed two games to be presented in Section 3.3. Strategies in the population may be treated uniformly or grouped as different populations according to their fitness and/or other properties (see Chapter 4).

3.2 The Intransitive Numbers Game

One problem associated with coevolutionary algorithm is that some traits are lost only to be needed later. To rediscover/relearn these traits may require more effort. Ficici and Pollack [8] proposed a game-theoretic memory mechanism for coevolution with evaluation using Watson and Pollack's Intransitive Numbers Game (ING) [32]. The ING game is in a co-evolutionary domain that allows intransitive relations. This section explains the game and the nature of how intransitive relations make coevolution harder. Evaluation results are presented after the introduction of a generalized version of such memory mechanism in Section 4.4.1.

Representation of Strategies Each pure strategy of the ING game is an n-dimensional vector of integers $\alpha = \langle i_1, \ldots, i_n \rangle$. Each integer i is represented as a list of b bits. For example, $\alpha = \langle 4,3 \rangle = \langle 0100, 0011 \rangle$ [1].

Evaluation Function For two pure strategies α and β, the payoff for the first player $E(\alpha,\beta)$ is calculated in three steps as follows:

$$E(\alpha,\beta) = \begin{cases} 0 & \text{if } min(h_i) = \infty \\ sign(\sum_{i=1}^{n} g_i) & \text{otherwise} \end{cases} \tag{3.1}$$

$$g_i = \begin{cases} \alpha_i - \beta_i & \text{if } h_i = min(h) \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

$$h_i = \begin{cases} |\alpha_i - \beta_i| & \text{if } |\alpha_i - \beta_i| \geq \varepsilon \\ \infty & \text{otherwise} \end{cases} \tag{3.3}$$

In Formulae 3.3, ε is the smallest magnitude difference and we take $\varepsilon = 1$. The $sign(x)$ function gives 1 if x is positive, 0 if $x = 0$ and -1 otherwise. As a zero-sum game, the payoff

[1] Note that the original game described in [8] suffers from an information theoretical problem: when an integer is represented as the sum of a list of n bits, the probability of getting the number n is $1/(2^n)$, while numbers around $n/2$ are significantly easier to obtain in evolution. Thus the searching process is influenced significantly by this distribution and, as described in the paper, very hard to converge. That's why this thesis presents a slightly different version.

is $E(\beta-\alpha)$ = 0 - $E(\alpha,\beta)$. We keep consistent with [8] and take our strategy space to be n-dimensional vectors of natural numbers, where each dimension spans the interval $[o,2^b-1]$; This results in $(2^b)^n$ distinct pure strategies. The game has a single Nash strategy when $\varepsilon < k$ and multiple otherwise [8]. The single Nash strategy is the pure strategy with value k in all n dimensions, i.e., $\underbrace{\langle 2^b-1, 2^b-1, \ldots, 2^b-1\rangle}_{\text{n}}$.

This 3-step calculation may not be obvious. The following is an example: take $\alpha_1 = \langle 4,2\rangle$, $\alpha_2 = \langle 2,6\rangle$, $\alpha_3 = \langle 7,3\rangle$. To calculate $E(\alpha_1,\alpha_2)$, we first obtain $h_1 = 2$ and $h_2 = 4$ and $min(h_1,h_2) = 2$. So $g_1 = 2$ and $g_2 = 0$ which makes $\sum_{i=1}^{n} g_i = 2 > 0$. Thus $E(\alpha_1,\alpha_2) = 1$. Likewise, we get $E(\alpha_2,\alpha_3) = 1$ and $E(\alpha_1,\alpha_3) = -1$. This gives an example of intransitive relations between three individuals in the search space.

We consider this game as a coevolutionary process where two players P_1 and P_2 each maintain a list of strategies [2]. The score w of a strategy α of P_1 is the $w = \sum_{i=1}^{n} E(\alpha,\beta_i)$ with $\beta_i \in P_2$ and vice versa. The score of all strategies is $W = [w_1,\ldots,w_n]$ and the fitness of each strategy is $f_i = w_i - min(w) + 0.1$. This guarantees the fitness of a strategy to be positive.

Population Management and Reproduction In each iteration, each player manage x strategies and the best y are selected to generate the g offspring asexually. Each bit has a mutation rate of 0.1 for example (i.e., each bit has 10% of chance to become $1-b$).

Termination The game terminates after 1000 generations for instance. Evaluation results are in Section 4.4.1.

3.3 The PNG and RNG Game

Percent Number Game (PNG) and Real Number Game (RNG) are two similar games introduced for the understanding of how the relation between two players impact coevolution. They also serve as test cases for non-zero games and bisexual reproduction.

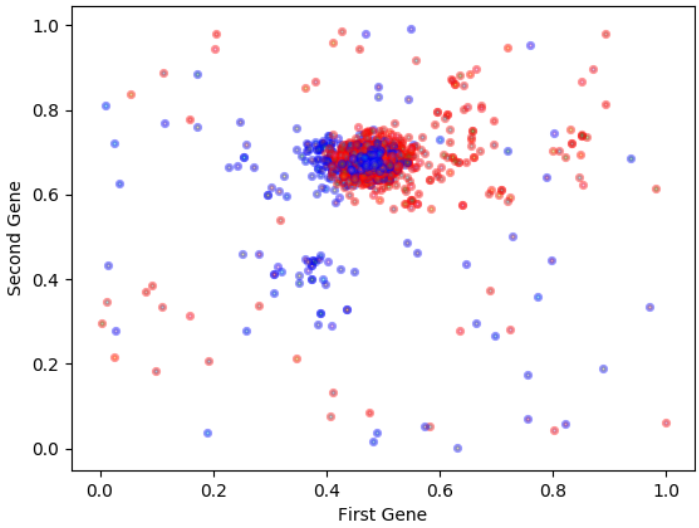
Representation of Strategies Similar as the ING game, for the PNG/RNG game, there are two players P_1 and P_2, each maintain a set of strategies. Each strategy is a point in a n-dimensional space within the range $[0,B]$, where B is the bound. For the PNG game, the bound is $B = 1$ while RNG takes $B = 100$. In 2D space, a strategy could be $\alpha = (0.5,0.3)$ for example. Before the a game starts, we fix two points $x = (x_1,x_2)$ and $y = (y_1,y_2)$ in the n-dimensional space as pre-defined attractors and assign one to each player. For each gene, we also associate each gene with a (positive) mutation rate, denoted as σ as described in Section 2.4.2. Interestingly, when $\varepsilon > 1$, the two population would eventually cluster to the same point rather than their attractors (see Figure 3.2).

Evaluation Function For a game in N-dimensional space, we define an evaluation function $E(\alpha_1,\alpha_2) = (w_1,w_2)$ with $\alpha_1 = (s_1,s_2)$ and $\alpha_2 = (t_1,t_2)$ as the strategies of the two players. When $N = 2$, w_1 and w_2 are defined as follow:

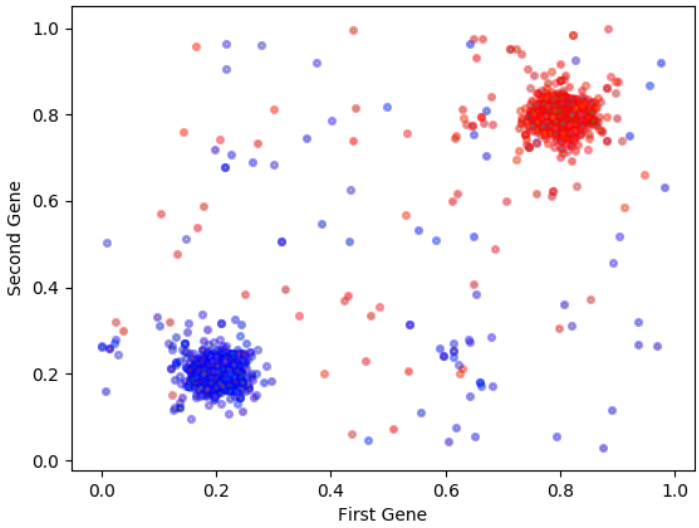
$w_1 = ((1-|s_1-x_1|)^2 + (1-|s_2-y_1|)^2) + \varepsilon * ((1-|s_2-x_2|)^2 + (1-|s_2-y_2|)^2)$.

$w_2 = ((1-|t_1-x_1|)^2 + (1-|t_2-y_1|)^2) + \varepsilon * ((1-|t_2-x_2|)^2 + (1-|t_2-y_2|)^2)$.

[2] In Ficici's paper [8], the game was transformed to an evolutionary domain.



(a) the population over 30 iterations with $\varepsilon = 2$



(b) the population over 30 iterations with $\varepsilon = 0.2$

Figure 3.2: A simulation of the PNG game with attractors at (0.2, 0.2) and (0.8, 0.8) with 20 strategies in each population

ε is the coefficient for correlation. The payoff values represent how close a point is from its attractor and the opponent's strategy. The larger ε is, the more attractive an opponent strategy than the pre-defined attractor (as a point in 2D space).

Population Management and Reproduction In each iteration, there are 20 strategies for each player and the best fit ones are selected to generate the offspring bisexually. For example, a pair of selected parents are $\alpha_1 = (s_1, \ldots, s_n)$ and $\alpha_2 = (t_1, \ldots, t_n)$, with mutation rate of $\sigma_{s1}, \ldots, \sigma_{sn}$ and $\sigma_{t1}, \ldots, \sigma_{tn}$ respectively. We reproduce a child $\alpha = (k_1, \ldots, k_n)$, with mutation rate $\sigma_{k1}, \ldots, \sigma_{kn}$.

1. First, we need to compute some intermediate values. Each k'_i has a equal chance to be either s_i or t_i while $\sigma'_{ki} = (\sigma_{si} + \sigma_{ti})/2$.
2. Then we need random numbers μ for this reproduction process and μ_i for each gene, both following Gaussian distribution with a mean of 0 and standard deviation of 1.
3. Finally, $\sigma_{ki} = \sigma'_{ki} * exp(\tau' * \mu + \tau * \mu_i)$ where $\tau' = 1/(2 * \sqrt{2n})$ and $\tau = 1/(2 * \sqrt{2 * \sqrt{n}})$.

Termination The game terminates after a given number of iteration.

3.4 Turing Learning

3.4.1 Inspiration and Background

Robots and other automation systems have greatly transformed our world over the past decades. *Swarm Intelligence* (SI) studies the design of intelligent mulit-robot/agent systems inspired by collective behavior of social insects [4]. Collectively, they can achieve much more complex tasks in cooperation than that of an individual. *Ethology* is about the study of animal behavior. Ethologists observe that some animals have similar appearance or behavior as another different species to increase the survival chance while there are still significant distinction between the two species. This is known as the *convergent evolution* where the independent evolution of unrelated or distantly related organisms evolve similar features including body forms, coloration, organs, and adaptations. For instance, in Central America, some species of harmless snakes mimic poisonous coral snakes that only an expert can tell apart [14]. Natural selection can result in evolutionary convergence under various circumstances. Some mimicry is disadvantageous to the model species since it takes longer for the predator to learn to avoid them. Ethologists often need to observe the animals and analyze the data manually to extract meaningful information from the records of animals under investigation, which can be time-consuming and tedious. With the help of different automation systems, it is much easier to conduct experiments more efficiently and accurately. The question is whether it is possible to construct a machine/system that can accomplish the whole process of scientific investigation, reasoning, automatic analysis of experimental data, generation of a new hypothesis, and the design new experiments. The development of "robot scientists" Adam and Eve by the team of Prof. Ross King gave a positive answer [17].

Nature has inspired scientist of all domains. In 1950, the father of Artificial Intelligence Alan M. Turing proposed the *Imitation Game*, which was later known as the *Turing test* [31]. A machine could

pass the Turing test if it could learn and adapt as we humans do. In June 2014, a Russian chatter bot named Eugene Goostman, which adopts the persona of a 13-year-old Ukrainian boy, fooled a third of the judges in 5-minute conversations [34]. The competition's organizers believed that the Turing test, for the first time, had been passed. However, the discussion and study of computational models continues for better simulation of the capabilities of human being in cognition, decision making, cooperation and many more aspects. Most recently, Turing Learning provides a new automated solution to the learning of animal and agent behavior [20].

Table 3.1: Turing Learning v.s. Coevolutionary Computing v.s. Turing Test v.s. Convergent Evolution.

	Turing Learning	Coevolutionary Computing	Turing Test	Convergent Evolution
Entities	Real agents, strategies and the corresponding GAOs	Genes	Human and computer programs	Model species, mimic species and the predator
Aim	Learn the behaviour of individuals thought interaction	Study the dynamics of two population of individuals	Distinguish computer programs from human by analyzing dialogues	Behave or appear like the model species
Evaluation	The classification result of an experiment	fitness computation	The accuracy of distinction	The predator's distinction

3.4.2 Turing Learning Basics

Turing Learning is a coevolutionary approach concerning the automated reverse engineering of agent behaviors [20]. While a machine passes the Turing test if its behavior is indistinguishable from that of a human, the models in Turing Learning can pass the tests by coevolving until behaving indistinguishably from the individuals being studied in tests. These individuals could be natural animals, robots, virtual agents or even groups of such individuals. The approach has been validated using two canonical problems in swarm robotics: the aggregation problem and the object clustering problem [20]. In this thesis, we introduce a game theoretical view and, as in Section 3.1.1, managers of models and classifiers are *players*. Models and classifiers are abstract representations of candidate solutions managed by the players and therefore referred to as the *model player* and the *classifier player*. For simplicity, this thesis covers only the learning of the behavior of robots in simulation. In other words, no animal, or physical robots are involved. The actual parameter of real robots/agents under investigation are *real* parameters. The generated individuals according to a strategy is *replica* robot/agent. To avoid inconsistency in glossary, from this point onwards, the explicit representation of parameters is called a *strategy* while the actual agents/robots/replicas as well as neural networks

strategies execute on are *Generative Adversarial Objects* (GAOs)[3]. From a biological point of view, while the strategies are like the genotype, GAOs are the corresponding phenotype. Overall, a system in such a setting is a *Turing Learner*.

Table 3.1 illustrates a comparison with other concepts. In Turing Learning, the coevolving entities are the unique strategies (and the GAOs generated accordingly) while the real agents under investigation remain the same in each generation. In contrast, in classical coevolution, the two species under investigation generate offspring in each generation. Turing learning shares similarities with convergent evolution: the classifiers in Turing Learning function in a way similar to predators in convergent evolution.

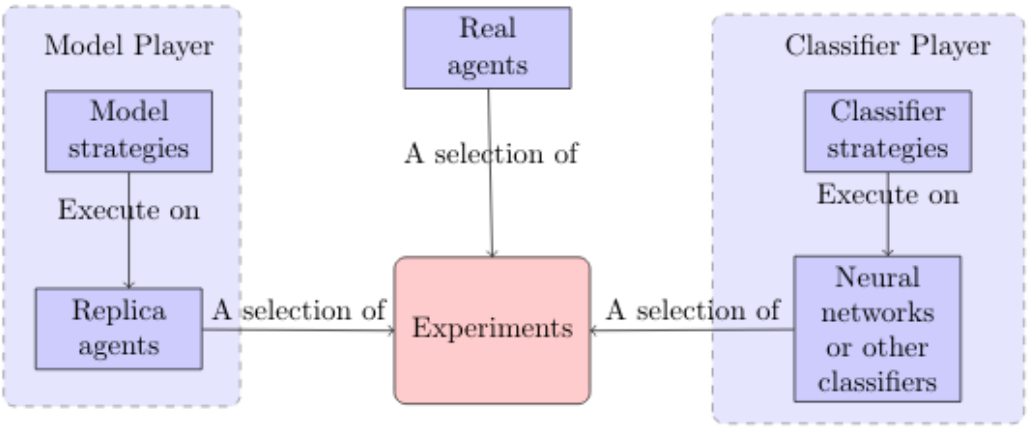


Figure 3.3: The architecture of a Turing Learner.

3.4.3 Architecture and System

A Turing Learner maintains three entities: a model player, a classifier player and the agents under investigation. A Turing Learner first initializes a model player and a classifier player; each manage a population of random strategies. In each iteration, for each strategy, the model player generates one or many replica agent(s) accordingly. An experiment is an interactive process of these agents in a sequence of timesteps. The model player then generates a classifier from a chosen strategy and take the observed data in the experiment as input. The output is the classification results for of the individuals in the experiment. The fitness of a model strategy is the probability of wrong judges by the classifiers. The fitness of a classifier is the sensitivity plus the specificity. Formally, for an experiment using C classifiers with A real agents and R replicas, the fitness values of a model and a classifier, f_m and f_c, are defined as:

[3]In the paper [20] and the PhD thesis [19] of Turing Learning, models can be represented explicitly as parameters or implicitly as replica agents or neural networks. This thesis introduces a object-oriented design, which would benefit the object-oriented architecture of Turing Learner to be presented in Chapter 5

$$f_m = 1/C \sum_{r=1}^{C} (1 - O_r)$$

$$f_c = 1/2((1/A \sum_{a=1}^{A} (1 - O_a)) + (1/R \sum_{r=1}^{R} O_r))$$

In the formula above, O_r is the output of the classifier when evaluating on replica agents. O_a is the output of the classifier when the input is from a real agent. At each generation, both players then select among a given number of strategies as parents according to their fitness value and generate the offspring. The learning process terminates after a given number of generations.

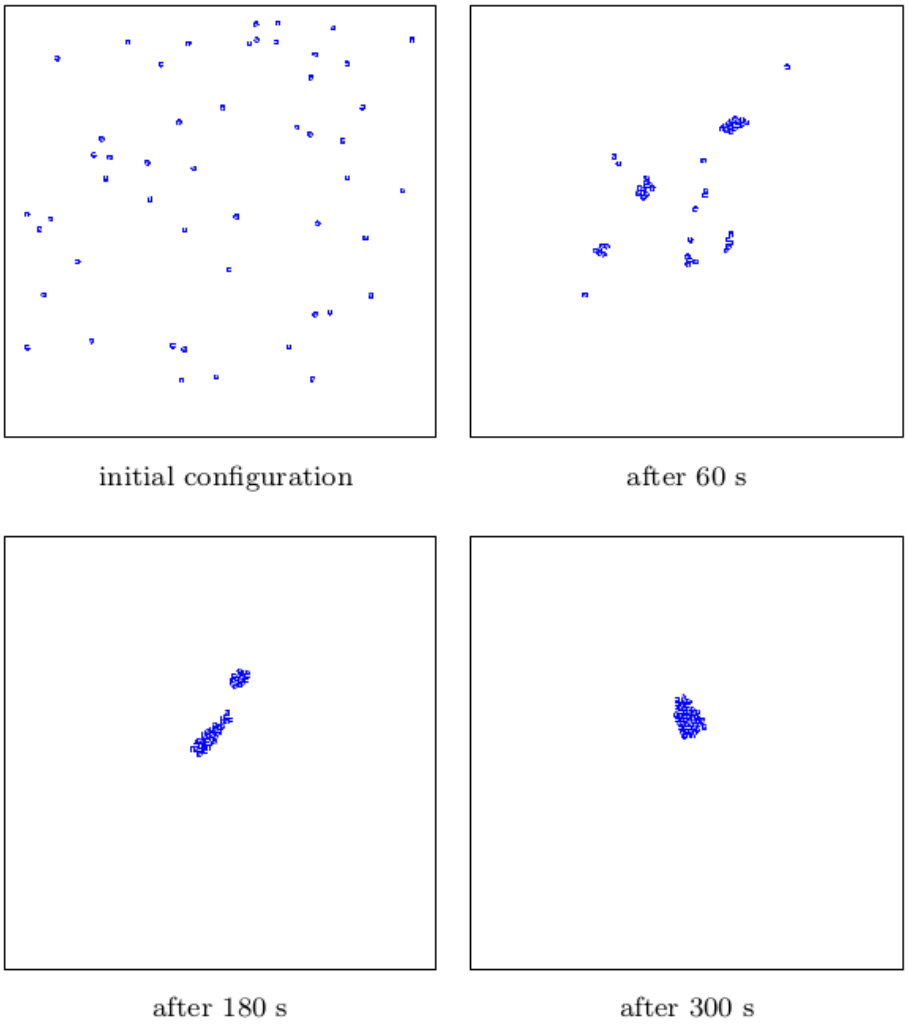
Different from existing behavior learning approaches, Turing Learning focus on the learning of agent behavior through interaction. This learning approach has its requirement in setting: an experiment consists of several real agents and replica agents. Agents must interact with each other to form observable evidence. Classifiers are not only to examine the evidence of the replica agents, but also the real agents. The goal is to learn the interactive behavior of some real agents rather than obtaining an optimal solution of a problem. Note that the result of Turing Learning might not be optimal or efficient. This thesis is to introduce Nash Memory to Turing Learning and examine efficiency improvement if any. For the purpose of comparison, we take the aggregation game as the test case.

3.5 Aggregation Game

In this thesis, we study the aggregation game in detail [20]. The game is a typical problem in swarm robot system where the mission of robots is to aggregate into a cluster in an environment free of obstacles [10]. In this aggregation experiment, we use the simulation platform Enki[4] (see Section A.1 for details). Enki has a built-in 2-D model of the e-puck robot. An agent in simulation is represented as a epuck robot of a disk of diameter 7.0 cm with mass 150 g. The speed of each wheel can be set independently (see Section A.1.1 for the complete specification). More specifically, the agents follows a reactive control architecture. The motion of each agent solely depends on the state of its line-of-sight sensor. Each possible sensor state is mapped to a predefined specification of the control of motors. In this thesis, we deal with only the sensor at the front. When the agent is free from obstacles along sight, the sensor reads 0 ($I = 0$) and the velocity of the left wheel is v_l and that of the right is v_r. When the sensor reads 1 ($I = 1$), the velocity values are v'_l and v'_r respectively. This reactive behavior of an agent depends solely on these parameters. For simplicity of representation, we collect all the parameters as a vector $p = \{v_l, v_r, v'_l, v'_r\}$ with $v \in [-1, 1], v \in p$. The marginal values correspond to the maximal values of the wheel rolling backward and forward respectively. In this thesis, we follow the setting of [20] and assume that the replica has the same differential drive

[4] `https://github.com/enki-community/enki`

Figure 3.4: The aggregation behavior of 50 agents in simulation [19].



and the line-of-sight sensor. The system identification task is therefore to infer the control parameters in p. This setting makes it possible to objectively measure the quality of the obtained models in the post-evaluation analysis. For a replica following $q = \{u_l, u_r, u'_l, u'_r\}$, the error is measured as follows:

For each parameter p_i and q_i ($i \in \{1,\ldots,N\}$, $N = |p| = |q|$), the Absolute Error (AE) is:

$$AE_i = |p_i - q_i|$$

The Mean Absolute Error (MAE) over all parameters is:

$$MAE = 1/N * \sum_{i=1}^{N} AE_i$$

All simulations are conducted in an unbounded area where there are A real agents and R replica agents. The initial position of a robot is uniformly distributed within an area of $((A+R)*Z)cm^2$ where $Z = 10,000$ (average area per individual). In this thesis, we follow the setting proposed by [10] and take $p = \{-0.7, -1.01.0, -1.0\}$ to achieve this aggregation behavior [5].

[5] This setting would lead to collisions.

4. Nash Memories

4.1 Nash Memory

4.1.1 Memory Methods and Solution Concepts

In coevolution, each player manages a population of candidate solutions. Using the "methods of fitness assignment" some better fit ones are "remembered" and remain in the population while the rest are "forgotten" before the next generation. The problem of designing a heuristic "memory mechanism" is to prevent forgetting good candidate solutions with certain traits. The contribution of a trait to the fitness may be highly contingent upon the context of evaluation. Even when candidate solutions with/without certain traits are equally fit, the candidates are at risk of drifting due to sampling error in the population dynamic leading to the lose of traits [8]. A *solution concept* specifies precisely a subset of candidate solutions qualify as optimal solutions to be kept in the population [25]. This is a more general definition of "methods of fitness assignment" introduced in Section 3.1.2. The study of solution concepts is important for evolutionary mechanisms that deal with candidate solutions with many traits while only maintain a limit amount of solutions in the memory [9]. As we described in Section 3.2, the ING game is a coevolutionary domain that is permeated by intransitive cycles. For a coevolutionary domain where a strategy consists of n integers, there are n traits to be "watched" by the solution concept.

Most solution concepts in the literature are instances of a general Best-of-Generation (BOG) model where the most fit individuals over the past few generations are retained in the memory. In

contrast to this approach, Stanley and Miikkulainen proposed *dominance tournament* (DT) [29]. The principle is to add the most fit individual of the generation only if it beats all the individuals in the memory. This prevents the case of intransitive relations as those of the ING game for example.

4.1.2 Introduction to Nash Memory

Nash Memory [8] is a solution concept introduced by Ficici and Pollack as a memory mechanism for coevolution. It was designed to better deal with cases where one or more previously acquired traits are only to be needed later in a coevolutionary process. This memory mechanism maintains two set of solutions and uses a game-theoretical method for the selection of candidates to better balance between different traits in coevolution. Before introducing the memory mechanism in detail, some concepts need to be clarified:

Support $Sup(s)$ The support of a mixed strategy s is the set of pure strategies s_i with non-zero probability: $Sup(s) = \{s_i | Pr(s_i|s) > 0\}$. For a set of mixed strategies, we define $Sup(S) = \bigcup_i Sup(s_i), s_i \in S$ [1].

Security set $Sec(s)$ The security set of strategy of s is a set of pure strategies against which s has an expected non-negative payoff. In other words, $Sec(s) = \{s_i | E(s,s_i) \geq 0\}$. Similarly, $Sec(S) = \bigcap_i Sec(s_i), s_i \in S$ [2].

Domination A strategy s_a dominates a strategy s_b if for any strategy s, $E(s_a,s) \geq E(s_b,s)$ and there exists a strategy s' such that $E(s_a,s') > E(s_b,s')$.

Support-secure If the security set contains the support set, then the strategy is *support-secure*.

Most population management in coevolutionary computation maintain a population of candidate solutions according to some explicit organizing principle discovered over generations. In contrast, Nash Memory uses an implicit measurement [8] using pairs of mixed strategies retrieved from Nash equilibria. These pairs are recommendations to the players on which action to take. The following (non-zero-sum) example gives a taste of this approach. The game is an extended version of the battle-of-sex game described in Section 2.2.3. Now there are four options for the man: a) watch a football match, b) watch an opera, c) prepare for an exam and d) write a thesis.

$$M_{man} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \\ 1 & 2 \\ 8 & -1 \end{bmatrix}$$

[1] The support of a strategy s was named $C(s)$ in the original paper [8].

[2] The security set of a strategy was named $S(s)$ in the original paper [8].

$$M_{woman} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \\ -1 & -6 \\ -8 & 4 \end{bmatrix}$$

Classical explicit methods would calculate the fitness of each strategy by summing the payoff uniformly. That is, the fitness values of the four strategies are 3, 2, 3, 7 respectively. However, Nash Memory gives a different solution concept. It computes the Nash equilibria first:

1. The man plays b) and c) with a probability of 0.625 and 0.375 respectively while the woman plays b).
2. The man plays c) and d) with a probability of 0.705 and 0.294 respectively while the woman goes to the opera with a chance of 0.7.
3. Both go to the opera (i.e. the man plays strategy b).

Imagine the man has to remove an option for next week (i.e. forget a strategy in the memory). According to the fitness using explicit method, he should forget about going to the opera next week. However, a) is not participating in any Nash equilibria, thus using the approach of Nash memory, the man will not consider watching a football match next week instead.

4.1.3 The Operation of Nash Memory

The Nash memory mechanism consists of two mutually exclusive sets of pure strategies: $\mathscr{N}$ and $\mathscr{M}$ [8]. $\mathscr{N}$ is defined as the support of the Nash strategy s_N that is at least secure against the elements of $\mathscr{N}$ and $\mathscr{M}$ ($Sec(\int_{\mathscr{N}}) \supseteq \mathscr{N} \cup \mathscr{M}$). The objective of this set is to represent a mixed strategy that is secure against the candidate solutions discovered thus far. The security set is expected to increase monotonically as the search progresses, thereby forming a better and better approximation of strategies for the game. On the other hand, $\mathscr{M}$ plays the role of a memory or an accumulator. It contains pure strategies that are not currently in $\mathscr{N}$, but were in the past. They may be in $\mathscr{N}$ again in the future. By the definition as in [8], $\mathscr{N}$ is unbounded in size and $\mathscr{M}$ is defined as a finite set. The size limit of $\mathscr{M}$ is c, which is known as the capacity of the memory. For simplicity, in this section we assume there is only one Nash equilibrium.

Initialization

Both $\mathscr{N}$ and $\mathscr{M}$ are initialized as an empty set. Let the first set $\mathscr{T}$ to be delivered by the searching heuristics [3]. We initialize $\mathscr{N}$ so that $Sup(\mathscr{N}) \subseteq T$ and $\mathscr{T} \subseteq Sec(\mathscr{N})$.

[3] It was named $\mathscr{Q}$ in Ficici's paper.

Data: The size limit of M, b_m
Result: The best strategy of at the end of iteration
Initialize $\mathscr{N}$ and $\mathscr{M}$ as described in Section 4.2. **while** *termination condition is not satisfied* **do**
  Obtain a set of new strategies $\mathscr{T}$;
  Evaluate strategies in $\mathscr{T}$ against the current Nash equilibria and store the winner $\mathscr{W}$;
  Obtain $U = \mathscr{N} \cup \mathscr{M} \cup \mathscr{W}$;
  Evaluate the candidate strategies in $\mathscr{U}$ and obtain the payoff values;
  Compute the Nash equilibria and update $\mathscr{N}$;
  Update $\mathscr{M}$ as described in Section 4.1.3 using b_m.
end
return the best fit pure strategies in $\mathscr{N}$.

Algorithm 4: Nash Memory.

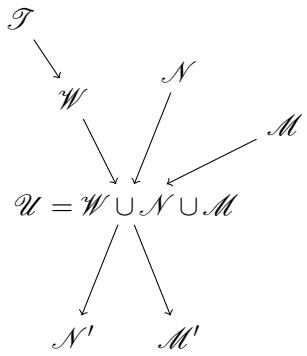


Figure 4.1: The Updating of $\mathscr{N}$ and $\mathscr{M}$ in Nash Memory.

Updating $\mathscr{N}$ and $\mathscr{M}$

We compute a set $\mathscr{W} = \{t \in \mathscr{T} | E(t, s_N) > 0\}$ as the winners against the current best strategy s_N. Using Linear Programming, we can obtain a new Nash strategy s'_N out of the set $\mathscr{U} = \mathscr{W} \cup \mathscr{N} \cup \mathscr{M}$[4]. Next, $\mathscr{N}' := C(s'_N)$ and $\mathscr{M}' := \mathscr{U} - \mathscr{N}'$. This process guarantees that $\mathscr{W} \subseteq Sec(\mathscr{N})$. Figure 4.1 gives an illustration of this process. Note that the updated $\mathscr{M}'$ may have a capacity larger than b_m. A number of policies can be used to reduce the size of $\mathscr{M}'$:

- remove those with the least fitness value calculated in the classical way.
- remove those that participated the least over the past generations.
- remove items at random from M then those released from $\mathscr{N}$.

[4]To verify that a mixed strategy is Nash, Ficici, etc. proposed a simple testing method [8]. In this project, we use external game solvers so no verification is necessary.

Termination

The algorithm terminates after a fixed number of iterations. Other terminating conditions may be adopted. The best fit pure strategies are returned.

4.1.4 Nash Memory in Application

Intuitively, the $\mathscr{N}$ behave like a defending army and defend the attack from the newly discovered strategies $\mathscr{T}$. Those who survive form the set $\mathscr{W}$. However, obtaining a survivor can be hard in some cases. It is possible that multiple genes have to evolve to a certain extend at the same time to obtain a pure strategy, and thus requires the evolution of multiple generations. A way to deal with this is to evolve a few generations against $\mathscr{N}$ and then update $\mathscr{N}$ using such a winner (if any). The Nash Memory mechanism has been examined using the ING game to outperform the Best-of-Generation and Dominance Tournament [8] where the coevolutionary process was transformed to an evolutionary domain. There they introduced this concept of *epochs* to capture the searching heuristic described above. One epoch is 30 generations. At each epoch, only 10 best fit strategies are selected to generate 90 new strategies as offspring.

Nash Memory guarantees monotonicity approaching to convergence [8]. It is a memory mechanism designed to avoid being trapped in an intransitive loop. However, it was set to form a zero-sum and symmetric game each time updating $\mathscr{N}$ and $\mathscr{M}$. As we have discussed in Section 2.2.3 and Section 2.2.4, many games are either asymmetric or not necessarily zero-sum. In the following sections, we explore possible ways to apply Nash Memory to such cases.

4.2 Parallel Nash Memory

Nash Memory guarantees monotonicity with respect to a given solution concept, but is limited to the use of Nash equilibria of symmetric games. Many problems of practical interest are asymmetric (see Section 2.2.3 for concepts and detailed discussions). Oliehoek, et al. extended Nash Memory and introduced *Parallel Nash Memory* [25]. Any coevolutionary algorithm can be combined with it and guarantee monotonic convergence [5]. We introduce $\mathscr{N}_1$, $\mathscr{M}_1$, and $\mathscr{T}_1$ for player P_1 and $\mathscr{N}_2$, $\mathscr{M}_2$, and $\mathscr{T}_2$ for player P_2. The algorithm works similar as that of the Ficic's Nash Memory as introduce above:

[5] In case of confusion, we refer to the previous memory mechanism as Ficici's Nash memory. We call all memory mechanisms extending Ficici's work as Nash memory for the sake of simple reference.

Initialization

$\mathscr{N}_1$, $\mathscr{M}_1$, $\mathscr{N}_2$ and $\mathscr{M}_2$ are initialized as empty sets. Let the first sets $\mathscr{T}_1$ and $\mathscr{T}_2$ to be delivered by the searching heuristics. We choose $\mathscr{N}_1$ and $\mathscr{N}_2$ so that $Sup(\mathscr{N}_1) \subseteq T_1$, $\mathscr{T}_1 \subseteq Sec(\mathscr{N}_1)$, and $Sup(\mathscr{N}_2) \subseteq T_2$, $\mathscr{T}_2 \subseteq Sec(\mathscr{N}_2)$.

Updating $\mathscr{N}$ and $\mathscr{M}$

In zero-sum games, both players are able to provide a security level payoff and the sum should be zero. Assume that the search heuristic delivers a single test strategy for both players. We can test the compound strategy $t = (t_1, t_2)$ against the compound Nash strategy $s = (s_1, s_2)$ as:

$$E(t,s) = E(t_1,s_2) + E(t_2,s_1)$$

In the original paper, when $E(t,s) > 0$, the Nash strategy s is not secure against t and therefore should be updated [25]. However, the following example shows that it is not the case.

Again, we modify the Battle of Sex game. This time, both of them find it an option to go swimming. The payoff matrices are now:

$$M_{man} = \begin{bmatrix} 3 & 0 & 2 \\ 0 & 2 & -3 \\ 2 & 1 & 0 \end{bmatrix}$$

$$M_{woman} = \begin{bmatrix} -3 & 0 & -2 \\ 0 & 2 & 3 \\ -2 & 2 & 0 \end{bmatrix}$$

Before discovering the third option, the Nash equilibrium is that the man watches football match and the woman listens to the opera; both with payoff zero. Evaluating the new option against the current Nash, we have $E(t_1,s_2) = 1$ and $E(t_2,s_1) = -2$ and that makes $E(t,s) = -1 < 0$. However, there is a need to update the Nash equilibria since the new equilibrium is now $(1/3, 0, 2/3)$ for the man with a payoff of $2/3$ and $(0, 2/3, 1/3)$ for the woman. This might be a minor error in the paper [6]. We update the Nash when either $E(t_1,s_2) > 0$ or $E(t_2,s_1) > 0$.

Similarly, for non-zero-sum games, the situation is as follows. When a strategy is evaluated against the current Nash equilibria, as long as it gets the same or better payoff, it should be in the support of Nash in the next generation. For example, the man can now discover a new strategy to play, with which, the player can tie the payoff of a Nash. Thus, the strategy has to participate in the Nash forming a fourth Nash equilibrium. However, since it contributes to only one Nash equilibrium,

[6] In the paper, $\mathscr{T}$ was calculated using best-response heuristic so we always have both $E(t_1,s_2) > 0$ and $E(t_2,s_1) > 0$. However, this is not the case in general.

in case the man has to remove a strategy, it is logical to remove this newly discovered one. Note that the corresponding payoff of the opponent may decrease.

$$M_{man} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \\ 1.5 & 1 \end{bmatrix}$$

$$M_{woman} = \begin{bmatrix} -3 & 0 & -2 \\ 0 & 2 & 3 \\ -2 & 2 & 0 \end{bmatrix}$$

Termination Condition

The algorithm terminates after a given number of generation, after reaching convergence, or satisfies some given conditions.

Data: The size of $\mathscr{M}_1$ and $\mathscr{M}_2$: b_{m1} and b_{m2}
Result: The best pure strategies at the end of iteration
initialize $\mathscr{N}_i$ and $\mathscr{M}_i$ ($i \in \{1,2\}$, same below) as described above.
while *termination condition is not satisfied* **do**
  obtain two sets of new strategies $\mathscr{T}_1$ and $\mathscr{T}_2$;
  evaluate strategies in $\mathscr{T}_i$ against the current Nash equilibria of P_{3-i} and store the winner $\mathscr{W}_i$;
  get $U_i = \mathscr{N}_i \cup \mathscr{M}_i \cup \mathscr{W}_i$;
  evaluate the candidate strategies in $\mathscr{U}_i$ against $\mathscr{U}_{3-i}$ and obtain the payoff values;
  compute the Nash equilibria and update $\mathscr{N}_i$;
  update $\mathscr{M}_i$ as described in Section 4.1.3 and reduce size using b_{m1} and b_{m2};
end
return best fit pure strategies in $\mathscr{N}_1$ and $\mathscr{N}_2$.

Algorithm 5: Parallel Nash Memory.

Monotonic convergence is guaranteed for Parallel Nash Memory. In other words, we have $\mathscr{U}_1 \subseteq Sec(\mathscr{N}_1)$ and $\mathscr{U}_2 \subseteq Sec(\mathscr{N}_2)$ at each iteration. The algorithm has been evaluated using the 8-card poker game with best response search heuristic [25].

4.3 Universal Nash Memory

4.3.1 Introduction to Universal Nash Memory

So far we have encountered different games. Table 4.1 shows that games vary on many aspects. This raises the need of a universal solution concept for all games and coevolution in general. To

do so, we first define a memory mechanism for bimatrix games: *Universal Nash Memory* (UNM in short). In this section, we present the algorithmic design of UNM and evaluation using the ING game. Further extension and integration with Turing Learning will be presented in Chapter 5. To guarantee monotonic convergence, we need to make sure that $\mathscr{N}_i \cup \mathscr{M}_i \subseteq Sec(\mathscr{N}_i)$ ($i \in \{1,2\}$).

Games / Features	ING	PNG, RNG	Aggregation Game
Representation	integer vectors	vector of bounded real numbers	vector of bounded real numbers
Reproduction	asexual	bisexual	bisexual
Evaluation function	simple direct calculation	simple direct calculation	based on simulation and classification results
Termination	both players converge/after some epochs	after some epochs	after some epochs
Symmetry	symmetric	symmetric	assymmetric
Sum	zero-sum	non-zero-sum	non-zero-sum

Table 4.1: A comparison of coevolutionary games.

Data: The size references r_1 and r_2
Result: The best pure strategies at the end of iteration
initialize $\mathscr{N}_i$ and $\mathscr{M}_i$ ($i \in \{1,2\}$, same below) as PNM.;
while *termination condition is not satisfied* **do**
  obtain two sets of new strategies $\mathscr{T}_1$ and $\mathscr{T}_2$;
  evaluate strategies in $\mathscr{T}_i$ against the current Nash equilibria of P_{1-i} and store (a selection of) the winner $\mathscr{W}_i$;
  get $\mathscr{U}_i = \mathscr{N}_i \cup \mathscr{M}_i \cup \mathscr{W}_i$;
  evaluate the candidate strategies in $\mathscr{U}_i$ against $\mathscr{U}_{1-i}$ and obtain the payoff matrices Ξ_1 and Ξ_2;
  compute the Nash equilibria using Ξ_1 and Ξ_2;
  obtain a new $\mathscr{N}_i'$ and $\mathscr{M}_i'$ and reduce the size according to the size refereces r_i.
end
return best fit pure strategies in $\mathscr{N}_1$ and $\mathscr{N}_2$.

Algorithm 6: Universal Nash Memory.

4.3.2 Towards Universal Nash Memory in Application

The current design of Turing Learning employs a simple coevolutionary algorithm. The existing implementations of Turing Learning are hard-coded proof-of-concept test cases. Each player manage a population of candidate solutions and update the population using only one criteria. Nash memory

was designed to better deal with cases where one or more previously acquired traits are only to be needed later in a coevolutionary process. This memory mechanism reports better balance between different traits in coevolution and guarantees monotonicity. This may help increase the learning efficiency of agent behavior. This project experiments the integration of Universal Nash Memory with Turing Learning and analyze the efficiency of learning of agent behaviors.

From Table 4.1, we see that Turing Learning differs from other games in many ways. The two players in Turing Learning are different in setting. The number of genes may differ significantly. Simulation results may lead to inaccuracy in fitness. It helps to take a closer look at UNM before applying to different coevolutionary games.

- A strategy of a classifier is typically a description of a neural network. Thus, the number of parameters could be huge. In comparison, the number of genes for a model could be significantly less.
- When searching strategies with a large number of parameters, it may take many evolutionary steps to tune all the parameters to beat the current Nash equilibria. In other words, it is more likely to have no winner for the player maintaining strategies with a large number of parameters.
- Different from zero-sum problems, bimatrix games are much more complex and harder to solve. For the sake of efficiency, we would have to consider reduce the size of payoff matrices.
- The size of the game depends on $\mathscr{U}_i$, which is a sum of $|\mathscr{N}_i|$, $|\mathscr{M}_i|$ and $|\mathscr{W}_i|$ ($i \in \{1,2\}$). Careful balance between the three sets can be tricky when the size reference r_i is small.
- There could be cases where there is no winner, or where there are too many winners.
- Balancing between the evolution of players is also an important issue.

It is worth to consider small test cases before fixing an algorithm with limitation on the size of $\mathscr{U}$ for Turing Learning. This section presents an attempt integrating UNM with the ING games. The results and analysis are presented in Section 4.4.1. With this experience, UNM is carefully tested on non-zero sum cases: the PNG and RNG games. Finally we integrate UNM with Turing Learning. Implementation and evaluation details are in Chapter 5.

4.4 Evaluation

As listed in Table 4.1, the properties of different games may vary significantly. Before integrating UNM with Turing Learning, it would help to know some properties of UNM and obtain some experience. The following are some test cases on the ING game. A comparative study of Turing Learning with/without UNM is to be presented in Chapter 5 in detail. All the experiments were conducted on a 64-bit machine with a CPU of AMD A10-7870K Radeon R7 (12 Compute Cores 4C+8G $\times$ 4) and a memory of 6.8GB running Ubuntu 16.04.

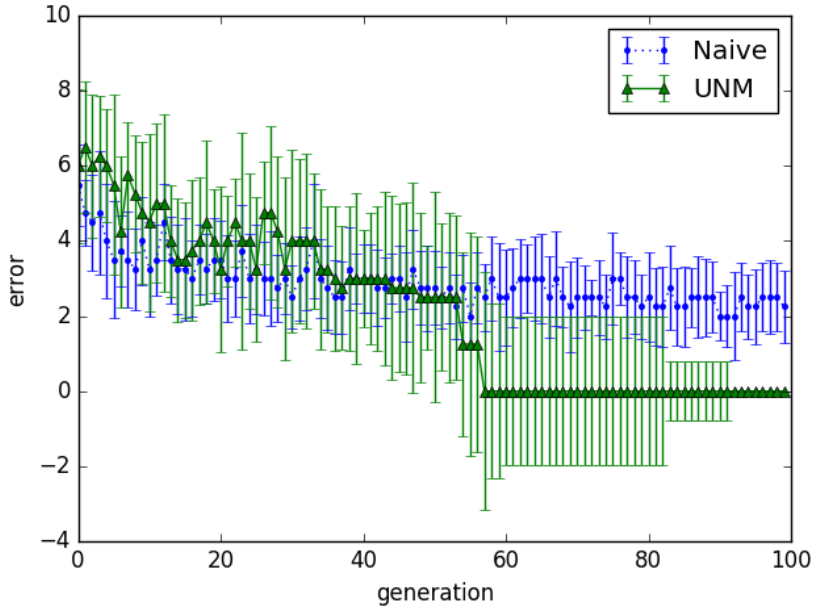


Figure 4.2: Convergence analysis of the ING game with/without UNM.

4.4.1 ING Game

The ING game is a very good test case where the algorithm deals with intransitive relations to reach convergence. Here we take $\varepsilon = 0$ and 4 genes for each strategy, each of which consists of 4 bits. Every example completes 100 generations.

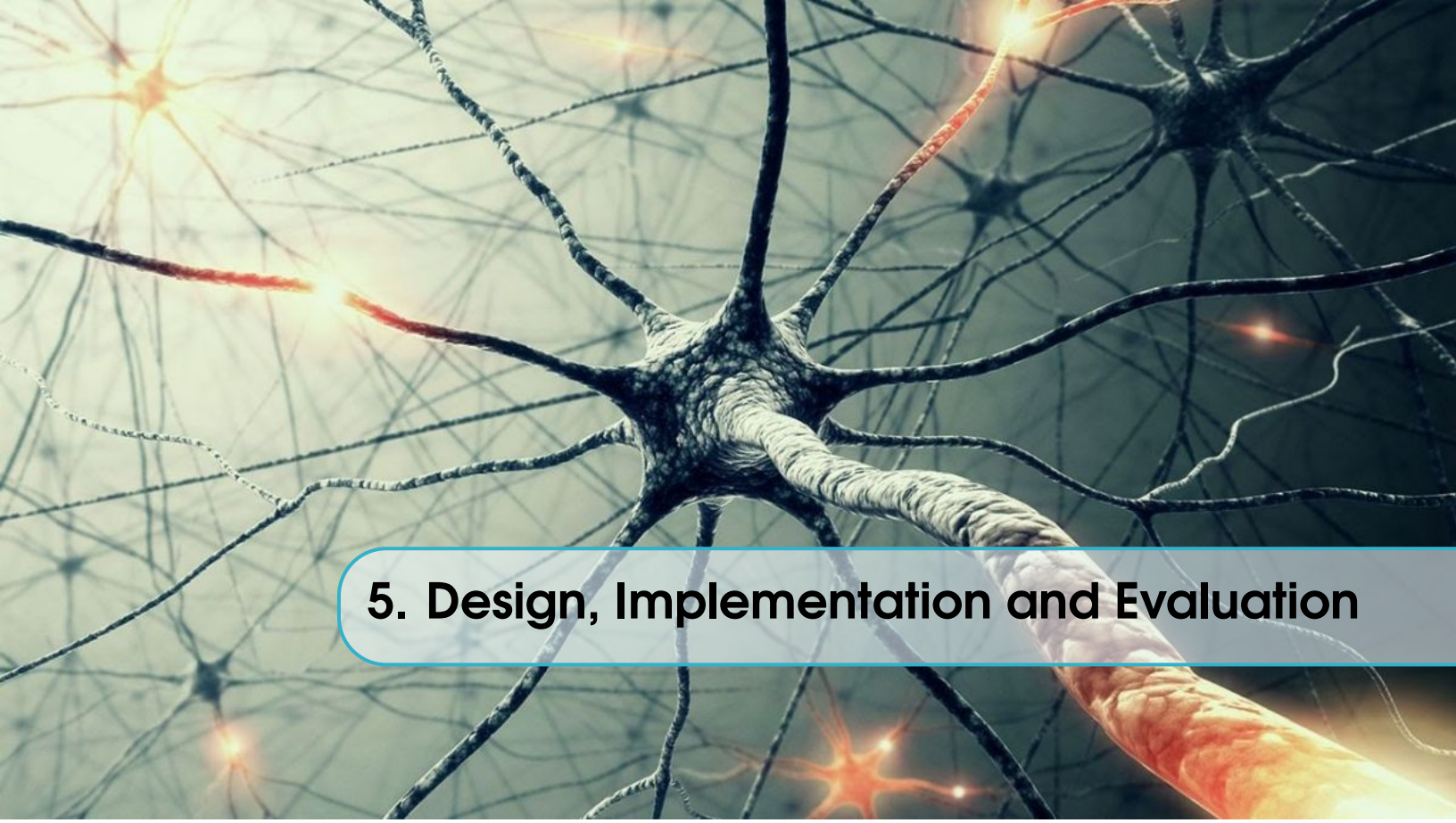
A crucial issue that was not discussed in [8] is the size of $\mathscr{N}$. In the ING game, when $\mathscr{N}$ is free from size bound, the $|\mathscr{N}|$ can easily go beyond 50 (when there are four genes, each of which consists of four bits). Generally speaking, we would expect the size of $\mathscr{N}$ not to be significantly bigger than that of $\mathscr{M}$ and $\mathscr{T}$. However, this is not the case due to complicated intransitive relations (even with an epoch of 30). In this project, we use one of the state-of-the-art bimatrix game solvers, the *lrs* solver (see Appendix A.4 for details) [1]. Although the time to compute each Nash equilibrium varies, the more the equilibria there are, the longer it takes in general. For instance, when the matrices are of size 10×10, it usually takes less than 0.1s to find all the Nash equilibria. When the matrices are of size 15×20, there could be over 1000 Nash equilibria and take several minutes. It may take up to half an hour to compute all the equilibria (over 10,000) when we have matrices of size 30×30.

At each generation, we obtain the $\mathscr{N}_i'$ and $\mathscr{M}_i'$ step by step regarding the size reference r_i. In this test case, we take $e_1 = 1.5$ and $e_2 = 2$.

1. Let $\mathscr{U}_i = \mathscr{N}_i$
2. Add $\mathscr{W}_i$ to $\mathscr{U}_i$ until $|\mathscr{W}_i| = e_1 * r_i$

3. Add $\mathscr{M}_i$ to $\mathscr{U}_i$ until $|\mathscr{W}_i| = e_2 * r_i$
4. For two sets of strategies $\mathscr{U}_0$ and $\mathscr{U}_1$, compute the payoff matrices Ξ_1 and Ξ_2
5. Compute (all) the Nash equilibria and extract $\mathscr{N}_i$ and let $\mathscr{M}_i = \mathscr{U}_i \setminus \mathscr{N}_i$
6. Reduce the size $\mathscr{N}_i$ to r_i and add the discarded ones to the memory $\mathscr{M}_i$
7. Reduce $\mathscr{M}_i$ to the memory size

Figure 4.2 illustrates the result based on 10 runs with/without UNM (with $r_i = 10$). The median and the standard deviation of the Naive coevolution method are plotted in blue. Those using the UNM are in green. It is clear that using UNM makes it much easier to reach convergence (after around 60 generations). An explanation is that, in the Naive approach, a strategy is evaluated against each strategy in the population and thus influencing the fitness of the strategy. However, when using UNM, strategies are simply evaluated against those in the set $\mathscr{N}$, which can be considered the most competitive candidates in the population. Intuitively, this gives the population a more specific direction to evolve towards (i.e., with less "disturb" from the others). However, it is significantly slower using UNM due to external calls to lrs. For 100 generations, it takes less than 5s for the Naive approach while it takes up to 10 minutes for the UNM approach with most time (over 90%) used by lrs. Our evaluation results match the results of [8].



5. Design, Implementation and Evaluation

Turing Learning is a novel system identification method for inferring the behavior of interactive agents. Over the past few years, some test cases have been developed and showed the success of Turing Learning in the learning of collective behavior. However, the source code of none of these projects is available for readers. Even worse, some of these projects are domain-specific and hard-coded. This chapter presents a plug-and-play Turing Learning platform, namely the *TuringLearner*.

5.1 *TuringLearner*: Design and Modelling

The project follows a modular design and an object-oriented modelling. At an abstract level, the *TuringLearner* platform is an instance of coevolutionary manager, which is an abstract class template. A `Player` class is an instance of abstract class templates using meta-programming concepts. Given two players of specific types and the agents under investigation, a manager conduct the coevolution. At each generation, it maintains a simulation platform for players to evaluate their strategies. In case of the use of Nash Memory, the `CoEvoManager` interacts with an external game solver, the default of which is the *lrs* solver. A player manages the evolution of a certain type of strategy following a solution concept: initialize a certain amount of strategies, interact with the coevolution manager and compute fitness explicitly or implicitly, select some parents and generate the offspring. More specifically, there are two types of players in the platform. Instances of the `SimplePlayer` class maintains strategies and their fitness values. A player of `UNMPlayer` class manages two sets of

Table 5.1: Integrated strategies and their players and games

	IntVectorStg	Pct-/RealVectorStg	ElmanStg	EpuckStg
Reproduction	asexual	bisexual	bisexual	bisexual
Players	INGPlayer	PNG-/RNGPlayer	model player	classifier player
Game	ING	PNG & RNG	aggregation game	

strategies: the set $\mathscr{N}$ stores strategies participate in the Nash equilibria while $\mathscr{M}$ serves as a memory of strategies not yet to be forgotten.

5.1.1 Strategies, Players and Games

Strategy

In this project, a strategy is explicitly a vector of genes. For the sake of simplicity in demonstration and evaluation, several strategies comes along with the *TuringLearner* platform. We have `IntVectorStrategy`, `RealVectorStrategy` and `PctVectorStrategy` for the ING, RNG and PNG games as described in Chapter 3. A strategy of type `IntVectorStrategy` is a vector of lists of Boolean values evaluated to an integer number each. The other two consists of genes of real values. The reproduction can be asexual or bisexual. The reproducing process is managed by the corresponding player at the end of each generation. Two other types of strategies come together with the platform are the `ElmanStrategy` and `EpuckStrategy`, which correspond to the Elman Neural Network and the spefication of an epuck robot respectively. It worth addressing that each of these genes is associated with a mutation value (Figure 5.1). Table 5.1 gives a summary of these strategies and their players and the corresponding games.

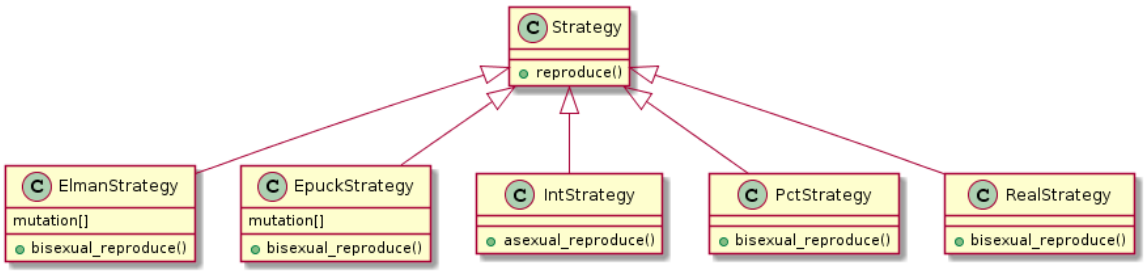


Figure 5.1: Class inheritance of built-in Strategy Classes.

Player

Players are managers of strategies and participate in a coevolutionary process. `UNMPlayer` differs from `SimplePlayer` where there are two sets of strategies $\mathscr{N}$ and $\mathscr{M}$. A player of `UNMPlayer` class manages two sets of strategies: the set $\mathscr{N}$ stores strategies participate in the Nash equilibria

while $\mathscr{M}$ serves as a memory of strategies not yet to be forgotten. `UNMPlayer` also maintain a record of the Nash equilibria of the previous generation. The `Player` class are further divided into four sub-classes depending on their relation with the Generative Adversarial Objects:

1. `SimpleStrategyPlayer`: players that maintain a set of strategies for naive coevolution (e.g., the players of the (naive) ING game).
2. `SimpleGAPlayer`: players that maintain a set of strategies and the corresponding GAOs for naive coevolution (e.g., the players of the (naive) aggregation game).
3. `UNMStrategyPlayer`: players that maintain two sets of strategies for coevolution using UNM (e.g., the players of the ING game using Nash memory).
4. `UNMGAPlayer`: players that maintain two sets of strategies and their corresponding GAOs (e.g., the players of the aggregation game using Nash memory).

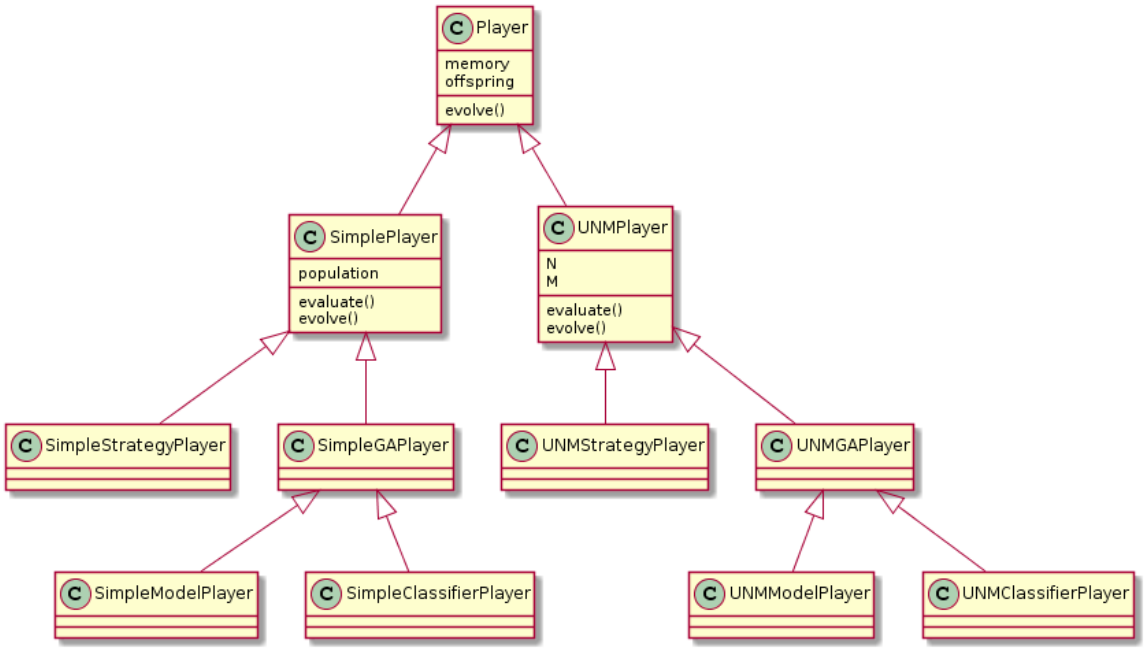


Figure 5.2: Class inheritance of players.

Depending on the solution concept, a model player playing `ElmanStrategy` and the maintain the Elman Neural Networks could be of class `SimpleGAPlyer` or `UNMGAPlayer`.

Coevolution Manager

The coevolution Manager template takes the type of its players as well as their strategies as type parameters. Coevolution Managers implement solution concepts and manage the coevolution of players. Figure 5.4 illustrates the inheritance relations of the aggregation game and the ING game. That of PNG and RNG are similar as ING. A `SimpleCoEvoManager` uses the naive $\lambda + \mu$ algorithm while `UNMCoEvoManager` takes advantage of UNM. A coevolution manager interact with external

game solvers. The default solver is *lrs*. Since this project follows a modular design, other solvers may be integrated. Coevolution managers is also incharge of interacting with the simulation platform and takes GAOs instead of strategies to compute the payoff using the simulation results if specified.

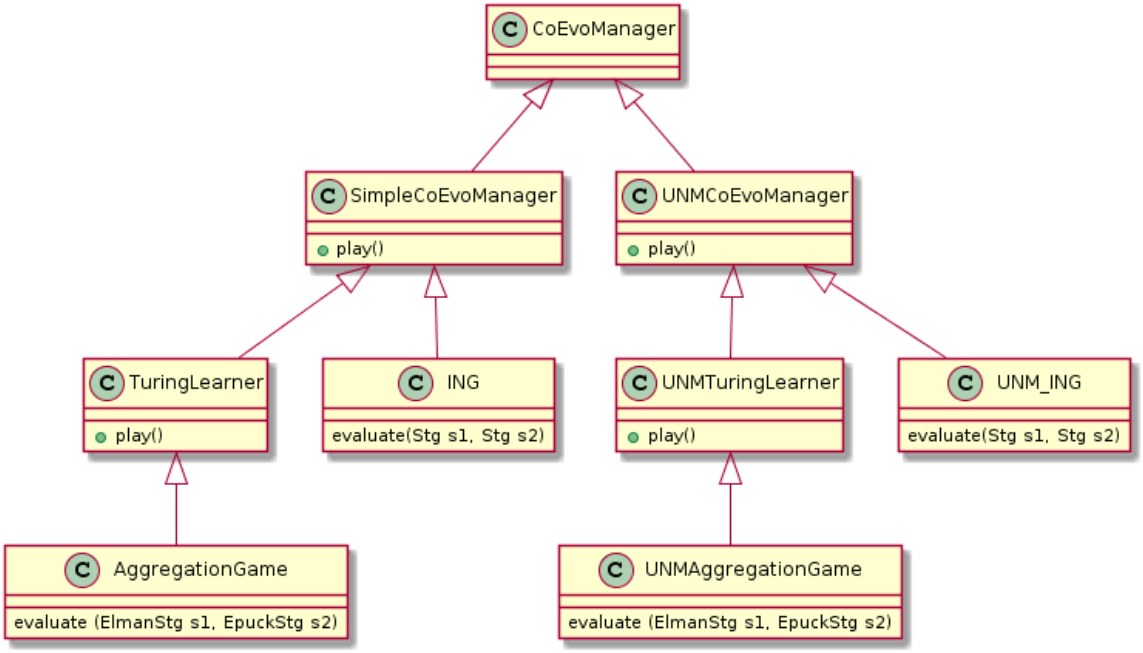


Figure 5.3: Class inheritance of coevolution managers.

5.1.2 Turing Learning and Aggregation Game

An instance of the class `TuringLearner` maintains also a set of real agents under investigation. Real agents are individuals that are invariant in behavior over generations. In the canonical reference of Turing Learning [20], models are at the same conceptual level as agents. This project follows an object-oriented implementation and emphasis the difference. A strategy is a vector of genes and evolve to better fit the game. A GAO is the entity that participate in a simulation. One strategy may generate many GAOs. More specifically, a GAO is a complete representation of an object including its shape, size, parameters, and behaviour. For example, an epuck robot is specified by its weight, its height, the radius, the reaction corresponding to different sensor signals. A classifier is also a GAO. For example, a neural network is determined by the number of layers, the bias, the input/output nodes, the connection between nodes as well as the weights. A corresponding strategy is a much simpler and only encodes the weights for the sake of evolution. In contrast, GAOs are functional entities: an agent can move around and interact, a classifier can distinguish the behaviour of a real agent from that of a replica.

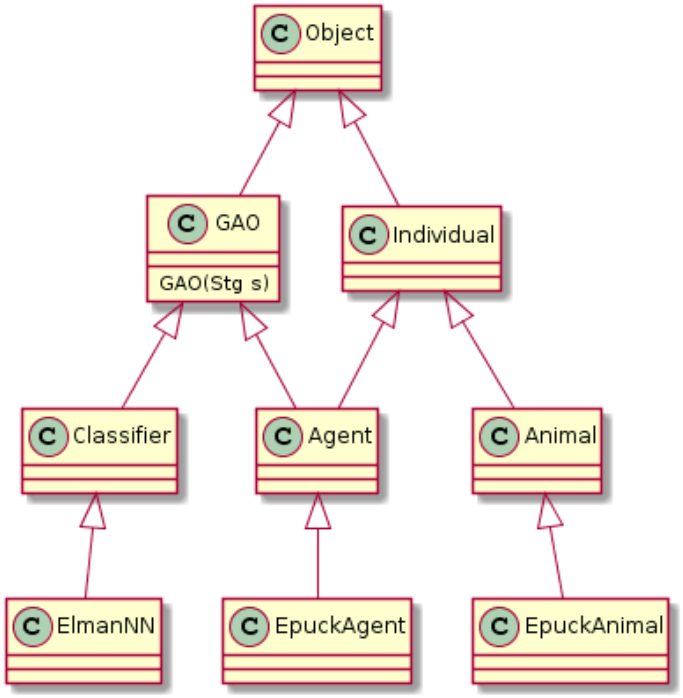


Figure 5.4: Class inheritance of Objects, Agents, Classifiers and Animals.

5.2 Implementation

This *TuringLearner* platform is implemented in C++. All games in this project are implemented in a bottom-up fashion due to the use of metaprogramming: from strategies to players, then to coevolution manager. Metaprogramming enables developers to write programs that falls under the same generic programming paradigm. Together with object-oriented programming, developers can reuse existing implementation of solution concepts and write elegant code.

1. Strategy classes are first implemented. Realize the evolution functions.
2. Implement real agent classes (optional).
3. Specify each Player class with a strategy class as type parameter. Realize the reproduction function and other functions corresponding to a solution concept.
4. Specify the coevolution manager with two player classes (and the real agents). Realize coevolution functions with terminating condition. Implement the interaction with external game solvers. Maintain a simulation platform if needed.

Coevolution and Turing Learning are frameworks rather than concrete algorithms. The strategies, players, agents and how they interact are all specified step by step, level by level. This project uses metaprogramming concepts and allows the reuse of code. For example, a player of `IntVectorStrategy` can easily be transformed to a player of `PctVectorStrategy`. The following is the code in C++:

```
typedef SimpleStrategyPlayer <IntVectorStrategy> UNMINGPlayer;
typedef SimpleStrategyPlayer <PctVectorStrategy> UNMPNGPlayer;
```

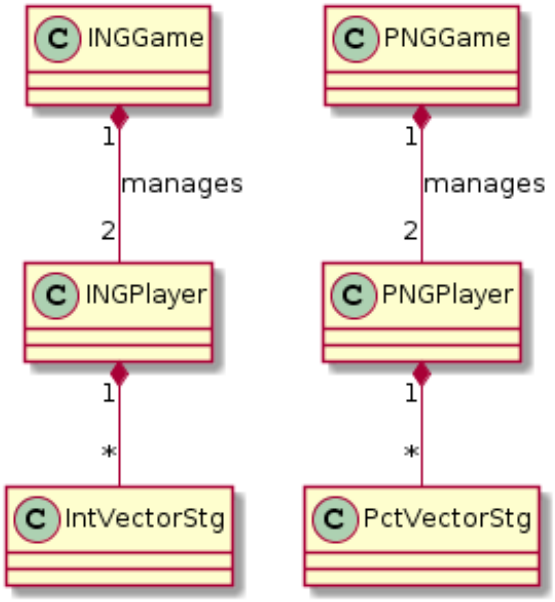


Figure 5.5: The architecture of ING and PNG game manager.

Similarly, a coevolution manager class takes the type of players and their strategies as parameter. Games using the same solution concept and architecture can reuse the code of the coevolution manager (e.g., the ING and PNG game; see Figure 5.5). In the following example, the code of `SimpleCoEvoManager` is reused. In fact, apart from output functions, the only function to be uniquely defined between classes is the `evaluate` function.

```
class ING: public SimpleCoEvoManager <IntVectorStrategy,
    IntVectorStrategy, INGPlayer, INGPlayer>{}
class PNG: public SimpleCoEvoManager <PctVectorStrategy,
    PctVectorStrategy, PNGPlayer, PNGPlayer>{}
```

The aggregation game was re-implemented on our new framework *TuringLearner* in C++. Compared with the three simple test cases, the aggregation game is much more complicated and involves the interaction of different external programs. A Turing Learner manages an Enki simulation world and triggers simulation. At the beginning of each simulation process, the model player fix a strategy and generates R replica agents. The platform first adds A real agents to the Enki world, and then the replica agents (the order is indifferent). Enki then conduct a simulation of 100 time steps. The linear velocity and angular velocity are calculated as described in Section 3.5 and recorded. The classifier player then generates a neural network and performs classification. The fitness values of the strategies are calculated respectively as in Section 3.4.3. The Turing Learner repeats this process until the payoff matrices are complete and then calls the external solver if necessary. At the end of generation, each player evolves the strategies according the fitness. In detail, the project follows a bottom-up fashion:

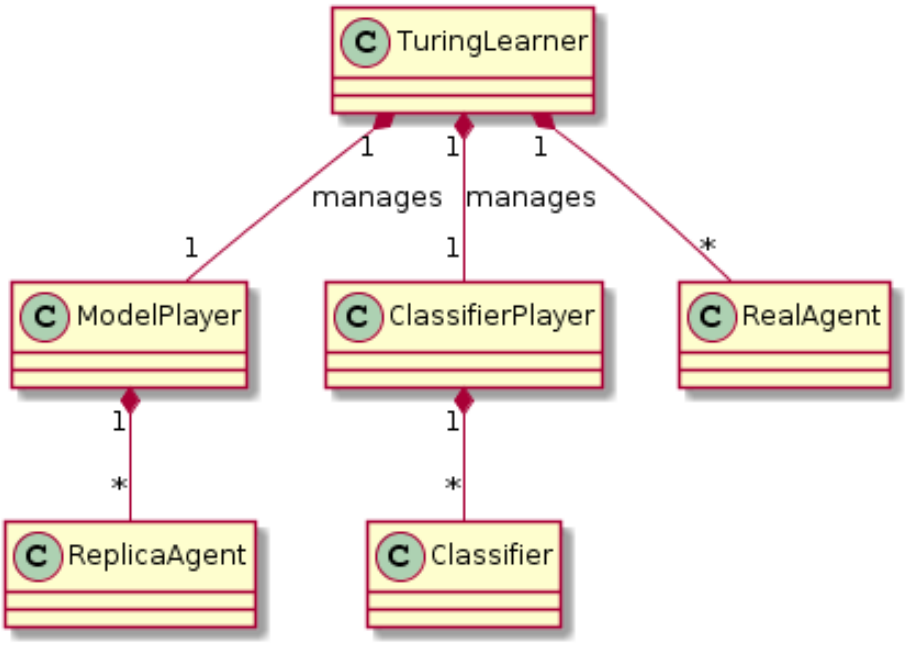


Figure 5.6: The architecture of Turing Learner.

1. Realize the `EpuckStrategy` class.
2. Realize the `EpuckAgent` class with `EpuckStrategy` as a type parameter.
3. Realize the EpuckAnimal class for real agents[1].
4. Generate some replica agents according to strategy and run simulations on Enki. Record the linear and angular velocity of some agents for later.
5. Realize the `ElmanStrategy` class.
6. Realize the `ElmanNN` class with `ElmanStrategy` as a type parameter for neural networks.
7. Take the data from previous steps and perform classification (regardless of the accuracy at this stage).
8. Implement the model player and classifier player regarding solution concepts.
9. Implement Turing Learner classes regarding solution concepts.
10. Implement the interaction with external game solvers.
11. Realize output functions and write scripts for testing.
12. Evaluate regarding differnt parameters.

This project first performs an examination of the hard-coded implementation from [19]. Not all parameters are taken the same as that of the original work. In both implementation, the maximum speed of epuck robot is 12.8cm/s. Different from the original work, in this implementation, we bound the velocity between -1 and 1 (inclusive) so the robots not misbehave totally out of order. In fact, this can be problematic in interaction. We will discuss this in Chapter 6 As for the mutation step, we take a scale parameter $sc = 0.4$. This way, it is less probable that the new value σ_i'' is beyond the range in the second step (see Section 2.4.2 for the complete mutation):

[1] Real agents are temporally named “animals” in implementation to remove confusion from replica agents.

$$\sigma_i'' := sc * \sigma_i'' * exp(\tau' * r + \tau * r_i), i = 1, \ldots, n; \tag{5.1}$$
$$g_i'' := (g_i'' + \sigma_i'' * r_i), i = 1, \ldots, n \tag{5.2}$$

A significant drawback of the original work is taking only $R = 1$ replica in each simulation. We take $R = A = 5$ in our implementation to make the fitness values more accurate and benefit the accuracy of UNM. Another problem is the imbalance of population regarding the size of genes in the original work. There are 4 genes for a strategy of the model player while there are 92 genes for each strategy of the classifier player. In the original work, there are 50 strategies managed by each player and they reproduce 50 offspring each. For the model player, each gene shares $50/4 = 12.5$ offspring. In contrast, for the classifier player, each gene shares at little as $50/92 = 0.54$, over 20 times less than that of the model player. In this project, we let the classifier reproduce 100 offspring instead [2]. Further discussion is included in Section 6.1. In addition, we take $r_0 = 8$ and $r_1 = 10$ to help the update of population of the classifier player.

5.3 Evaluation

5.3.1 The PNG and RNG Game

This project takes the ING game as a cases study for the correctness of implementation (Section 4.4). Similarly, the PNG and RNG games are introduced as test cases for a quantitative measurement on how UNM would speed up the convergence regarding different correlated populations. We introduced ε as a coefficient in Section 3.3 in the evaluation function. The larger ε is, the more attractive the point of an opponent strategy in comparison with the pre-defined attractor (as a point in 2D space) and thus harder to converge to the attractor. We show the difference between a naive coevolutionary process and that using UNM. When $\varepsilon < 1$, the two populations converge and settle at the pre-defined attractors respectively (Figure 5.7). Otherwise the population may settle at a point in between (see Figure 5.8). It is clear from the figure that there are significantly less strategies explored in the space when using UNM [3].

We further examine the impact of ε on coevolution for PNG and RNG games respectively. We take two points in 4D space: $p_1 = \{0.2, 0.2, 0.2, 0.2\}$ and $p_2 = \{0.8, 0.8, 0.8, 0.8\}$ as attractors. For the PNG game, Figure 5.11 shows that a larger ε makes the entanglement more significant and thus harder for the population to converge. However Figure 5.12 shows that for RNG game, this impact is not as significant. In both cases, the use of UNM can significantly boost convergence but slow down the computation due to the calls to the external solver. Different from the ING game, there

[2] This is still far from "enough" to be balanced, considering the combination of all the values of each gene.

[3] The four plots are with the same population size. In the plots with UNM, the same strategies are repeatedly plotted over and over again on the same spots.

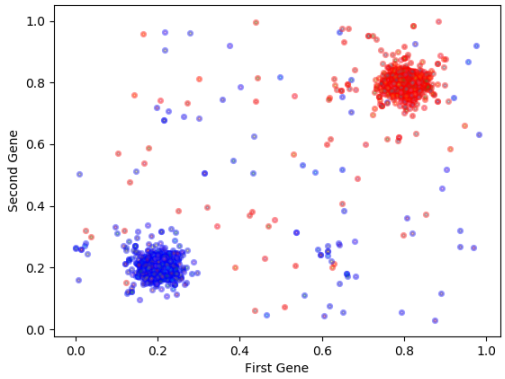


Figure 5.7: RNG with $\varepsilon = 0.2$.

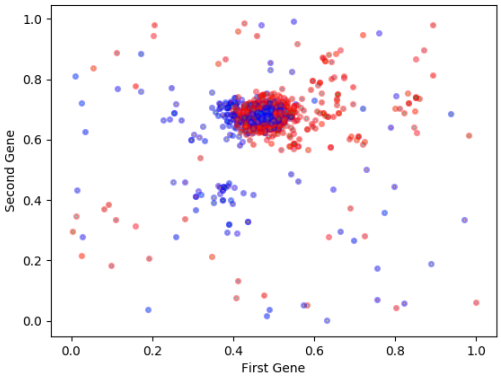


Figure 5.8: RNG with $\varepsilon = 2$.

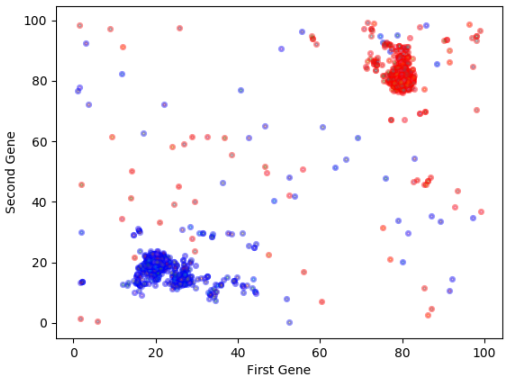


Figure 5.9: RNG (UNM) with $\varepsilon = 0.2$.

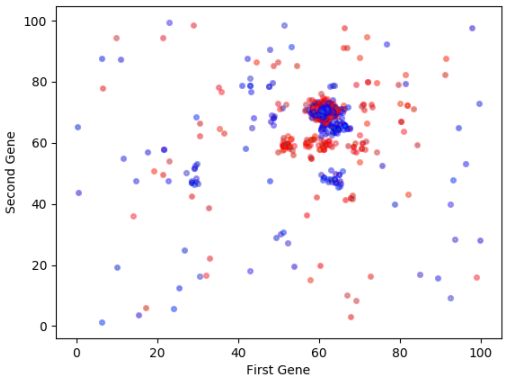
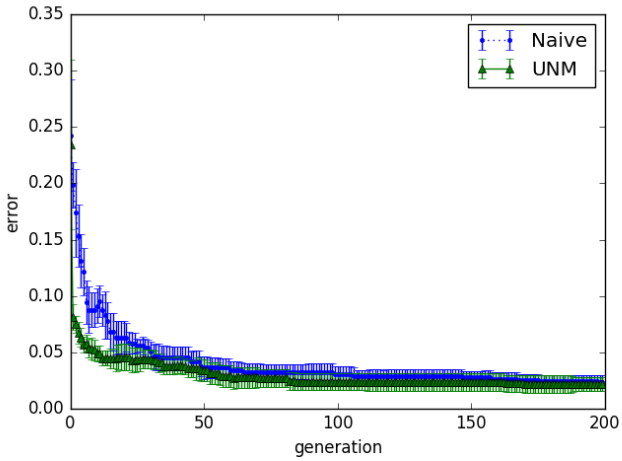
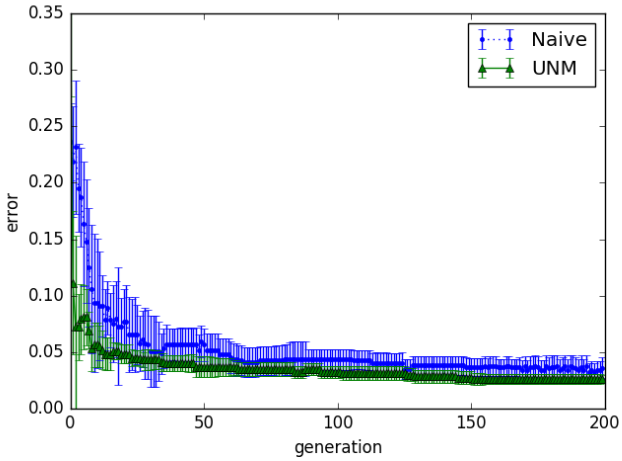


Figure 5.10: RNG (UNM) with $\varepsilon = 2$.

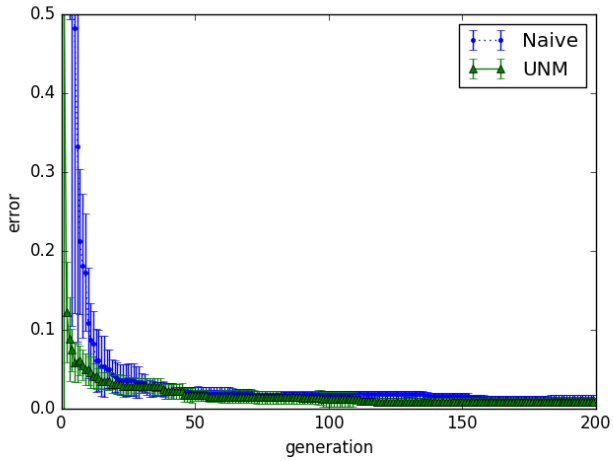


(a) coevolution with $\varepsilon = 0.2$)

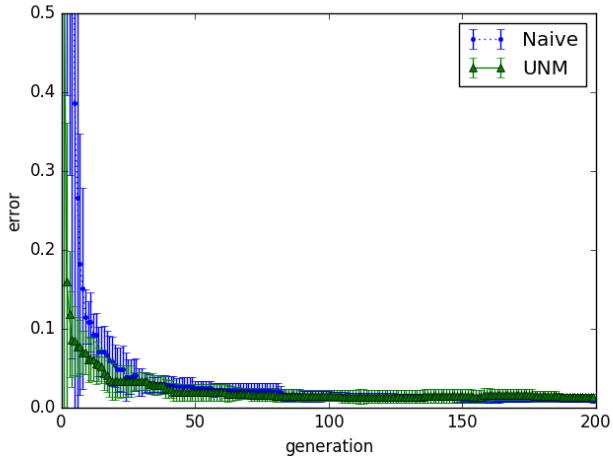


(b) coevolution with $\varepsilon = 0.8$

Figure 5.11: Convergence analysis of the PNG game with/without UNM.



(a) coevolution with $\varepsilon_2 = 0.2$



(b) coevolution with $\varepsilon_2 = 0.8$

Figure 5.12: Convergence analysis of the RNG game with/without UNM.

are less intransitive relations. Before convergence, there are a few Nash equilibria. After reaching convergence, there is usually only one Nash equilibrium. Therefore the external solver returns the result much faster. For the the PNG game, the average time to complete 200 generations is 1.38s in comparison with 18.65s for that using UNM. For the RNG game, the naive approach takes 1.35s on average while the UNM approach takes 17.95s.

5.3.2 Aggregation Game

TuringLearner features a plug-and-play design using metaprogramming and makes the code shorter and more elegant for developers. Moreover, this object-oriented modular framework distributes

different tasks to the corresponding managers to reduce the complexity of debugging. At each generation, we obtain the $\mathscr{N}_i'$ and $\mathscr{M}_i'$ step by step regarding the size reference r_i. From the previous evaluation results, it is clear that the bigger the matrices, the harder to be solved.

We use the definition of Mean Absolute Error (MAE) and define two additional measurements of errors corresponding to the two modes epuck robots have.

$$\text{MAE} = \frac{1}{4}\sum_{i=1}^{4}|v_i - g_i|$$

$$\text{MAE-A} = \frac{1}{2}\sum_{i=1}^{2}|v_i - g_i| = \frac{1}{2}(AE_1 + AE_2)$$

$$\text{MAE-B} = \frac{1}{2}\sum_{i=3}^{4}|v_i - g_i| = \frac{1}{2}(AE_3 + AE_4)$$

MAE-A measures the error when the input $I = 0$ (mode A) and MAE-B measures that with $I = 1$ (mode B). An epuck robot switches to mode B when there are obstacles at the front. We define the *degree of observation* f_A and f_B as frequency values for each mode indicating the probability that the agent operates under the mode. In the aggregation game, the epuck robots are in mode A with a chance of $f_A = 91.2\%$ v.s. $f_B = 8.8\%$ in mode B on average [19].

Balance of Strategies in UNM

Due to the limit of size of matrices, UNM can not guarantees monotonicity in convergence. A careful balance of $\mathscr{N}_i$, $\mathscr{W}_i$ and $\mathscr{M}_i$ for player i at each generation is therefore an important issue. This subsection presents some experiments exploring possible values of the parameters to achieve faster convergence.

At each generation, $\mathscr{U}$ is obtained in the following steps with the parameter e_1, e_2 and e_1' corresponding to the size reference $r_1 = 8$ and $r_2 = 10$:

1. Let $\mathscr{U}_i = \mathscr{N}_i$.
2. *If* $|\mathscr{W}_i| > 0$, add strategies in $\mathscr{W}_i$ to $\mathscr{U}_i$ until $|\mathscr{W}_i| = e_1 * r_i$; add strategies in $\mathscr{M}_i$ to $\mathscr{U}_i$ until $|\mathscr{W}_i| = e_2 * r_i$.
3. *Else*, add strategies in $\mathscr{M}_i$ to $\mathscr{U}_i$ until $|\mathscr{W}_i| = e_1' * r_i$.

At first, we take $e_1 = 1.2$, $e_2 = 1.5$ and $e'_1 = 1.2$. Figure 5.13 plots the values at each generation for 200 generations. The first four plots correspond to the genes $g_1, \ldots, g_4$ for the epuck strategy and the last two illustrate the mean absolute error (MAE-A and MAE-B) over all parameters. The real parameters are $p = \{-0.7, -1.0, 1.0, -1.0\}$ respectively. The values corresponding to the experiment without UNM (i.e., the naive approach) are marked with red dots while the green ones are the values collected using UNM. Due to the small number of winners, the change of population is slow. From the plots, we can see that it is possible that the best strategy for the model player remains the same for consecutive numbers of generations. Another reason is the lack of new and better strategies in the classifier player [4].

In the next experiment, we take more winners. Here we set $e_1 = 1.8$, $e_2 = 2$ and $e'_1 = 1.5$. As illustrated in Figure 5.14 after some iterations, the optimal strategy starts to switch between different local minimums. This experiment shows that taking a large number of winners suffers from overfitting and unstable convergence.

After careful tuning of parameters manually, the most balanced and the best results achieved so far is when we take $e_1 = 1.5$, $e_2 = 2.0$ and $e'_1 = 1.4$. Figure 5.15 shows two example cases with/without UNM. Similarly, the first four plots correspond to the genes and the last two illustrate the mean absolute error (MAE-A and MAE-B) over all parameters.

Figure 5.15 illustrates the convergence of two runs of the aggregation game with and without UNM. It is easy to see that both approaches result in quick convergence of the values on all values. For parameters of mode A, both approaches learn the values at ease. However, the convergence using UNM is slightly not as stable as that of the naive approach. As for mode B, the use of UNM makes it easier to balance between the evolution of two modes. The values converge a little faster than the naive approach. For a statistical examine of the convergence properties, we repeat the experiment 10 times each.

Figure 5.16 illustrates the median and the standard deviation of 10 tests, each with 150 generations. The convergence of MAE-A is not as stable as the naive approach. A possible reason is over-fitting. In the UNM approach, at each generation, the population evolves against only the strategies in $\mathcal{N}$ of the opponent. This may lead to over-fitting and cause unstable convergence. This issue is expected to be removed when setting the size of $|\mathcal{N}|$ to be bigger (see Section 6.1 for further discussion). While using UNM makes the convergence of parameters in mode B slightly faster and more stable. From statistics, we have $f_A = 91.2\%$ v.s. $f_B = 8.8\%$. That explains why the parameters in mode A converges much faster than that of mode B. Overall, Figure 5.17 shows that using UNM makes it easier to reach convergence but not significantly. Finally, Table 5.2 indicates that using UNM can speed up the computation to 35.9% that of the naive approach.

[4] It was observed that in this setting, there are a lot iterations where the is no winner for the classifier player.

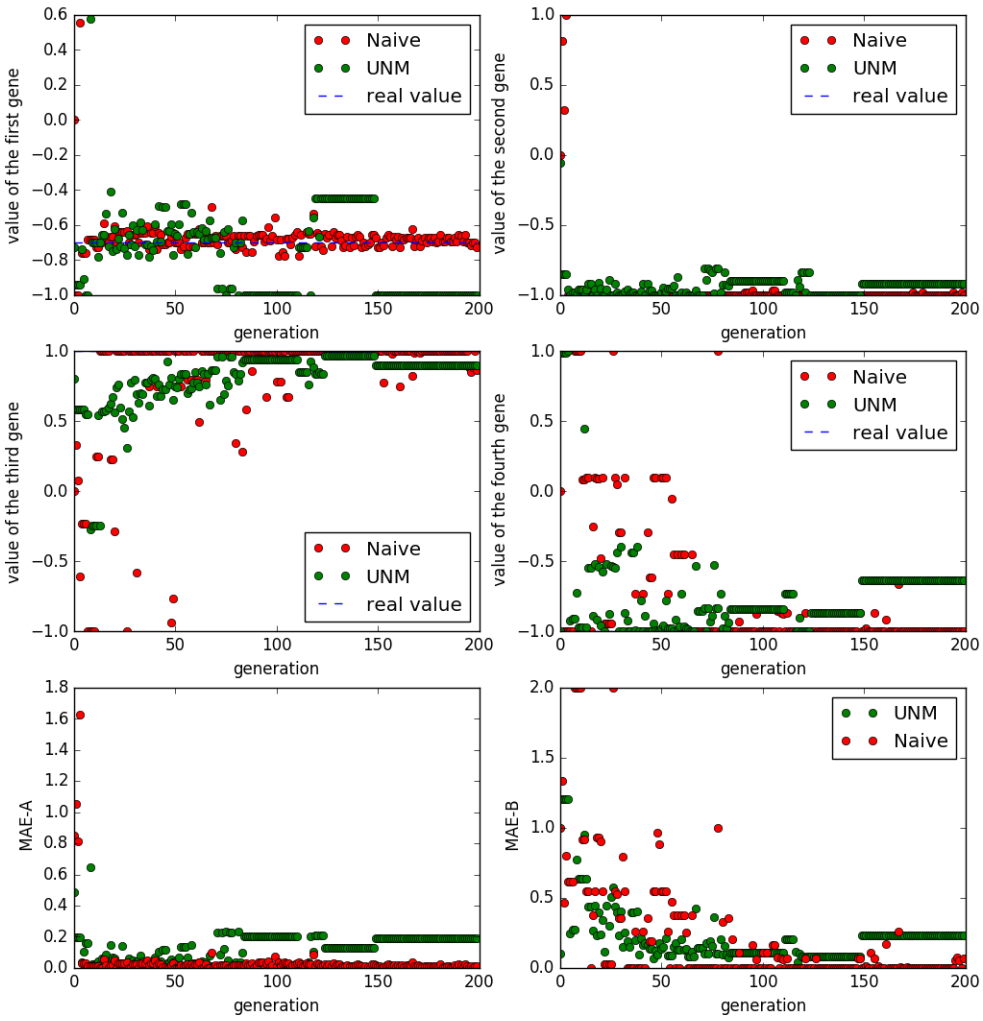


Figure 5.13: Convergence analysis taking a small number of winners.

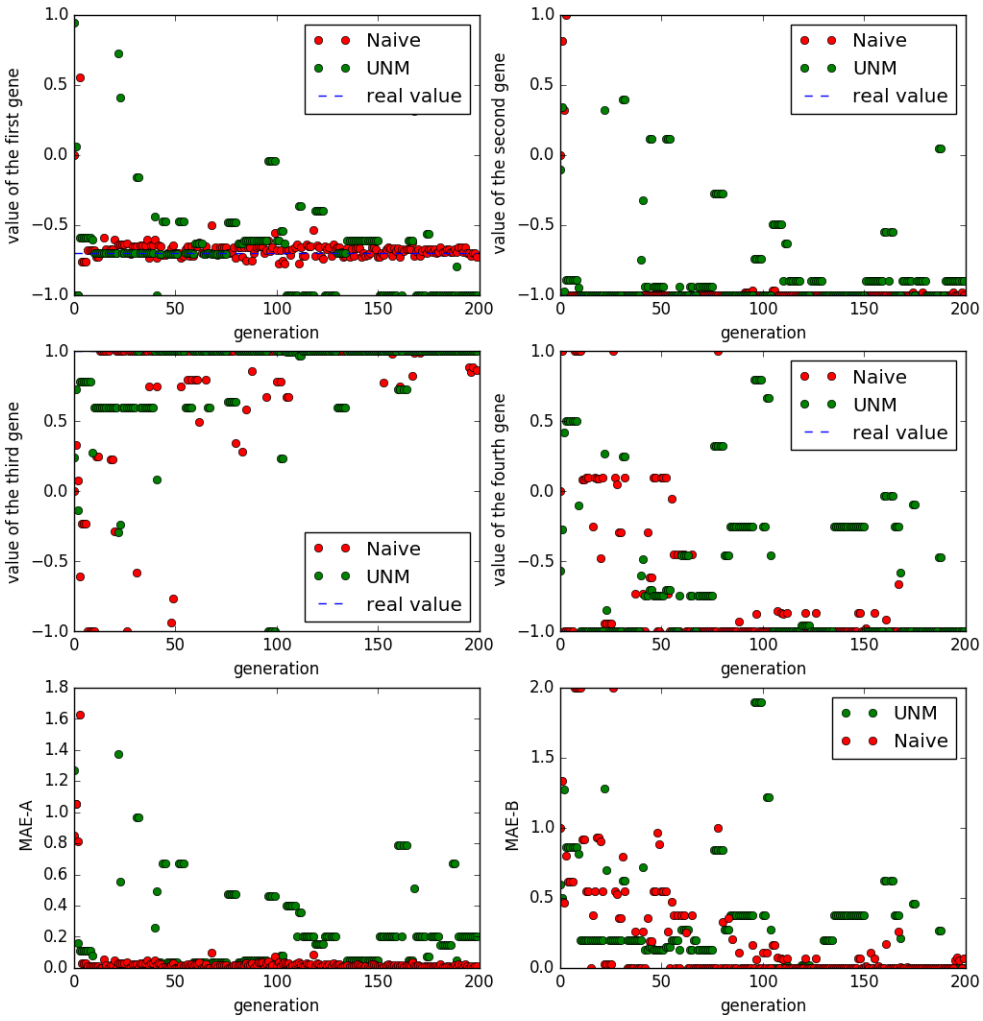


Figure 5.14: Convergence analysis taking a large number of winners.

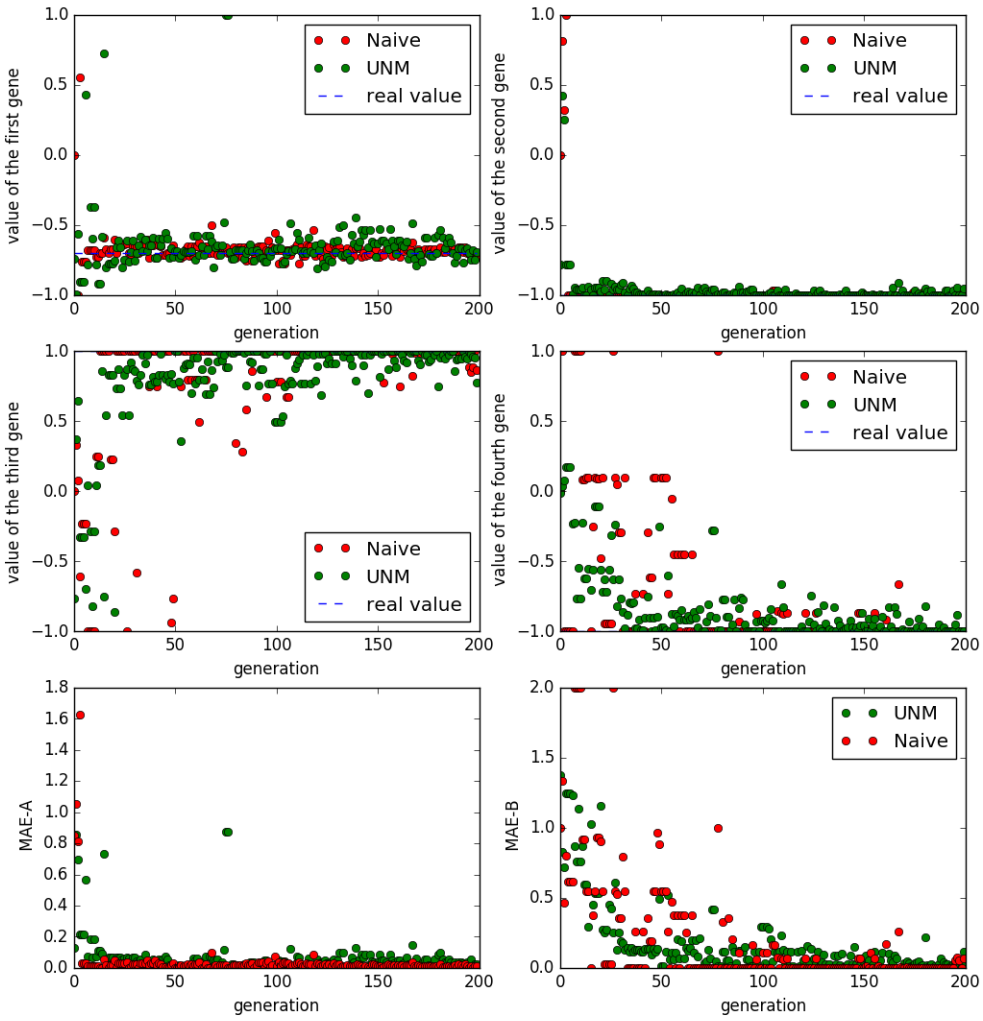


Figure 5.15: Convergence analysis of an example of the aggregation game with and without UNM.

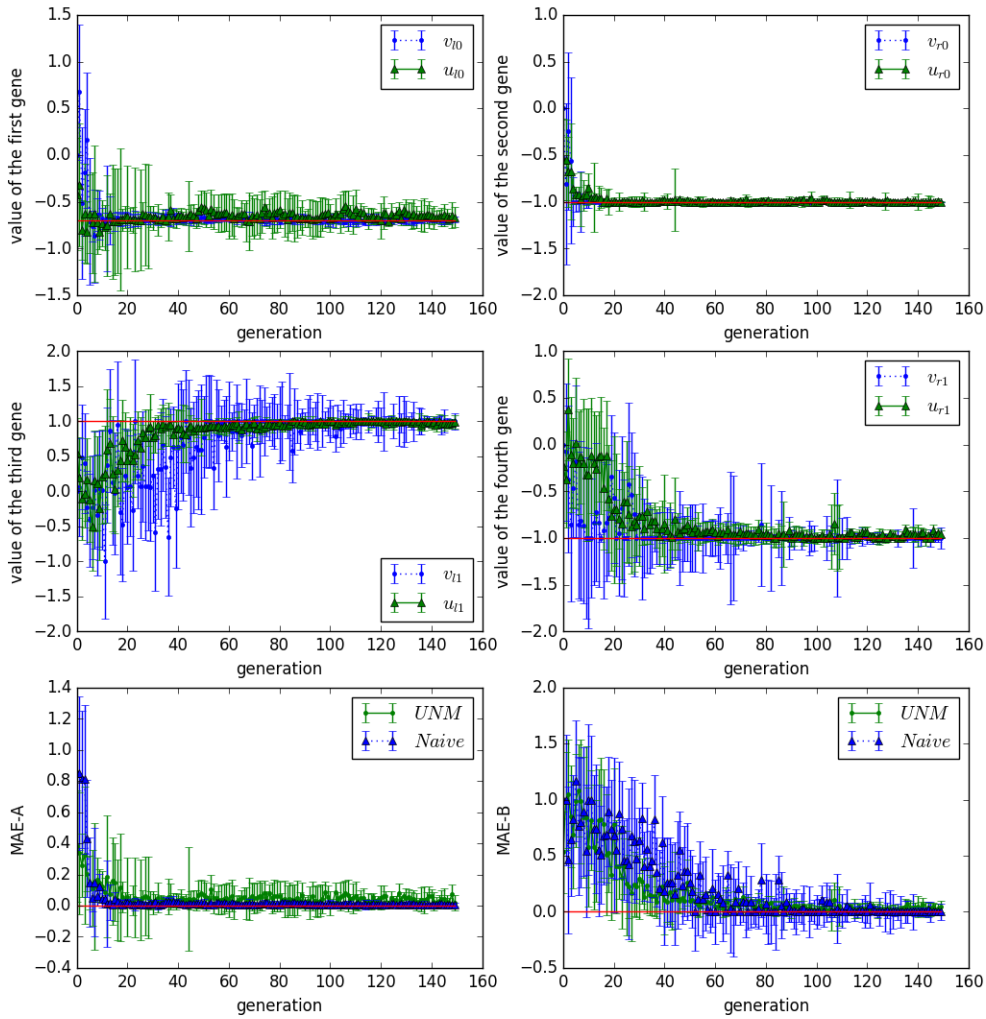


Figure 5.16: Convergence analysis of the parameters of aggregation game with/without UNM.

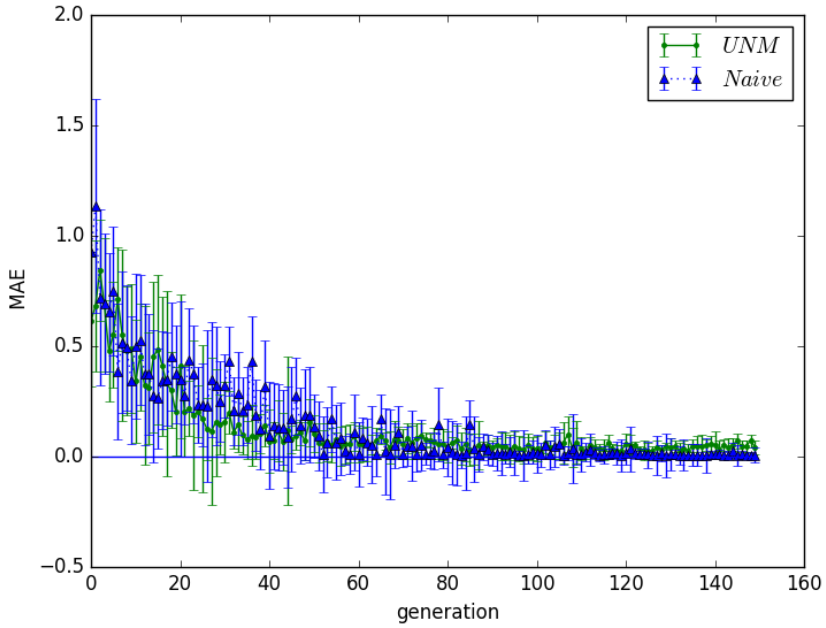


Figure 5.17: Convergence analysis of the aggregation game with/without UNM.

Interaction and Observation

Turing Learning is a new approach for the learning of the behavior of agents in interaction. The aggregation game is a good test case on the learning of swarm behavior. However the game is in an "optimal" setting to form a cluster [19]. The agents walk backwards when no obstacle presents at the front. As a result, they may run into each other and change the path of each other. This subsection studies the impact of UNM on the learning of parameters in agents with reactive control architecture. We define two new games:

Backwards Walking Game (BWG) This game is a variant of the aggregation game. This game differs from the aggregation game in two ways: the agents walk backwards in a straight line. The values are not marginal. More specifically, the agents move back at a speed of 6.4cm/s when no obstacle presents at the front. Otherwise, the agents turns to the left with a speed of -2.56 cm/s on the left wheel and 6.4cm/s on the right wheel. In other words, the parameters are $p = \{-0.5, -0.5, -0.2, 0.5\}$.

Safe Walking Game (SWG) In this game, agents are defined to avoid physical contact with other agents. The agents walk forward when seeing no obstacles, otherwise turn right. More specifically, the parameters are $p = \{0.7, 0.9, 0.3, -0.5\}$.

From previous studies we know that the learning of parameters of a given mode depends on the frequency of observation. For the aggregation game, the frequency is $f_A = 91.2\%$ and $f_B = 8.8\%$ for

Table 5.2: Time consumption of aggregation game with/without UNM.

No.	Naive	UNM
1	1592	548
2	1465	580
3	1471	532
4	1491	479
5	1470	493
6	1479	503
7	1475	528
8	1465	552
9	1455	527
10	1501	533
Average (s)	1486	527

mode A and mode B respectively. As for the BWG game, the frequency of observation is $f_A = 94.6\%$ for mode A and $f_B = 5.4\%$ for mode B. The corresponding frequency of observation values of SWG are $f_A = 95.4\%$ and $f_B = 4.6\%$ respectively.

In neither setting would the agents form a cluster, thus lose the swarm behavior of aggregation. In fact, in both cases, the agents avoid collision and spread all over the place after 100 time-steps. Similar as the aggregation game, we perform 10 tests for each game and plot the mean and standard deviation.

Similar as before, Figure 5.18 plots the four genes and the error measures of BWG. Figure 5.19 summarizes the MAE in one plot. It is clear that the parameters of mode A are learned at ease. In fact, due to high frequency of observation, the plots indicate that the standard deviation is less than that of the aggregation game. However, the parameters of mode B are significantly harder to learn. In both cases, the parameters do not converge within the generations limited by the computation power of our PC. Since there is little interaction between agents, and the paths of an agent may change due to the "push" of the others, the path is less organized and harder for the classifiers to tell apart. It is clear that the standard deviation corresponding to mode B is significantly larger than the aggregation game due to the low frequency of observation and uncertainty in paths. From Figure 5.19, UNM may be more capable of reducing the errors but not reaching convergence either.

As for the SWG game, both approaches manage to converge. Since the interaction is simpler, the convergence is more stable compared with that of the aggregation game (Figure 5.20 and Figure 5.21). The UNM approach results in slightly larger standard deviation values. A possible explanation is that in the UNM approach, strategies evolve against the opponent strategies in $\mathscr{N}$. In our setting, $|\mathscr{N}|$ is less than or equals to 10 while the naive approach evaluates against 50 strategies. This could lead to overfitting and results in larger standard deviation values.

In both games, the use of UNM reduces the computation time to around 32% compared with the

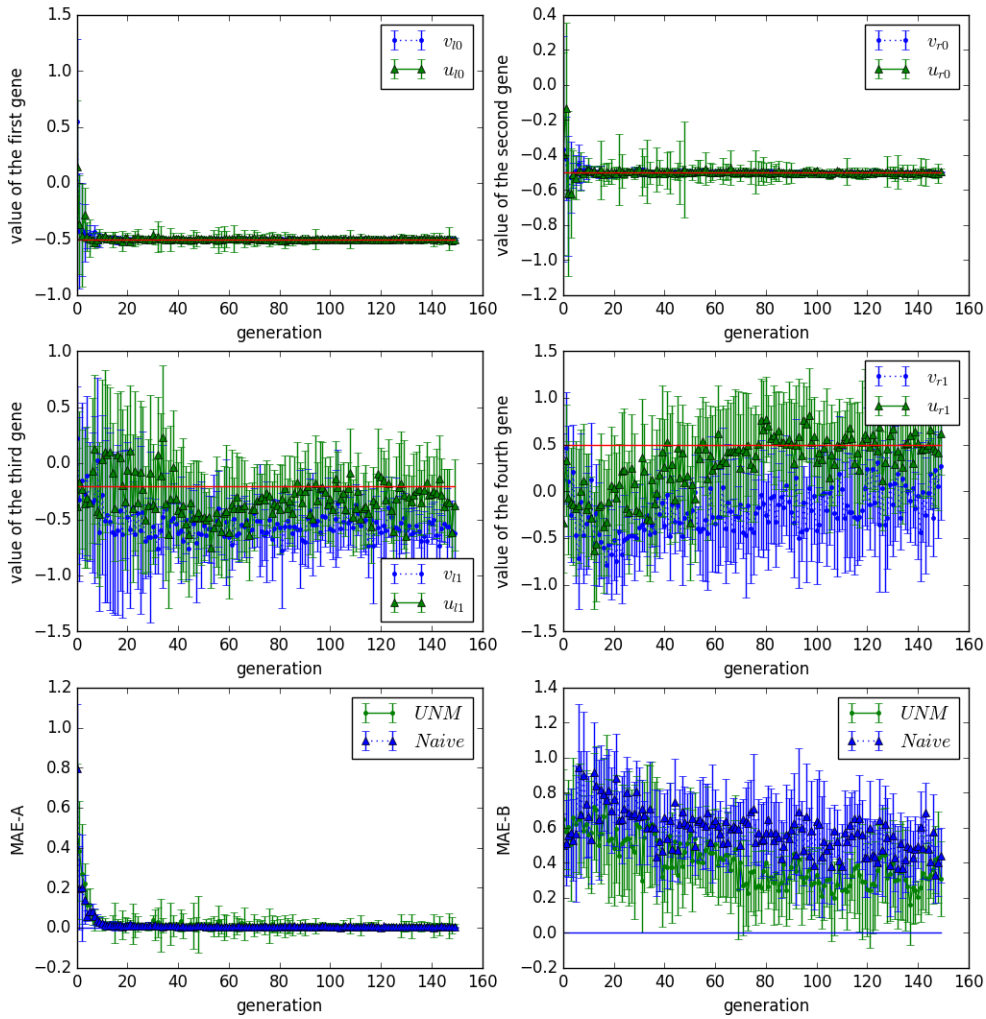


Figure 5.18: Convergence analysis of the parameters of BWG game with/without UNM.

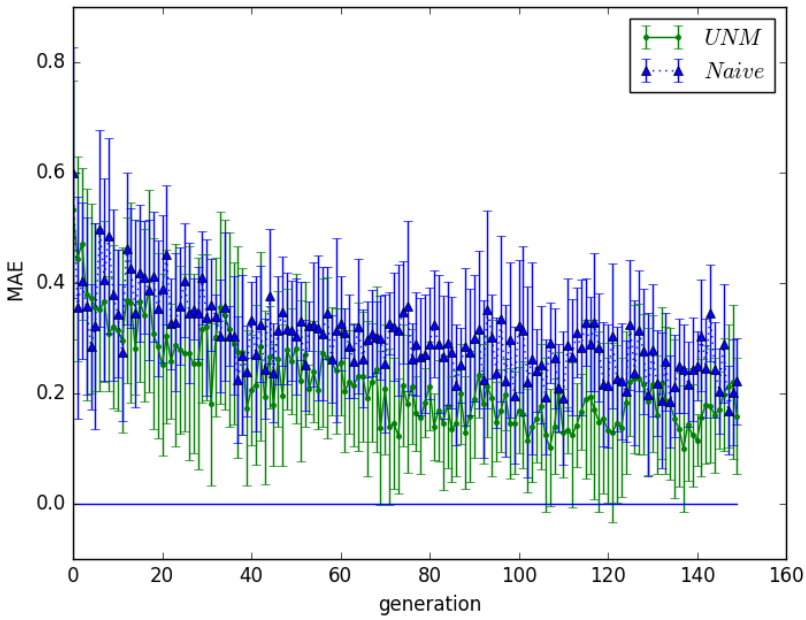


Figure 5.19: Convergence analysis of the BWG game with/without UNM.

naive approach.

Efficiency

This project makes use of a 64-bit PC with a CPU of AMD A10-7870K Radeon R7 (12 Compute Cores 4C+8G $\times$ 4) and a memory of 6.8GB running Ubuntu 16.04. The experiments of PNG/RNG game indicates that when the time taken for evaluation is very little, the computation time mostly depends on the time computing the Nash equilibria. As for the PNG game, it takes 1.38s in the naive approach in comparison with 18.65s for that using UNM. For the RNG game, the naive approach takes 1.35s on average while the UNM approach takes 17.95s. This indicates that around 90% of the computation time is dedicated to the interaction with external solvers. When each evaluation takes significantly longer, the advantage of UNM approach comes into sight. Since the strategies are only evaluated against those in $\mathscr{N}$ of the opponent player, the total number of evaluation is less. More specifically, the number of evaluation in the UNM approach is less than $50 \times 10 \times 4 = 4,000$ while that of the naive approach is $100 \times 100 = 10,000$. Table 5.2 indicates that when the size of matrices are bounded by a small number, this approach can be more efficient in time. The time used by the solver varies from 13.59% to 22.15% (excluding the small overhead reading and writing input and output files). In cases where physical experiments are involved, each evaluation step may take from minutes to hours. An example is the aggregation game in real world as described in detail in [19]. The total running time excluding human intervention time is 50 hours. This UNM approahc is promising in the reduction time in such cases.

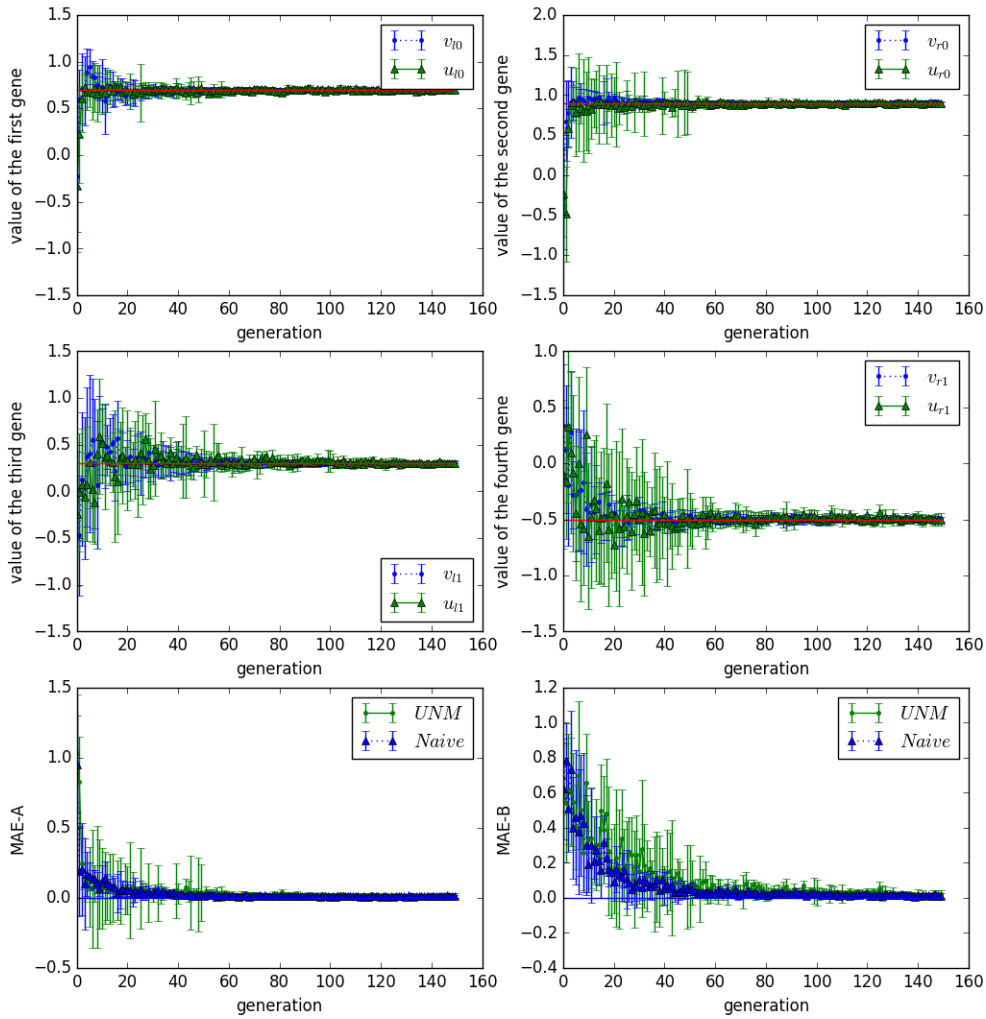


Figure 5.20: Convergence analysis of the parameters of the SWG game with/without UNM.

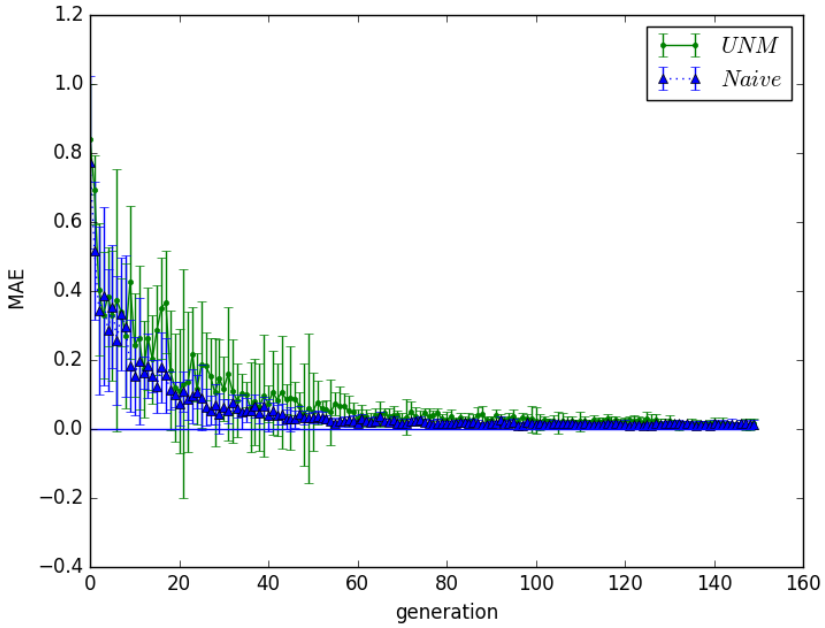


Figure 5.21: Convergence analysis of the BWG game with/without SNM.

In the original implementation, epuck robots take unbounded real values as parameters for the wheels in each mode. This is not only inefficient but also dangerous when it comes to real robots. In our implementation, these parameters are bounded to $[-1, 1]$ so the speed is within $[-12.8, 12.8]$ (cm/s). This makes the values of offspring in coevolution more focused on the domain of interest and thus more efficient. It takes 300 generations to reach convergence in the original work. With our improvements, convergence is achieved within 140 generations (Figure 5.17).

The main bottle neck of this UNM approach is the efficient computation of Nash equilibria of large bimatrix games. In this project, we make use of one of the state-of-the-art solvers, namely *lrs*. On a relatively powerful PC (see above for specification), as the size of matrices gets larger than 20×30, the computation time can be longer than 30 mins. Moreover, there are cases where the solver meets a bug and fails the computation. This bottleneck is further discussed in Chapter 6.

Noise and Uncertainty

While PNG and RNG have explicit evaluation functions, the aggregation game includes the agents' impact on each other. More specifically, in the aggregation game, epuck robots can run into each other since they walk backwards. This leads to the change of speed, orientation and location of the robots involved in the collision. The initial location and orientation of an epuck robot also have an impact on its behavior. For example, a robot located at the center of the environment is more likely to switch to mode B compared with a robot on the edge of a group. In this implementation, we

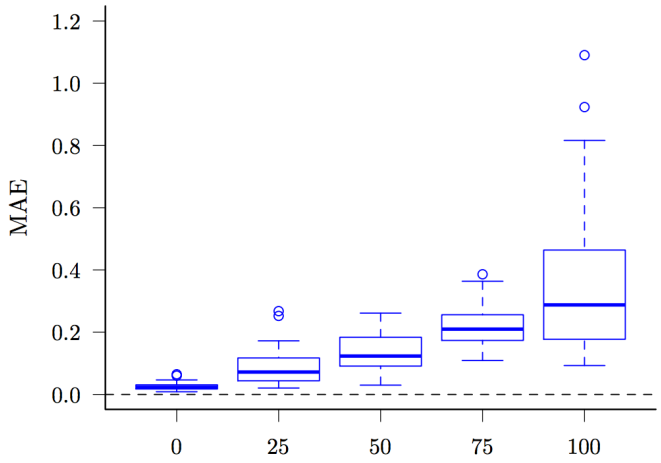


Figure 5.22: The impact of noise on the MAE [19].

take $R = A = 5$ in stead of $A = 10$ and $R = 1$ to make the fitness values more accurate and benefit the overall accuracy. Compared with the aggregation game (Figure 5.21), the SWG game (Figure 5.21) evaluation results show that when agents have no collision, the interaction is simpler, the behavior is more obvious to learn and thus the parameters converge in less generations. In reality, the observed behavior of a robot depends on many other aspects, for example, the error from the motion tracking device (e.g. a camera), etc. This results in noise in data. Figure 5.22 plots the MAE when increasing the percentage of noise on the measurements of the individuals' position and orientation in the aggregation behavior. Each box corresponds to 30 coevolution runs [19]. The plot reveals that the Turing Learning system still performs relatively well even when a considerable amount of noise affects the individual's motion tracking device [19]. Due to the limit of size of matrices in this project, it is unknown if this UNM approach can improve noise tolerance when equipped with a powerful bimatrix game solver that deals with large input matrices. Further experiments are needed.

6. Discussion and Conclusion

6.1 Discussion and Future Work

6.1.1 Games and Solvers

In practice, UNM does not guarantee monotonicity in convergence due to the limit in the size of input matrices. This is the main bottleneck of this approach. The Lemke-Howson algorithm is a widely-used pivoting algorithm. It has exponential worst case behaviour. It is PSPACE-hard to compute the equilibrium [7]. Most recent algorithms focuses on additive approximations. An $\varepsilon-$Nash equilibrium requires that both players achieve an expected payoff that is within σ of a best response. A stronger notion of σ-well supported Nash equilibrium (ε-WSNE) requires that both players only place probability on strategies that are within ε of a best response. The current state of the art for ε-Nash equilibria finds a (2/3 - 0.00591)-WSNE in polynomial time [7]. Some recent experimental work shows that it is possible to compute the ε-Nash equilibria of 2000 $\times$ 2000 matrices within several minutes ($epsilon$ = 0) [7]. A future step is to adapt some more efficient bimatrix solvers and integrate them with the current Turing Learning platform.

It was noticed that in this restricted version of UNM, identical strategies hurt the functionality of Nash memory. When there are two (or more) identical strategies with exactly the same payoff, the probability in a Nash equilibrium have to be shared between all these strategies. It is possible that when removing strategies from $\mathcal{N}$, these strategies are all kept for the next generation which may block new strategies from getting into the population.

Another issue is the efficiency and robustness of the game solver. The current default game solver is *lrs*. The solver is one of the most popular solvers in academia. However, it is heavy in the use of memory and not robust when dealing with large matrices. For the aggregation game, after 150 iterations the solver may run out of memory or return error. This leads to the failure of the whole test.

It is not possible to split the two matrices and convert a bimatrix game into a zero-sum asymmetric game and use PNM. The reason is simple: all the classifiers will reject all the models. This leads to the so-called *loss of gradient* where the strategy of one player dominates that of the other, thus making the strategies lost "guidance" while evolving. Another problem is the "cyclic behavior" where one player discovers the same set strategies after a few seemingly "better and better" generations, which is what the use of Nash memory is trying to prevent.

6.1.2 Architecture, Types and Metaprogramming

Turing Learning was introduced as a new approach for the learning of agent behavior in interaction. Existing implementations are hard-coded proof-of-concept test cases. There is a need for a general-purpose, object-oriented Turing Learning platform following a modular design. This thesis presents a new platform *TuringLearner* implemented in C++. The programming language was chosen initially for the convenience of interaction with Enki and the *lrs* solver. During this project, several implementations were attempted using different techniques for a modular and object-oriented design. The student obtained more experience and a better understanding of this coevolutionary platform after each attempt.

- Before the implementation of *TuringLearner*, the student studied the encoding of games and the interaction with LP solvers. Details are presented in Section 2.2.2 and Appendix A.3.
- In the first implementation, reproduction functions, evaluate functions and other elements of a solution concept were passed between players and the platform as functors. Despite the complexity, the implementation was complicated and domain-specific in design. Moreover, users were assumed to be familiar with function objects. In addition, the debugging was confusing and inefficient.
- The second attempt followed a top-down design using object-oriented programming. This led to multiple bugs due to the lack of experience. As a result, the players exhibited strange behaviors. Optimal strategies might be ignored due to bugs at multiple places. This attempt was given up.
- The third attempt introduced metaprogramming concepts. However, the top-down design led to a great trouble. Although the student managed to realize the PNG and RNG game, the implementation was messy and confusing.
- The final attempt was a careful redo of the previous work using a bottom-up design. This time, each module was tested separately before integration. Metaprogramming was taken advantage to make code more elegant and efficient. Existing implementation of solution concepts could be easily adopted to new objects. Based on this implementation, the student implemented the

aggregation game and its variants.

Although the current implementation is a successful attempt capturing desired features as a Turing Learning platform, there are several drawbacks:

- The use of metaprogramming reduces the complexity of the code and allows efficient reuse of implementation of solution concepts. However, this requires the developers' knowledge about metaprogramming. Taking types as input parameter is not common. Initially, C++ was chosen as the programming language for the convenience of interaction with Enki and *lrs*. The current platform has a strict type system. Developers will frequently encounter passing types as parameters and function overridding and overloading. This will result in confusion and bugs hard to discover. A possible solution is to relax on the types. A python implementation may reduce the complexity and make it easier to interact with state-of-the-art Machine Learning tools, especially those of neural networks and deep learning.
- During this project, there are occasions where values need to be collected for analysis. The current implementation has logging functions but those logging functions are not systematic. On top of this, additional visualization functions would help the users observe the process of coevolution. Reloading functions would also help to continue the coevolution process from a break point caused by the failure of external solvers.
- *TuringLearner* deals with the evolution of neural networks. The default coevolution method is the $\lambda + \mu$ algorithm, which is not an up-to-date approach for evolving neural networks. Most recent novel evolutionary optimization algorithms include the CMA-ES, which is based on the derandomized evolution strategy with covariance matrix adaptation [15], and the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which evolves not only not the weights but also the topology of the network [30].
- Many functions can be parallelized. This will improve the efficiency computing the payoff matrices for example.

6.1.3 Models and Classifiers

One of the improvements in our work is the balance of real agents and replica agents in each simulation. The original Turing Learning implementation takes only one replica. It is possible that the replicas do not exhibit different behavior in a simulation. This would lead to misevaluation. In fact, a significant amount of time was spend on running simulations. A possible way to reduce the amount of time is to generate replicas from different strategies and run in one simulation for more efficient observations.

One of the problem is overfitting. The most direct reason is the limit on the size of $\mathscr{N}$. A possible method is to introduce a set $\mathscr{B}$ and collect the best strategies of some previous generations. Newly discovered strategies can be evaluated against $\mathscr{N} \cup \mathscr{B}$ to prevent overfitting. Another idea is to take previously generated results into evaluation.

Generative Adversarial Networks (GAN) [13] follow a similar concept of coevolution as Turing Learning. GAN forms a game between two neural network models: the generative model G creates samples that captures the data distribution, and the discriminative model D estimates the probability that a sample is real or fake. The two neural networks coevolve and train each other. In the case where G and D are defined by multiple player perceptrons, the entire system can be trained with backpropagation. Since its invention, GAN have attracted a lot attention and variants of GANs have been developed. When taking samples as agents the concept is very similar as Turing Learning. From the primitive results in this thesis, it is likely that UNM increases the learning efficiency and results in faster training of GAN networks. Some experiments are to be designed for concrete evaluation this idea.

6.2 Conclusion

This thesis extended Parallel Nash memory and introduced Universal Nash Memory (UNM). We studied the integration of UNM with Turing Learning for efficient reverse engineering of agent behaviors. As a first step, we tested the correctness of our first implementation on the Intransitive Number Game (ING). Then we presented *TuringLearner*, the first Turing Learning platform together with test cases of Percentage Number Game (PNG) and Real Number Game (RNG). Finally, we implemented the aggregation game together with its two variants, Backwards Walking Game (BWG) and Safe Walking Game (SWG). We investigated the balance of strategies in UNM and studied how UNM impact Turing Learning when facing changes and uncertainty in interaction. Our experiments indicate that this new approach can reduce the computation time to 35.4% and results in faster convergence for the aggregation game. Discussion and possible future work were also presented.

6.3 Acknowledgement

The student appreciates detailed guidance and feedback from Dr. Leen Torenvliet and Dr. Frans Oliehoek. Dr. Roderich Groß explained Turing Learning with great patience. The project took advantage of Dr. Wei Li's work. The source code of the aggregation game was kindly shared to the people involved in this project, which made it easier for this project. The project finished after the commerce of the student's doctoral studies. The student also appreciates the understanding of his supervisor, Prof. Frank van Harmelen. In addition, Mr. Eric Flaten and Mr. Ho-Yin Lui offered kind help reviewing some paragraphs.

A. External Programs and Test Cases

This project is an interdisciplinary project. Many platforms and tools were used in the project, among which, many were new to the student. This chapter sums up a brief description of each platform and tool involved for the convenience of the readers and quick reference. These sections may also be useful for readers who are not interested in the thesis but only a particular tool.

A.1 Enki

Enki is the main swarm robot simulation platform used in this project. However, there is little material to guide users step by step. This introduction is for those who need a tutorial of the Enki robot simulation platform. Some of the examples included are manipulated from the online Enki reference document[1].

Enki[2] is a simple and popular robot simulator. Physical objects include robots as well as static objects that do not suppose to move by themselves. All objects and are placed in a virtual world. Robots may interact with other robots and the environment. In the default Enki distribution, there are four kinds of robots included (three true robots and beacon). To keep our work consistent with Wei Li's work for comparison, the student used the EPuck robot [19].

[1] `http://lis2.epfl.ch/resources/download/doc1.0/libenki/index.html`
[2] `https://github.com/enki-community/enki`

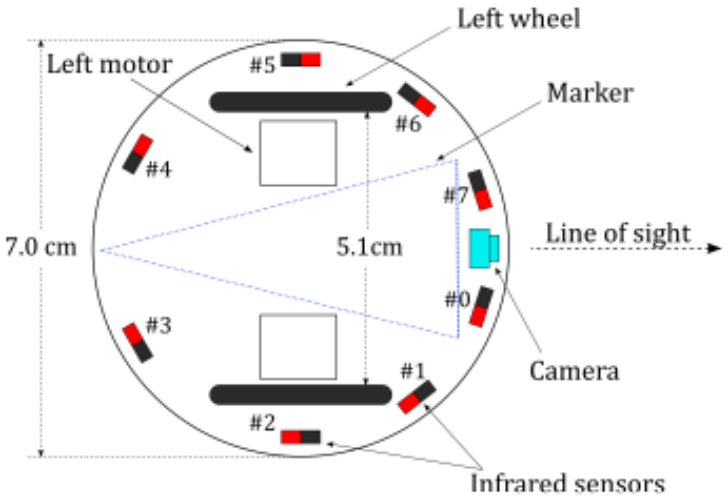


Figure A.1: A schematic top view of e-puck [19]

Figure A.2: The e-puck robot

A.1.1 The e-puck robots

The e-puck robot was designed at the EPFL (Ecole Polytechnique Federale de Lausanne) for teaching and research[3]. It is a very small and compact robot with many sensors and can be placed on top of a palm. The robot comes with a software, EpuckMonitor, which is a program developed in C++ that allows uers control the robot E-puck from PC via Bluetooth. In this project we took advantage of its simulation only. The robot has two wheels (left and right) and eight sensors.

In the Enki simulation, the epuck robot is represented as an object whose diameter is 7cm, height is 5cm with a mass of 150g. The speed of each wheel can be set independently and the maximum speed is 12.8cm/s. The control of robot is updated every 0.1s while the physics is updated every 0.01s [19].

A.1.2 A simple example: a robot and a world

We create a world and initialize it with a robot. The world is world is of size 200mm by 200mm with one robot located at (100, 100). The speed of the left wheel is 30mm/s and the right is 20mm/s. The location of the robot is printed in the console. This simulation is running virtually. To see the movement of the robot, a viewer is still to be added.

```
#include <enki/PhysicalEngine.h>
#include <enki/robots/e-puck/EPuck.h>
```

[3] http://www.e-puck.org/

```
#include <iostream>

int main(int argc, char *argv[])
{
        // Create the world
        Enki::World world(200, 200);

        // Create a robot and position it
        Enki::EPuck *ePuck = new Enki::EPuck;
        ePuck->pos = Enki::Point(100, 100);
        ePuck->leftSpeed = 30;
        ePuck->rightSpeed = 20;
        world.addObject(ePuck);
        // Run for some times
        for (unsigned i=0; i<10; i++)
        {
                // step of 50 ms
                world.step(0.05);
                std::cout << "E-puck pos is (" << ePuck->pos.x
                << "," << ePuck->pos.y << ")" << std::endl;
        }
}
```

A.1.3 Viewer

Enki uses Qt5[4] for simulation interface. Users can define a class as an inheritance of the `ViewerWidget` class. The following code corresponds to a simulation of three robots in a bounded round environment (Figure A.3) with two video records online[5]. Another simulation with a bounded square environment as in Figure A.4 was also recorded for clearer understanding of the process [6].

```
#include "Viewer.h"
#include <enki/PhysicalEngine.h>
#include <enki/robots/e-puck/EPuck.h>
#include <enki/robots/marxbot/Marxbot.h>
#include <enki/robots/thymio2/Thymio2.h>
#include <QApplication>
#include <QtGui>
#include <iostream>

using namespace Enki;
```

[4] `http://doc.qt.io/qt-5/`
[5] `http://youtu.be/zY7gPqs5VpM` and `http://youtu.be/NQ3pThASh68`
[6] `http://youtu.be/qWm1hoD9sCc`

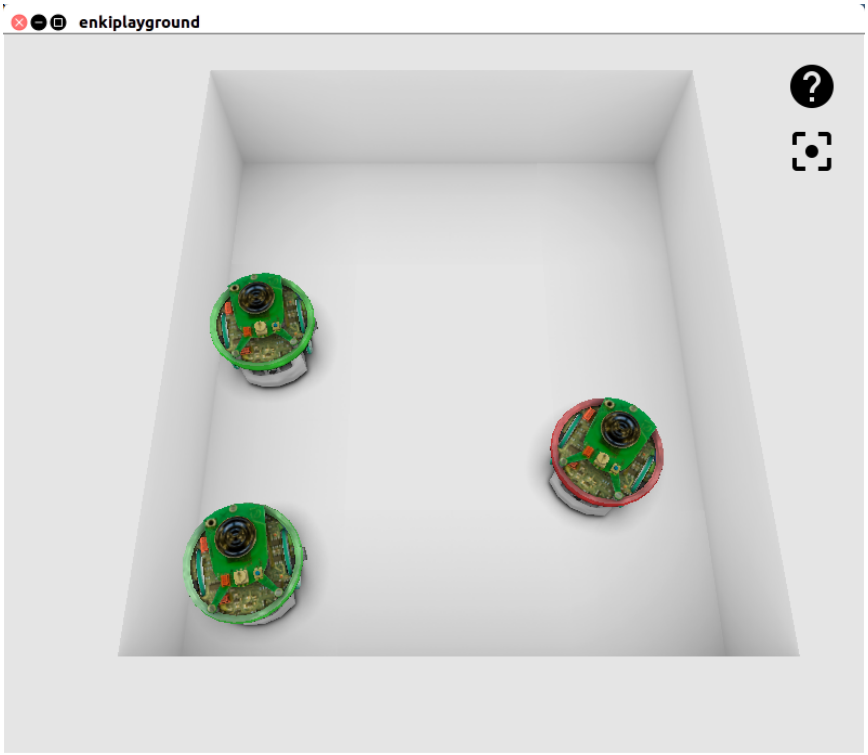


Figure A.3: Three epuck robots in a bounded empty environment

```
using namespace std;

class EnkiPlayground : public ViewerWidget
{
protected:
        QVector<EPuck*> epucks;
        QMap<PhysicalObject*, int> bullets;

public:
        EnkiPlayground(World *world, QWidget *parent = 0) :
                ViewerWidget(world, parent)
        {
                EPuck *epuck = new EPuck;
                epuck->pos = Point(20, 15);
                epuck->leftSpeed = 4;
                epuck->rightSpeed = 5;
                world->addObject(epuck);

                epuck = new EPuck;
                epuck->pos = Point(20, -10);
                epuck->leftSpeed = 5;
                epuck->rightSpeed = 2;
```

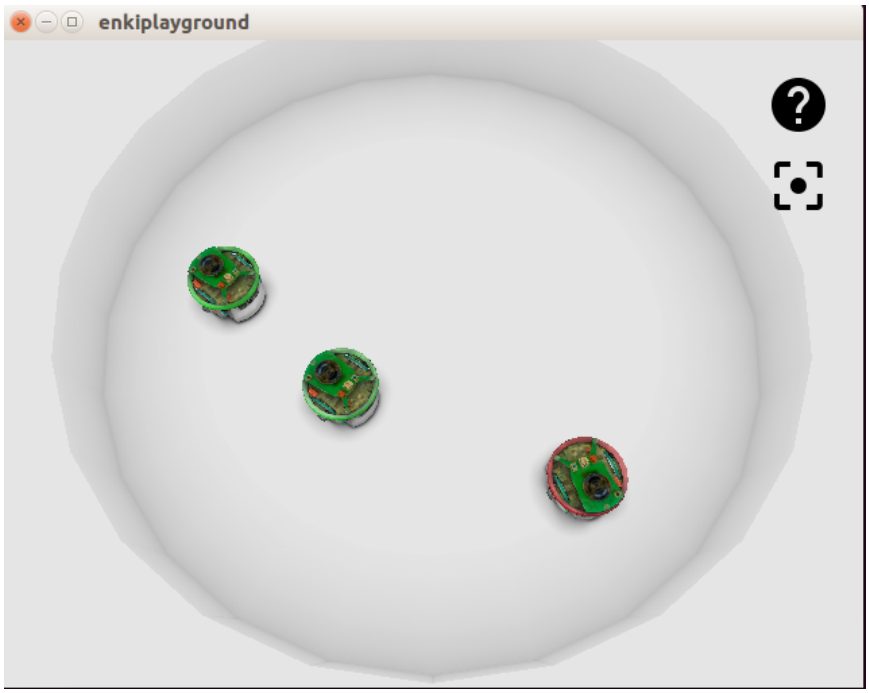


Figure A.4: Three EPuck robots in a bounded empty environment

```
                epuck->setColor(Color(1, 0, 0));
                world->addObject(epuck);

                epuck = new EPuck;
                epuck->pos = Point(0, 30);
                epuck->leftSpeed = 2;
                epuck->rightSpeed = 3;
                epuck->setColor(Color(0, 1, 0));
                world->addObject(epuck);
        }
};

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);
        // Create the world and the viewer
        bool igt(app.arguments().size() > 1);
        QImage gt;
        World world(35, Color(0.9, 0.9, 0.9),
            World::GroundTexture());
        EnkiPlayground viewer(&world);
        viewer.show();
        return app.exec();
```

```
}
```

A.1.4 Object

The following code adds a polygone object in the middle of the world (Figure A.5). A corresponding simulation video is online[7]. In the video, the a robot can move the object around when the object blocks its way.

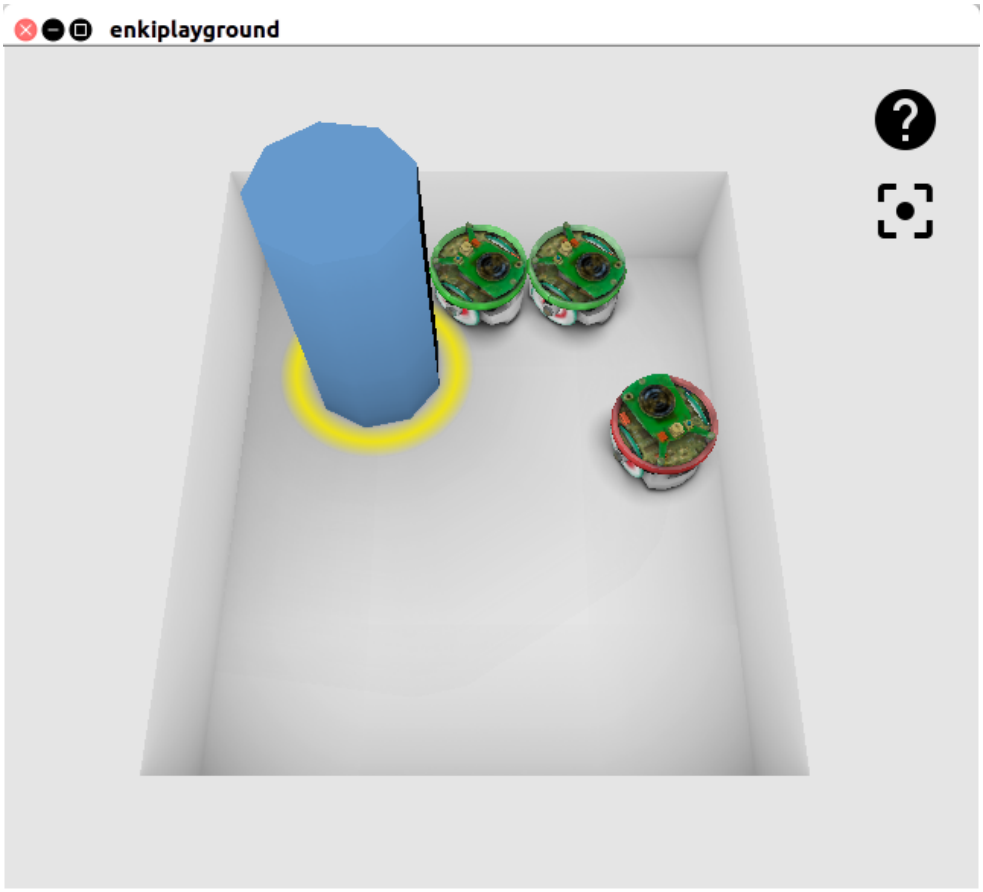


Figure A.5: The viewer of a simulation of robots and an object

```
Enki::Polygone p;
    const double amount = 9;
    const double radius = 5;
    const double height = 20;
    // std::cout <<"M_PI = " <<M_PI <<std::endl;
    for (double a = 0; a < 2*M_PI; a += 2*M_PI/amount)
            p.push_back(Enki::Point(radius * cos(a), radius * sin(a)));
    Enki::PhysicalObject* o = new Enki::PhysicalObject;
```

[7] http://youtu.be/CcFyJYymimI

```
Enki::PhysicalObject::Hull hull(
    Enki::PhysicalObject::Part(p, height));
o->setCustomHull(hull, 1);
o->pos = Enki::Point(100, 100);
o->setColor(Enki::Color(0.4,0.6,0.8));
world->addObject(o);
```

A.2 GNU Scientific Library (GSL)

The GNU Scientific Library (GSL) provides a over 1000 functions in a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. It was used in this project to obtain a random number following uniform distribution within a given range[8] or a specified Gaussian (i.e., normal) distribution[9]. To keep this project consistent with Wei Li's work for comparison, the project used the same parameters at the beginning [19]. The following is a short description of some functions where *r* is the number generator (defined as a global variable in the project).

- `gsl_ran_gaussian (const gsl_rng * r, double sigma)` The function returns a Gaussian random variate, with mean zero and standard deviation sigma. The probability distribution for Gaussian random variates is, $p(x)dx = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$
- `gsl_rng_uniform (const gsl_rng * r)` The function returns a double precision floating point number uniformly distributed in the range [0,1). The range includes 0.0 but excludes 1.0.

A.3 Solve the Rock-Paper-Scissors Game with PuLP

The Rock-Paper-Scissors game was the first sub-project implemented. In this project we evaluated PuLP[23] and attempted implementing the Nash Memory Mechanism (even though there are only three actions). The code is open-source at `https://github.com/airobert/RockPaperScissors`. A hard-coded Python implementation of the RPS game using PuLP [10] as follows:

```
from pulp import *

prob = LpProblem("test1", LpMaximize)

# Variables
```

[8] `https://www.gnu.org/software/gsl/manual/html_node/Sampling-from-a-random-number-generator.html`

[9] `https://www.gnu.org/software/gsl/manual/html_node/The-Gaussian-Distribution.html`

[10] The code is in the file `https://github.com/airobert/RockPaperScissors/blob/master/lp.py`

```
x1 = LpVariable("x1", 0, 1)
x2 = LpVariable("x2", 0, 1)
x3 = LpVariable("x3", 0, 1)
v = LpVariable("v", 0)

# Objective
prob += v

# Constraints
prob += v <= -1*x2 + x3
prob += v <= x1 + -1*x3
prob += v <= -1*x1 + x2
prob += x1 + x2 + x3 == 1

GLPK().solve(prob)

# Solution
for v in prob.variables():
        print (v.name, "=", v.varValue)

print ("objective=", value(prob.objective))
```

A.4 The *lrs* Solver

lrslib [2] is a self-contained ANSI C implementation of the reverse search algorithm for vertex enumeration/convex hull problems[11]. It can be used to find Nash equilibria of a bimatrix game. All computations are done exactly in either multiple precision or fixed integer arithmetic. This project takes advantage of *lrsnash* and computes all Nash equilibria (NE) for a two person noncooperative game. They are computed using two interleaved reverse search vertex enumeration steps. Since the output is not stored in memory, during the project, the student have carefully modified the program to output the answers in sequence. The input for the problem are two m by n matrices A,B of integers or rationals. The first player is the row player, the second is the column player. An example run is the Battle of the sexes game as described in Section 2.2.3. The input are two matrices of size 2×2.

```
2 2

3 0
0 2

2 0
0 3
```

[11] http://cgm.cs.mcgill.ca/~avis/C/lrs.html

The output is a description of the Nash equilibria. There are three Nash equilibria. The output is in a format of player-distribution-payoff. The first Nash equilibria is when the man goes to the opera with a chance of 40% while the woman does so with a probability of 60%. This way, they both get a payoff of 1.2. When both goes to the same place, the payoff increases to 2 and 3.

```
*Copyright (C) 1995,2016, David Avis    avis@cs.mcgill.ca
*lrsnash:lrslib v.6.2 2016.3.28(64bit,lrsgmp.h gmp v.6.1)
2   2/5   3/5   6/5
1   3/5   2/5   6/5

2   0   1   2
1   0   1   3

2   1   0   3
1   1   0   2

*Number of equilibria found: 3
*Player 1: vertices=3 bases=3 pivots=5
*Player 2: vertices=3 bases=1 pivots=6
```

lrs can compute to the most $MAXSTRAT * MAXSTRAT$ number of Nash equilibria. In this project, we take $MAXSTRAT = 100$.

Bibliography

[1] David Avis. A revised implementation of the reverse search vertex enumeration algorithm. In *DMV Seminar*, volume 29, pages 177–198. Springer, 1998.

[2] David Avis, Gabriel D Rosenberg, Rahul Savani, and Bernhard Von Stengel. Enumeration of nash equilibria for two-player games. *Economic theory*, 42(1):9–37, 2010.

[3] Ken Binmore. *Fun and games, a text on game theory*. D.C. Heath and Company, 1992.

[4] Christian Blum and Xiaodong Li. *Swarm Intelligence in Optimization*, pages 43–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] Agoston E. Eiben and James E. Smith. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[6] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179 – 211, 1990.

[7] John Fearnley, Tobenna Peter Igwe, and Rahul Savani. *An Empirical Study of Finding Approximate Equilibria in Bimatrix Games*, pages 339–351. Springer International Publishing, Cham, 2015.

[8] Sevan G. Ficici and Jordan B. Pollack. A game-theoretic memory mechanism for coevolution. In *Genetic and Evolutionary Computation Conference*, pages 286–297. Springer, 2003.

[9] Sevan Gregory Ficici. *Solution concepts in coevolutionary algorithms*. PhD thesis, Brandeis University, 2004.

[10] Melvin Gauci, Jianing Chen, Wei Li, Tony J. Dodd, and Roderich Groß. Self-organized aggregation without computation. *The International Journal of Robotics Research*, 33(8):1145–1161, 2014.

[11] Andrea Gaunersdorfer, Josef Hofbauer, and Karl Sigmund. On the dynamics of asymmetric games. *Theoretical Population Biology*, 39(3):345–357, 1991.

[12] Paul W. Goldberg, Christos H. Papadimitriou, and Rahul Savani. The complexity of the homotopy method, equilibrium selection, and lemke-howson solutions. *ACM Trans. Economics and Comput.*, 1(2):9:1–9:25, 2013.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[14] Harry W Greene and Roy W McDiarmid. Coral snake mimicry: does it occur. *Science*, 213(4513):1207–1212, 1981.

[15] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003.

[16] Danica Janglová. Neural networks in mobile robot motion. *International Journal of Advanced Robotic Systems*, 1(1):2, 2004.

[17] Ross D. King, Jem Rowland, Stephen G. Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N. Soldatova, Andrew Sparkes, Kenneth E. Whelan, and Amanda Clare. The automation of science. *Science*, 324(5923):85–89, 2009.

[18] F. L. Lewis, A. Yesildirek, and Kai Liu. Multilayer neural-net robot controller with guaranteed tracking performance. *IEEE Transactions on Neural Networks*, 7(2):388–399, 1996.

[19] Wei Li. *Automated Reverse Engineering of Agent Behavoirs*. 2016. PhD Thesis, The University of Sheffield.

[20] Wei Li, Melvin Gauci, and Roderich Groß. Turing learning: a metric-free approach to inferring behavior and its application to swarms. *Swarm Intelligence*, 10(3):211–243, 2016.

[21] Alex McAvoy and Christoph Hauert. Asymmetric evolutionary games. *PLoS Computational Biology*, 11(8), 2015.

[22] Thomas Miller, Paul J. Werbos, and Richard S. Sutton. *Neural networks for control*. MIT press, 1995.

[23] Stuart Mitchell and Iain Dunning. PuLP: a linear programming toolkit for python. Technical report, Department of Engineering Science, The University of Auckland, 09 2011.

[24] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[25] Frans A Oliehoek, Edwin D De Jong, and Nikos Vlassis. The parallel nash memory for asymmetric games. In *Proceedings of the 8th annual conference on genetic and evolutionary computation*, pages 337–344. ACM, 2006.

[26] Ryan Porter, Eugene Nudelman, and Yoav Shoham. Simple search methods for finding a nash equilibrium. *Games and Economic Behavior*, 63(2):642–662, 2008.

[27] Sara J Shettleworth. *Cognition, evolution, and behavior*. Oxford University Press, 2010.

[28] John Maynard Smith. *Evolution and the Theory of Games*. Cambridge university press, 1982.

[29] Kenneth O Stanley and Risto Miikkulainen. The dominance tournament method of monitoring progress in coevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002) Workshop Program*, pages 242–248. Morgan Kaufmann, San Francisco, CA, 2002.

[30] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[31] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[32] Richard A Watson and Jordan B Pollack. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 702–709. Morgan Kaufmann Publishers Inc., 2001.

[33] Rudolf Paul Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. PhD thesis, Fairfax, VA, USA, 2004.

[34] Jia You. Beyond the turing test. *Science*, 347(6218):116–116, 2015.