UNIVERSITY OF THE WITWATERSRAND
JOHANNESBURG

School of Computer Science and Applied Mathematics

Crowd Behavioural Simulation via Multi-Agent Reinforcement Learning

Sheng Yan Lim (0710918P)
Supervised by Prof. Clint Van Alten and Mr. Pravesh Ranchod

August 2015

Abstract

Crowd simulation can be thought of as a group of entities interacting with one another. Traditionally, an animated entity would require precise scripts so that it can function in a virtual environment autonomously. Previous studies on crowd simulation have been used in real world applications but these methods are not learning agents and are therefore unable to adapt and change their behaviours. The state of the art crowd simulation methods include flow based, particle and strategy based models. A reinforcement learning agent could learn how to navigate, behave and interact in an environment without explicit design. Then a group of reinforcement learning agents should be able to act in a way that simulates a crowd. This thesis investigates the believability of crowd behavioural simulation via three multi-agent reinforcement learning methods. The methods are Q-learning in multi-agent markov decision processes model, joint state action Q-learning and joint state value iteration algorithm. The three learning methods are able to produce believable and realistic crowd behaviours.

Declaration

I, Sheng Yan Lim, hereby declare the contents of this research report to be my own work. This report is submitted for the degree of Masters of Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

Signature ______________________________ Date ______________________________

Acknowledgements

I would like express my deepest appreciation to Professor Clint Van Alten, who has given me great guidance since my honours year. The advices and suggestions that he has given me, have helped me to understand and value the work. Without his clear supervision and continuous help, this dissertation would not have been possible.

A heartfelt thanks to Mr. Pravesh Ranchod, who has pointed me towards the right direction countless times. He has shared and provided his past experiences and practical advices with me to make the process of finishing this thesis smoother. His help is highly appreciated.

I would also like to thank Orry Messer for his sharp eyes in pointing out an error in my work. A special thanks to Dean Wookey for providing me programming assistance when I needed the most.

Contents

Abstract i

Declaration ii

Acknowledgements iii

List of Figures vi

List of Tables vii

List of Acronyms viii

1 Introduction 1

2 Background and Related Work 4

2.1 Reinforcement Learning 4
2.2 Single Agent Reinforcement Learning 5
2.2.1 Value Iteration 7
2.2.2 Q-Learning Algorithm 8
2.2.3 Example: Path Finding 9
2.3 Multi-Agent Machine Learning 13
2.3.1 Multi-Agent Markov Decision Process 13
2.3.2 Game Theory 13
2.3.3 Markov Game 18
2.3.4 Coordinating Q-Learning 22
2.4 Crowd Simulation 24
2.4.1 Macroscopic Model 25
2.4.2 Microscopic Models 26
2.4.3 Crowd Simulation using Reinforcement Learning 28
2.5 Chapter Summary 30

3 Multi-Agent Crowd Simulation 32

3.1 The Environment 32
3.2 Formation of Bottlenecks 34
3.3 Simultion Recordings 36
3.4 Chapter Summary 36

4 Rule Based and MMDP Agents **37**
4.1 Rule Based Agents 37
4.1.1 Rule Set 37
4.2 MMDP Agents 37
4.2.1 State Space 38
4.3 Experiments 38
4.4 Results and Analyses 39
4.5 Chapter Summary 43

5 JSA-Q_1 and JS-VI Agents **44**
5.1 JSA-Q_1 Agents 44
5.1.1 Joint State Action Space 44
5.1.2 The *JSA*-Q_1 Algorithm 45
5.2 JS-VI Agents 47
5.2.1 Joint State and Joint Action 47
5.2.2 The *JS-VI* Algorithm 47
5.3 Experiments 47
5.4 Results and Analysis 49
5.4.1 JSA-Q_1 Agents 49
5.4.2 JS-VI Agents 52
5.5 Chapter Summary 54

6 Conclusion **55**

Appendix A **58**

Appendix B **59**

References **59**

List of Figures

2.1 Example showing immediate and long-term rewards 5
2.2 Example: path finding 9
2.3 Framework for learning on two levels 23
2.4 Traffic flow 25
2.5 The 2D School Domain [Torrey 2010] 29

3.1 Crowd simulation platform 33
3.2 Small scaled simulation platform 33
3.3 An example of the colour scale representing the density intensity. Lighter to darker shades depicting lower to higher density. 34
3.4 Density Graph - This density graph represents the movement of 30 agents in the grid world from figure 3.1a over a period of time. 35
3.5 The Y or V shape forms around the exit where the clogging occurs 35

4.1 The state information (shaded regions) that is available to the agent at position X. 38
4.2 20-agent simulations - 50, 100 and 200 rounds 40
4.3 50-agent simulations - 50, 100 and 200 rounds 40
4.4 100-agent simulations - 50, 100 and 200 rounds 40
4.5 Average number of exits in Experiment A and B 41
4.6 Density graphs generated by 200 rounds of simulations trained over 2000 episodes 42
4.7 Average number of collisions in Experiment A and B 42
4.8 An example showing that a collision is unavoidable in the MMDP model. . . . 43

5.1 A 2×2 grid world illustrating the joint state action space concept 45
5.2 Two policies to following from the initial states 46
5.3 Collision Avoidance 50
5.4 Graph showing average number of steps to reach the goal state 51
5.5 Two-agent Collision Avoidance 53
5.6 Three-agent Collision Avoidance 53
5.7 Four-agent Collision Avoidance 53

List of Tables

2.1 First 3 iterations of Value Iteration 10
2.2 Policy 1 generated from the optimal value function 10
2.3 Policy 2 generated from the optimal value function 11
2.4 These are the first five steps of the training episodes 11
2.5 The final Q-table 12
2.6 Policy generated from the final Q-function 12
2.7 The *Prisoner's Dilemma* - utility function 15
2.8 The *Coordination Game* - utility function 15
2.9 *Bach or Stravinsky?* - utility function 15
2.10 *Matching Pennies* - utility function 16
2.11 *Chicken* - utility function 16
2.12 Zone classification 27
2.13 General Steering Behaviour [Reynolds 1999] 28

5.1 Comparisons of the Number of Entries in Q-table 49
5.2 Q-values of the joint state $(1, 1)$ and $(2, 2)$ 51
5.3 Average steps obtained from the *JSA*-Q_1 optimal policies 52
5.4 Average steps obtained from the *JS-VI* optimal policies 54

B.1 Value Function generated from a two-agent example - Part 1 59
B.2 Value Function generated from a two-agent example - Part 2 60
B.3 Value Function generated from a two-agent example - Part 3 61
B.4 Value Function generated from a two-agent example - Part 4 62

List of Acronyms

RL	Reinforcement Learning
MDP	Markov Decision Processes
MARL	Multi-Agent Reinforcement Learning
MMDP	Multi-Agent Markov Decision Processes
JSA-Q_1	Joint State Action Q_1
JS-VI	Joint State Value Iteration
PPAD	Polynomial Parity Arguments on Directed graphs

Chapter 1

Introduction

One of the challenges in crowd simulation is to generate large background groups of virtual agents in a believable manner [Torrey 2010]. Applications of crowd simulation can be found in architecture, computer graphics, physics, robotics, safety science, training systems and sociology [Thalmann and Musse 2007]. Methods to simulate crowds and their behaviour include rule-based approach where the agents follow some predefined scripts, flow models where individuals are non-distinguishable, particle models where crowds are particles and move according to the physics of particle motion and strategy based models where crowds follow probabilistic behavioural rules.

Animating crowd behaviour started with the work done by Reynolds [1987]. It was argued that group behaviour is just the result of individuals (termed boids) interacting with one another. Therefore, by simulating individual, small and unsophisticated boids, a more complex behaviour would develop through the interactions. This method was inspired by observing behaviours present in flocks of birds and schools of fishes.

There are two main directions in crowd simulations, *realism of behavioural aspects* and *high-quality visualisation* [Thalmann and Musse 2007]. In the former, simulations are achieved in a simple 2D environment so that more quantitative validation can be compared to real world observations, for example, a simulated emergency evacuation and a hazard drill. In the latter idea, one considers visual results that can be presented in a convincing and believable manner, for example, animated movies and 3D games.

In machine learning, reinforcement learning [Sutton and Barto 1998] is a learning paradigm where a learning agent is able to navigate an unknown environment by performing actions that cause transitions between states within the environment. A numerical reward is associated with each transition of states. The objective of a reinforcement learning agent is to maximise this numerical reward based on the actions performed. The key aspect of reinforcement learning is that an agent learns through *interactions* within an environment by trial-and-error. A framework called *Markov Decision Processes* is often used for representing reinforcement learning problems.

Two notable methods for solving reinforcement learning problems are *Q-learning* and *value iteration*. The value iteration algorithm, a class of dynamic programming algorithms, attempts to compute optimal policies in a model-based environment. The Q-learning algorithm, a class of temporal-difference learning algorithms, updates the estimates based on previously learned estimates (experiences) in a model-free environment.

A multi-agent reinforcement learning task is similar to the single-agent approach with the introduction of more agents interacting with one another in an environment. However, reinforcement learning was initially designed for single-agent applications, adding more agents that apply reinforcement learning and interacting simultaneously in a shared environment is often discouraged as the guarantee of convergence is violated [Torrey 2010; Sutton and Barto 1998].

Fortunately, the framework of *Markov games*, derived from game theory is able solve the multi-agent reinforcement learning problem. However, Markov games are not without their limits and complications. The first and foremost problem is the exponential explosion of state space making a large multi-agent system impractical to implement. The exponential state space is due to the fact that agents not only need to coordinate but also take into account the state of the every other agent in the environment. Secondly, solving a game theory problem requires the searching of one equilibrium (global optimum) from multiple equilibria (local optimum) and this is generally difficult. Despite the restrictions, Markov games still provide a good theoretical framework for multi-agent reinforcement learning.

Reynolds's argument that a crowd behaviour is the consequence of interactions between agents and the notion that reinforcement learning is the study of interactions of an agent in an environment, makes it logical to combine these two ideas together. Furthermore, we can extend reinforcement learning in a single-agent environment to a multi-agent environment as crowd simulation is inherently a multi-agent problem. Consequently, we propose to study the multi-agent reinforcement learning paradigm through crowd simulation.

To investigate this proposition, we have formulated the following research question:

> *Given a 2D environment with a set of actors and objectives, can multi-agent reinforcement learning algorithm be used to produce policies that create believable crowd behaviour?*

To answer the question, we present four methods of crowd behaviour simulation where the crowd behaviour being examined is that of a group of agents navigating to the goal state.

The four methods of crowd simulation are rule based, multi-agent Q-learning, joint-state-action Q-learning and joint-state value iteration. In the rule based approach, the simulation agents navigate around the environment using a set of defined rules. In multi-agent Q-learning, we will explore the effectiveness of extending the single-agent Q-learning method to a multi-agent environment. The first two methods are adapted from the work of [Torrey 2010]. In joint-state-action Q-learning, the agents will learn to navigate in the environment using the Q-learning method with a joint-state-action approach. To our knowledge, we believe our version of the joint-state-action Q-learning is a novel approach. Lastly, we will extend the value iteration method from the single-agent environment to a multi-agent environment. The comparisons between these four methods in the environment considered in this thesis have not been considered elsewhere.

This research thesis is organised as follows: The background and related work is covered in chapter 2. The chapter contains a discussion of reinforcement learning background including both single-agent and multi-agent approaches; the discussion also covers methods of crowd simulation. Chapter 3 contains the research question for this research; it also explains the multi-agent environments that are used in the experiments and the method of measuring the effectiveness of the four crowd behaviour simulation methods. Chapters 4 and 5 present the experiments and analyses of the four testing methods. Lastly, chapter 6 concludes the thesis and presents a brief overview of future work.

Chapter 2

Background and Related Work

2.1 Reinforcement Learning

Reinforcement Learning is a machine learning paradigm wherein an agent learns to achieve objectives through experience by means of exploring the world where rewards are associated with each step of the learning phase. Other types of machine learning, e.g. supervised and unsupervised learning, are widely used but they do not learn while interacting with the world [Sutton and Barto 1998].

In general, the framework of reinforcement learning has three basic components: *Policy, Reward function,* and *Value function* [Sutton and Barto 1998]. A policy dictates the agent's decision at any given time. It is a mapping from the observed state to an action when the policy is deterministic. A deterministic policy is also denoted by $\pi(s) \in A = \{a_1, a_2, ..., a_n\}$ where s is a state and A is a set of n actions. On the other hand, a non-deterministic or stochastic policy, $\pi(s, a)$, is the probability of choosing action a in state s.

A reward function is a guiding mechanism for the learning agent when exploring the unknown environment. It is a mapping of states (usually including the current state and next state) and actions to a number (the reward). The reward function can take the form of lookup table which is the simplest representation. The agent's objective is defined through optimising the reward function. The reward that the agent receives is also known as the immediate reward. Note that the term *penalty* is sometimes used to refer to a negative reward.

The highest immediate reward may not be the best action as the long-term reward could be a better option. For example, consider the simple scenario shown in figure 2.1. The green cell $(0, 1)$ is the goal with the associated reward of 10, black cells $(0, 2)$ and $(1, 2)$ are obstacles and they carry -10 penalty, the red cell $(2, 3)$ carries a penalty of -1 and each of the empty cells have a reward of 0. An agent starts at state s $(1, 3)$, as indicated in the figure and tries to navigate to the goal. Consider the two actions *up* and *down*. The action that yields the highest reward is *up*, giving the agent an immediate reward of 0 which is better than -1 if choosing the action *down*. However, an obstacle ahead of that new resultant state will cause a negative impact for which the reward received will be -10. Therefore, the highest immediate reward (greedy approach) does not mean that it is the best option asymptotically.

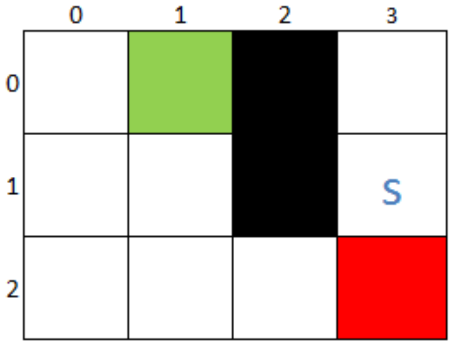


Figure 2.1: Example showing immediate and long-term rewards

Unlike the reward function that specifies an agent's immediate reward, a value function provides a long-term reward. The value of a state is the sum of rewards that an agent can collect from that state by following the specified policy.

One important idea in reinforcement learning is the balance of *exploitation* and *exploration*. In exploitation, the agent follows some previously known policy to obtain higher rewards. However, in exploration, the agent tries to discover a new policy to gain higher future rewards. A method is provided later as an approach to decide when an agent should exploit or explore.

Since an agent interacts with the world, reinforcement learning is suitable for solving problems where the environment is unknown and this is called model-free reinforcement learning. Similarly, if the environment's dynamics are known, it is called model-based reinforcement learning.

2.2 Single Agent Reinforcement Learning

Reinforcement learning attempts to solve problems that can be formulated as a *Markov Decision Process* (MDP). The definitions and equations discussed in this section are taken from Sutton and Barto [1998], unless stated otherwise.

A *finite* MDP is a tuple

$$M = (S, A, P, R),$$

where $S = \{s_1, s_2, ..., s_n\}$ is a finite set of all possible states, $A = \{a_1, a_2, ..., a_m\}$ is a finite set of all actions, $P : S \times A \times S \to [0, 1]$ is the *transition probability function*; for $s, s' \in S$ and $a \in A$, $P(s, a, s')$ is the probability that the agent ends in state s' given that the agent was in state s and took action a. Lastly, $R : S \times A \times S \to \mathbb{R}$ is the *reward function* where $r = R(s, a, s')$ is the real valued reward that the agent receives in state s' given that the agent was in state s and took action a.

Note that throughout this paper, we will focus solely on discrete state spaces, where states can be represented using lookup tables or similar structures. The discrete state space representation allows the agent to interact in the environment over a sequence of discrete time steps, $t = 0, 1, 2, 3, ..., T$. At each time step t, the agent is placed in the state s_t and chooses an action a_t. In the next time step $t + 1$, the agent has moved to the new state s_{t+1} and receives a reward

r_{t+1}. This process ends when $t = T$, which is the final time step. Moreover, an *episode* denotes the sequence of agent-environment interactions from $t = 0$ to $t = T$ or from $t = 0$ until the agent arrives at a *terminal* or *absorbing* state.

The MDP environment is *stationary*. That is, the states will remain unchanged over the duration of the episodes.

The objective of a reinforcement learning agent is to choose actions so that the sum of the discounted rewards it receives over the future is maximised. The sum of discounted future rewards is also known as the *discounted return* and it is defined as

$$\mathcal{R}_t = \sum_{i=0}^{\infty} \gamma^i r_{i+t+1}$$

where $0 \leq \gamma \leq 1$ is the discount rate which affects the weighting of future rewards. If $\gamma = 0$, the learning agent will then disregard future rewards and will be interested in immediate rewards only. As γ approaches 1, the learning agent increasingly takes future rewards into consideration and thus being far-sighted.

The *value function* of a policy π, denoted $V^\pi : S \to \mathbb{R}$, is formally defined in equation 2.1.

$$\begin{aligned} V^\pi(s) &= E_\pi\Big\{\mathcal{R}_t | s_t = s\Big\} \\ &= E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\Big\} \\ &= \sum_a \pi(s,a) \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma V^\pi(s')\Big]. \end{aligned} \tag{2.1}$$

Equation 2.1 is the expected return when starting in s and following π thereafter. V^π is also called the *state-value function* for policy π.

Similarly, we can define the action-value function, denoted $Q^\pi : S \times A \to \mathbb{R}$, to be the expected return when starting in s, taking action a and thereafter following the policy π as

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\Big\{\mathcal{R}_t | s_t = s, a_t = a\Big\} \\ &= E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\Big\} \\ &= \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')\Big]. \end{aligned}$$

This function Q^π is called the *action-value function* for policy π.

A deterministic policy $\pi(s)$ can be determined from any function $V : S \to \mathbb{R}$, provided P is known, is expressed in equation 2.2.

$$\pi(s) = arg\,\underset{a}{max} \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma V(s')\Big],\ \forall s \in S. \tag{2.2}$$

Note that P must be known is equivalent to having a model-based environment. Moreover, a deterministic policy can also be determined from any function $Q : S \times A \to \mathbb{R}$ is shown in equation 2.3.

$$\pi(s) = arg\,\underset{a}{max}\, Q(s,a),\ \forall s \in S. \tag{2.3}$$

Similarly, the missing P indicates the environment is model-free.

A policy π is better than or equal to a policy π' if and only if π's expected return is greater than or equal to the expected return of π'. That is $\pi \geq \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in S$. As such, there is at least one policy that is greater than or equal to all the other policies. This policy is called the *optimal policy*, π^*. The optimal policy, by definition, has a corresponding *optimal state-value function* V^* and is given by

$$V^*(s) = \underset{\pi}{max}\, V^{\pi}(s).$$

Similarly, the optimal policy also has a corresponding *optimal action-value function* Q^* and is defined as

$$Q^*(s,a) = \underset{\pi}{max}\, Q^{\pi}(s,a).$$

Policies constructed from equations 2.2 or 2.3 are also known as *greedy policies* as the action that yields the highest value at each state is chosen. The greedy policy may be the best policy to use at times (given a particular non-optimal value function), however, it is not without restriction. The nature of the action selection method tends to avoid other actions that could result in higher future rewards. That is, if the agent always selects the action that it currently calculates is optimal based on incomplete information, it cannot acquire new information about the alternative states, and can therefore not discover even better policies. This means that the policy will fail to explore other potentially 'good' states. On the other hand, if a policy explores too much, the overall rewards that the learning agent receives will be significantly lower.

The greedy policy does not balance the important concept of *exploitation* (follow states with the highest values) and *exploration* (discover new states) described in section 2.1. The greedy policy does not explore the environment fully. That is, the greedy policy only employs exploitation. To add in exploration, we can introduce a control $\epsilon \in [0, 1]$ to the greedy policy. This new policy is called the ϵ-*greedy* policy where ϵ is the probability that the policy chooses a random action (state exploration), while $1 - \epsilon$ is the probability that the policy chooses an optimal action (state exploitation). In the event of multiple optimal actions (actions that yield the same maximum value), one of the actions is chosen randomly.

2.2.1 Value Iteration

The recursive function described in equation 2.1 is also known as the *Bellman equation* for V^{π}. This equation describes the relationship between the value of a state s, $V(s)$ and the value of its successor state s', $V(s')$ [Bellman and Kalaba 1965; Sutton and Barto 1998]. The optimal value function, V^* satisfies the *Bellman optimality equation*

$$V^*(s) \;=\; \underset{a}{max} \sum_{s'} P(s,a,s')\Big[R(s,a,s') + \gamma V^*(s')\Big].$$

By turning this equation into an update rule to approximate the values of each state, it becomes

$$V_{k+1}(s) \quad = \quad \underset{a}{max} \sum_{s'} P(s,a,s') \Big[R(s,a,s') + \gamma V_k(s') \Big],$$

and it can be shown that for any arbitrary V_0, the sequence V_k will converge to V^*. This update rule can be used to approximate the values of all states through an iterative process. This iterative process stops when the value function changes by only a small amount (i.e. the current value and the new value differs by a small amount). The outline of this method, *Value Iteration*, is described in algorithm 1. Value iteration is a dynamic programming tool for finding optimal policies.

Algorithm 1 Value Iteration

```
Require:
  Initialise $V(s)$ arbitrarily, $\forall\ s \in S$
  repeat
    $\Delta \leftarrow 0$
    for $s \in S$ do
      $v \leftarrow V(s)$
      $V(s) \leftarrow \underset{a}{max} \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V(s')]$
      $\Delta \leftarrow max(\Delta, |v - V(s)|)$
    end for
  until $\Delta < \theta$
```

A deterministic policy, $\pi(s)$, can be determined from the final value function V^*. See equation 2.2. Once again, the presence of $P(s,a,s')$ requires that the value iteration to be implemented in a model-based environment.

2.2.2 Q-Learning Algorithm

While Value Iteration attempts to find an optimal policy from the optimal state-value function, we can also find an optimal policy from the optimal action-value function, Q^*. The value of this function is also known as the Q-value and one of the learning algorithms for learning the optimal action-value function is called the Q-learning algorithm [Watkins and Dayan 1992; Sutton and Barto 1998].

The Q-learning algorithm is an iterative update solution to estimate the optimal action-value function, $Q^*(s,a)$. The update is defined in equation 2.4 where $0 < \alpha \leq 1$ is the learning rate , $0 \leq \gamma \leq 1$ is the discount rate.

$$\begin{aligned} Q(s_t,a_t) &= Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \underset{a}{max}\ Q(s_{t+1},a) - Q^{\pi}(s_t,a_t)] \\ &= (1-\alpha)Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \underset{a}{max}\ Q(s_{t+1},a)] \end{aligned} \tag{2.4}$$

The benefit of applying Q-learning is that the implementation is straightforward and with sufficient learning episodes, the solution will converge. The general outline of Q-learning is in algorithm 2. The agent maintains a record of Q-values. Starting from an initial state s_0, the agent chooses an action a from a policy then observes a reward r and the next state s_1. The agent is then able to update $Q(s_0, a)$ using equation 2.4. Again, starting from s_1, the process continues until an episode ends.

Algorithm 2 The *Q-Learning* Algorithm

```
Require:
  Initialise $Q(s, a)$ arbitrarily, $\forall\ s \in S, a \in A$
  Repeat for specified number of episodes
  repeat
    initialise $s$
    for each step of episode do
      Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy policy)
      Take action $a$ and observe $r$ and $s'$
      $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \underset{a}{max}\, Q(s', a)]$
      $s \leftarrow s'$
    end for
  until episode ends
```

As oppose to the value iteration algorithm, Q-learning can function in a model-free environment.

2.2.3 Example: Path Finding

To put the above two methods into perspective, we can consider a simple grid world (figure 2.2). A grid world is a simple representation of a complex environment where each state is discretised. This is a 5×4 grid world, indexed from 0, where the goal is at $(0, 3)$, the original (starting) position is at $(4, 0)$ and obstacles are at positions $(1, 2), (2, 2), (1, 3), (2, 3)$. The objective is to find an optimal path from the starting position to the goal and to avoid obstacles.

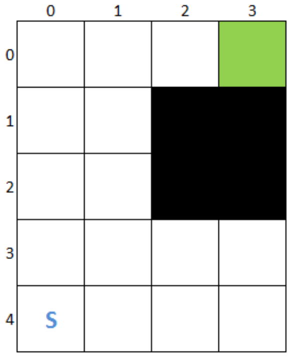


Figure 2.2: Example: path finding

There are four actions available: *up, right, down* and *left* and they are denoted as $0, 1, 2$ and 3 respectively. The rewards are 100 when the agent arrives at the goal state, -10 when the agent moves to an obstacle state or it moves off the grid world (i.e. to an out of bounds state) and lastly, a penalty of -1 for every action taken except when agent is in the goal state. Note that if an action takes the agent to an obstacle state or an out-of-bounds state, the agent will remain in the original state.

Value Iteration

Let $\gamma = 1$ and $\theta = 0.01$. The values $V(s)$ are initialised to 0 for all $s \in S$. Following the value iteration algorithm (refer to Algorithm 1), the algorithm converges after 4 iterations, the values of the states of the first 3 iterations are as follows:

	0	**1**	**2**	**3**
0	-1	-1	99	0
1	-1	-1	-	-
2	-1	-1	-	-
3	-1	-1	-1	-1
4	-1	-1	-1	-2

(a) First iteration

	0	**1**	**2**	**3**
0	-2	98	99	0
1	-2	97	-	-
2	-2	96	-	-
3	-2	95	94	93
4	-2	94	93	92

(b) Second iteration

	0	**1**	**2**	**3**
0	97	98	99	0
1	96	97	-	-
2	95	96	-	-
3	94	95	94	93
4	93	94	93	92

(c) Third iteration

Table 2.1: First 3 iterations of Value Iteration

In practice, we can see that value iteration propagates the values starting from the goal state, and it updates the value of the neighbouring states of the goal state. Note that by convention, the value of a terminal state is usually 0. The values approximated on the fourth iteration are exactly the same as the values found in table 2.1c (the third iteration). Since the algorithm terminates after the fourth iteration, this indicates that the value functions generated from the third and fourth iterations differ by a small amount (in this case, they are the same). Therefore, we can conclude that the values listed in table 2.1c are the final optimal values.

Note that the number of iterations is dependent on the visiting order of the states. In this example, the iteration order starts from state $(0, 0)$, moving from left to right and continuing at the next row until state $(4, 3)$. If the iteration order starts from state $(4, 0)$, moving from left to right and continuing at the previous row until state $(0, 3)$, the number of iterations will be larger.

A policy generated from this optimal value function, using equation 2.2, is shown below:

	0	**1**	**2**	**3**
0	$1_{\rightarrow}$	$1_{\rightarrow}$	$1_{\rightarrow}$	-
1	$0_{\uparrow}$	$0_{\uparrow}$	-	-
2	$0_{\uparrow}$	$0_{\uparrow}$	-	-
3	$0_{\uparrow}$	$0_{\uparrow}$	$3_{\leftarrow}$	$3_{\leftarrow}$
4	$0_{\uparrow}$	$0_{\uparrow}$	$0_{\uparrow}$	$0_{\uparrow}$

Table 2.2: Policy 1 generated from the optimal value function

It is clear that there is more than one optimal policy that can be obtained. For example, from the starting position, there are two possible actions that the agent can take (*up* or *right*). The two actions are optimal because the resulting state from taking the either action has the same value. A random tie breaker will be able to generate a new optimal policy (a policy that traverses a different route to the goal state). Another possible policy is shown in table 2.3.

	0	**1**	**2**	**3**
0	$1_{\rightarrow}$	$1_{\rightarrow}$	$1_{\rightarrow}$	-
1	$1_{\rightarrow}$	$0_{\uparrow}$	-	-
2	$1_{\rightarrow}$	$0_{\uparrow}$	-	-
3	$1_{\rightarrow}$	$0_{\uparrow}$	$3_{\leftarrow}$	$3_{\leftarrow}$
4	$1_{\rightarrow}$	$0_{\uparrow}$	$3_{\leftarrow}$	$3_{\leftarrow}$

Table 2.3: Policy 2 generated from the optimal value function

Q-Learning

Let the parameters α, γ and ϵ be $0.25, 1$ and 0.1 respectively. The $Q(s, a)$ entries are initialised to 0 for all $s \in S$ and $a \in A$. Initially, the agent is placed randomly in the environment. One training/learning episode ends when the agent reaches the goal state and the agent will be placed randomly again for the next episode. The agent will return to its current state if the next state is an obstacle state or an out-of-bounds state. The number of episodes is 1000 and an episode ends when the agent arrives at the goal state. The update rule used is the Q-Learning update from equation 2.4.

Substituting the parameters, the update rule becomes

$$\begin{aligned} Q(s,a) &= (1-\alpha)Q(s,a) + \alpha[r + \gamma \underset{a}{max}\, Q(s',a)] \\ &= 0.75\, Q(s,a) + 0.25[r + \underset{a}{max}\, Q(s',a)]. \end{aligned}$$

The first 5 steps of the first episode, where the actions: *up, right, up, right* and *up* or $0, 1, 0, 1, 0$ are chosen, are tabulated in table 2.4.

Step	s	a	s'	r	$Q(s,a)$
1	$(4,0)$	0	$(3,0)$	-1	-0.25
2	$(3,0)$	1	$(3,1)$	-1	-0.25
3	$(3,1)$	0	$(2,1)$	-1	-0.25
4	$(2,1)$	1	$(2,1)$	-10	-2.5
5	$(2,1)$	0	$(1,1)$	-1	-0.25

Table 2.4: These are the first five steps of the training episodes

Table 2.5 is the final Q-value as represented in a table form.

State/Action	**0**	**1**	**2**	**3**
(0, 0)	97	96	96	94
(0, 1)	96	96	95	93
(0, 2)	95	95	94	92
(1, 0)	86	94	94	92
(1, 1)	86	85	85	83
(2, 0)	98	97	94	93
(2, 1)	97	96	94	92
(3, 0)	96	95	93	91
(3, 1)	96	95	92	85
(3, 2)	87	86	83	82
(3, 3)	99	95	95	93
(4, 0)	98	95	94	93
(4, 1)	97	94	93	92
(4, 2)	88	93	93	82
(4, 3)	88	84	93	82

Table 2.5: The final Q-table

A policy can be generated using the greedy approach by choosing the action that has the highest value in all states (see equation 2.3). The policy is shown in table 2.6.

	0	**1**	**2**	**3**
0	$1_{\rightarrow}$	$1_{\rightarrow}$	$1_{\rightarrow}$	-
1	$0_{\uparrow}$	$0_{\uparrow}$	-	-
2	$0_{\uparrow}$	$0_{\uparrow}$	-	-
3	$0_{\uparrow}$	$0_{\uparrow}$	$3_{\leftarrow}$	$3_{\leftarrow}$
4	$0_{\uparrow}$	$0_{\uparrow}$	$0_{\uparrow}$	$0_{\uparrow}$

Table 2.6: Policy generated from the final Q-function

The policy for finding the goal at state $(0, 3)$ starting from state $(4, 0)$ is 0, 0, 0, 0, 1, 1, 1 or *up, up, up, up, right, right, right.* Furthermore, it is clear that there is a policy for every state that will lead to the goal and this is learned by randomly assigning the agent a starting position in each episode.

Remarks

The two methods are able to compute optimal policies in this example grid world. We can see that the two algorithms function differently in the same environment. The value iteration algorithm functions in a model-based environment as the agent needs to possess a full knowledge of the environment in order to consider all possible actions when visiting a state. By contrast, the Q-learning algorithm only requires a model-free environment and the agents are able to learn the transitions in the environment through trial and error. The Q-learning agent is able to learn its way in an unknown environment without prior knowledge.

2.3 Multi-Agent Machine Learning

In this section, we will discuss the multi-agent reinforcement learning framework. In particular, the extension of the single-agent framework and the overview of game theory leading to Markov games.

2.3.1 Multi-Agent Markov Decision Process

In a multi-agent environment, the simplest method of applying reinforcement learning is to implement n MDP's if there are $n > 1$ agents. Treating each agent as if it is acting alone in the environment makes this model simple to use. This model is also called the multi-agent MDP (MMDP) model [Buşoniu *et al.* 2008]. The MMDP model is one of the four research methods used in this thesis (see chapter 4.2).

Furthermore, we can extend the Q-learning algorithm discussed in section 2.2.2 to a MMDP model (see algorithm 3). Each agent i maintains a Q-table and updates the contents according to the update rule. The procedure is similar to the single-agent Q-learning algorithm.

Algorithm 3 The Multi-Agent *Q-Learning* Algorithm

```
Require:
  Initialise $Q^i(s,a)$ arbitrarily, $\forall\ i \in n, s \in A, a \in A$
  Repeat for specified number of episodes
  repeat
    initialise $s$ for all agents
    for each step of episode do
      for each agent $i$ do
        Choose $a$ from $s$ using policy derived from $Q^i$
        Take action $a$ and observe $r$ and $s'$
        $Q^i(s,a) \leftarrow (1-\alpha)Q^i(s,a) + \alpha[r + \gamma \max_a Q^i(s',a)]$
        $s \leftarrow s'$
      end for
    end for
  until one episode ends
```

Note that, the addition of agents present in the environment is a violation of the MDP requirement that the environment must remain stationary. Introducing multiple agents makes the environment *non-stationary*. That is, each agent causes a non-stationary environment for the other agents while it is navigating in the environment.

Despite the shortfall of losing theoretical guarantees, multi-agent Q-learning in MMDP models is able to produce significant results [Torrey 2010; Tesauro and Kephart 2002].

2.3.2 Game Theory

This section contains the literature review on game theory. This section is included as game theory is the foundation of the Markov Game model for multi-agent reinforcement learning discussed in the next section.

Game theory is the mathematical study of interactions (*decision-making*) among two or more agents [Shoham and Leyton-Brown 2008; Osborne 2004]. Formally, a simple game (*strategic game*) consists of

- a set of *players* ($P = \{p_1, p_2, ..., p_n\}$)
- for each player p_i, a set of *actions* A_i with m actions
- for each player p_i, a *utility* or *payoff* function u_i

A utility or payoff function is a description of all possible outcomes (by taking actions) denoted by some numerical values. The utility function expresses a player's preference to all possible outcomes in a game. The utility function for player i can be represented as

$$u_i(a_1, a_2, ..., a_i, ..., a_m).$$

To simplify, in a two-player two-action game, $u_i(a_1, a_2)$ denotes the payoff that player i receives if player 1 chooses action a_1 and player 2 chooses action a_2. Generally, the utility function of a game can be tabulated for ease of reference. Note that the terms *player* and *agent* are interchangeable, they denote an entity in an intelligent system.

This form of strategic game is also known as a *normal form game* where players simultaneously select an action and it is done so once off. In *repeated games*, players play the games repeatedly. Repeated games are also known as *stateless games* as there is no state transition.

An *action profile* is a set of all players' actions or a collection of strategies. Player i's strategy $\sigma_i : A_i \rightarrow [0, 1]$ is a probability distribution over A_i. A *pure strategy* is a strategy where the player always chooses the same action, $\sigma_i(a) = 1$ where $a \in A_i$ and 0 for all other actions. A non pure strategy is called *mixed strategy*.

Types of Games

In general, games can be classified by their utility functions into three types, namely *purely cooperative*, *purely competitive* and *mixed*. In purely cooperative games, all players' utility functions are the same ($u_1 = u_2 = ... = u_n$). In this scenario, the players have a common goal and they interact with each other in a collaborative manner. However, in purely competitive games or zero-sum games, the total utility functions add to zero ($u_1 + u_2 + ... + u_n = 0$). The last type of game is neither fully cooperative nor competitive, and in this game, all players have different utility functions ($u_1 + u_2 + ... + u_n = R$). This type of game is called a *mixed game* [Buşoniu *et al.* 2010; Bowling and Veloso 2000]. Mixed and purely cooperative games are also known as *general-sum games*.

Example: Prisoner's Dilemma

To put the definitions into perspective, one well illustrated example of game theory is the *Prisoner's Dilemma* problem, a cooperative game. The problem states that two criminals are arrested. Each prisoner is in solitary confinement which means they are not allowed to communicate with each other. However, there isn't enough evidence to convict the pair but only one of the two. The two are now faced with two choices, confess or deny the charges. If they both confess, they will each get a lesser sentence, say 1 year. If one confesses and the other denies, then they will get 0 and 3 years respectively and vice versa as the confessor will testify against the other. If they both deny, they will both get 2 years of sentencing.

In this problem, the players are the two suspects, their actions are confess and deny. The utility function for the players consist of the number of years behind bars with respect to the actions they have chosen.[1] The utility function that represents this game is shown in table 2.7 where suspect 1's payoffs are given by the first element of the utility pair and suspect 2's payoffs are given by the second element of the utility pair.

		Suspect 1	
		Confess	Deny
Suspect 2	Confess	-1,-1	-3,0
	Deny	0,-3	-2,-2

Table 2.7: The *Prisoner's Dilemma* - utility function

The *Prisoner's Dilemma* illustrates a scenario where cooperation gives the best result. See the section on *Nash Equilibrium* below for more discussion.

The Coordination Game

An example of a pure cooperative game, is the *Coordination* game involving players that have a common interest. Two players have the same utility function (table 2.8) for the task of choosing to go left or right. This simple problem shows that the players have to coordinate to get some optimal payoff or risk the chance to get 0 payoff.

		Player 1	
		Left	Right
Player 2	Left	5,5	0,0
	Right	0,0	2,2

Table 2.8: The *Coordination Game* - utility function

Example: Bach or Stravinsky?

The third classic game theory example is the *Bach or Stravinsky* (BoS) or the *Battle of the Sexes* (as it was formally known). Two players decide to go to a concert together. Player 1 prefers Bach while player 2 prefers Stravinsky. The utility function is shown in table 2.9.

		Player 1	
		Bach	Stravinsky
Player 2	Bach	2,1	0,0
	Stravinsky	0,0	1,2

Table 2.9: *Bach or Stravinsky?* - utility function

This example illustrates that the players agree to cooperate (a cooperation game), however, they disagree about the best outcome.

[1] Usually the value of a payoff signifies a gain (positive payoff) or a loss (negative payoff).

Example: Matching Pennies

This game is a simplification of the *Rock, Paper, Scissors* game. Two players each have a coin and they need to decide whether to show their coins to be head or tail. If the coins show the same side, player 1 gains a point from player 2; if the coins show different sides, player 2 gains a point from player 1. Unlike the previous two examples, this game is conflictual or competitive where each player wants to take the opposite action of the other. This is a zero-sum game.

		Player 1 (same)	
		Head	Tail
Player 2 (different)	Head	1,-1	-1,1
	Tail	-1,1	1,-1

Table 2.10: *Matching Pennies* - utility function

Example: Chicken

A well studied mixed game is the two-player game called *Chicken. Chicken* is a game where two drivers are approaching from opposite direction in a straight and narrow road. The ‘chicken’ is the driver who backs up so that the other driver can drive pass. In this case, both drivers will try not to be the chicken, however, if both drivers are not willing to be the chicken could result in a collision (worst outcome). A similar game is the Hawk-Dove game [Osborne and Rubinstein 1994].

To illustrate the *Chicken* game, table 2.11 represents the utility function of the game. There are two possible actions that each player can perform. For example, if player 1 decides to be chicken and player 2 decides to dare, then player 1 gets a reward of 3 and player 2 gets a reward of 9. On the contrary, player 1 gets a reward of 9 and player 2 gets a reward of 3 if their roles are exchanged. It is clear that this game has two pure strategies (a pure strategy clearly defines how a player plays a game), (chicken, dare) and (dare, chicken) as mentioned above with rewards 3 and 9. However, the pure strategies state that each player prefers daring, as opposed to be ‘chicken’. This will lead to the actions (dare, dare) where both players will be in a collision. There is one more equilibrium that can be obtained by applying mixed strategies where the choice of action is influenced by a probabilistic measure. In this game, the third equilibrium is for both players to be ‘chicken’. In this scenario, both players will try to avoid daring each other and if possible, they want to be the only player who dares with some independent probability (say $\frac{1}{3}$ chance of daring). The solution turns out to be ‘chicken’ for both players as both players get the same reward.

		Player 1	
		Chicken	Dare
Player 2	Chicken	7,7	9,3
	Dare	3,9	-10,-10

Table 2.11: *Chicken* - utility function

Nash Equilibrium - Solution to Games

Suppose in a two-player game, the players have agreed on some actions. However, if one player was to deviate from its agreed action to another action where the resulting payoff is better than the others, the situation becomes unstable or infeasible. To put it in context, in the Prisoner's Dilemma problem, if both prisoners choose to deny, they will each get a sentence of 3 years. However, if prisoner 1 were to confess instead of denying, then the payoff will be more favourable towards prisoner 1 which shows the strategy is unstable. To illustrate, the payoffs for both the prisoners are -2 each if they choose to deny. If prisoner 1 deviates his action to confess, the payoffs are changed to 0 and -3 for the prisoner 1 and 2 respectively. This clearly show that prisoner 1 will benefit with such a strategy.

From the above, we can define an action profile to be a *Nash Equilibrium* (equilibria for plural) so that no player can deviate from its strategy to out-perform any other players. In other words, when players have reached the Nash equilibrium, there is no need for any player to deviate its strategy so that it can do better. If Nash equilibrium is achieved, the strategies are then in a steady state [Osborne 2004]. Formally, we can define a to be the action profile, and a_i denotes the action of player i. Then, a'_i is the action selected by player i, where it can equal a_i or differ. Let $a_{\text{-i}}$ be the collection of every players' actions (following a) except i's. Hence, $(a'_i, a_{\text{-}i})$ is the action profile where all players' actions follow a and player i follows a'_i. Therefore, the action profile a^* of a Nash equilibrium in a normal form game is shown in equation 2.5.

$$u_i(a^*) \;\geq\; u_i(a_i, a^*_{\text{-}i}) \quad \text{for every action } a_i \text{ of player } i. \tag{2.5}$$

Finding the Nash equilibrium is an iterative process. Consider the four possible pairs of actions (action profiles) in the Prisoner's Dilemma (see table 2.7). The Nash equilibrium here is the pair (Confess, Confess). According to equation 2.5, the resulting inequalities hold:

$$\begin{aligned} u_1(\text{Confess, Confess}) &\geq u_1(\text{Deny, Confess}) \\ -1 &\geq -3 \end{aligned}$$

and

$$\begin{aligned} u_2(\text{Confess, Confess}) &\geq u_2(\text{Confess, Deny}) \\ -1 &\geq -3 \end{aligned}$$

There is only one Nash equilibrium in the Prisoner's Dilemma, hence it is a unique equilibrium. This equilibrium indicates that no players can change their choice of action without reducing their payoffs. The other three pairs are not Nash equilibria as they do not satisfy equation 2.5. This is because players always have incentives to deviate their action so to achieve better outcome. For example, in the case where both players choose to deny, it is always an incentive for a player to deviate the action to confess and thereby getting a higher payoff (reducing the sentence from 2 years to 0). Furthermore, in the case where one player chooses to confess and the other chooses to deny, it is always beneficial for the denying player to choose to confess (resulting in both players confessing) so as to obtain a higher payoff (reducing the sentence from 3 years to 1).

The process of finding a Nash equilibrium is iterative. During the learning phase, player i tries to find an optimal strategy (π_i) while interacting with other players. Then, a *Nash Equilibrium* is a collection of strategies from all players such that no player can perform better by changing strategies provided that all other players follow the equilibrium.

There are two Nash equilibria in the *BoS* game, namely (Bach, Bach) and (Stravinsky, Stravinsky). Similarly, there are also two Nash equilibria in the *Coordination* game: (Left, Left) and (Right, Right). In *Matching Pennies*, the game has a mixed strategy Nash equilibrium as there is no pure strategy. Lastly, *Chicken* has three Nash equilibria namely: two pure strategies (Chicken, Dare) and (Dare, Chicken) and the *mixed* strategy where players choose Dare with a probability of $\frac{1}{3}$ as discussed above.

It is clear that in a game, the number of Nash equilibra can be none or many. However, the possibility that there are one or more Nash equilibria means that there may exist sub-optimal equilibria and this poses the problem of choosing the optimal equilibrium [Kolokoltsov and Malafeyev 2010].

2.3.3 Markov Game

Markov games or *stochastic games* developed by Shapley [1953] are the extension and generalisation to multi-agent settings of multi-agent *markov decision processes* and *repeated games*. Markov games are the natural platform for solving multi-agent problems using reinforcement learning. This section is a literature review of Markov Games and the methods to solve the problem. The methods are not part of the research methods as they are examples of showing the limitations of Markov Games.

In general, a *Markov game* is a tuple

$$M = (S, A_1, ..., A_n, P, R_1, ..., R_n),$$

where $n > 1$ is the number of agents[2], $S = \{s_1, s_2, ..., s_m\}$ is the unified finite state space, $A_i = \{a_1, a_2, ..., a_k\}$ is the finite action set available for agent i, $P : S \times A_1 \times ... \times A_n \times S \rightarrow [0, 1]$ is the transition probability function and $P(s, a_1, ..., a_n, s')$ is the probability that the agents end in state s' given the agents were in state s and the joint action[3] $a_1, ..., a_n$ or $\vec{a}$ is taken. Lastly, $R_i(s, a_1, ..., a_n, s') : S \times A_1 \times ... \times A_n \times S \rightarrow \mathbb{R}$ is a reward function for agent i that returns a real value r when given any action $\vec{a}$, state s and new state s'.

The *discounted return* with respect to the joint policy $\vec{\pi} = (\pi_1, ..., \pi_n)$ for agent i at time t is now defined as

$$\mathcal{R}^{\vec{\pi}}_{i(t)} = \sum_{j=0}^{\infty} \gamma^j r_{j+t+1}$$

where the rewards r_{j+t+1} are dependent on the reward function R_i. Furthermore, the *state-value function* $V_i^{\vec{\pi}}(s)$ for agent i following the joint policy $\vec{\pi}$ is now defined as

$$\begin{aligned} V_i^{\vec{\pi}}(s) &= E_{\vec{\pi}}\Big\{\mathcal{R}^{\vec{\pi}}_{i(t)} | s_t = s\Big\} \\ &= E_{\vec{\pi}}\Big\{\sum_{j=0}^{\infty} \gamma^j r_{j+t+1} | s_t = s\Big\}. \end{aligned}$$

[2] Note that if $n = 1$, a Markov game will be reduced to an MDP

[3] A joint action is a set of actions taken by agents at any given step

Similarly, the *action-value function* $Q^{\vec{\pi}}_{i(t)}(s, \vec{a})$ for agent i is defined w.r.t $\vec{\pi}$ as

$$Q^{\vec{\pi}}_{i(t)}(s, \vec{a}) \;\; = \;\; E_{\vec{\pi}}\left\{ \sum_{j=0}^{\infty} \gamma^j r_{j+t+1} | s_t = s, \vec{a}_t = \vec{a} \right\}.$$

Moreover, let $\vec{\pi}_{-i}$ be a joint policy excluding π_i. Then, in an n player stochastic game, the Nash equilibrium is a collection of n policies $\vec{\pi}^*$ (strategies) such that

$$V_i^{\vec{\pi}^*}(s) \geq V_i^{\vec{\pi}^*_{-i}, \pi_i}(s) \, \forall \, \pi_i \in \Pi_i, s \in S$$

where Π_i is the set of policies available to agent i.

Let n be the number of agents, $|S|$ be the number of states and $|A|$ be the number of actions (assuming all agents have the same number of actions, $|A_1| = |A_2| = ... = |A_n|$). The state-action space required for Q-learning in a MDP model is $\mathcal{O}(|S|\cdot|A|)$; the state-action space required for multi-agent Q-learning in a MMDP model is $\mathcal{O}(n\cdot|S|\cdot|A|)$ as agents need to maintain n Q-tables. However, the state-action space required for a Markov game is $\mathcal{O}(n\cdot|S|^n\cdot|A|^n)$. The presence of joint state and joint action cause the complexity of the state-action space to be exponential instead of linear. It is, therefore, very important to keep in mind that joint actions, or even joint state implementation is expensive.

Consider the single-agent Q-learning update rule (equation 2.4). The Q-learning update rule in a Markov game model for agent i can be extended as

$$\begin{aligned} Q_i^{\pi}(s_t, \vec{a}_t) &= Q_i^{\pi}(s_t, \vec{a}_t) + \alpha[r_{i(t+1)} + \gamma V_i^{\pi}(s_{t+1}) - Q_i^{\pi}(s_t, \vec{a}_t)] \\ &= (1 - \alpha) \, Q_i^{\pi}(s_t, \vec{a}_t) + \alpha[r_{i(t+1)} + \gamma V_i^{\pi}(s_{t+1})]. \end{aligned}$$

In single-agent Q-learning, the update takes the highest value, $\underset{\vec{a}}{max} \; Q(s_{t+1}, \vec{a})$, with respect to the actions available in the next state s_{t+1}, as an estimate of $V_i^{\pi}(s_{t+1})$ under the *greedy* policy. However, in multi-agent Q-learning under a Markov game model, the agents must take into account other agents' actions as well.

Most multi-agent reinforcement learning algorithms that follow the model of the Markov game focus on how to choose the best optimal value $V_i^{\pi}(s')$. In general, these algorithms follow the template in algorithm 4. The algorithm follows the usual Q-learning update rule (algorithm 2) with the addition of multiple agents, together with joint action. The notable difference is the way of estimating $V_i^{\pi}(s')$.

NashQ-Learning

Hu and Wellman [2003] have extended the concept of Nash equilibrium to Q-learning called *NashQ*-learning. As mentioned above, the key factor is to choose a function that represents the best optimal future value (next state s'), $V_i(s')$. Under the joint-action condition, *NashQ*-learning redefines the Q-function to be the expected sum of discounted rewards, where all agents follow unique Nash equilibrium strategies from the future. The *NashQ*-function for agent i is

$$Q_i^*(s, \vec{a}) = r_i + \gamma \sum_{s' \in S} p(s'|s, \vec{a}) V_i^{\vec{\pi}^*}(s'),$$

Algorithm 4 The General Multi-Agent *Q-Learning* Algorithm

```
Require:
  $Q_i(s, \vec{a}) = 0 \,\forall\, i \in n, s \in S^n, \vec{a} \in A$
  Initialise state $s$
  Repeat for specified number of episodes
  repeat
    for all agents $i$ do
      Select action $a_i$ using policy derived from $Q_i$
    end for
    Execute joint action $\vec{a} = (a_1, ..., a_n)$
    Observe new state $s'$ and rewards $r_i$
    for all agents $i$ do
      $Q_i(s, \vec{a}) \leftarrow Q_i(s, \vec{a}) + \alpha[r_i + \gamma V_i(s') - Q_i(s, \vec{a})]$ where some estimated $V_i(s')$ is used.
    end for
  until termination condition
```

where $\vec{\pi}^*$ is the joint Nash equilibrium strategy, r_i is agent i's reward, and $V_i^{\vec{\pi}^*}(s')$ is agent i's total discounted reward from state s' provided that agents follow $\vec{\pi}^*$.

The extended Q-learning will be changed to update future Nash equilibrium rewards instead of each agent's own maximum reward. Initially ($t = 0$), agent i starts learning its Q-*value* by guessing arbitrarily. A common approach is to initialise all $Q_i(s, \vec{a}) = 0$ for all $s \in S$ and $\vec{a} \in \vec{A}$. Then, at time t, agent i observes the current state, and perform its action. Besides observing its reward, agent i also inspects actions and rewards taken by all other agents. Instead of searching for the maximum Q-value when taking action a in state s_{t+1} in single-agent Q-learning, agent i needs to calculate the future Nash Equilibrium payoffs, $u_i(\pi^1(s')...\pi^n(s'))$, as well as an approximation of other agents' immediate rewards and actions, $Q_{i(t)}(s', \vec{a})$. The Nash Equilibrium is calculated using the *Lemke-Howson* method [Cottle *et al.* 1992].

The extended Q-learning update rule now becomes

$$Q_i(s_t, \vec{a}) \leftarrow Q_i(s_t, \vec{a}) + \alpha[r_{i(t+1)} + \gamma \; NashQ_i(s_{t+1}, \vec{a})],$$

where $0 < \alpha \leq 1$ is the learning rate, $0 \leq \gamma \leq 1$ is the discount rate and

$$NashQ_i(s, \vec{a}) = u_i(\pi_1(s)...\pi_n(s')).Q_i(s, \vec{a}),$$

is the Nash equilibrium payoffs under the joint policy. Following algorithm 4, the value function $V_i(s')$ is defined to be $NashQ_i(s', \vec{a})$.

The disadvantage of *NashQ*-learning is the existence of multiple Nash equilibria. Sub-optimal Nash equilibria form part of the solution, however, choosing one of them could result in a lower average payoff. Consider the *Coordination* game in table 2.8 where there are two Nash equilibria (Left, Left) and (Right, Right). It is clear that (Right, Right) is the sub-optimal Nash equilibrium, if both players agree on this joint action, the payoff will be lower than (Left, Left). Generally, the Nash equilibrium method is implemented in two-player games where the worst case complexity is exponential.

Currently, the complexity of finding Nash equilibrium is PPAD-complete even in a two-player game [Daskalakis *et al.* 2006]. PPAD-complete is a sub-class of *FNP* (*function-NP*), an extension of the decision problem class *NP*. It shows that currently there is no efficient solution for computing a specific Nash equilibrium, even though it is guaranteed that solutions (many Nash equilibria) exist.

Correlated Q-Learning

Mixed strategies do not provide *fairness* as players are always trying to deviate to perform actions that could maximise their rewards as shown in the *Chicken* game on page 16. Fairness is a social phenomena where players not only are concerned with their payoffs but also the payoffs of the others. By introducing a judge or mediator in the game, players follow the mediator's decision and understand that other players also follow the same decision and never deviate. The resultant expected reward payoff from this correlated strategy is better than using mixed strategy [Osborne and Rubinstein 1994]. An equilibrium obtained using this correlated strategy is known as a *Correlated Equilibrium*.

Consider an n-player game with strategy π_i for player i and $\Pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_n$ [Papadimitriou and Roughgarden 2004]. Define x to be the distribution on Π and $\bar{\pi} \in \Pi_{\text{-}i}$, then, let $x^i_{\kappa,\bar{\pi}}$ be the probability that player i chooses strategy κ while other players choose $\bar{\pi}$. Furthermore, $u^i_{\kappa,\bar{\pi}}$ is the utility (reward) of player i for taking the strategy κ while other players take $\bar{\pi}$. Then x is a correlated equilibrium, provided that the expected utility of player i taking strategy κ is no smaller than that of playing another strategy τ. The correlated equilibrium is expressed in equation 2.6.

$$\begin{aligned} \sum_{\bar{\pi}\in\Pi_{\text{-}i}} u^i_{\kappa,\bar{\pi}}\, x_{\kappa,\bar{\pi}} &\geq \sum_{\bar{\pi}\in\Pi_{\text{-}i}} u^i_{\tau,\bar{\pi}}\, x_{\kappa,\bar{\pi}} \quad \forall\, \kappa,\tau \in \Pi_i \\ \sum_{\bar{\pi}\in\Pi_{\text{-}i}} u^i_{\kappa,\bar{\pi}}\, x_{\kappa,\bar{\pi}} &- \sum_{\bar{\pi}\in\Pi_{\text{-}i}} u^i_{\tau,\bar{\pi}}\, x_{\kappa,\bar{\pi}} \geq 0. \end{aligned} \tag{2.6}$$

Equation 2.6 can also be rewritten in the form of a linear program:

$$\begin{aligned} &\sum_{\bar{\pi}\in\Pi_{\text{-}i}} \left[u^i_{\kappa,\bar{\pi}} - u^i_{\tau,\bar{\pi}} \right] x_{\kappa,\bar{\pi}} \geq 0, \\ &x_\pi \geq 0 \quad \forall \pi \in \Pi, \\ &\sum_{\pi\in\Pi} x_\pi = 1. \end{aligned}$$

The inequalities satisfy the problem-constraints and non-negative values conditions. The objective function to be maximised by this linear program will be players' utility. Consider the *Chicken* game with utility shown in table 2.11. Suppose a mediator randomly chooses the three joint actions (Dare, Chicken), (Chicken, Dare) and (Chicken, Chicken) with equal probability [Papadimitriou and Roughgarden 2004]. This means that there is a $\frac{1}{3}$ probability of randomly choosing the three joint actions. The joint action (Dare, Dare) is not chosen because of the unfavourable consequence. Moreover, let Dare $= D$ and Chicken $= C$. We can construct the probabilities into constraints

$$\begin{aligned} x_{CC} + x_{CD} + x_{DC} + x_{DD} &= \tfrac{1}{3} + \tfrac{1}{3} + \tfrac{1}{3} + 0 \\ &= 1 \\ x_{CC}, x_{CD}, x_{DC}, x_{DD} &\geq 0, \end{aligned}$$

and from inequality 2.6, the first corresponding constraint (from player 1) is

$$[u^1_{CC} - u^1_{DC}]x_{CC} + [u^1_{CD} - u^1_{DD}]x_{DD} \geq 0$$

$$(7-9)x_{CC} + (3-(-10))x_{DD} \geq 0$$

$$-2x_{CC} + 13x_{DD} \geq 0.$$

Similarly, the next constraint can be reduced to

$$2x_{CC} - 13x_{DD} \geq 0.$$

Note that there are four rational constraints in this game. However, since the game is symmetric, player 2's rational constraints will be identical to player 1's. In this problem, one possible objective function is to maximise the expected sum of all player's utilities over the total distribution,

$$\begin{aligned} \text{maximise:} \quad & \sum_{\pi \in \Pi} \left[x_\pi \sum_{i=0}^{n} u^i_\pi \right] \\ = \quad & 14x_{CC} + 12x_{CD} + 12x_{DC} - 20x_{DD} \end{aligned}$$

Greenwald and Hall [2003] has further extended the multi-agent Q-learning method using the correlated equilibrium. A correlated equilibrium is more general than a Nash equilibrium. A Nash equilibrium is a correlated equilibrium but a correlated equilibrium is not necessary a Nash equilibrium. In general, correlated equilibria are easier to compute than Nash equilibria. A Nash equilibrium is a distribution over each player's action, while a correlated equilibrium is a distribution over agents' joint action space.

Following the template in algorithm 4, the value function $V_i(s')$ is now defined as player i's reward corresponding to some correlated equilibria estimated by other player's rewards. That is,

$$V_i(s') = CE_i(Q_i(s, \vec{a})) = \sum_{\vec{a} \in A} \pi^*(\vec{a}) Q_i(s, \vec{a}),$$

where π^* is a correlated equilibrium chosen from many equilibria using four selection mechanisms. The four selection mechanisms are used for selecting the objective function. The selection mechanisms are to maximise: $1)$ the sum of the players' rewards, $2)$ the minimum of the players' rewards, $3)$ the maximum of the players' rewards and $4)$ the maximum of each player i's rewards.

Though computing correlated equilibria requires less work, the existence of multiple equilibria coupled with the fact that the state-action space is exponential poses a critical problem in large scale environments.

2.3.4 Coordinating *Q*-Learning

The use of MMDP model in the multi-agent environment violates the requirement of the transition probability being stationary in an MDP, thus MMDP learning may fail to converge; and the exponential cost of implementing the Markov game in a large scale environment is, therefore, not practical. However, Nowé *et al.* [2012] observed that by making use of the benefits of the two extreme methods (coordinate using multi-agent techniques and act independently using single-agent techniques), it is possible for agents to observe their surrounding states while updating. Essentially, an agent observes if there is an interaction needed between other agents in some states. In such a case, the agent will use multi-agent techniques to coordinate, or act independently otherwise. This learning algorithm is closely related to the research method discussed in chapter 5.

The aforementioned technique is called *Coordinating Q-Learning* or *CQ-learning* and it follows the concept of learning on multiple levels (see figure 2.3). An agent in this framework identifies the states where it needs to take other agents into consideration (so as to coordinate) when it is choosing its action. The primary goal in this framework is to reduce the joint action space in multi-agent environments.

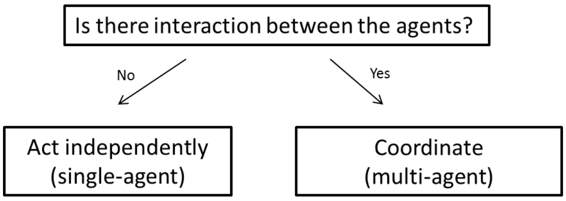

Is there interaction between the agents?
No
Yes
Act independently (single-agent)
Coordinate (multi-agent)

Figure 2.3: Framework for learning on two levels

The basis of *CQ*-learning is to assume that all agents have a learned policy. This policy tells the agent the expected reward of taking an action in some states. This is equivalent to having the agents trained an MDP individually (as if other agents are not present) and retain the Q-function as the baseline for this algorithm. This baseline serves the purpose for agents to identify the states where coordination is needed.

Suppose agents arrive at a state where they are influencing one another, and so the resulting expected reward received by agents, and the observed reward will be inconsistent. The Student t-test, is used to detect if there is a discrepancy between the rewards received when agents are alone (expected), and the rewards received while agents are learning together (observed) in some states. There are two steps to finding coordinating states. Firstly, the statistics detect discrepancies in the observed rewards against expected rewards. Then, it attempts to identify the agents that have contributed to the discrepancies. The second step is done by collecting new samples (joint state action pairs) from a joint state space. The joint state action pairs are tested against the observed rewards.

Furthermore, when selecting an action, agents need to verify that if the current local state is the state which is causing the discrepancy, otherwise agents choose actions according to the learned policy (this tells the agents to act as if they were alone). After identifying the local state, agents need to observe the global state (joint state space) to determine if the state information of other agents is the same when the discrepancy was detected. If this is the case, then agents choose actions according to the global state information.

Agents in *CQ*-learning update their Q-values using a joint state Q-value function. That is, for agent i, the joint state Q-value function is denoted as $Q_i^j(js, a_i)$ where js is the joint state information. The Q-learning update rule for agent i is now defined as

$$Q_i^j(js, a_i) \leftarrow (1 - \alpha_t)Q_i^j(js, a_i) + \alpha_t[r(js, a_k) + \gamma \underset{a}{max}\, Q(s'_k, a)].$$

The state-action space in *CQ*-learning could be reduced only if the environment is sparse. If the environment is dense (the agents are placed in the environment compactly), then the agents would have to coordinate almost at all the states as each step of learning there will be conflict or collision. The *CQ*-learning algorithm is described in Appendix A on page 58.

The *CQ*-learning method is the foundation of our novel approach, the joint-state-action Q-learning in chapter 5.

2.4 Crowd Simulation

The sections covered so far contain the literature on single-agent and multi-agent reinforcement learning and their respective methods to solve the problems. The goal of this thesis is to study multi-agent reinforcement learning using crowd simulation as a domain. This section introduces the concept of crowd simulation and the current techniques in crowd simulation.

Crowd simulation is the process of reproducing interactions, and movements of a large number of entities in an environment. Crowd simulation can be found in a number of fields such as computer graphics, safety science, architecture, physics and sociology. In these fields, the main concern is to simulate a crowd with a collection of entities. There are two areas in crowd simulation: focusing on the *realism* of simulated crowd behaviour and achieving *visualisation* of high-quality [Thalmann and Musse 2007].

Reynolds [1987]'s work had revolutionised the task of animating and simulating crowds. The main idea was based on observations from the behaviour of flocks of flying birds. It was realised that each bird in a flock follows some common behaviour, that is, it maintains its position in the flock (to stay close), and avoids collisions. Therefore, the complex motion formed by a flock of birds was just a collection of actions taken by each bird perceived in its entirety.

Reynolds simulated the actions of these birds, naming them *boids* for the purposes of his simulation. By allowing interactions between simple boids individually, a more complex fluid-like, flocking behaviour can be accomplished. There are only three simple rules that govern the behaviour of each boid : 1) avoid collisions with nearby boids, 2) match velocity with nearby boids, and 3) stay close to nearby boids. The simulated results confirmed that the boids were able to portray complex flock-like behaviours.

Extending Reynolds's work, Tu and Terzopoulos [1994] had simulated schools of fishes. To achieve realistic behaviours for fishes, several factors need to be considered. The fishes' physical bodies are modelled as spring-mass systems representing the muscle movements through contraction. The fishes' mobility is modelled after hydrodynamics to simulate the visualisation of volume displacement as the fishes swim. Furthermore, the fishes are equipped with vision and temperature sensing models. The behaviours of the fishes are much more complicated than birds. The behaviours (dependent on the internal state) include hunger, libido, fear, and some habit conditions. The actions corresponding to the behaviours are avoiding obstacles (including other fishes), eating food, mating, leaving, wandering, escaping and schooling. The fishes are grouped into predators and prey.

Producing a realistic and *believable* simulated human is a complex task, as human beings are complex creatures in nature. To animate a seemingly believable humanoid (a reference to Reynold's term boid, a simulated human-like object), an animator has to manually adjust every detail over the course of the animation period. Extending to simulating a crowd of humanoids (say ten humanoids), the workload of the animator has now increased more than ten fold. Besides giving instructions to each individual humanoid, there is a need to take into account interactions between humanoids, particularly collisions. Note that the instructions given to each individual humanoid must be unique so that humanoids do not appear *predictable*.

There are two distinct views on the crowd simulation models: macroscopic and microscopic. Macroscopic models treat the crowd as a whole (flow and fluid characteristics of moving crowd), while microscopic models examine and analyse the individual behaviour and interactions between the entities.

2.4.1 Macroscopic Model

Fluid Dynamics

In the macroscopic view of simulating a crowd, the model considers the individuals as a collection. One such model uses fluid dynamics [Kachroo *et al.* 2008]. Consider the one-dimensional problem of modelling car traffic flow as continuous flow. Let p be the density and v be the velocity. Given an initial density p_0, an initial velocity v_0, and a point x, the flow (with time t) is defined as

$$f(p, v) = p(x, t)v(x, t)$$

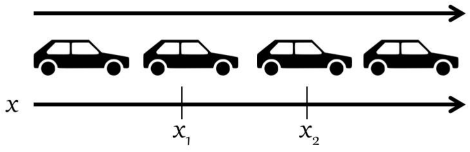


Figure 2.4: Traffic flow

Given a segment x_1 and x_2 (figure 2.4), the number of cars within $[x_1, x_2]$ at time t is shown in equation 2.7.

$$N = \int_{x_1}^{x_2} p(x, t)dx. \tag{2.7}$$

Furthermore, suppose no cars were destroyed, and the change of the number of cars is caused by cars entering and leaving the boundaries only. We can define the rate of change of the number of cars as expressed in equation 2.8.

$$\frac{dN}{dt} = f_{in}(p, v) - f_{out}(p, v). \tag{2.8}$$

Combining equations 2.7 and 2.8, we get

$$\frac{d}{dt}\int_{x_1}^{x_2} p(x, t)dx = f_{in}(p, v) - f_{out}(p, v).$$

This equation represents the change in number of cars due to the flows at the two boundaries. Suppose the boundary points are independent, then

$$\frac{\partial}{\partial t}\int_{x_1}^{x_2} p(x,t)dx = f_{in}(p,v) - f_{out}(p,v),$$

and the change in number of cars with respect to distance is

$$f_{in}(p,v) - f_{out}(p,v) = -\int_{x_1}^{x_2} \frac{\partial}{\partial x} f(p,v)dx,$$

then combining the two, we can show that the law of conservation holds, that is,

$$\int_{x_1}^{x_2} \left[\frac{\partial}{\partial t}p(x,t) + \frac{\partial}{\partial x}f(p,v)\right] dx = 0.$$

If $p(x,t)$ and $f(p,v)$ are smooth, this equation can be deduced to

$$\frac{\partial}{\partial t}p(x,t) + \frac{\partial}{\partial x}f(p,v) = 0.$$

Extending to crowd flow, the domain is now two-dimensional. The initial velocity along the y-axis is denoted by u_0 and the two velocities (v,u) direct the flow of the crowd in any direction. The density function is now additionally a function of y, and is denoted by $p(x,y,t)$. Then, the flow rates f and g along the two axes are

$$\begin{aligned} f(p,v) &= p(x,y,t)v(x,t) \\ g(p,u) &= p(x,y,t)u(y,t) \end{aligned}$$

Similarly, the law of conservation is maintained

$$\frac{\partial}{\partial t}p(x,y,t) + \frac{\partial}{\partial x}f(p,v) + \frac{\partial}{\partial y}g(p,u) = 0.$$

Given initial density and velocities, the future density can be predicted. Therefore, it is important to choose density dependent velocity functions that simulate the model behaviour.

2.4.2 Microscopic Models

Particle Motion Model

The particle motion model discussed by Heïgéas *et al.* [2010] is used to simulate crowd behaviours. The particles were modelled using geometrical estimates to detect collisions with some physical properties. In the model, the moving crowd (people) are defined as two-dimensional particles, and obstacles (non-mobile objects like buildings) are defined as fixed particles.

Interactions among the crowd, and ultimately the collective behaviour can be represented as obstacle avoidance by the crowd. The interacting forces between two particles are defined to be

$$\vec{F} = \begin{cases} (K_1 D + Z_1 V)\vec{u} & \text{if } D < D_1 \\ (K_{i+1} D + Z_{i+1} V)\vec{u} & \text{if } D_i < D < D_{i+1} \quad \text{for } i = 1,2 \\ 0 & \text{if } D_3 < D \end{cases},$$

where D is the distance between the particles, K is the stiffness, V is the norm relative velocity between the two particles, Z is the viscosity and $\vec{u}$ is the unit vector between the particles. The stiffness of a body (particle) is the measure of rigidity, the resistance applied by some force. The viscosity of fluid (pertaining to flow) is the measure of the fluid's resistance asserted by some pressure. The three thresholds (D_1, D_2 and D_3) regulate the explicit behaviour within each divided zone between two particles (fixed and non-mobile particles). See table 2.12. Note that there will be no interacting forces when the particles are separated by a distance greater than D_3.

Zones	**Distance**	**Stiffness and Viscosity**	**Threshold Name**
A ($D < D_1$)	Closest	High	Impenetrable Zone
B ($D_1 < D < D_2$)	Medium	Medium	Avoidance Zone
C ($D_2 < D < D_3$)	Furtherest	Low	Anticipation Zone

Table 2.12: Zone classification

In the impenetrable zone (zone A), there is a need for collision detection and avoidance. Hence, the high stiffness value will generate high repulsive force, coupled with the high value of viscosity, the velocity behaviour can be regulated. The high stiffness and viscosity values will produce lower velocity. Similarly, there will be low stiffness and viscosity values in the anticipation zone, as collision detection and avoidance produce lesser impact. The argument in avoidance zone will be the same.

The simulated crowd was unrealistic as they tended to move past obstacles along similar paths, thereby producing predictable behaviours. To remedy this, the thresholds were modified so that the stiffness and viscosity were uniformly distributed between some minimum and maximum values.

Helbing's Model

[Helbing *et al.* 2000]'s model (Helbing's model in short) is a combination of sociopsychology and physical forces found in humans. This combined model is also called the social force model and it had been able to capture the behaviours of crowd evacuation simulated using a force model.

In Helbing's model, there are N pedestrians where each pedestrian i has a mass m_i. Each pedestrian has a desired (initial) velocity v_i^0 in a direction e_i^0 over a time interval τ_i, while observing its instantaneous velocity v_i. Pedestrian i keeps a velocity-dependent distance from other pedestrians j and walls W, using interaction forces $\mathbf{f}_{ij}$ and $\mathbf{f}_{iW}$. Formally, given time t, Helbing's model can be expressed as the change in velocity, using the acceleration equation

$$m_i \frac{d}{dt} v_i = m_i \frac{v_i^0(t) e_i^0(t) - v_i(t)}{\tau_i} + \sum_{j \neq i} \mathbf{f}_{ij} + \sum_{W} \mathbf{f}_{iW}.$$

Suppose A_i and B_i are constants, the *repulsive interaction force* describes how two pedestrians i and j would stay away from each other. Let the distance between two pedestrians' centres of mass be $d_{ij} = ||r_i - r_j||$, where r_i is the change of position of pedestrian i, and $\mathbf{n}_{ij} = (n_{ij}^1, n_{ij}^2) = \frac{r_i - r_j}{d_{ij}}$ be the normalised vector (perpendicular) pointing from pedestrian j to pedestrian i. Then, the repulsive interaction force is $A_i e^{(r_{ij}-d_{ij})/B_i}\mathbf{n}_{ij}$. The two pedestrians will come in contact (touching each other) when the sum $r_{ij} = r_i + r_j$ is greater than their distance d_{ij}. Furthermore, there are two additional forces that can be observed, a body force counteracting the body compression $k(r_{ij} - d_{ij})\mathbf{n}_{ij}$, and a sliding friction force disrupting the relative tangential motion $\mathsf{k}(r_{ij} - d_{ij})\triangle v^t_{ji}\mathbf{t}_{ij}$, where $\mathbf{t}_{ij} = (-n^2_{ij}, n^1_{ij}) = \frac{-r_j - r_i}{d_{ij}}$ is the direction of the tangent, and $\triangle \mathbf{v}^t_{ji} = (\mathbf{v}_j - \mathbf{v}_i)\mathbf{t}_{ij}$ is the tangential velocity difference. Putting it all together, we get

$$\mathbf{f}_{ij} = \left[A_i e^{\frac{r_{ij}-d_{ij}}{B_i}} + kg(r_{ij} - d_{ij})\right]\mathbf{n}_{ij} + \mathsf{k}g(r_{ij} - d_{ij})\triangle v^t_{ji}\mathbf{t}_{ij},$$

where $g(x)$ is zero if $(r_{ij} < d_{ij})$, and it is $r_{ij} - d_{ij}$ otherwise. k and k are some large constants.

Similarly, denote d_{iW} to be the distance to the wall, n_{iW} to be the direction perpendicular to the wall, and $\mathbf{t}_{iW}$ to be the direction tangential to the wall. The interaction force between a pedestrian i and wall W is

$$\mathbf{f}_{iW} = \left[A_i e^{\frac{r_{ij}-d_{iW}}{B_i}} + kg(r_{ij} - d_{iW})\right]\mathbf{n}_{iW} - \mathsf{k}g(r_{iW} - d_{iW})(\mathbf{v}_i \cdot \mathbf{t}_{iW})\mathbf{t}_{iW}$$

Helbing's model was able to simulate realistic behaviour of *pushing* in an evacuation scenario, through the use of repulsion and attraction forces. However, in high density cases, the simulated pedestrians appeared to be 'shaking' or 'vibrating', as there are too many inter-personal frictions when pedestrians are too close together in a large volume. Nevertheless, the simulated result is similar to the empirical observations of escape panics (evacuation events).

Rule-Based Model

Reynold's boids is an example of a rule-based model. In addition to the three stated rules at the beginning of this section, Reynolds [1999] further discussed a more general crowd behaviour called *steering*. The steering behaviour further enhances the original behaviour model by adding more behavioural rules. The rules are listed in table 2.13.

seek and flee	crowd path following
pursue and evade	leader following
wander	unaligned collision avoidance
arrive	queuing
obstacle avoidance	flocking
containment	
wall following	
path following	
flow field following steer behaviour	

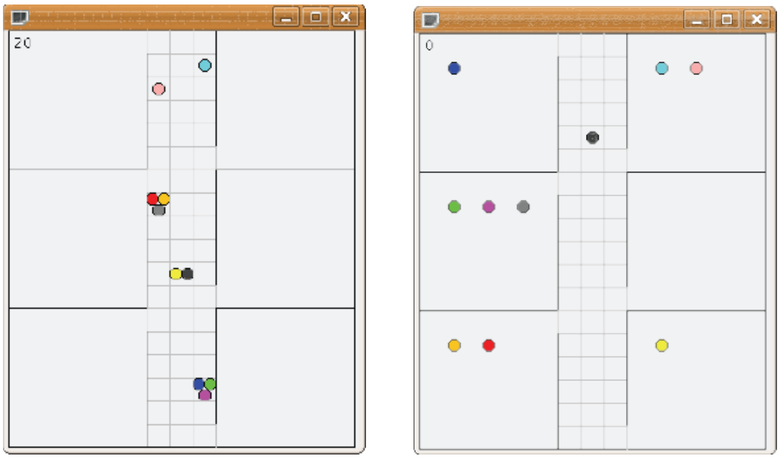
Table 2.13: General Steering Behaviour [Reynolds 1999]

2.4.3 Crowd Simulation using Reinforcement Learning

Multi-agent learning is a natural framework for crowd simulation where a collection of agents interact in an open environment. Rule-based methods have been able to generate crowds, however, the crowd behaviours are believed to be predictable. Current crowd simulation techniques discussed above are able to simulate believable crowds, however, the agents which are involved do not perform any learning. Multi-agent reinforcement learning models accommodate this criterion. This section describes a multi-agent reinforcement learning approach to crowd simulation.

Recent work on comparing the use of a multi-agent reinforcement learning method, and a rule based method in simulating crowd behaviour in a school domain by Torrey [2010] had given a new direction in crowd simulation. This work introduces the school domain, which simulates the behaviours from students' interactions during recess (break) time. There are two behaviours to be studied, agents socialising with one another and trying to get to classrooms on time. The agents' interaction area is a 18×3 grid world and this area denotes the hallway (corridor) that connects to six classrooms. Agents are placed randomly in the hallway, and their goals are to get to their assigned classrooms while socialising (see figure 2.5). There are 3 available actions where at each time step, each agent chooses to

1. Stay in the current state
2. Move one state closer towards the goal classroom
3. Move one state closer towards the closest agent



(a) Before training (b) After training

Figure 2.5: The 2D School Domain [Torrey 2010]

More than one agent can occupy one state space (cell) in the hallway while socialising at each time step. The classrooms are the absorbing states. The rewards are 30 for agents successfully arriving at the assigned classroom, and reward of 1 for each time step while socialising. The length of the learning episode is 25. The episode length ensures that arriving at classroom is more beneficial than constantly socialising.

Using the Q-learning algorithm in the MMDP environment (see section 2.3.1), with parameters $\alpha = 0.1$ and $\epsilon = 0.4$, the average episode reward produces good results for 10 agents in the school domain. Agents with high rewards (over 30) generally socialise at the beginning of the episode and manage to arrive at the goal classrooms, whereas agents with low rewards (below 30) do not arrive at the goal classrooms as they prefer socialising. The average reward obtained by each agent is 38 per episode.

The rules given in the rule-based approach are:

1. If the time left is less than or equal to the goal distance, move towards the goal.
2. Otherwise, move towards the closest agent.

In the rule-based approach, all agents managed to receive substantially higher rewards (above 30), i.e. they all arrived at their classrooms on time. The average reward obtained by each agent is 46 per episode, which is much higher than reinforcement learning agents' average rewards.

In reality, not all students are always on time. Rule based agents may achieve higher rewards but they behave in a mechanical and predictable manner. Reinforcement learning agents capture human personalities where some agents prefer to socialise more than the others. This shows that reinforcement learning agents portray human traits, in this case they are either extroverts or introverts. To simulate human qualities in a rule based method, animators have to lay down more rules and introduce probability into the system. Reinforcement learning is flexible and is able to acquire these qualities without explicit configuration.

It could be noted that the erratic or unpredictable behaviours presented by the reinforcement learning agents in the school domain are due to the violation of an MDP property (the MDP environment needs to be stationary). However, it is important to realise that in visual simulations, the qualitative result surpasses the quantitative result. That is, getting higher rewards does not mean the observed (visual) result is better in this case. There is a trade-off between obtaining highest rewards, found in a rule-based method, and higher believability, found in reinforcement learning method.

Agents are able to portray life-like personalities autonomously using the reinforcement learning approach whereas extra effort is needed to create probabilistic rules in rule-based agents. Owning to the fact that reinforcement learning agents influence one another in the learning process, each agent learns a different policy (optimal or sub-optimal) and thus can produce less predictable behaviour than that of rule-based agents.

2.5 Chapter Summary

In this chapter, we presented the background and related work on reinforcement learning and crowd simulation. A reinforcement learning agent is able to learn an optimal policy by means of gaining experience while interacting in environment. The single-agent reinforcement learning problem can be formulated as an MDP. Two methods were discussed on solving the MDP: value iteration and Q-learning. In multi-agent settings, the simplest form of applying reinforcement learning in this framework is having MDPs for each agent and solving via the Q-learning method. This technique provides overall good results albeit losing theoretical guarantees. Markov game is the natural extension of MDP in a multi-agent environment. Agents in a Markov game co-exist with one another in the environment and are aware of actions taken by the other agents (coordination) but they are exposed to the multi equilibria selection problem. Current multi-agent reinforcement techniques focus on the two-layer design where the multi-agent framework activates if there's a need for agents to coordinate, otherwise the single-agent framework is sufficient and thereby reducing the state-space.

The goal of crowd simulation is to produce the interactions and movements of entities in an environment. The approaches to simulating crowd are divided into two categories: macroscopic and microscopic. The techniques are able to depict the general behaviour found in crowds, but they do not involve learning. Recent work in simulating crowds using reinforcement learning produces believable crowd behaviour in a school environment. The 'students' are able to exhibit distinct (unpredictable) characteristics in the interactions with each other.

Chapter 3

Multi-Agent Crowd Simulation

The multi-agent domain is inherently difficult to implement effectively. As discussed in the previous chapters, the main challenge in the multi-agent domain is the exponential explosion found in the state-action space. Extending the single-agent reinforcement learning paradigm to the multi-agent environment is often trivial to implement, but computationally expensive.

In this dissertation, we study four methods to simulate crowd behaviour in a multi-agent environment. The methods are:

1. Rule Based (Chapter 4)
2. Multi-Agent Markov Decision Process (MMDP) (Chapter 4)
3. Joint State Action Q Learning (JSA-Q_1) (Chapter 5)
4. Joint State Value Iteration (JS-VI) (Chapter 5)

The objective is to obtain some results from the four methods, so as to answer our research question:

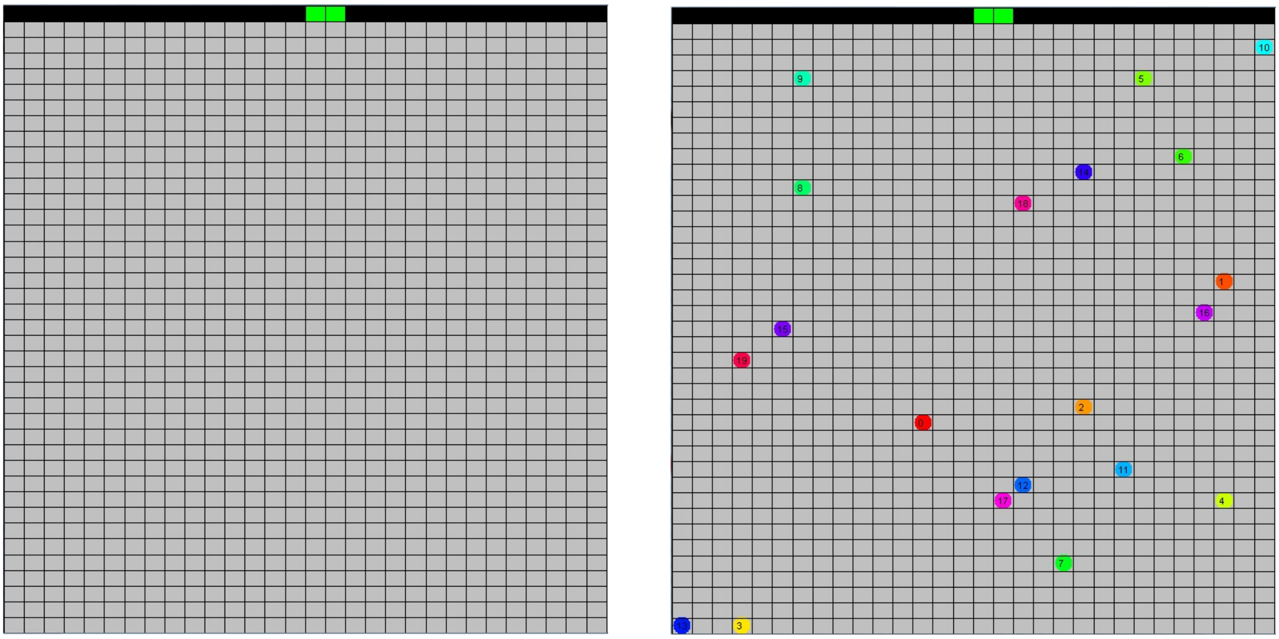
> *Given a 2D environment with a set of actors and objectives, can multi-agent reinforcement learning algorithm be used to produce policies that create believable crowd behaviour?*

3.1 The Environment

The simulated environment in this research is a grid world. Figure 3.1 is a general depiction of a 40×30 grid world. Each cell represents one state, and each state can be either a goal state, obstacle state or empty state at each time step. Empty states and goal states are states that an agent can occupy. The states are colour coded for reference, where goal states are in green, obstacle states are in black and lastly empty states are coded in grey (figure 3.1a).

A state is occupied when an agent is placed in that state, and this agent is represented as a coloured circle with a number referencing the agent's identification (figure 3.1b). Different agents will never occupy the same state at any given time step.

Figure 3.2 is a scaled down version of the grid world. The smaller grid world is 5×5. Refer to chapter 5 for more discussions.



(a)

(b)

Figure 3.1: Crowd simulation platform

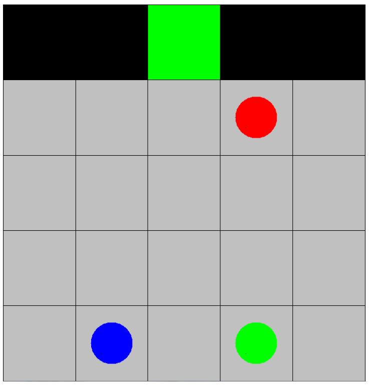


Figure 3.2: Small scaled simulation platform

The objective of the agents is to navigate to the goal states from any starting positions in an efficient manner, while steering clear of obstacle states and avoiding collisions with other agents. The goal states are the *absorbing states* as agents that enter the goal state will stay in the goal state for the remainder of the episode, irrespective of the actions (if any) chosen.

To simplify the simulation task, we assume that the following conditions hold:

1. agents move at the same velocity
2. each cell in the world represents a uniform space occupied by the average human size
3. any move performed by an agent is executed precisely in one time step

The third condition allows an agent to move or turn, say, 180° in one time step. That is, agents do not face any direction specifically, they can move in any direction without facing the intended orientation. There are 5 actions that an agent can perform: *up, right, down, left* and *stop*.

Collision and Out-of-Bounds

Special attention is needed in the event of agent collisions and when an action takes the agent out of the boundary of the environment. In the learning phase, suppose two or more agents have executed their respective actions and moved to the same resulting state, this scenario leads to the notion of an agent collision. As explained above, a state cannot be occupied by more than one agent at the same time. One solution to manage this problem that we use here is to let the colliding agents move back to their respective previous states and *randomly* choose one agent to proceed to the new (previously collision) state. Similarly, if an action takes an agent to an out-of-bounds state, the agent moves back to its previous state.

3.2 Formation of Bottlenecks

The study of crowd behaviour in a simulated environment is a subjective evaluation process, and is often difficult to quantify [Schadschneider *et al.* 2009; Martinez-Gil *et al.* 2012]. To simplify the evaluation process, we have decided to generate *density graphs* for our simulations.

A density graph or *heat map* in crowd simulation is a representation of density (the number of objects, humans, agents that are within a specific radius) over a period of time. Figure 3.3 is the colour scale that will be used to depict the density intensity. The darker the shade of a state the higher the density of population in that state. Similarly, a lower density of population will have a lighter colour representation over that state.

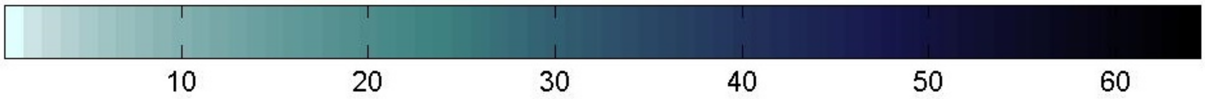


Figure 3.3: An example of the colour scale representing the density intensity. Lighter to darker shades depicting lower to higher density.

The *local density* ρ of a state $\vec{r} = (x, y)$ at time t is defined as

$$\rho(\vec{r}, t) = \sum_i f(\vec{r_i}(t) - \vec{r}), \tag{3.1}$$

where $\vec{r_i}(t)$ is agent i's current position at time t and

$$f(\vec{z}) = \frac{1}{\pi R^2}\, exp[\frac{-||\vec{z}||^2}{R^2}], \tag{3.2}$$

is a distance dependent weight function. Note that this density is obtained by averaging over a circular region of radius R [Johansson *et al.* 2007]. In a grid world, the radius corresponds to the number of immediate neighbours to the state $\vec{r}$. Figure 3.4 is an example of a density graph.

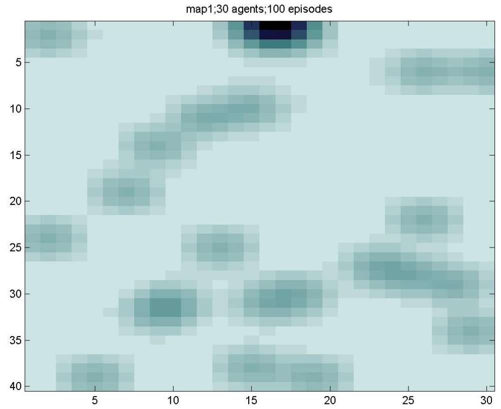


Figure 3.4: Density Graph - This density graph represents the movement of 30 agents in the grid world from figure 3.1a over a period of time.

It is inevitable that in any kind of crowd movement, especially in larger crowds, there will be congestion at exits (near and around) where the congestion will eventually build up long lines of lanes. The clogging and blockage of crowd movements contribute to the formation of bottlenecks. Observations of the formation of bottlenecks in both real-life simulation and computer generated simulation, have suggested that there is a definite shape that can be recognised. This shape is in the form of a *zipper* (the `Y` or `V` shape) [Schadschneider *et al.* 2009] (see figure 3.5). This zipper shape, otherwise is also known as the *zipper effect* is used to analyse our simulated results in chapters 4.1 and 4.2.

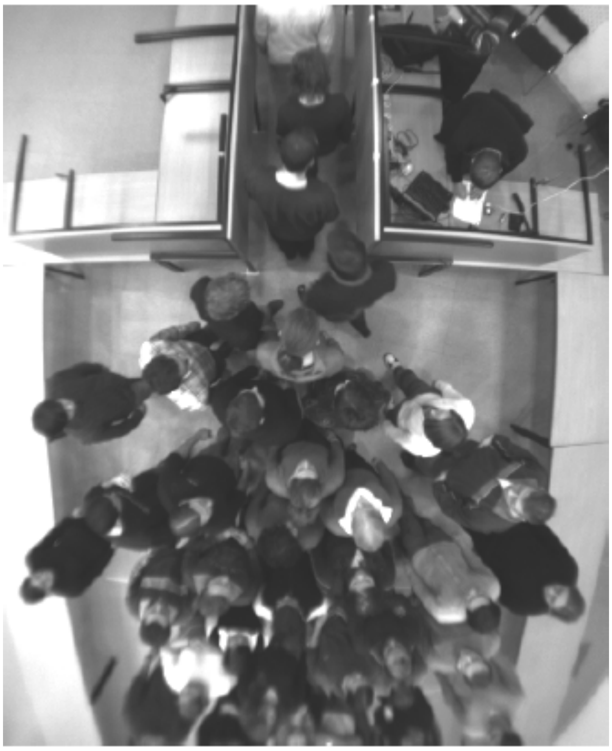


Figure 3.5: The `Y` or `V` shape forms around the exit where the clogging occurs

3.3 Simultion Recordings

Selected actual simulation recordings are available at `pages.ms.wits.ac.za/~fred` for viewing.

3.4 Chapter Summary

In this chapter, we formulated the research question and described the general domains for the four research methods. There are two environments available, a 40×30 large grid world and a smaller 5×5 grid world. The general depiction of the states in the grid world is discussed. Special attention to collision and out-of-bounds is also included.

Lastly, we introduced the notion of bottlenecks forming at the space around an exit. The bottleneck is a consequence of clogging and blockage of crowd movements which produce a recognisable zipper shape. We have devised density graphs as tools to identify the zipper shape in our experiments.

Chapter 4

Rule Based and MMDP Agents

In this chapter, we investigate the performance of the rule based and MMDP approach to crowd simulation. The 40×30 grid world described in section 3.1 is used for both methods.

4.1 Rule Based Agents

As discussed in section 2.4.3, rule based agents in crowd simulation present predictable behaviours. However, the rule based mechanism is a simple and intuitive solution. With the use of simple rules, it is possible to simulate life-like simple agents without carrying large computational cost. On the other hand, to simulate complex objects would require explicit rules and this will eventually become too tedious to implement in large size crowd simulations. Nevertheless, rule based methods have merit as they are simple and precise.

4.1.1 Rule Set

The rules that govern the actions of the agents at each time step are to move a new position, say s', which is closer to the goal state, while making sure that s' is not occupied by another agent and s' is not an obstacle state. Agents can move to any of the four immediate neighbouring states, s', with actions: *up, right, down* and *left* provided the action takes the agent to a valid state. That is, the new state is valid if it is within the grid world. Failing to meet the three rules, the agent will remain in the current state s'.

For each action, in the order of *up, right, down* and *left*, the three rules are listed below

1. determine if new state is a valid state, then
2. if the new state is not an obstacle or an occupied state, then
3. if the distance to the goal state is lower, move to the new state

Note that the distance between two states is determined by taking the Manhattan distance between two points.

4.2 MMDP Agents

Recall that in an MMDP model, the goal is to iterate over n agents so that their respective Q-table values are updated using the Q-learning algorithm. The aim is to simulate a large crowd navigating to exits in a large environment using the MMDP model. We also investigate whether the loss of theoretical guarantees will have any impact on the simulation.

4.2.1 State Space

We consider two ways to represent the state space. Firstly, an elementary representation of states with size $\mathcal{O}(n{\cdot}|S|{\cdot}|A|)$ where n is the number of agents, $|S|$ is the size of the dimension and $|A|$ is the number of actions. That is, each agent maintains a Q-table of size $\mathcal{O}(|S|{\cdot}|A|)$. The Q-value is indexed by the state and action pair. For example $Q^i(s_1, a_1)$ denotes the Q-value that agent i is in state s_1 and chooses action a_1.

Secondly, additional state information is included such that agents receive the information of neighbouring states. The additional states are shown in figure 4.1.

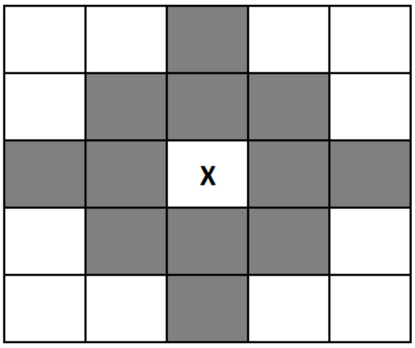


Figure 4.1: The state information (shaded regions) that is available to the agent at position X.

There are a total of 12 extra bits of state information where each bit indicates the presence of an agent. Therefore, the size of the Q-table that each agent has is $\mathcal{O}(n{\cdot}|S|{\cdot}|A|{\cdot}2^{12})$. The Q-value is indexed by the state-action pair and the 12 additional bits. For example, $Q^i(s_1, a_1, b_1, b_2, ..., b_{11}, b_{12})$ denotes the Q-value that agent i is in state s_1 and chooses action a_1 while taking account of the 12 neighbouring states. The additional state information serve to expedite the learning process.

4.3 Experiments

Rule based agents do not require any pre-training or pre-learning prior to the simulations as agents react to the environment and perform actions during the simulation. Agents are placed in the simulation environment randomly and each agent navigates the environment following the above described rules as per simulation.

Contrary to rule based simulations, the MMDP agents need to learn to navigate in the environment first. Refer to the MMDP Q-learning algorithm as described in section 2.3.1, Algorithm 3.

The length of one learning episode is 500 moves (steps) or until all agents have reached the goal state. An agent will stay in the terminal (absorbing) state for the duration of an episode once it enters a terminal state.

The rewards and penalties associated with each action are as follows: reward of 100 for arriving at goal state, penalty of -10 for colliding with other agent(s), penalty of -10 for moving to an obstacle state, lastly a penalty of -1 for every action taken except when the agent is in the goal state. The -1 penalty is designed to ensure that agents can find short paths to the goal.

Following the discussion on collision and out-of-bounds in section 3.1, a learning agent will move back to its previous state if an action takes the agent to an out-of-bounds state. However, in the event of a collision, there is no need for agents to move back to their previous states as the agents make their moves asynchronously.

The policy is obtained from the final Q^i-tables for each agent by

$$\pi^i(s) = \underset{a}{arg\ max}\ Q^i(s,a),\ \forall s \in S.$$

In the simulation phase, a full simulation or round ends when all agents have arrived at the goal state or when the number of moves are more than 500. At each step, the order of agents taking actions is randomly selected.

The parameters α, γ and ϵ are $0.25, 0.99$ and 0.1 respectively are used for all the experiments. Note that α is set to decay.

A complete simulation is obtained when all agents have arrived at the goal state. The order in which the agents move to new states is chosen by their identification number in ascending order. At every time step of the simulation, the density of the crowd is calculated using equation 3.1.

4.4 Results and Analyses

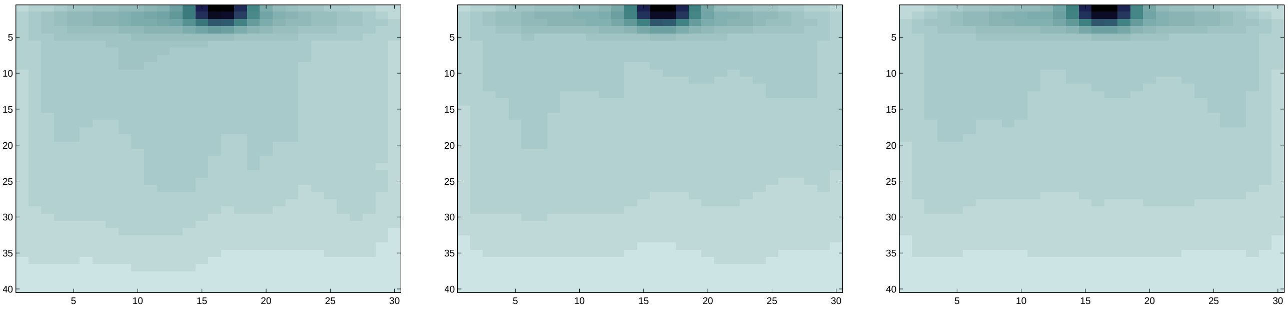
The number of agents used in both methods are $20, 50$ and 100.

Rule Based Agents

The simulation results from the rule based approach are shown in figures 4.2 to 4.4. As the number of rounds increases for each simulation, we can see that the agents (or crowd) are following a fixed path. The movement of the crowd is predictable as they are inclined to move straight ahead (in a straight line) and then move across to the exits. The crowd moves in such a predictable manner because we have forced them to choose actions in the *up, right, down* and *left* order.

This phenomenon can be seen from the denser horizontal region at the top of the graph. We can also confirm that the zipper effect or the shape is not found in the density graph. Even though the crowd will always try to find the shortest path to the exits, however, without introducing any *variance* or *randomness* in choosing actions, the path that the crowd follows will be similar in every simulation.

The concept of adding randomness is that each individual in a crowd could portray some traits of *uniqueness* or *individuality*, but it must also preserve the quality of a crowd as a whole. Furthermore, as evidenced from section 2.4.2, more rules have to be introduced in order to produce more believable crowd behaviour. The process will eventually become tedious and completely infeasible when simulating large crowds (especially as humans), as more complex rules must be designed for each agent to achieve a higher degree of believability.

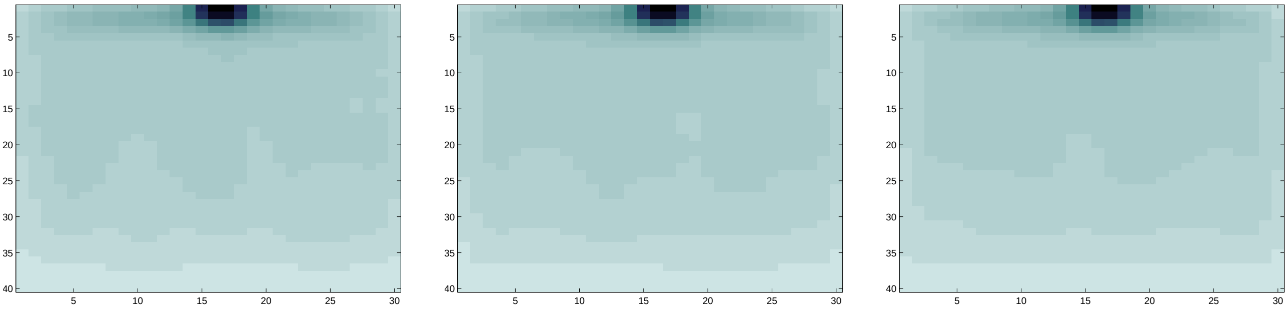


(a) 50 rounds

(b) 100 rounds

(c) 200 rounds

Figure 4.2: 20-agent simulations - 50, 100 and 200 rounds

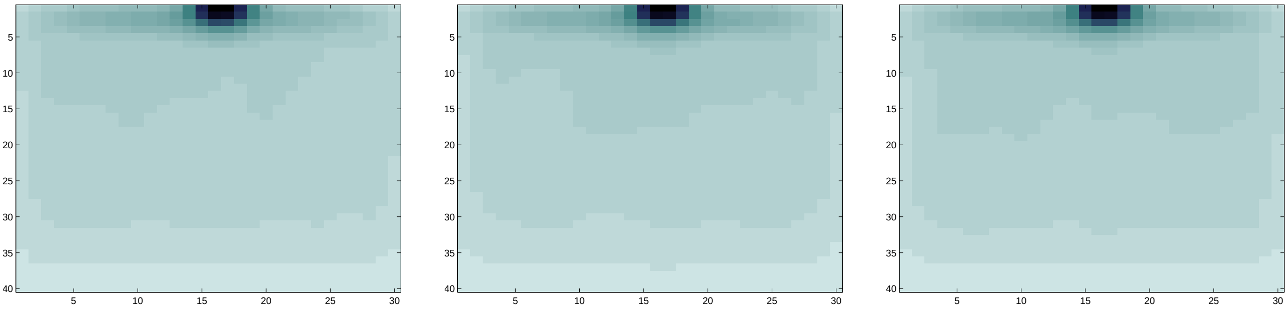


(a) 50 rounds

(b) 100 rounds

(c) 200 rounds

Figure 4.3: 50-agent simulations - 50, 100 and 200 rounds



(a) 50 rounds

(b) 100 rounds

(c) 200 rounds

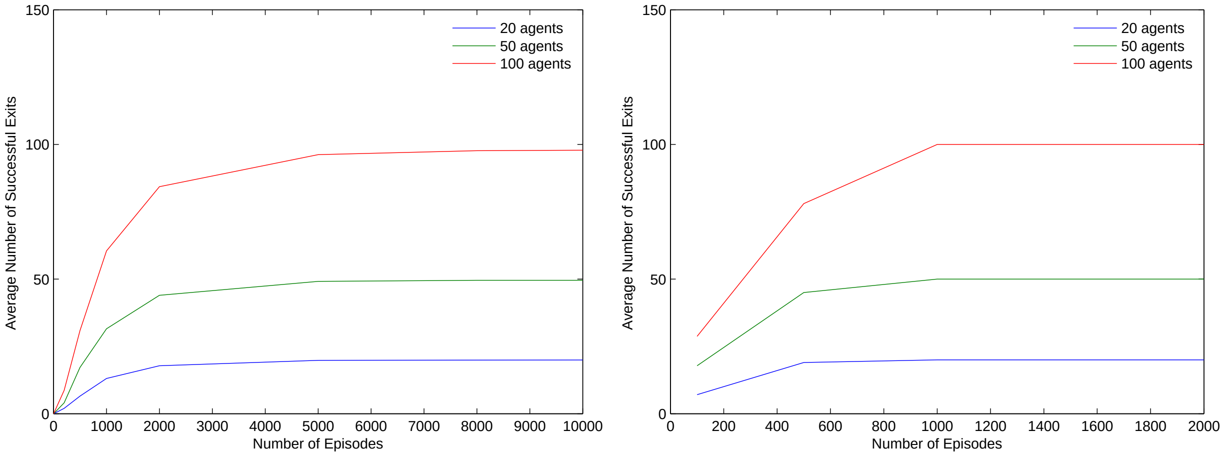
Figure 4.4: 100-agent simulations - 50, 100 and 200 rounds

Moreover, the complexity analysis indicates that the order is $\mathcal{O}(n)$ where n is the number of agents. This linear complexity is comprised of two parts, a $\mathcal{O}(1)$ complexity at determining the three rules and this is done over n times ($\mathcal{O}(n)$). Combining both parts, we get $\mathcal{O}(n)$.

MMDP Agents

To differentiate the two methods of representing states, we denote the elementary representation (where each agent maintains a Q-table of size $\mathcal{O}(|S|\cdot|A|)$) as Experiment A and the second representation (where each agent maintains a Q-table of size $\mathcal{O}(n\cdot|S|\cdot|A|\cdot 2^{12})$) as Experiment B. An interesting phenomenon is observed from the two experiments. In the 20-agent environment, agents in Experiment A are able to learn optimal policies after 2000 episodes, whereas agents in Experiment B are able to learn optimal policies after 500 episodes. This indicates that agents in Experiment B learn faster. However, owing to the fact that 2^{12} additional bits of information is present in the state space, the time taken to learn is substantially longer than Experiment A.

The average number of successful exits over the number of episode graphs in figure 4.5 indicate that with sufficient learning, all the agents from both experiments are able to locate and move to the goal states. Note that the average number is obtained by running 200 simulations.

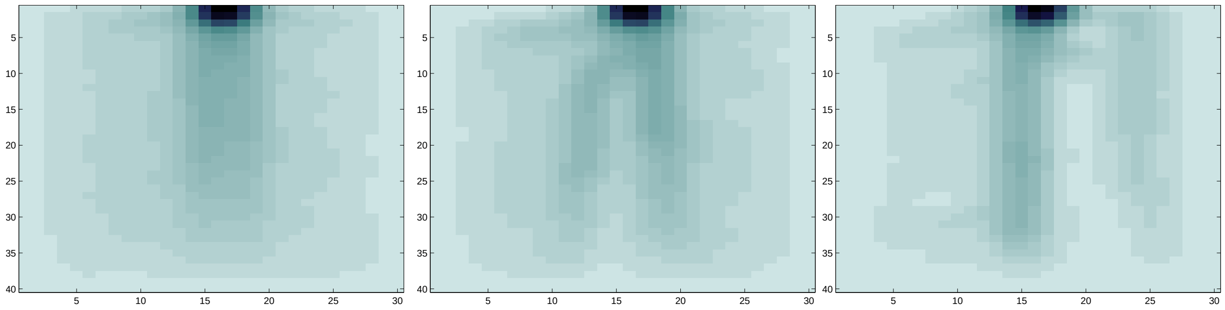


(a) Graph showing average number of exits in Experiment A

(b) Graph showing average number of exits in Experiment B

Figure 4.5: Average number of exits in Experiment A and B

Furthermore, the agents in the two experiments are following similar policies as the density graphs show close similarities. In addition, the `Y` shape can be identified in the density graphs. The `Y` shape or the zipper effect is an indication the agents are portraying common behaviours found in and around exits. Figure 4.6 from Experiment B shows the presence of the `Y` pattern. The density graphs are obtained by averaging 200 rounds of simulations.



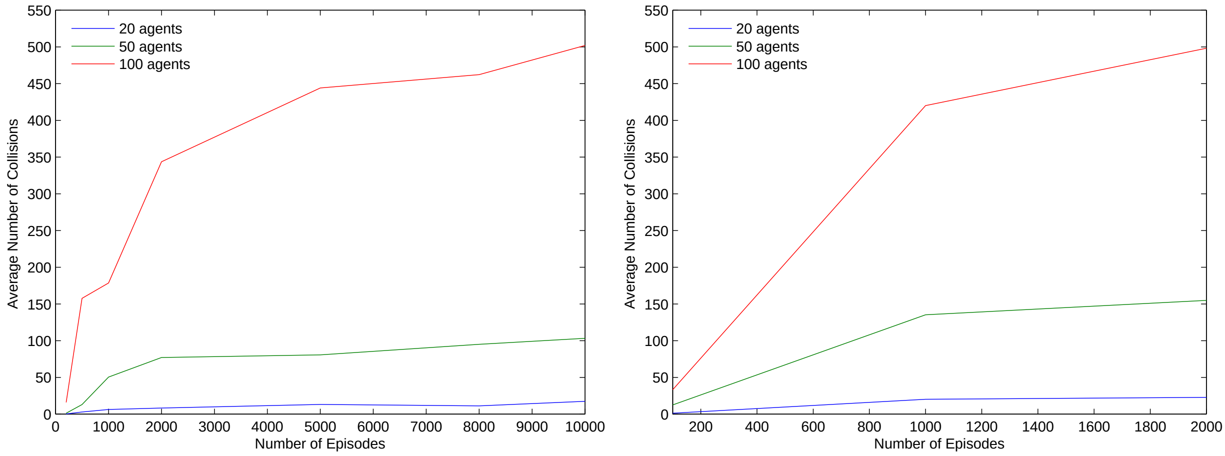
(a) 20 agents

(b) 50 agents

(c) 100 agents

Figure 4.6: Density graphs generated by 200 rounds of simulations trained over 2000 episodes

On the other hand, consider the average number of collisions over the number of episodes graphs in figure 4.7. The average number of collisions per episode is increasing as indicated from both graphs. While the agents are able to navigate to the goal states successfully, they are colliding with one another rather frequently.



(a) Graph showing average number of collisions in Experiment A

(b) Graph showing average number of collisions in Experiment B

Figure 4.7: Average number of collisions in Experiment A and B

One simple deduction of such an occurrence is the non-stationary environment that the agents reside. As discussed previously, the theoretical guarantee of an MDP converges is no longer valid if it is extended to a multi-agent environment. The reward function specifies that agents will receive penalties if collisions occur, thus an attempt is made to prevent the same collision from occurring again. However, the non-stationary environment coupled with the fact that agents are not aware of other agents' actions and thus prompting the frequency of collisions.

The motivation for including the additional information of the neighbouring states is to let the agents be aware of their surroundings while learning so that they are able to learn to coordinate. However, being aware of the presence of the other agents is not sufficient for the agents to coordinate in this case. The additional information indicate the occupancy of the surrounding states, but they do not inform what actions the other agents are performing.

To demonstrate, consider the scenario in figure 4.8. In the learning phase, agent 0 understands that agent 1 is in its neighbouring state, however because agent 0 does not know which action agent 1 will perform, agent 0 eventually learns that the best action is the action *up*. Similarly, agent 1 learns that the best action is the action *left*. In this scenario, the collision is inevitable. The agents update their Q-values as if they are alone in the environment.

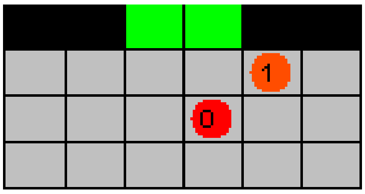


Figure 4.8: An example showing that a collision is unavoidable in the MMDP model.

4.5 Chapter Summary

The simulated rule based agents in crowd simulation behaved in a predictable way. An observation found in our rule based simulations is that the cost of the implementation is linear in nature due to the three simple rules. However, the crowds resemble no individuality as the agents are not exposed to randomness when choosing actions and more complex instructions have to be tailored for each agent.

The lack of a theoretical guarantee of convergence is a shortfall in MMDP models as the environment is non-stationary. However, with adequate learning episodes, the agents were able to find the goal states while navigating in the environment as supported by the learning curves presented. The fact that the density graphs conformed to the zipper shape further demonstrated that the agents were presenting crowd-like behaviour in our simulations. The MMDP Q-learning algorithm can produce policies that would allow a fixed number of agents (up to 100 agents) to locate and navigate to the goal states.

On close inspection, the graphs in figures 4.5 and 4.7 indicate that agents are certainly navigating to the goal states, however, with collision tendencies. This scenario resembles an evacuation simulation where a group of agents (a crowd) attempts to escape an environment in an emergency situation.

Finally, despite efforts made to encourage agents to coordinate by including additional information from their surrounding states, agents in an MMDP model failed to prevent collisions. Without being informed of the actions that the other agents take, agents cannot coordinate. Consequently, we investigate the effectiveness of the joint state action space for agents to coordinate in the next chapter.

Chapter 5

JSA-Q_1 and JS-VI Agents

In this chapter, we introduce two multi-agent reinforcement learning methods: joint state action Q-learning and joint state value iteration. The methods are tested in the 5×5 environment described in section 3.1. In contrast to the environment used in the MMDP model in the previous chapter, this smaller environment is necessary as large sized environment makes joint state action (even joint state) representation infeasible. The joint state action space, allows the agents to be aware of the actions taken by one another, suggests that agents should be able to learn to coordinate.

5.1 JSA-Q_1 Agents

We introduce a novel approach to implement multi-agent Q-learning in joint state action spaces. Essentially, this is equivalent to having one centralised agent learning the tasks of the other agents. We also try to address the problem of exponential growth which occurs in joint state action spaces and attempt to find a way to reduce the computational cost. Suppose an optimal policy for one agent (single-agent) is available. Then, for two agents to learn in the same environment, there is no need to learn further when one agent arrives at the goal state as there will only be one agent left in the environment. This observation reduces the two-agent problem to a single-agent problem which in turn reduces the number of joint state action spaces that the algorithm needs to visit and update.

5.1.1 Joint State Action Space

The joint state and joint action spaces concept is discussed briefly in section 2.3.3. In an n agent environment, let $|S|$ be the size of the environment and $|A|$ be the number of actions. The size of the joint state space is $|S|^n$ and the size of the joint action space is $|A|^n$. Hence, the required size of a joint state action space is $\mathcal{O}(|S|^n \cdot |A|^n)$. Note that the joint state space is not exactly $|S|^n$ as agents are not allowed to occupy the same state at the same time.

One motivation for using the joint state action space is to correlate the movements of the agents during the simulation phase. Realistically, the agents should perform actions simultaneously rather than in some defined order as shown in the previous two methods.

The full joint state action set is a set of all possible states and actions for n agents. That is, suppose in a 2×2 grid world with 2 agents, the agents are allowed to perform 4 actions $(0, 1, 2$ and 3). Then the total number of states is 4 and the number of joint states is $4^2 = 16$. However, since we are limiting the agents to not occupy the same state, the actual number of joint states available is 12. The number of joint actions is $4^2 = 16$. The total size of the joint state action space needed is $12 \times 16 = 192$.

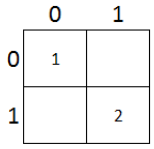


Figure 5.1: A 2×2 grid world illustrating the joint state action space concept

The two agents, 1 and 2, are placed in states $(0, 0)$ and $(1, 1)$ which constitute a joint state, while a joint action comprises the collection of two actions, say $(0\ 0)$ where both agents take the action 0. A pair of joint state and joint action represent a joint state action pair. The joint state action set is used to access the Q-value from the Q-table, $Q(\vec{s}, \vec{a})$, where $\vec{s}$ is the joint state and $\vec{a}$ is the joint action.

5.1.2 The *JSA*-Q_1 Algorithm

Extending the general multi-agent Q-learning method from Algorithm 4 in section 2.3.3, we have derived the *JSA*-Q_1 algorithm as shown in Algorithm 5. This algorithm is based on the notion that it is relatively easy for one agent to learn in a simple environment (single-agent reinforcement learning). Suppose an optimal policy for one agent exists. Then for two agents to learn in the same environment, the learning iteration per episode terminates when an agent has arrived at the goal. That is, there is no need for the remaining agent (one agent in this case) to learn an optimal policy since we already have the optimal policy for one agent. This process is then repeated until an optimal policy for two agents is achieved.

Similarly, once an optimal policy for two agents has been obtained, this two-agent policy can now serve as the basis for finding an optimal policy for three agents. The learning iteration per episode terminates when an agent has arrived at the goal, thus leaving two agents behind. Therefore, using this method, we can find some policy for n agents navigating in an environment. The learning iteration per episode ends when one agent has found the goal, hence the name Q_1. Note that, for n agents to learn, the algorithm needs to be executed n times in the order of $1, 2, 3, ..., n$ agents.

The objective of applying this method is to reduce the size of the joint state action space and accelerate the learning process. The assumption is to limit the algorithm to explore additional joint state action set when an agent arrives at the goal state, thus, trimming down the size of the joint state action space which can then accelerate the learning process.

There are two parts to the *JSA*-Q_1 algorithm (shown in algorithm 5). The first segment (the while loop) handles the learning mechanism for n agents until one agent has found a goal state. The second segment is necessary as we need an extra step to allow the remaining agent(s) to follow the correct policy. To illustrate this motive, consider a two-agent example (see figure 5.2a). Let the rewards be 100 for arriving at a goal state, -10 for an agent collision, -10 for moving to an obstacle state and -1 in addition for every move. The agents' positions are at $(1, 2)$ and $(1, 4)$. Suppose agent 0 has chosen to move up, thus moving to the goal state. At this

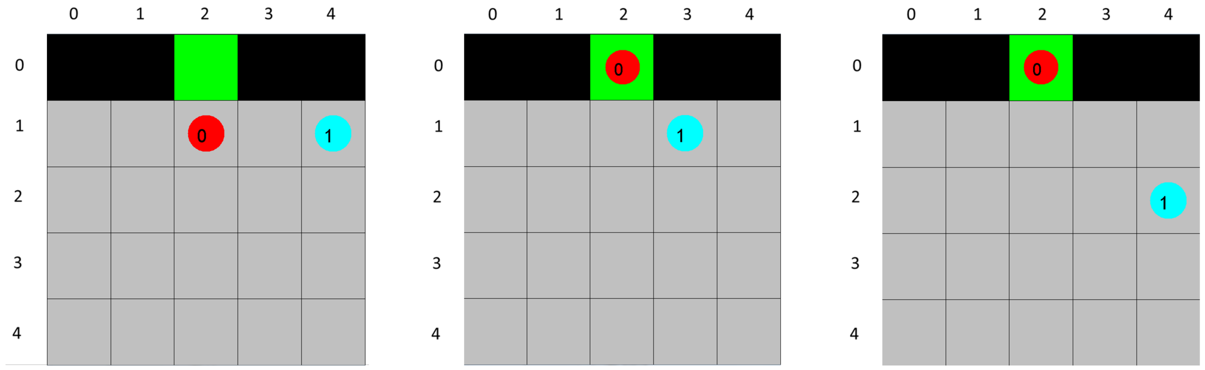
Algorithm 5 The *JSA*-Q_1 Algorithm for n agents

```
Require:
  Initialise $Q(\vec{s},\vec{a}) = 0 \; \forall \; \vec{s} \in S^n, \vec{a} \in A^n$ where $n$ is the number of agents
  Initialise state $\vec{s}$
  while no agent has arrived at goal do
    Choose $\vec{a}$ from $\vec{s}$ using policy derived from $Q(\vec{s},\vec{a})$
    Take action $\vec{a}$ and observe $r$ and $\vec{s}^{\,'}$
    $Q(\vec{s},\vec{a}) \leftarrow (1-\alpha)Q(\vec{s},\vec{a}) + \alpha[r + \gamma \max_{\vec{a}} Q(\vec{s}^{\,'},\vec{a})]$
    $\vec{s} \leftarrow \vec{s}^{\,'}$
  end while
  if number of agents remaining is more than 0 then
    $\vec{s} \leftarrow$ Backtrack all the agents to previous states
    Choose $\vec{a}$ from $\vec{s}$ using policy derived from $Q(\vec{s},\vec{a})$
    Take action $\vec{a}$ and observe $r$ and $\vec{s}^{\,'}$
    $Q(\vec{s},\vec{a}) \leftarrow (1-\alpha)Q(\vec{s},\vec{a}) + \alpha[r + \gamma \max_{\vec{a}} Q_{\text{-}1}(\vec{s}^{\,'}_{\text{-}1},\vec{a}_{\text{-}1})]$
  end if
```

point, the algorithm will perform one last update to the Q-table. However, this update does not take into account the action of agent 1. That is, a two-agent policy is optimal if one agent is able to navigate to the goal state, regardless of the other agent's actions. Therefore, the two policies where agent 0 and agent 1 move

1. *up* and *left*
2. *up* and *down*

are considered as optimal policies as one agent in both policies is able to locate the goal state. Nevertheless, the second policy is not optimal as agent 1 is moving a step further away from the goal state.



(a) Initial (b) Policy 1 (c) Policy 2

Figure 5.2: Two policies to following from the initial states

To rectify this flaw, the Q-value of the conflicting states needs to be adjusted, hence, we have the second part of the algorithm. Before the update begins, let i^* be the agent that is in the goal state and the algorithm backtracks all the agents (including agent i^*) to their previous states, $\vec{s}^{\,'}$. Similar to the first part, the adjustment occurs at the update of Q-value. Instead of finding the future Q-value from the n agent policy after finding and executing the joint action, this update now finds the value from the $n-1$ agent policy using the joint state $\vec{s}^{\,\prime}_{\text{-}1}$ and joint action $\vec{a}_{\text{-}1}$ where agent i^*'s state and action are excluded. This extra step allows the two policies (n and $n-1$ policies) to be coherent.

Consider the example above, if the single-agent policy is optimal, then agent 1's maximum future Q-value will be from the state to its left, $(1, 3)$. This value will then be used to update the two-agent Q-value, and with sufficient learning, there will only be one optimal policy (Policy 1, figure 5.2b).

As an extension from Q-learning , the *JSA*-Q_1 learning algorithm assumes no prior knowledge of the environment, thus making the environment model-free.

5.2 JS-VI Agents

We also look at the value iteration (VI) algorithm, which is a dynamic programming approach, to crowd simulation (multi-agent). The multi-agent value iteration algorithm is almost identical to the single-agent value iteration algorithm discussed previously. The single-agent value iteration algorithm is shown in algorithm 1 in section 2.2.1.

5.2.1 Joint State and Joint Action

The only modification to the single-agent value iteration algorithm is the use of joint states, $\vec{s}$, to index the value function $V(\vec{s})$. The size of the joint state space is still exponential, $\mathcal{O}(|S|^n)$ where $|S|$ is the size of the environment and n is the number of agents.

Similar to the joint state action space discussed in the previous chapter, the definition of joint state and joint action remain unchanged. The difference is that the *JSA*-Q_1 algorithm requires the presence of the full joint state action set in order to index the values from the Q-table, whereas the algorithm used in this chapter handles the joint states and joint actions separately. The algorithm calls for the existence of joint states to obtain the values from a value function and joint actions for when agents are selecting actions.

5.2.2 The *JS-VI* Algorithm

The revised algorithm, *joint state value iteration* or *JS-VI*, is outlined below (algorithm 6).
The value iteration algorithm requires the full knowledge of the transition probability thus requiring the *JS-VI* algorithm to work in model-based environments.

5.3 Experiments

Owing to the nature of exponential growth in the state space with joint state action set, it is infeasible to use the same environment as the previous two methods discussed in chapter 5. A simple calculation shows that in a 40×30 grid with 10 agents where each agent has 5 actions, the state space requirement is $1200^{10} \times 5^{10} \approx 6 \times 10^{44}$, which is astronomical. Similar argument holds for the joint state approach. We therefore test the two methods in a smaller environment described in section 3.1.

Algorithm 6 The Joint State Value Iteration (*JS-VI*) Algorithm

```
Require:
  Initialise $V(\vec{s}) = 0 \; \forall \; \vec{s} \in S^n$ where $n$ is the number of agents
  repeat
    $\Delta \leftarrow 0$
    for $\vec{s} \in S^n$ do
      $v \leftarrow V(\vec{s})$
      $V(\vec{s}) \leftarrow \underset{\vec{a}}{max} \sum_{\vec{s}'} P(\vec{s}, \vec{a}, \vec{s}')[R(\vec{s}, \vec{a}, \vec{s}') + \gamma V(\vec{s}')]$
      $\Delta \leftarrow max(\Delta, |v - V(\vec{s})|)$
    end for
  until $\Delta < \theta$
```

JSA-Q_1 Agents

A single agent Q-value (or Q-table) is obtained by letting one agent learn in the environment using the single agent Q-learning method from Algorithm 2 in section 2.2.2. This single agent Q-table is used as the base case as described above.

A full episode of learning ends when one agent has arrived at the goal state. The handling of agent collision and out-of-bounds occurrences in the learning phase is as discussed in section 3.1. The rewards and penalties for different states are as follows: reward of 100 for arriving at a goal state, penalty of -10 for colliding with other agent(s), penalty of -10 for moving to an obstacle or out-of-bounds state, lastly a penalty of -1 for every action taken except when agent is in the goal state. The parameters for the experiments α, γ and ϵ are $0.25, 0.99$ and 0.1 respectively.

A random selection is made when there is a tie. The ties are usually found in choosing the highest next joint state value ($\underset{\vec{a}}{max}\ Q(\vec{s'}, \vec{a})$) in the learning phase and in choosing the joint action that has the highest value in the simulation phase.

JS-VI Agents

The JS-VI algorithm uses the same reward function as the JSA-Q_1 algorithm. The parameters for the experiments γ and θ are 0.99 and 0.01 respectively.

In the process of evaluating $V(\vec{s})$, it is safe to assume that a joint action that results in a collision will not have the highest value. That is, there exists at least one joint action where one of the agents will 'give way' which in turn will have a higher value. This assumption indicates that we do not need to handle the reassignment of agents when there is a collision or an out-of-bounds occurrence as the corresponding joint action will always have lower values than other joint actions.

Once the learning phase is complete, the deterministic policy $\pi(\vec{s})$ is determined by

$$\pi(\vec{s}) = arg\,\underset{\vec{a}}{max} \sum_{\vec{s}'} P(\vec{s}, \vec{a}, \vec{s}') \Big[R(\vec{s}, \vec{a}, \vec{s}') + \gamma V(\vec{s}') \Big], \; \forall \vec{s} \in S^n. \tag{5.1}$$

A random tie breaker is introduced if there are two or more joint actions that have the same values when determining a policy.

5.4 Results and Analysis

Both algorithms are tested on $2, 3$ and 4 agents and the results are discussed below.

5.4.1 JSA-Q_1 Agents

The length of the learning episodes for each experiment with different numbers of agents can range from $2,000$ to $4,500,000$ episodes. The number of joint state action sets visited are recorded from each experiment and they form the total number of entries found in the Q-table. The number of entries (joint state action set) is to be compared with the theoretical calculations of the joint state action space as discussed above (section 5.1.1).

Table 5.1 lists the number of entries (joint state action set) present in the Q-table per each episode, and the theoretical calculation of the joint state action space.

Number of Agents	Number of Episodes	Number of Entries in the Q table	Theoretical Calculation
2	500,000	10,700	15,625
	1,000,000	10,669	
	2,000,000	10,691	
	2,500,000	10,708	
	3,000,000	10,710	
3	2,000,000	56,8676	1,953,125
	2,500,000	606,036	
	3,000,000	632,849	
	4,000,000	678,609	
4	2,000,000	2,914,979	244,140,625
	2,500,000	3,550,623	
	3,000,000	4,163,536	
	3,500,000	4,759,815	
	4,000,000	5,338,670	
	4,500,000	5,898,055	

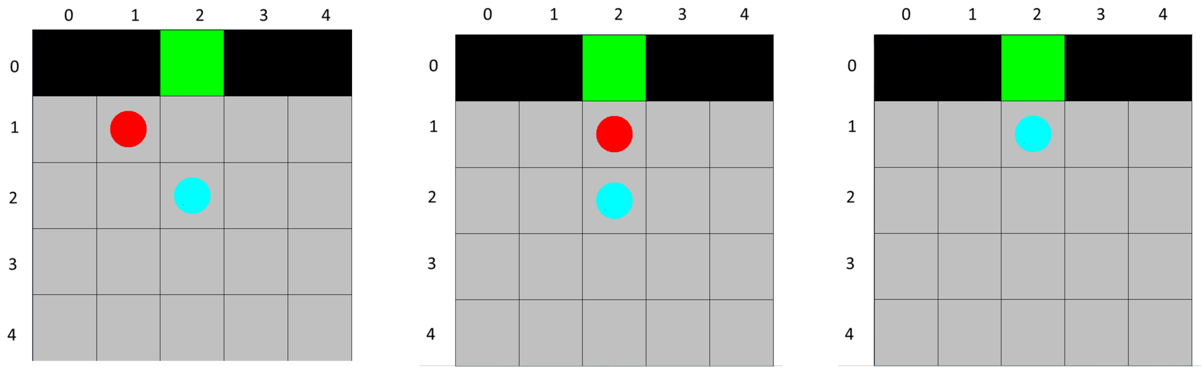
Table 5.1: Comparisons of the Number of Entries in Q-table

We have chosen the starting policy π_1^* (one agent policy) from the Q-table of 1000-episodes. This policy is optimal as we have tested that the agent manages to find optimal paths to the goal state when placed randomly in the environment. Using this policy π_1^*, we are able to apply the algorithm for experiments involving two agents over a range of episodes and thus obtain a two-agent optimal policy, π_2^*. Again, using this two-agent optimal policy, we are able to obtain a three-agent optimal policy, π_3^*, and so forth.

Unfortunately, it is infeasible to obtain an optimal policy for 5 agents. The size of the joint state space of a five-agent, 5×5 environment is $25^5 \times 5^5 = 30,517,578,125$, which is approximately 30 billion. This number is impractical to be implemented as the physical computer memory will be inadequate. In fact, the optimal policies for the three-agent and four-agent scenario are sub-optimal policies. Without running *out of memory*, the number of visited joint state action sets is only a fraction of the number of theoretical joint state action set. We can see that in the three-agent and four-agent policies, the number of visited joint state action sets is only roughly 35% and 2% of the theoretical number respectively. As shown in table 5.1, the number of entries escalates exponentially as the number of agents increases.

Agent Collision

In an optimal policy, agents learn to coordinate in order to avoid collisions. Consider a two-agent environment (figure 5.3a) where the agents' (0 and 1) positions are $(1, 1)$ and $(2, 2)$. From the contents of the Q-table, table 5.2 is extracted for the joint state $(1, 1)$ and $(2, 2)$ where each unique joint action is associated with a Q-value. The joint action is of the form a_0 a_1 which means agent 0 performs action a_0 and agent 1 performs action a_1.



(a) Initial (b) Step 1 (c) Step 2

Figure 5.3: Collision Avoidance

Table 5.2 lists 25 joint actions and the Q-value pairs from the joint state $(1, 1)$ and $(2, 2)$. The number of joint action and Q-value pairs corresponds to the total number of joint actions in our calculation, which is $|A|^n = 5^2$. This indicates that the agents have attempted all possible joint actions from all possible states, and with sufficient learning, the values of each joint action should be able to reflect the consequence of performing such joint actions. There are two joint action sets that have the same maximum Q-value (195), namely $(1\ 4)$ and $(0\ 4)$. The joint action pairs are indeed the optimal actions. Evidently, looking at figure 5.3a we can easily conclude that one agent has to give way to the other agent in order to avoid a collision, and it is clearly represented in the two joint action sets $(1\ 4)$ and $(0\ 4)$. A random tie breaker is used in the event of multiple optimal actions.

By allowing the fifth action *stop*, the agents are able to *coordinate* when they are moving to a conflicting state. Since there is no preference or priority on which agent gets to move to the new state, the selection is done randomly. Suppose the joint action $(1\ 4)$ is selected and the agents perform the actions where the results are shown in figure 5.3b. In this scenario, there is no threat of collision and therefore there should only be one joint action set that has a maximum

Joint Action	**Q-value**
0 0	185
0 1	183
0 2	183
0 3	183
0 4	183
1 0	173
1 1	194
1 2	194
1 3	194
1 4	195
2 0	194
2 1	191
2 2	191
2 3	173
2 4	193
3 0	194
3 1	191
3 2	191
3 3	191
3 4	193
4 0	195
4 1	193
4 2	193
4 3	193
4 4	193

Table 5.2: Q-values of the joint state $(1,1)$ and $(2,2)$

value $(0\ 0)$. After performing this joint action, agent 1 will be the only agent left in the environment (figure 5.3c). This simulation ends with agent 1 taking action 0 from the one agent policy π_1^*.

Values, as listed in table 5.2, can be calculated by adding up the total rewards of the joint action taken by the agents upon arriving at the goal state (by following some policy)[1]. For example, if the agents perform the action $(0\ 0)$, the rewards will be -10 for moving to an obstacle state and -2 for the penalty. In the next move, the joint action that carries the highest value is $(1\ \ 0)$ and this has a reward of 100 from agent 1 moving to a goal state and -2 for the penalty. Lastly, agent 0 moves to the goal state and the rewards are 100 and -1. Therefore, the total rewards for the agents to navigate to the goal state from the starting positions are $(-10) + (-2) + 100 + (-2) + 100 + (-1) = 185$. This total reward correspond to the $Q-$value of the joint action $(0\ 0)$.

Performance Curves

To validate the performance of this algorithm, consider the graph in figure 5.4.

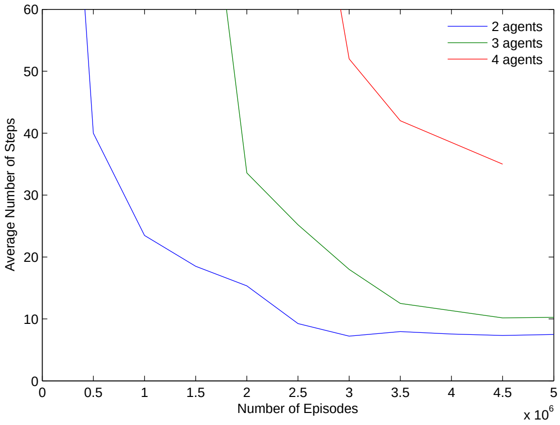


Figure 5.4: Graph showing average number of steps to reach the goal state

[1] Assuming $\gamma = 1$

The average number of steps for the agents to reach the goal state is steadily decreasing as the number of training episodes increases where the average is taken by running the simulations 100 times. This graph validates that the algorithm converges for the 2-agent and 3-agent environments. The graph of the 4-agent case is terminated after $4,500,000$ learning episodes as continuing the learning triggers out of memory errors. The higher number of average steps can be justified as the explored joint state action set is already 2% (as discussed above), signifying the agents have not visited adequate number of joint state action sets at training due to the large state space and thus render the agents immobile or forcing the agents take longer paths to the goal state.

The average number of steps obtained from the simulations using policies π_2^* and π_3^* are as follows:

Optimal Policy	Average steps
π_2^*	7.56
π_3^*	10.16

Table 5.3: Average steps obtained from the *JSA*-Q_1 optimal policies

5.4.2 JS-VI Agents

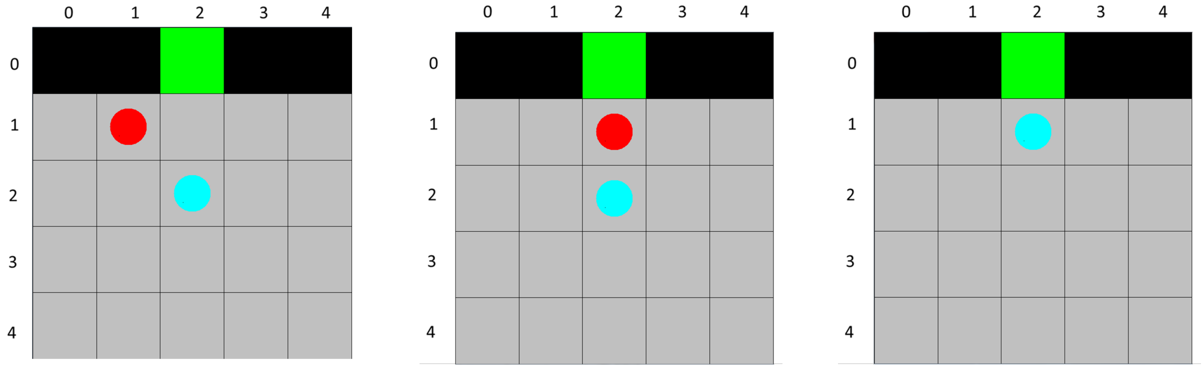
Recall that value iteration (in a single-agent environment) is a dynamic programming approach to finding an optimal policy. Value iteration, provided it converges, will always guarantee an optimal policy. This concept applies and extends to the *JS-VI* algorithm.

The convergence criterion of the value iteration algorithm is that the algorithm will stop the iteration process when the value function changes only by a small amount. This criterion can be used in the *JS-VI* algorithm as the algorithm terminates when evaluating the value functions in the experiments. Furthermore, the only adjustment to the *JS-VI* algorithm from the single-agent value iteration algorithm is the use of joint states. This adjustment only changes the representation of the value function, so the theoretical limits and properties are still preserved. Therefore, we can formulate the *JS-VI* algorithm as a standard (single-agent) value iteration problem trivially and thus maintain the convergence criterion.

The *JS-VI* algorithm is able to terminate (converge) for the $2, 3$ and 4 learning agents. This indicates that we can generate optimal policies using equation 5.1. Unfortunately, the state space of a 5-agent environment is too large for the algorithm to handle as the algorithm is forced to abort due to memory limitations.

Collision Avoidance

Consider a two-agent example, where agent 0 (red) and agent 1 (blue) are at positions $(1, 1)$ and $(2, 2)$ initially (figure 5.5a). The value of this joint state is 195. Using equation 5.1, we have obtained two joint actions $(right\ stop)$ and $(stop\ up)$. The two joint actions are indeed the optimal strategy in this scenario. Suppose that the joint action $(right\ stop)$ is chosen randomly, then the new joint state is $(1, 2)$ and $(2, 2)$ (figure 5.5b). In this new joint state, the value is 197 and the only optimal joint action is $(up\ up)$. Lastly, the new joint state is $(0, 2)$ and $(1, 2)$ and the value of this joint state is 199 with optimal joint action $(stop\ up)$ (figure 5.5c). Finally, the last agent will move to the goal state and the simulation ends. The values of all the joint states in this two-agent environment are listed in appendix B.



(a) Initial (b) Step 1 (c) Step 2

Figure 5.5: Two-agent Collision Avoidance

Similarly, in a three-agent example (figure 5.6), the value function is optimal as there are 3 joint actions that would give a value of 291. Also, in a four-agent example (figure 5.7), there are 4 joint actions that would give a value of 386. The agents in the two examples are able to avoid collisions as the policies are optimal.

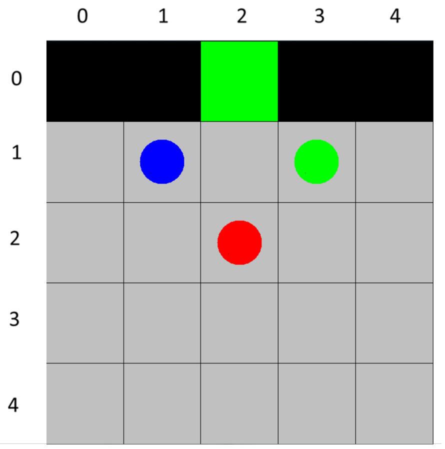


Figure 5.6: Three-agent Collision Avoidance

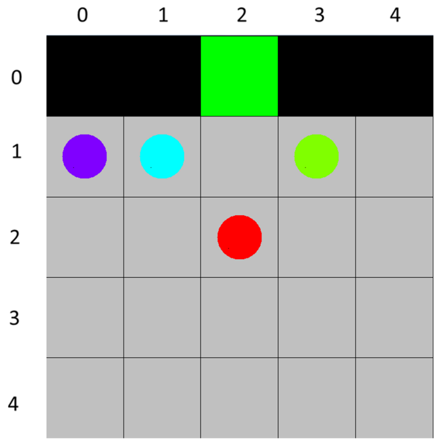


Figure 5.7: Four-agent Collision Avoidance

Average Steps

Similar to table 5.3, the average steps for the agents to arrive at the goal state is tabulated in table 5.4.

Number of Agents	Average steps
2	7.42
3	10.95
4	14.8

Table 5.4: Average steps obtained from the *JS-VI* optimal policies

The number of steps is averaged over 100 simulations for each set of policies (the $2, 3$ and 4-agent polices). The average steps obtained from the 2 and 3-agent policies are almost equivalent. Since, policies generated from value iteration are always guaranteed optimal provided the algorithm converges, we can conclude that the policies π_2^* and π_3^* described in 5.4.1 are optimal.

5.5 Chapter Summary

The fundamental difference between the JSA-Q_1 learning algorithm and the MMDP Q-learning algorithm is that while learning, agents take into account other agents' actions. And provided there is sufficient learning, agents are able to learn to coordinate their actions when arriving at a conflicting state so that a collision is avoided.

The limit to this learning algorithm is the exponential growth of a joint state action space. Despite this limitation, in the two-agent experiments, we are able to extract an optimal policy using this algorithm. The number of visited joint state action sets in the optimal policy π_2^* contains roughly 68% of the number of theoretical joint state action sets and the average steps for the agents to reach the goal state declines to a steady number, suggest that the *JSA*-Q_1 learning algorithm is converging and it is able to reduce the joint state action space.

The benefit of using the value iteration method is the guarantee of convergence and thus obtaining an optimal policy. The fact that the value function is optimal in the four-agent experiment shows that the *JS-VI* algorithm is a favourable option for finding an optimal policy where the agents are required to coordinate in conflicting states. The optimal policy is able to avoid agent collisions while providing accurate navigation.

Chapter 6

Conclusion

The single-agent reinforcement learning paradigm allows an agent to interact within an environment to achieve a given objective by means of trial and error. By extension, the multi-agent reinforcement learning paradigm allows a group of agents to interact within an environment as well as interact with one another.

While the Markov game provides a theoretically guaranteed framework for solving multi-agent reinforcement learning problems, it is nonetheless impractical to implement. This is caused by the PPAD-complete complexity of finding a Nash equilibrium, the existence of multiple equilibria and the exponential nature of the joint action or joint state action space.

To overcome these problems, work has been done on shifting multi-agent reinforcement learning problems from Markov games to learning on multiple levels. The *Coordinating Q-Learning* algorithm from section 2.3.4 is an example of learning on multiple levels. However, this method only works in a sparse environment.

Existing crowd simulation methods cover a wide array of advanced artificial intelligence approaches. However, the virtual crowd agents generated from these methods do not learn. As such, we have investigated the learning agents group behaviours by means of experimenting on simulating crowd behaviours using multi-agent reinforcement learning techniques.

To incorporate multi-agent reinforcement learning and crowd simulation, we have formulated the research question:

> *Given a 2D environment with a set of actors and objectives, can multi-agent reinforcement learning algorithm be used to produce policies that create believable crowd behaviour?*

To answer the question, we have devised four methods to simulate crowd. The methods are rule based, multi-agent Q-learning, joint-state-action Q-learning (*JS*-Q_1) and joint-state value iteration (*JS-VI*).

We begin the experiments with a rule based approach. Although it is not a reinforcement learning method, the rule based method has provided a ground rule for measuring the believability of the simulated crowd behaviour. The simulated rule based agents are predictable. To achieve a higher degree of realism or believability requires the design of complex rules and the adding of randomness which will be infeasible when simulating large crowds.

Despite violating the theoretical guarantee of convergence, the multi-agent Q-learning method under the MMDP framework has shown promising results. The density graphs have shown that the MMDP agents are able to follow some optimal policies and navigate to the goal states, provided there is sufficient learning. The density graphs have further confirmed that the simulated behaviour correlates to the behaviour observed from real-life simulations. The MMDP framework also has the capacity to cater for a larger environment and a larger number of agents.

On the other hand, agents learning in the MMDP framework are experiencing collisions. A simple reasoning regarding the collision is that the loss of theoretical guarantees caused by the chaotic (non-stationary) environment as agents are not able to account for other agents' actions. Although the agents failed to coordinate, this scenario best describes an emergency evacuation simulation where the collisions can be considered as stampedes occurring at mass gatherings.

It is inevitable that in a joint state action or joint state environment, the dimensions and the number of agents have to be greatly reduced. Although a crowd constitutes a large number of agents, we follow the multi-agent convention of equating a crowd to having more than 1 agent. Nevertheless, the results obtained from the simulations of smaller scaled environment are still worthy of note. One important concept in crowd navigation is the notion of a collision. Agents need to learn to avoid collisions by means of making way for other agents to pass through. Provided with sufficient learning in the *JSA*-Q_1 learning algorithm, the agents are able to learn policies that portray the coordination behaviour. Similarly, the policies generated from the *JS-VI* learning algorithm also illustrate such behaviour.

The *JSA*-Q_1 algorithm attempts to reduce the state space requirement by eliminating the need of exploring the full joint state action space. The algorithm shows promising results as the joint state action space needed for a sufficiently learnt policy is lower than one with a full joint state action space. Moreover, the joint state action scheme is a realistic representation of crowds moving simultaneously rather than moving in an orderly fashion as seen in the rule-based and MMDP approaches. Despite these advantages, this method is limited to a small scaled environment.

The fundamental difference between the *JSA*-Q_1 and *JS-VI* algorithms is that the former operates in a model-free environment while the latter operates in a model-based environment. Although both algorithms are able to produce believable behaviours, we can assume that in a perfect world, agents should not be able to have full knowledge of the environment they reside. Additionally, policies generated from the *JS-VI* algorithm are always optimal (global), indicating that agents will always make the optimal actions which are not the case in real crowds.

In conclusion, the three multi-agent reinforcement algorithms (Q-learning in MMDP, *JSA*-Q_1 and *JS-VI*) have answered the research question. The methods are able to generate policies that produce believable behaviours in multi-agent environments. The agents in the *JS-VI* algorithm are able to learn to coordinate, however, the nature of a model-based environment remains a challenge as real crowds do not possess full knowledge of the environment.

Future work

As mentioned, the main challenge of multi-agent reinforcement learning problem is the representation of states and actions. This challenge makes scaling up, both the environment dimension and the number of agents, difficult. State representation can be implemented directly as we have done in our experiments, however, only in a small scaled environment. One solution to improve the method of state representation is the use of *function approximation* [Sutton and Barto 1998]. Function approximation is a method of generalisation that aims at reducing the dimensions of the state space.

One method of function approximation is *tile coding* and it works by partitioning states into tiles where each partition is called a tiling. Each tile maintains a binary weight value where 1 indicates a particular state falls within that tile and 0 otherwise. By summing weights of the tiles, an approximate value of a state can be determined at any given point. Tile coding is easy to implement and is suitable for discrete state space problems.

Appendix A

The *Coordinating Q-Learning* algorithm discussed in section 2.3.4 on page 22 is outlined here.

Algorithm 7 *Coordinating Q-Learning*

```
Require:
  initialise $Q_i$ through single agent learning and $Q_i^j$
  while true do
    if state $s_i$ of Agent $i$ is marked then
      Select $a_i$ for Agent $i$ from $Q_i$
    else
      if the joint state information $js$ is safe then
        Select $a_i$ for Agent $i$ from $Q_i$
      else
        Select $a_i$ for Agent $i$ from $Q_i^j$ based on the joint state information $js$
      end if
    end if
    Sample $(s_i, a_i, r_i)$
    if t-test detects difference in observed rewards vs expected rewards then
      mark $s_i$
      for ∀ other state information present in the joint state$js$ do
        if t-test detects difference between independent state$s_i$and joint state$js$ then
          add $js$ to $Q_i^j$
          mark $js$ as dangerous
        else
          mark $js$ as safe
        end if
      end for
    end if
    if $s_i$ is unmarked for Agent $i$ or $js$ is safe then
      No need to update $Q_i(s_i)$
    else
      Update $Q_i^j(js, a_i) \leftarrow (1 - \alpha_t)Q_i^j(js, a_i) + \alpha_t[r(js, a_i) + \gamma \underset{a}{max}\, Q(s'_k, a)]$
    end if
  end while
```

Appendix B

The tables in this appendix contain the joint state values of a two-agent environment.

Joint State	Value
(0, 2) (0, 2)	0
(0, 2) (1, 0)	197
(0, 2) (1, 1)	198
(0, 2) (1, 2)	199
(0, 2) (1, 3)	198
(0, 2) (1, 4)	197
(0, 2) (2, 0)	196
(0, 2) (2, 1)	197
(0, 2) (2, 2)	198
(0, 2) (2, 3)	197
(0, 2) (2, 4)	196
(0, 2) (3, 0)	195
(0, 2) (3, 1)	196
(0, 2) (3, 2)	197
(0, 2) (3, 3)	196
(0, 2) (3, 4)	195
(0, 2) (4, 0)	194
(0, 2) (4, 1)	195
(0, 2) (4, 2)	196
(0, 2) (4, 3)	195
(0, 2) (4, 4)	194
(1, 0) (0, 2)	197
(1, 0) (1, 1)	195
(1, 0) (1, 2)	196
(1, 0) (1, 3)	195
(1, 0) (1, 4)	193
(1, 0) (2, 0)	193
(1, 0) (2, 1)	193
(1, 0) (2, 2)	195
(1, 0) (2, 3)	193
(1, 0) (2, 4)	193
(1, 0) (3, 0)	192
(1, 0) (3, 1)	193
(1, 0) (3, 2)	193
(1, 0) (3, 3)	193
(1, 0) (3, 4)	192
(1, 0) (4, 0)	191
(1, 0) (4, 1)	192
(1, 0) (4, 2)	193
(1, 0) (4, 3)	192
(1, 0) (4, 4)	191
(1, 1) (0, 2)	198
(1, 1) (1, 0)	195
(1, 1) (1, 2)	197
(1, 1) (1, 3)	195
(1, 1) (1, 4)	195
(1, 1) (2, 0)	194
(1, 1) (2, 1)	195
(1, 1) (2, 2)	195
(1, 1) (2, 3)	195
(1, 1) (2, 4)	194
(1, 1) (3, 0)	193
(1, 1) (3, 1)	194
(1, 1) (3, 2)	195
(1, 1) (3, 3)	194
(1, 1) (3, 4)	193
(1, 1) (4, 0)	192
(1, 1) (4, 1)	193
(1, 1) (4, 2)	194
(1, 1) (4, 3)	193
(1, 1) (4, 4)	192
(1, 2) (0, 2)	199
(1, 2) (1, 0)	196
(1, 2) (1, 1)	197
(1, 2) (1, 3)	197
(1, 2) (1, 4)	196
(1, 2) (2, 0)	195
(1, 2) (2, 1)	196
(1, 2) (2, 2)	197
(1, 2) (2, 3)	196
(1, 2) (2, 4)	195
(1, 2) (3, 0)	194
(1, 2) (3, 1)	195
(1, 2) (3, 2)	196
(1, 2) (3, 3)	195
(1, 2) (3, 4)	194
(1, 2) (4, 0)	193
(1, 2) (4, 1)	194
(1, 2) (4, 2)	195
(1, 2) (4, 3)	194
(1, 2) (4, 4)	193
(1, 3) (0, 2)	198
(1, 3) (1, 0)	195
(1, 3) (1, 1)	195
(1, 3) (1, 2)	197
(1, 3) (1, 4)	195
(1, 3) (2, 0)	194
(1, 3) (2, 1)	195
(1, 3) (2, 2)	195
(1, 3) (2, 3)	195
(1, 3) (2, 4)	194
(1, 3) (3, 0)	193
(1, 3) (3, 1)	194
(1, 3) (3, 2)	195
(1, 3) (3, 3)	194
(1, 3) (3, 4)	193
(1, 3) (4, 0)	192
(1, 3) (4, 1)	193
(1, 3) (4, 2)	194
(1, 3) (4, 3)	193
(1, 3) (4, 4)	192
(1, 4) (0, 2)	197
(1, 4) (1, 0)	193
(1, 4) (1, 1)	195
(1, 4) (1, 2)	196
(1, 4) (1, 3)	195
(1, 4) (2, 0)	193
(1, 4) (2, 1)	193
(1, 4) (2, 2)	195
(1, 4) (2, 3)	193
(1, 4) (2, 4)	193
(1, 4) (3, 0)	192
(1, 4) (3, 1)	193
(1, 4) (3, 2)	193
(1, 4) (3, 3)	193
(1, 4) (3, 4)	192
(1, 4) (4, 0)	191
(1, 4) (4, 1)	192
(1, 4) (4, 2)	193
(1, 4) (4, 3)	192

Table B.1: Value Function generated from a two-agent example - Part 1

Joint State	Value
(1, 0) (2, 4)	193
(1, 0) (3, 0)	192
(1, 0) (3, 1)	193
(1, 0) (3, 2)	193
(1, 0) (3, 3)	193
(1, 0) (3, 4)	192
(1, 0) (4, 0)	191
(1, 0) (4, 1)	192
(1, 0) (4, 2)	193
(1, 0) (4, 3)	192
(0, 2) (0, 2)	200
(0, 2) (1, 0)	197
(0, 2) (1, 1)	198
(0, 2) (1, 2)	199
(0, 2) (1, 3)	198
(0, 2) (1, 4)	197
(0, 2) (2, 0)	196
(0, 2) (2, 1)	197
(0, 2) (2, 2)	198
(0, 2) (2, 3)	197
(0, 2) (2, 4)	196
(0, 2) (3, 0)	195
(0, 2) (3, 1)	196
(0, 2) (3, 2)	197
(0, 2) (3, 3)	196
(0, 2) (3, 4)	195
(0, 2) (4, 0)	194
(0, 2) (4, 1)	195
(0, 2) (4, 2)	196
(0, 2) (4, 3)	195
(0, 2) (4, 4)	194
(1, 0) (0, 2)	197
(1, 0) (1, 1)	195
(1, 0) (1, 2)	196
(1, 0) (1, 3)	195
(1, 0) (1, 4)	193
(1, 0) (2, 0)	193
(1, 0) (2, 1)	193
(1, 0) (2, 2)	195
(1, 0) (2, 3)	193
(1, 0) (2, 4)	193
(1, 0) (3, 0)	192
(1, 0) (3, 1)	193
(1, 0) (3, 2)	193
(1, 0) (3, 3)	193
(1, 0) (3, 4)	192
(1, 0) (4, 0)	191
(1, 0) (4, 1)	192
(1, 0) (4, 2)	193
(1, 0) (4, 3)	192
(1, 0) (4, 4)	191
(1, 1) (0, 2)	198
(1, 1) (1, 0)	195
(1, 1) (1, 2)	197
(1, 1) (1, 3)	195
(1, 1) (1, 4)	195
(1, 1) (2, 0)	194
(1, 1) (2, 1)	195
(1, 1) (2, 2)	195
(1, 1) (2, 3)	195
(1, 1) (2, 4)	194
(1, 1) (3, 0)	193
(1, 1) (3, 1)	194
(1, 1) (3, 2)	195
(1, 1) (3, 3)	194
(1, 1) (3, 4)	193
(1, 1) (4, 0)	192
(1, 1) (4, 1)	193
(1, 1) (4, 2)	194
(1, 1) (4, 3)	193
(1, 1) (4, 4)	192
(1, 2) (0, 2)	199
(1, 2) (1, 0)	196
(1, 2) (1, 1)	197
(1, 2) (1, 3)	197
(1, 2) (1, 4)	196
(1, 2) (2, 0)	195
(1, 2) (2, 1)	196
(1, 2) (2, 2)	197
(1, 2) (2, 3)	196
(1, 2) (2, 4)	195
(1, 2) (3, 0)	194
(1, 2) (3, 1)	195
(1, 2) (3, 2)	196
(1, 2) (3, 3)	195
(1, 2) (3, 4)	194
(1, 2) (4, 0)	193
(1, 2) (4, 1)	194
(1, 2) (4, 2)	195
(1, 2) (4, 3)	194
(1, 2) (4, 4)	193
(1, 3) (0, 2)	198
(1, 3) (1, 0)	195
(1, 3) (1, 1)	195
(1, 3) (1, 2)	197
(1, 3) (1, 4)	195
(1, 3) (2, 0)	194
(1, 3) (2, 1)	195
(1, 3) (2, 2)	195
(1, 3) (2, 3)	195
(1, 3) (2, 4)	194
(1, 3) (3, 0)	193
(1, 3) (3, 1)	194
(1, 3) (3, 2)	195
(1, 3) (3, 3)	194
(1, 3) (3, 4)	193
(1, 3) (4, 0)	192
(1, 3) (4, 1)	193
(1, 3) (4, 2)	194
(1, 3) (4, 3)	193
(1, 3) (4, 4)	192
(1, 4) (0, 2)	197
(1, 4) (1, 0)	193
(1, 4) (1, 1)	195
(1, 4) (1, 2)	196
(1, 4) (1, 3)	195
(1, 4) (2, 0)	193
(1, 4) (2, 1)	193
(1, 4) (2, 2)	195
(1, 4) (2, 3)	193
(1, 4) (2, 4)	193
(1, 4) (3, 0)	192
(1, 4) (3, 1)	193
(1, 4) (3, 2)	193
(1, 4) (3, 3)	193
(1, 4) (3, 4)	192
(1, 4) (4, 0)	191
(1, 4) (4, 1)	192
(1, 4) (4, 2)	193
(1, 4) (4, 3)	192
(1, 4) (4, 4)	191
(2, 0) (0, 2)	196
(2, 0) (1, 0)	193
(2, 0) (1, 1)	194
(2, 0) (1, 2)	195
(2, 0) (1, 3)	194
(2, 0) (1, 4)	193
(2, 0) (2, 1)	193
(2, 0) (2, 2)	194
(2, 0) (2, 3)	193
(2, 0) (2, 4)	191
(2, 0) (3, 0)	191
(2, 0) (3, 1)	191
(2, 0) (3, 2)	193
(2, 0) (3, 3)	191
(2, 0) (3, 4)	191
(2, 0) (4, 0)	190
(2, 0) (4, 1)	191
(2, 0) (4, 2)	191
(2, 0) (4, 3)	191
(2, 0) (4, 4)	190
(2, 1) (0, 2)	197
(2, 1) (1, 0)	193
(2, 1) (1, 1)	195
(2, 1) (1, 2)	196
(2, 1) (1, 3)	195
(2, 1) (1, 4)	193
(2, 1) (2, 0)	193
(2, 1) (2, 2)	195
(2, 1) (2, 3)	193

Table B.2: Value Function generated from a two-agent example - Part 2

Joint State	Value
(2, 1) (2, 4)	193
(2, 1) (3, 0)	192
(2, 1) (3, 1)	193
(2, 1) (3, 2)	193
(2, 1) (3, 3)	193
(2, 1) (3, 4)	192
(2, 1) (4, 0)	191
(2, 1) (4, 1)	192
(2, 1) (4, 2)	193
(2, 1) (4, 3)	192
(2, 1) (4, 4)	191
(2, 2) (0, 2)	198
(2, 2) (1, 0)	195
(2, 2) (1, 1)	195
(2, 2) (1, 2)	197
(2, 2) (1, 3)	195
(2, 2) (1, 4)	195
(2, 2) (2, 0)	194
(2, 2) (2, 1)	195
(2, 2) (2, 3)	195
(2, 2) (2, 4)	194
(2, 2) (3, 0)	193
(2, 2) (3, 1)	194
(2, 2) (3, 2)	195
(2, 2) (3, 3)	194
(2, 2) (3, 4)	193
(2, 2) (4, 0)	192
(2, 2) (4, 1)	193
(2, 2) (4, 2)	194
(2, 2) (4, 3)	193
(2, 2) (4, 4)	192
(2, 3) (0, 2)	197
(2, 3) (1, 0)	193
(2, 3) (1, 1)	195
(2, 3) (1, 2)	196
(2, 3) (1, 3)	195
(2, 3) (1, 4)	193
(2, 3) (2, 0)	193
(2, 3) (2, 1)	193
(2, 3) (2, 2)	195
(2, 3) (2, 4)	193
(2, 3) (3, 0)	192
(2, 3) (3, 1)	193
(2, 3) (3, 2)	193
(2, 3) (3, 3)	193
(2, 3) (3, 4)	192
(2, 3) (4, 0)	191
(2, 3) (4, 1)	192
(2, 3) (4, 2)	193
(2, 3) (4, 3)	192
(2, 3) (4, 4)	191
(2, 4) (0, 2)	196
(2, 4) (1, 0)	193
(2, 4) (1, 1)	194
(2, 4) (1, 2)	195
(2, 4) (1, 3)	194
(2, 4) (1, 4)	193
(2, 4) (2, 0)	191
(2, 4) (2, 1)	193
(2, 4) (2, 2)	194
(2, 4) (2, 3)	193
(2, 4) (3, 0)	191
(2, 4) (3, 1)	191
(2, 4) (3, 2)	193
(2, 4) (3, 3)	191
(2, 4) (3, 4)	191
(2, 4) (4, 0)	190
(2, 4) (4, 1)	191
(2, 4) (4, 2)	191
(2, 4) (4, 3)	191
(2, 4) (4, 4)	190
(3, 0) (0, 2)	195
(3, 0) (1, 0)	192
(3, 0) (1, 1)	193
(3, 0) (1, 2)	194
(3, 0) (1, 3)	193
(3, 0) (1, 4)	192
(3, 0) (2, 0)	191
(3, 0) (2, 1)	192
(3, 0) (2, 2)	193
(3, 0) (2, 3)	192
(3, 0) (2, 4)	191
(3, 0) (3, 1)	191
(3, 0) (3, 2)	192
(3, 0) (3, 3)	191
(3, 0) (3, 4)	189
(3, 0) (4, 0)	189
(3, 0) (4, 1)	189
(3, 0) (4, 2)	191
(3, 0) (4, 3)	189
(3, 0) (4, 4)	189
(3, 1) (0, 2)	196
(3, 1) (1, 0)	193
(3, 1) (1, 1)	194
(3, 1) (1, 2)	195
(3, 1) (1, 3)	194
(3, 1) (1, 4)	193
(3, 1) (2, 0)	191
(3, 1) (2, 1)	193
(3, 1) (2, 2)	194
(3, 1) (2, 3)	193
(3, 1) (2, 4)	191
(3, 1) (3, 0)	191
(3, 1) (3, 2)	193
(3, 1) (3, 3)	191
(3, 1) (3, 4)	191
(3, 1) (4, 0)	190
(3, 1) (4, 1)	191
(3, 1) (4, 2)	191
(3, 1) (4, 3)	191
(3, 1) (4, 4)	190
(3, 2) (0, 2)	197
(3, 2) (1, 0)	193
(3, 2) (1, 1)	195
(3, 2) (1, 2)	196
(3, 2) (1, 3)	195
(3, 2) (1, 4)	193
(3, 2) (2, 0)	193
(3, 2) (2, 1)	193
(3, 2) (2, 2)	195
(3, 2) (2, 3)	193
(3, 2) (2, 4)	193
(3, 2) (3, 0)	192
(3, 2) (3, 1)	193
(3, 2) (3, 3)	193
(3, 2) (3, 4)	192
(3, 2) (4, 0)	191
(3, 2) (4, 1)	192
(3, 2) (4, 2)	193
(3, 2) (4, 3)	192
(3, 2) (4, 4)	191
(3, 3) (0, 2)	196
(3, 3) (1, 0)	193
(3, 3) (1, 1)	194
(3, 3) (1, 2)	195
(3, 3) (1, 3)	194
(3, 3) (1, 4)	193
(3, 3) (2, 0)	191
(3, 3) (2, 1)	193
(3, 3) (2, 2)	194
(3, 3) (2, 3)	193
(3, 3) (2, 4)	191
(3, 3) (3, 0)	191
(3, 3) (3, 1)	191
(3, 3) (3, 2)	193
(3, 3) (3, 4)	191
(3, 3) (4, 0)	190
(3, 3) (4, 1)	191
(3, 3) (4, 2)	191
(3, 3) (4, 3)	191
(3, 3) (4, 4)	190
(3, 4) (0, 2)	195
(3, 4) (1, 0)	192
(3, 4) (1, 1)	193
(3, 4) (1, 2)	194
(3, 4) (1, 3)	193
(3, 4) (1, 4)	192
(3, 4) (2, 0)	191
(3, 4) (2, 1)	192
(3, 4) (2, 2)	193
(3, 4) (2, 3)	192
(3, 4) (2, 4)	191
(3, 4) (3, 0)	189
(3, 4) (3, 1)	191

Table B.3: Value Function generated from a two-agent example - Part 3

Joint State	Value
(3, 4) (3, 2)	192
(3, 4) (3, 3)	191
(3, 4) (4, 0)	189
(3, 4) (4, 1)	189
(3, 4) (4, 2)	191
(3, 4) (4, 3)	189
(3, 4) (4, 4)	189
(4, 0) (0, 2)	194
(4, 0) (1, 0)	191
(4, 0) (1, 1)	192
(4, 0) (1, 2)	193
(4, 0) (1, 3)	192
(4, 0) (1, 4)	191
(4, 0) (2, 0)	190
(4, 0) (2, 1)	191
(4, 0) (2, 2)	192
(4, 0) (2, 3)	191
(4, 0) (2, 4)	190
(4, 0) (3, 0)	189
(4, 0) (3, 1)	190
(4, 0) (3, 2)	191
(4, 0) (3, 3)	190
(4, 0) (3, 4)	189
(4, 0) (4, 1)	189
(4, 0) (4, 2)	190
(4, 0) (4, 3)	189
(4, 0) (4, 4)	187
(4, 1) (0, 2)	195
(4, 1) (1, 0)	192
(4, 1) (1, 1)	193
(4, 1) (1, 2)	194
(4, 1) (1, 3)	193
(4, 1) (1, 4)	192
(4, 1) (2, 0)	191
(4, 1) (2, 1)	192
(4, 1) (2, 2)	193
(4, 1) (2, 3)	192
(4, 1) (2, 4)	191
(4, 1) (3, 0)	189
(4, 1) (3, 1)	191
(4, 1) (3, 2)	192
(4, 1) (3, 3)	191
(4, 1) (3, 4)	189
(4, 1) (4, 0)	189
(4, 1) (4, 2)	191
(4, 1) (4, 3)	189
(4, 1) (4, 4)	189
(4, 2) (0, 2)	196
(4, 2) (1, 0)	193
(4, 2) (1, 1)	194
(4, 2) (1, 2)	195
(4, 2) (1, 3)	194
(4, 2) (1, 4)	193
(4, 2) (2, 0)	191
(4, 2) (2, 1)	193
(4, 2) (2, 2)	194
(4, 2) (2, 3)	193
(4, 2) (2, 4)	191
(4, 2) (3, 0)	191
(4, 2) (3, 1)	191
(4, 2) (3, 2)	193
(4, 2) (3, 3)	191
(4, 2) (3, 4)	191
(4, 2) (4, 0)	190
(4, 2) (4, 1)	191
(4, 2) (4, 3)	191
(4, 2) (4, 4)	190
(4, 3) (0, 2)	195
(4, 3) (1, 0)	192
(4, 3) (1, 1)	193
(4, 3) (1, 2)	194
(4, 3) (1, 3)	193
(4, 3) (1, 4)	192
(4, 3) (2, 0)	191
(4, 3) (2, 1)	192
(4, 3) (2, 2)	193
(4, 3) (2, 3)	192
(4, 3) (2, 4)	191
(4, 3) (3, 0)	189
(4, 3) (3, 1)	191
(4, 3) (3, 2)	192
(4, 3) (3, 3)	191
(4, 3) (3, 4)	189
(4, 3) (4, 0)	189
(4, 3) (4, 1)	189
(4, 3) (4, 2)	191
(4, 3) (4, 4)	189
(4, 4) (0, 2)	194
(4, 4) (1, 0)	191
(4, 4) (1, 1)	192
(4, 4) (1, 2)	193
(4, 4) (1, 3)	192
(4, 4) (1, 4)	191
(4, 4) (2, 0)	190
(4, 4) (2, 1)	191
(4, 4) (2, 2)	192
(4, 4) (2, 3)	191
(4, 4) (2, 4)	190
(4, 4) (3, 0)	189
(4, 4) (3, 1)	190
(4, 4) (3, 2)	191
(4, 4) (3, 3)	190
(4, 4) (3, 4)	189
(4, 4) (4, 0)	187
(4, 4) (4, 1)	189
(4, 4) (4, 2)	190
(4, 4) (4, 3)	189

Table B.4: Value Function generated from a two-agent example - Part 4

References

[Bellman and Kalaba 1965] Richard Bellman and Robert Kalaba. Dynamic programming and modern control theory, 1965.

[Bowling and Veloso 2000] Michael Bowling and Manuela Veloso. An analysis of stochastic game theory for multiagent reinforcement learning. Technical report, Computer Science Department, Carnegie Mellon University, 2000.

[Buşoniu *et al.* 2008] L. Buşoniu, R. Babuška, and B. De Schutter. A comprehensive survey of multi-agent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, March 2008.

[Buşoniu *et al.* 2010] L. Buşoniu, R. Babuška, and B. De Schutter. Multi-agent reinforcement learning: An overview. In D. Srinivasan and L.C. Jain, editors, *Innovations in Multi-Agent Systems and Applications – 1*, volume 310 of *Studies in Computational Intelligence*, chapter 7, pages 183–221. Springer, Berlin, Germany, 2010.

[Cottle *et al.* 1992] Richard W Cottle, Jong-Shi Pang, and Richard E Stone. *The linear complementarity problem*, volume 60. Siam, 1992.

[Daskalakis *et al.* 2006] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. pages 71–78. ACM Press, 2006.

[Greenwald and Hall 2003] Amy Greenwald and Keith Hall. Correlated-q learning. In *In AAAI Spring Symposium*, pages 242–249. AAAI Press, 2003.

[Heïgéas *et al.* 2010] Laure Heïgéas, Annie Luciani, Joëlle Thollot, and Nicolas Castagné. A physically-based particle model of emergent crowd behaviors. *CoRR*, abs/1005.4405, 2010.

[Helbing *et al.* 2000] Dirk Helbing, Illes Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000.

[Hu and Wellman 2003] Junling Hu and Michael P Wellman. Nash Q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4(6):1039–1069, 2003.

[Johansson *et al.* 2007] Anders Johansson, Dirk Helbing, and Pradyumn K Shukla. Specification of the social force pedestrian model by evolutionary adjustment to video tracking data. *Advances in complex systems*, 10(supp02):271–288, 2007.

[Kachroo *et al.* 2008] Pushkin P.E Kachroo, Sabiha Amin Wadoo, Sadeq J Al-nasur, and Apoorva Shende. *Pedestrian Dynamics: Feedback Control of Crowd Evacuation*. Understanding Complex Systems. Springer, Berlin, Heidelberg, 2008.

[Kolokoltsov and Malafeyev 2010] Vassili N Kolokoltsov and Oleg A Malafeyev. *UNDERSTANDING GAME THEORY:Introduction to the Analysis of Many Agent Systems with Competition and Cooperation*. Number 7564 in World Scientific Books. World Scientific Publishing Co. Pte. Ltd., October 2010.

[Martinez-Gil *et al.* 2012] Francisco Martinez-Gil, Miguel Lozano, and Fernando Fernández. Multi-agent reinforcement learning for simulating pedestrian navigation. In *Adaptive and Learning Agents*, pages 54–69. Springer, 2012.

[Nowé *et al.* 2012] A. Nowé, P. Vrancx, and Y-M. De Hauwere. *Reinforcement Learning: State-of-the-Art*, chapter Game Theory and Multi-agent Reinforcement Learning, pages 441–470. Springer, 2012.

[Osborne and Rubinstein 1994] Martin J. Osborne and Ariel Rubinstein. *A course in game theory*. The MIT Press, July 1994.

[Osborne 2004] Martin J. Osborne. *An introduction to game theory*. Oxford Univ. Press, New York, NY [u.a.], 2004.

[Papadimitriou and Roughgarden 2004] Christos H. Papadimitriou and Tim Roughgarden. Computing equilibria in multi-player games. In *In Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA*, pages 82–91. SIAM, 2004.

[Reynolds 1987] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphics*, pages 25–34, 1987.

[Reynolds 1999] Craig Reynolds. Steering behaviors for autonomous characters, 1999.

[Schadschneider *et al.* 2009] Andreas Schadschneider, Wolfram Klingsch, Hubert Klüpfel, Tobias Kretz, Christian Rogsch, and Armin Seyfried. Evacuation dynamics: Empirical results, modeling and applications. In *Encyclopedia of complexity and systems science*, pages 3142–3176. Springer, 2009.

[Shapley 1953] L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

[Shoham and Leyton-Brown 2008] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York, NY, USA, 2008.

[Sutton and Barto 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction*. Cambridge, Massachusetts: MIT Press, 1998.

[Tesauro and Kephart 2002] Gerald Tesauro and Jeffrey O. Kephart. Pricing in agent economies using multi-agent q-learning. *Autonomous Agents and Multi-Agent Systems*, 5(3):289–304, September 2002.

[Thalmann and Musse 2007] Daniel Thalmann and Soraia Raupp Musse. *Crowd Simulation*. Number May. Springer, 2007.

[Torrey 2010] Lisa Torrey. Crowd simulation via multi-agent reinforcement learning. In G. Michael Youngblood and Vadim Bulitko, editors, *AIIDE*. The AAAI Press, 2010.

[Tu and Terzopoulos 1994] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 43–50, New York, NY, USA, 1994. ACM.

[Watkins and Dayan 1992] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):272–292, 1992.