Leveraging Nature-inspired Techniques to Semi-automate the Design of Multi-agent Systems

Nicola Mc Donnell M.Sc.

Co-supervisor: Prof. Jim Duggan

Co-supervisor: Dr. Enda Howley

A thesis presented in fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computer Science, University of Galway, Galway City, Ireland,

June 2023

Table of Contents

List of Figures

List of Tables

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others except as specified in the text and Acknowledgements.

Nicola Mc Donnell, June 2023

Acknowledgements

I am deeply grateful to my esteemed, patient and highly encouraging supervisors, Prof. Jim Duggan and Dr Enda Howley. This thesis was made possible by their wisdom, guidance and unerring support.

I would like to thank my Graduate Research Committee members Dr Conor Hayes, Dr Michael Hartnett and Dr. Michael Schukat, for their suggestions and support.

I wish to acknowledge the DJEI/DES/SFI/HEA Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and the excellent ICHEC user support team.

I wish to thank my dearest friends who never complained when I arrived late and left early with excuses about needing to "study"; the neighbours and the hiking girls, you know who you are.

Lastly, and most importantly, I am eternally grateful to my amazing family, Mum, Dad, the Donabate crew, and the best cousins ever, Sue and Nolz.

Abstract

Multi-agent systems (MAS) applications are used in many fields, from telecommunications network management to traffic simulation. Their utilisation is increasing with emerging technologies such as Edge computing and the Internet of Things. This research aims to enhance the use of nature-inspired techniques that semi-automate the development of decentralised heuristics for resolving complex systems. It explores a series of frontiers in these areas focusing on different techniques and problem domains.

The Gossip Contracts (GC) protocol is a novel communication protocol for multi-agent systems to facilitate decentralised cooperation strategies. It is a generic protocol that can be applied to many problems. Hence, it was used to develop a novel decentralised dynamic virtual machine consolidation (DVMC) strategy. This strategy was compared to two well-known DVMC strategies, *Sercon* and *ecoCloud*, using a cloud data centre model derived from a real-world dataset. The assessment considered various criteria and found that the GC-based DVMC had lower service level agreement violations than the other strategies.

The Evolved Gossip Contracts (EGC) framework is a novel evolutionary computing-based framework for designing decentralised heuristics. Using EGC, a decentralised heuristic for the NP-hard bin packing problem (BPP) was developed and shown to be better than two popular BPP heuristics, *Best Fit* and *First Fit*, for the datasets on which it was trained. Based on a literature review, EGC is the first evolutionary computing-based framework evaluated on an NP-hard problem.

The QD(λ) learning algorithm is a novel value-based MARL algorithm that leverages networked agents. It incorporates the advantages of transfer learning from the MARL QD learning algorithm and faster learning when rewards are delayed from the RL Q(λ) learning algorithm. It attained the lowest accumulative error for larger environments that require agents to transition through several states before receiving a reward. Thus, QD(λ) learning is suited to sizeable multi-agent systems learning a communication protocol, an emerging research area that has seen significant activity in the last few years.

Acronyms

Acronym	Description or Citation
1D-BPP	One-Dimensional bin packing problem
AME	Accumulative Mean Error
Anthill	[Babaoglu et al., 2002]
ARMSE	Accumulative Root-mean-squared Error
AVF	Action-value Function
BPP	Bin Packing Problem
Contract Net	[Smith, 1980]
COHDA	Combinatorial Optimisation Heuristic for Distributed Agents [Hinrichs et al., 2014]
CSV	Comma-separated Value
DP	Dynamic Programming
DVMC	Dynamic Virtual Machine Consolidation
EC	Evolutionary Computing
ECJ	A Java-based evolutionary computation research system
ecoCloud	[Mastroianni et al., 2013]
EGC	Evolved Gossip Contracts
EPOS	Energy Plan Overlay Self- stabilisation [Pournaras et al., 2010]
eRBGP	Extended Rule-based Genetic Programming [Weise and Zapf, 2009]
GA	Genetic Algorithm
Gaia	[Wooldridge and Ciancarini, 2000]
GAPS	Gradient Ascent in Policy Space [Varshavskaya et al., 2009]
GC	Gossip Contracts
GCT	Google cluster-usage traces
GE	Grammatical Evolution
GESwarm	[Ferrante et al., 2013]
GP	Genetic Programming
I-EPOS	Iterative Economic Planning and Optimized Selection [Pilgerstorfer and Pournaras, 2017]
IaaS	Infrastructure as a Service
IQR	Interquartile Range

Acronym	Description or Citation
JA-BE-JA	A distributed balanced graph partitioning algorithm [Rahimian et al., 2015]
LGC	Learned Gossip Contracts
MAAD	MetaCompiler for Automatic Algorithm Discovery [Van Berkel et al., 2012]
MAOS	Multi-Agent Optimisation System [Xie and Liu, 2008]
MARL	Multi-agent Reinforcement Learning
MAS	Multi-agent systems
MaSE	Multiagent Systems Engineering [DeLoach et al., 2001]
MDP	Markov decision process
MIPS	Millions of Instructions Per Second
MW	Megawatts
OR	Operations Research
P2P	Peer-to-Peer
RL	Reinforcement Learning
RMSE	Root-mean-squared Error
SD	Standard Deviation
SelfLets	[Barbagallo et al., 2009]
Sercon	A centralised DVMC algorithm [Reiss et al., 2011]
SLA	Service Level Agreement
TD	Temporal Difference
THz	Terahertz
Tropos	An agent-oriented software development methodology [Bresciani et al., 2004]
TruCAOS+	A trust- and cooperation-based algorithm for open MAS [Anders et al., 2015]
V-MAN	A fully decentralised algorithm for consolidating VMs in large Cloud datacentres [Marzolla et al., 2011]
VM	virtual machine

List of Publications

- **Nicola Mc Donnell**, Enda Howley, and Jim Duggan. *"Dynamic virtual machine consolidation using a multi-agent system to optimise energy efficiency in cloud computing."* Future Generation Computer Systems 108 (2020): 288-301. https://doi.org/10.1016/j.future.2020.02.036

- **Nicola Mc Donnell**, Enda Howley, and Jim Duggan. *"Evolved Gossip Contracts-A Framework for Designing Multi-agent Systems."* Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I 16. Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-58112-1_44

- **Nicola Mc Donnell**, Enda Howley, and Jim Duggan. *"A Genetic Programming-based Framework for Semi-automated Multi-agent Systems Engineering."* ACM Transactions on Autonomous and Adaptive Systems (TAAS) (2023). https://doi.org/10.1145/3584731

- **Nicola Mc Donnell**, Enda Howley, and Jim Duggan. *"QD(λ) Learning: Multi-agent Reinforcement Learning algorithm with eligibility decay"* Adaptive and Learning Agents Workshop at AAMAS (2023) https://alaworkshop2023.github.io/papers/ALA2023_paper_41.pdf

CHAPTER 1

Introduction

1.1 Motivation

A review on the impact of Multi-agent systems (MAS) applications reported that they are now widely used in various fields, from telecommunications network management to traffic simulation [Herrera et al., 2020, Müller and Fischer, 2014]. MAS are collections of autonomous interacting agents operating in an environment to achieve a common goal [Răileanu and Borangiu, 2023, Balaji and Srinivasan, 2010]. This decentralised agent-based approach offers significant advantages over centralised methods for specific problem types, for example, systems composed of large numbers of independent entities, particularly when they can change rapidly [Davidsson et al., 2007]. MAS scale well, as there is no manager node to act as a bottleneck. They are resilient, as there is no single point of failure.

The popularity of MAS techniques is anticipated to grow in the near term with the advent of several critical emerging technologies, such as Edge computing and the Internet of Things [Mascardi and Weyns, 2018, Julian and Botti, 2019, Kong et al., 2022]. These technologies will impact many application domains. For example, on the individual level, wearables, personal devices, such as phones, [Subhan et al., 2023], and smart homes, especially given the ageing population [Kanchana et al., 2022]. While at the population level, the Internet of Things applications include smart grids, smart buildings, transportation and healthcare [Rejeb et al., 2022].

Serugendo et al. described self-organising within a system as "the mechanism or the process enabling a system to change its organisation without explicit external command during its execution time" [Di Marzo Serugendo et al., 2005]. One of the most exciting aspects of self-organising systems is when they exhibit emergence, where the whole behaviour is considered greater than the sum of the parts. More formally, an emergent property of a system does not exist at the lower level of the system but arises from the interactions of elements at the lower level [Gershenson, 2007].

However, the engineering of multi-agent systems is challenging. Mascardi states that "the systematic engineering of large-scale and open MAS still poses many challenges" [Mascardi et al., 2019]. Trianni coined this challenge "the design problem" [Trianni and Nolfi, 2011]. The most popular techniques, Gaia [Wooldridge and Ciancarini, 2000], Multiagent Systems Engineering (MaSE) [DeLoach et al., 2001], and Tropos [Bresciani et al., 2004], rely on human designers rather than automated or semi-automated approaches. Techniques to automate, or even semi-automate, the development process of MAS becomes even more valuable as demand for MAS grows. Van Berkel writes that "automatic algorithm generation for large-scale distributed systems is one of the 'holy grails' of artificial intelligence and agent-based modelling" [Van Berkel et al., 2012].

Self-organisation and emergence are pervasive in natural systems. As such, MAS designers employ nature-inspired techniques to aid their design. Examples include Gossip, a multicast routing algorithm for network communication inspired by how viral infections spread [Jelasity, 2011]. Evolutionary Computing (EC) is a family of algorithms inspired by natural/biological evolution. While Reinforcement Learning (RL), again a family of algorithms, is inspired by the trial and error learning exhibited by humans [Sutton and Barto, 2018].

The gossip protocol is inspired by gossiping in human societies, which is a highly effective means of spreading information [Jelasity, 2011] It relies on a network of connected nodes that can send messages to one another. The classic formulation is the Susceptible-Infected-Removed model, which mimics the spread of a viral infection. A *Susceptible* node has yet to receive the information, while an *Infected* node has already received the information and is actively sending it to its neighbouring nodes. It continues sending the information until it transitions to the *Removed* state.

Evolutionary computing (EC) is a trial-and-error method that evolves algorithms. It generates populations of candidate algorithms and tests to check their success. Then, these candidate algorithms evolve, mimicking the mechanisms of natural selection. It selects the best candidate algorithms as the parents of the next generation. Typically, EC alters the genomes of the selected parents by mutating or changing a part of the algorithms or swapping whole sections between parents.

Reinforcement learning (RL) is a single-agent learning framework

[Sutton and Barto, 2018]. With no prior knowledge, the agent wants to learn what action to take next, given the state of the environment at that time, to achieve its goal. It tries different actions, assesses their outcome, and learns the best actions to take. Typically, the agent meets its goal and realises the award at the end of a series of actions. Multi-agent Reinforcement Learning (MARL) extends single-agent RL, studying multiple agents operating in a shared environment.

1.2 Research Objectives

The central objective of this research is to:

> Enhance the use of nature-inspired techniques that semi-automate the development of decentralised heuristics for addressing challenges in complex systems.

This research focuses on various nature-inspired techniques: gossip protocol, evolutionary computing and reinforcement learning, and problem domains: the bin packing problem and the dynamic virtual machine consolidation problem. Three specific research objectives are listed below. They are presented in detail in the coming paragraphs.

1. Develop a genuinely decentralised DVMC strategy and evaluate it in a cloud data centre modelled using a real-world dataset, considering packing efficiency, the number of migrations and SLA violations.
2. Develop an EC-based framework to design decentralised heuristics evaluated on an NP-hard problem.
3. Develop a value-based MARL algorithm that combines the advantages of transfer learning while minimising the initial loss of learning speed.

Research objective 1 Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources and services. One of the service models in cloud computing is Infrastructure as a service (IaaS), where cloud computing providers supply infrastructure, such as compute resources, as a utility to consumers to meet their requests to host virtual machines (VMs). VMs are computing resources that use software instead of a physical computer to run programs. IaaS has seen rapid growth since the launch of Amazon EC2 in 2006.

Many VMs can run on one physical machine. The dynamic virtual machine consolidation (DVMC) problem is a resource allocation problem where VMs are packed into as few hosts as possible to minimise energy use in the cloud data centre. Decentralised

DVMC strategies should scale to large data centres and be resilient to coordinator node failures.

Several DVMC strategies that are classified as distributed or decentralised still rely on a manager [Mastroianni et al., 2011, Mastroianni et al., 2013, Yazir et al., 2010]. Other research oversimplified the cloud data centres assuming that the hosts and VMs are homogeneous [Marzolla et al., 2011, Sedaghat et al., 2014, Pantazoglou et al., 2015], or they neglect to model the VMs' workload or simply model it with synthetic random workloads [Barbagallo et al., 2009, Feller et al., 2012, Masoumzadeh and Hlavacs, 2016]. While all techniques considered the packing efficiency or the power consumption during their evaluations, and a number also considered the number of migrations [Murtazaev and Oh, 2011, Yazir et al., 2010], only one considered the performance degradation due to overloading a host [Masoumzadeh and Hlavacs, 2016].

Research objective 1 is to develop a genuinely decentralised DVMC strategy and evaluate it in a cloud data centre modelled using a real-world dataset, considering packing efficiency, the number of migrations and SLA violations.

Research objective 2 The idea of using evolutionary computing (EC) to assist in the development of MAS has a long history. Early research focused on proving EC could be an effective strategy for evolving a MAS and exploring which EC techniques and features best suit the challenge [Mitchell et al., 1994, Koza, 1994, Manela, 1994, Haynes et al., 1995b, Haynes et al., 1995a, Luke and Spector, 1996, Iba, 1996, Iba et al., 1997, Iba, 1998, Luke et al., 1997, Aronsson, 2003, Tavafi and Banzhaf, 2017]. However, a framework which can be applied to many unique problems would be more valuable. Several of these have been proposed; namely, Anthill [Babaoglu et al., 2002], MetaCompiler for Automatic Algorithm Discovery (MAAD) [Van Berkel et al., 2012], and GESwarm [Ferrante et al., 2013].

These frameworks are a significant contribution, and while they were evaluated on challenging distributed systems problems, none have been assessed on an NP-hard problem — a problem not solvable in polynomial time. One NP-hard problem is the Bin Packing Problem (BPP), where items of different sizes are packed into a finite number of bins of equal volume such that the number of bins used is minimal. The bins can be one too many dimensions. BPP has many variations, such as the knapsack problem and the cutting stock problem. These problems have many practical applications, from scheduling tasks to loading cargo into trucks, including the DVMC problem mentioned in research objective 1.

Research objective 2 is to develop an EC-based framework to design decentralised heuristics evaluated on an NP-hard problem, the bin packing problem.

Research objective 3 Transfer learning is a machine learning technique which leverages previous knowledge to improve learning speed or performance [Taylor et al., 2007]. When applied to MARL, this knowledge can come from another agent. Two value-based techniques have been proposed where MARL agents share information. They are QD-Learning [Kar et al., 2012] and an approach which augments classical temporal difference, TD(0), by incorporating updates received from neighbouring agents by a gossip-like mechanism [Mathkar and Borkar, 2016]. However, research on MARL notes that "agents engaging in partnership can significantly outperform independent agents although they may learn slowly in the beginning" [Tan, 1993].

Research objective 3 is to develop a value-based MARL algorithm that combines the advantages of transfer learning while minimising the initial loss of learning speed.

1.3 Research contributions

In addressing the above research objectives, this thesis makes a number of contributions to the advancement of the state of the art. Below is a summary the four key contributions of this thesis.

- **Gossip Contracts** — a novel communication protocol for networked multi-agent systems to facilitate decentralised cooperation strategies [Mc Donnell et al., 2020a]. Gossip Contracts (GC) is inspired by Contract Net, a communication protocol for a system of distributed problem solvers to facilitate allocating tasks across the system's agents [Smith, 1980]. GC leverages a Gossip-like multi-cast protocol to send tender messages further across the network.

- **Gossip Contracts-based DVMC strategy** — a novel decentralised DVMC strategy built using Gossip Contracts [Mc Donnell et al., 2020a]. It was evaluated using a cloud data centre model derived from a real-world dataset, the Google cluster-usage traces [Reiss et al., 2011]. The assessment considered packing efficiency, power consumption, the number of migrations and SLA violations. It was compared to two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, *ecoCloud*. The GC strategy outperformed the other strategies with respect to minimising SLA violations. However, it had similar power consumption to *Sercon*, which was lower than *ecoCloud*. In addition, it had equivalent or fewer migrations to *ecoCloud* but a higher number of migrations than *Sercon*.

- **Evolved Gossip Contracts** — a novel EC-based framework to design decentralised heuristics that leverage the GC protocol [Mc Donnell et al., 2020b]. The Evolved Gossip Contracts (EGC) framework uses Genetic Programming (GP) to semi-automate the search for a specific GC protocol. The EGC framework was used to develop GC-based decentralised heuristics for the NP-hard bin packing problem (BPP). The best BPP strategy developed by EGC is shown to be better than two BPP heuristics that are popular in the literature, *Best Fit* and *First Fit*, for the datasets on which it was trained.

- **QD(λ) Learning** — a novel value-based MARL algorithm that leverages networked agents. It incorporates the advantages of transfer learning from the MARL QD learning [Kar et al., 2012] algorithm and the faster learning when rewards are delayed from the RL Q(λ) Learning algorithm [Watkins, 1989]. The QD(λ) learning algorithm was evaluated in specially designed RL environments requiring agents to transition through several states before receiving a reward. The QD(λ) learning algorithm achieved the lowest accumulative root-mean-squared error when the environment has eight or more states.

- **Open Source Software Packages** — the software packages developed as part of this research are available online in support of Open Science [Mc Donnell, 2022]. They include the packages for the three research contributions: the GC protocol, the EGC framework and the QD(λ) learning algorithm, and the two packages of the simulators for evaluations: the Bin Packing Problem simulator and the DVMC simulator. Making these packages easily accessible will make this research easier to reproduce and thus extend the life of this work.

1.4 Structure of Thesis

The diagram in Figure 1.1 presents the overall structure of this thesis. It highlights the contributions, nature-inspired techniques and problem domains employed in each chapter.

- **Chapter 2 - Literature Review** highlights the research gaps and presents the foundational research for the Gossip Contracts protocol, the Evolved Gossip Contracts framework and the QD(λ) Learning algorithm. It introduces multi-agent systems, the Gossip and Contract Net protocols and existing methodologies for designing MAS. It describes the bin packing problem and the dynamic virtual machine consolidation (DVMC) problem, which were used to evaluate the contributions. It describes evolutionary computing, genetic programming and several

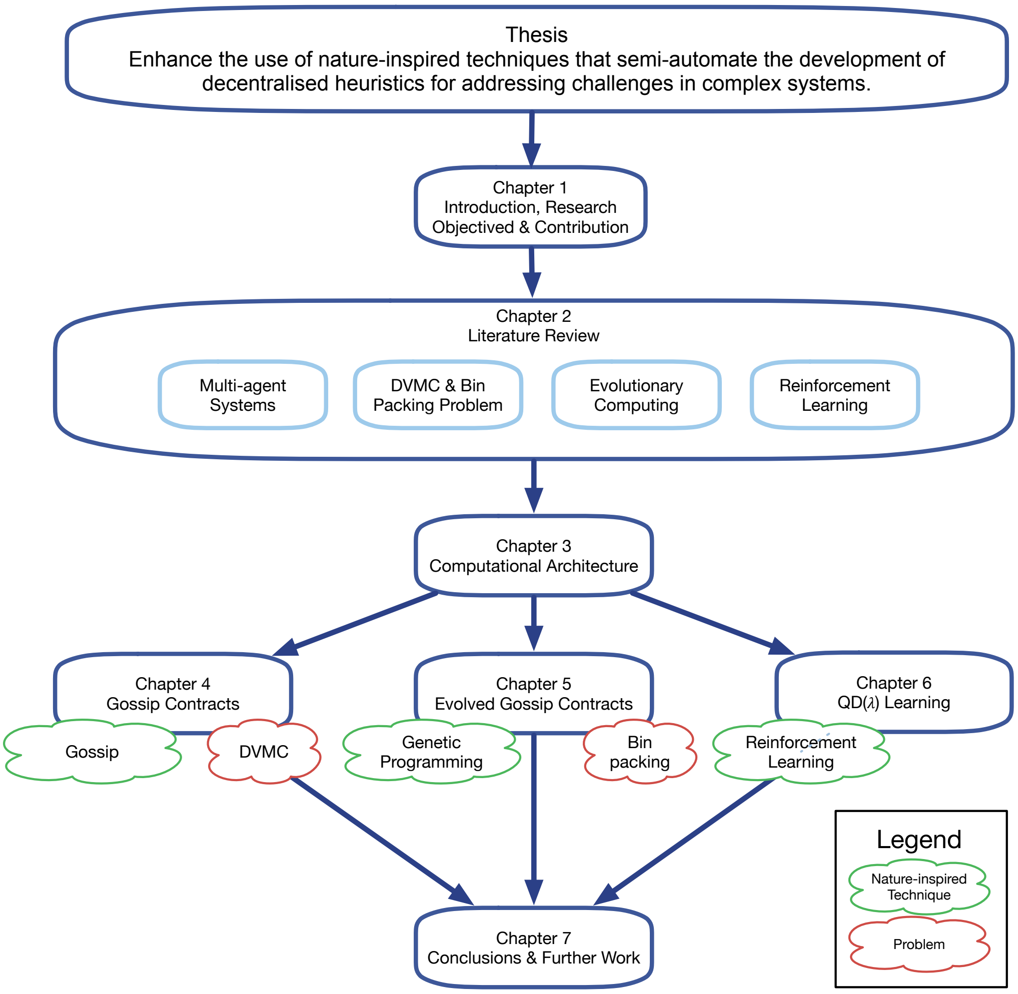


Figure 1.1: The broad structure of the thesis, highlighting the contributions, nature-inspired techniques and problem domains employed in each chapter

frameworks which use evolutionary computing to design MAS. It presents Reinforcement learning (RL), multi-agent reinforcement learning (MARL), and popular RL and MARL algorithms. The last section summarises the research highlighting the gaps which the research objectives address.

⇨ **Chapter 3 - Architecture** presents the architecture of the software and the packages developed as part of this research. Three of these packages are the core research output: the Gossip Contracts protocol, Evolved Gossip Contracts framework and the QD(λ) Learning algorithm, while two are the simulators used for evaluation: the Bin Packing Problem simulator and the DVMC simulator.

⇨ **Chapter 4 - Gossip Contracts** describes the Gossip Contracts protocol and how it was used to develop a fully-decentralised DVMC strategy. It presents the four experiments and the cloud data centre model they used. The experiments increase in size and complexity, with the last using real data centre data from a Google

Borg cell.

⇨ **Chapter 5 - Evolved Gossip Contracts** describes the Evolved Gossip Contracts (EGC) framework. EGC uses genetic programming to automate the search for a GC-based decentralised protocol. The chapter explains the four-step process of developing a multi-agent system with a desired macro behaviour. It shows how a MAS designer uses the Evolved Gossip Contracts framework to develop a decentralised bin packing heuristic. The experiments compare the new decentralised bin packing heuristic to two popular bin packing heuristics: *First Fit* and *Best Fit*.

⇨ **Chapter 6 - QD(λ) Learning** presents the QD(λ) learning algorithm, a novel value-based MARL algorithm that leverages networked agents. The MARL algorithm was evaluated in test reinforcement learning environments designed to require the agents to transition through several states before receiving a reward. The experiments compared it against two other RL algorithms and one MARL algorithm assessing its convergence speed and quality.

⇨ **Chapter 7 - Conclusion** completes the thesis with a review of the research objectives and the research contributions. It describes some of the limitations of this work and outlines suggestions for further research.

CHAPTER 2

Literature Review

2.1 Introduction

This chapter presents the foundational research for this thesis and the Gossip Contracts (EC) protocol, the Evolved Gossip Contracts (EGC) framework and QD(λ) Learning algorithm.

- Section 2.2 covers multi-agent systems (MAS), which are systems of autonomous interacting agents acting in an environment to achieve a common goal [Răileanu and Borangiu, 2023, Balaji and Srinivasan, 2010]. A recent survey on the impact of MAS applications reported that MAS is now widely used in various fields, from telecommunications network management to traffic simulation [Herrera et al., 2020, Müller and Fischer, 2014]. Two MAS protocols, Contract NET [Smith, 1980] and Gossip [Jelasity, 2011], inspired the Gossip Contracts protocol, which is the main contribution presented in Chapter 4. The PeerSim framework, a widespread network simulator commonly used to model and evaluate novel network protocols [Montresor and Jelasity, 2009], is used as the simulation technology for all the experiments in this thesis.

  Subsection 2.2.3 describes the most used methodologies for designing MAS in the literature. They are Gaia [Wooldridge and Ciancarini, 2000], Multiagent Systems Engineering (MaSE) [DeLoach et al., 2001], and Tropos [Bresciani et al., 2004]. These techniques rely on human designers rather than automated or semi-automated approaches.

- Section 2.3 presents bin packing problems where items of different sizes are packed into a finite number of bins of equal volume such that the number of bins used is minimal. These problems have arisen from the Operations Research (OR) field, which is introduced briefly in Subsection 2.3.1. Three classes of bin packing problems are described: the bin packing problem (BPP) and the Dynamic Virtual Machine Consolidation Problem (DVMC). These three problems build in complexity, and the most complex, the DVMC problem, is a real-world problem from the field of Cloud computing [Prabha et al., 2021, Chekuri and Khanna, 2004].

  The DVMC problem arises from increasing concern about the energy requirement for cloud computing data centres. Several strategies for DVMC are critiqued. Two are described in detail *Sercon*, a centralised strategy, and *ecoCloud*, a distributed strategy, as they are compared against the DVMC strategy developed in Chapter 4 using Gossip Contracts. Google's real-world dataset [Reiss et al., 2011] is describe as it is used for the evaluation of Gossip Contracts. Lastly, Subsection 2.3.4 gives examples of using multi-agent systems for solving optimisation problems.

- Section 2.4 presents Evolutionary Computing (EC), a family of algorithms inspired by natural/biological evolution. Subsection 2.4.1 describes Genetic Programming (GP) [Koza, 1992], the EC technique used in EGC, Chapter 5. Subsection 2.4.2 critiques research in multi-agent Systems and Evolutionary Computing. The early research teams developed bespoke solutions for unique problems, often the pursuit game or RoboCup agents, and focused on the GP techniques and features.

  Several frameworks have been proposed: Anthill by Babaoglu et al. [Babaoglu et al., 2002], MetaCompiler for Automatic Algorithm Discovery (MAAD) by Van Berkel et al. [Van Berkel et al., 2012], and GESwarm by Ferrante et al. [Ferrante et al., 2013]. While these frameworks are a significant contribution towards semi-automating the development of a MAS, none were evaluated against an NP-Hard problem, as is EGC in Chapter 5.

- Section 2.5 describes another nature-inspired technique commonly used with MAS, Reinforcement learning (RL), a trial-and-error learning framework [Sutton and Barto, 2018]. In RL, with no prior knowledge, the agent learns what action to take, given the state of the environment at that time, to achieve its goal. Two popular RL techniques, Q learning [Watkins and Dayan, 1992] and Watkins Q(λ) learning [Watkins, 1989], are described in detail as they are used in the experiments which evaluate QD(λ) learning in Chapter 6.

  Subsection 2.5.1 describes the research field of multi-agent reinforcement learning (MARL), which extends the single-agent RL paradigm to multiple agents operating in a shared environment. One classification of MARL techniques relates to whether the agents are controlled centrally or in a decentralised manner. In the fully decentralised learning scheme, each agent makes observations, takes actions, and receives rewards, essentially learning independently. While another, decentralised learning with networked agents, blends MARL and transfer learning [Taylor et al., 2007]. One technique from this space is QD Learning [Kar et al., 2012], which brings information shared by neighbouring agents into its learning algorithm. The novel MARL technique for networked agents, QD(λ) learning, presented in Chapter 6, is inspired by Watkins Q(λ) learning and QD Learning.

❖ The final section, Section 2.6, summarises the foundational research, highlighting the research gaps and introducing how these gaps are addressed in the thesis.

2.2 Multi-agent Systems

Multi-agent systems are a promising technique for distributed, complex and heterogeneous problem domains [Weiss, 1999]. Agent-based approaches outperform centralised methods for large, dynamic, inherently-modular systems with unstable communication [Davidsson et al., 2007]. A recent survey on the impact of MAS applications reported that MAS is now widely used in telecommunications network management; manufacturing and logistics; crowd, pedestrian, and traffic simulation; security and surveillance, and it reported energy systems as an emerging area [Herrera et al., 2020, Müller and Fischer, 2014]. With the rise of Edge or Fog computing [Bonomi et al., 2012], Internet of Things [Atzori et al., 2010], Smart Cities [Albino et al., 2015] and Smart Grids [Li et al., 2010], there is a growing need for multi-agent systems [Mascardi and Weyns, 2018, Julian and Botti, 2019].

A multi-agent system, MAS, is a system of autonomous interacting agents acting in an environment in order to achieve a common goal [Răileanu and Borangiu, 2023, Balaji and Srinivasan, 2010]. A MAS has multiple decision-making agents in a shared environment. They have a goal or goals which they are trying to achieve. Each agent can make observations about what is happening by sensing its environment. They can take actions which may modify the environment, and most importantly, they can interact with other agents; thus, the actions of one agent can impact another agent's behaviour.

A MAS can comprise agents which cooperate or compete. Cooperative MAS will share a common goal towards which they all work. While in a competitive MAS, the agents have individual goals but still interact to achieve them. Figure 2.1 shows three agents in an environment that are interacting with other agents and taking action to achieve their common goal.

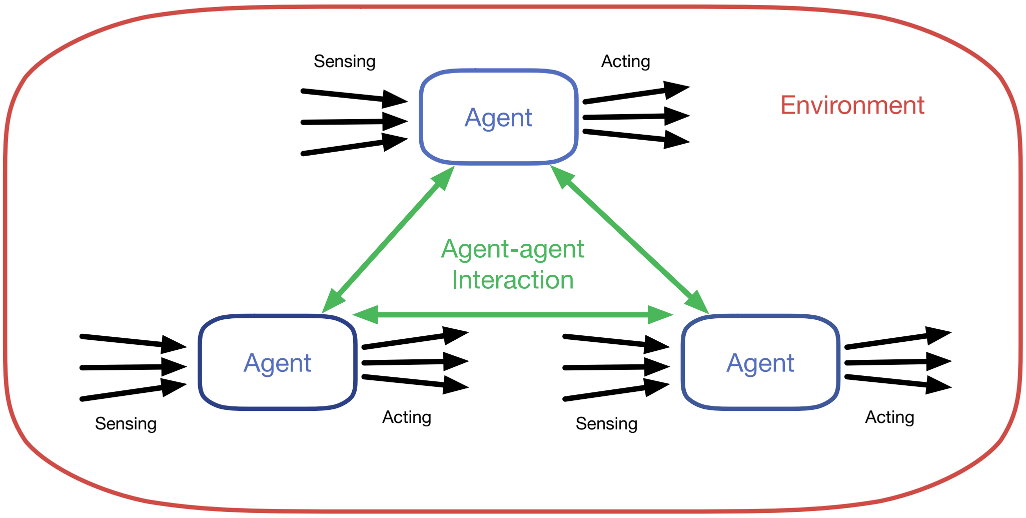


Figure 2.1: A Multi-agent System with three agents that are sensing the environment, interacting with other agents and taking actions to achieve their common goal [van Moergestel et al., 2013]

The field of MAS arose from the study of Distributed Artificial Intelligence in the 1980s. Parunak et al. presented several commercial applications of agent-based approaches used in manufacturing that primarily focused on shop-floor scheduling, and control [Parunak, 1996]. Also, around this time, Reid Smith developed the Contract Net protocol [Smith, 1980]. In is a task-sharing communication protocol used to allocate tasks among autonomous agents. It is prevalent and still used today, over forty years later. By the mid-1990s, the field of multi-agent systems was established as a research area in its own right.

One area of MAS research, practical reasoning, leverages classical planning systems. "Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes..." [Bratman, 1987]. It is distinct from theoretical reasoning, which relies on explicit logical reasoning. It comprises two different activities, deliberation and means-ends reasoning. While deliberating, an agent decides what it wishes to achieve. During the means-ends reasoning phase, the agent develops plans, which are sequences of actions, for how it will achieve its intentions. Belief-Desire-Intention is a software model for programming practical reasoning agents [Georgeff et al., 1998]. Beliefs are the agent's symbolic model of the environment. Desires represent the agent's motivations; objectives the agent wishes to accomplish. Intentions are the output of the deliberation phase; desires to which the agent chooses to commit [Wooldridge, 2009].

Contract Net and other early research assumed agents were benevolent rather than self-interested. However, by the mid-1990s, this was considered impractical for many real-world systems. Around this time, Jeffrey Rosenschein et al. introduced decision-theoretic and game-theoretic concepts to the multi-agent systems space to formalise and automate negotiations between the agents [Rosenschein and Zlotkin, 1994]. Research areas with a background in game theory include auctions, cooperative games and computational social choice. These remain active areas in MAS research.

In nature, self-organisation is pervasive, such as flocking and shoaling. Figure 2.2 shows a murmuration of starlings. It refers to a system composed of many elements, or agents, that exhibit coordinated behaviour without central control. Serugendo et al. [Di Marzo Serugendo et al., 2005] described self-organising within a system as "the mechanism or the process enabling a system to change its organisation without explicit external command during its execution time". One of the most exciting aspects of a self-organising or multi-agent system is when they exhibit emergence, where the whole behaviour is considered greater than the sum of the parts. More formally, an emergent property of a system does not exist at the lower level of the system but arises from the interactions of elements at the lower level [Gershenson, 2007]. Perhaps self-organisation is commonplace in natural systems because they are resilient and effective in a rapidly changing environment, as there is no single point of failure.

Figure 2.2: A murmuration of starlings by Walter Baxter, licensed under CC BY 2.0

2.2.1 Contract Net Protocol

The Contract Net protocol inspired the Gossip Contracts protocol. A Contract Net specifies the communication protocol between distributed problem solvers to facilitate allocating tasks across the system's agents [Smith, 1980]. It is customarily used to address optimisation problems in a decentralised way. Figure 2.3 shows the communication between a manager and a contracting agent using the Contract Net protocol. In Contract Net, the agent that tenders a task is called the *manager*. (a) It sends a *task announcement* to potential *contractors*. (b) Contractors can propose a bid to complete the task by sending a message back to the manager. Note that bids are binding, meaning the contractor may need to reserve resources to honour the bid if awarded. (c) The manager selects the best bid from all the bids and awards the contract to that contractor. (d) After the contractor completes the task, it sends a report to the manager, informing whether the task was completed successfully or if there was any failure.

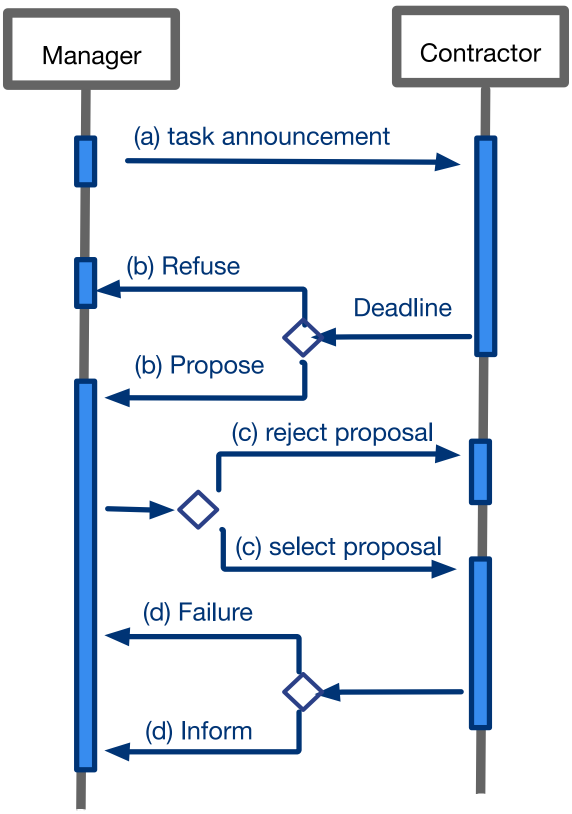


Figure 2.3: The Contract Net protocol showing the communication between a manager and contracting agent

Each task announcement has an expiry time, meaning a contractor does not need to bid immediately for a tender but can wait until just before it expires. Consequently, it can decide from all the tenders it has received by this time which, if any, is the best one to bid on. This limits the time a contractor is bound to a bid. Then, once a tender expires, all the bidding contractors are no longer bound.

Application of, and improvements to, Contract Net remains an active research area in diverse fields, from Unmanned Aerial Vehicles (UAVs) to Disaster Management. For example, Zhang et al. constructed a multi-constraint mathematical model and proposed a novel method based on a hybrid Contract Net protocol to address a multi-UAV dynamic task scheduling problem [Zhang et al., 2022]. While Richa et al. presented a geospatial agent-based simulator that utilises the Contract Net protocol for task assignment, allowing officials to visualise and analyse the influence of crowdsourcing in disaster management [Richa et al., 2023]. In Aerospace Engineering, Luo et al. proposed an asteroid terminal defence method utilising multi-satellite interception and an improved contract net protocol algorithm to allocate tasks and plan satellite orbits [Luo et al., 2022]. The wide-ranging application of the Contract Net protocol highlights its ongoing significance as an active research area.

2.2.2 Gossip Protocol

As the name suggests, the Gossip Contracts protocol leverages the Gossip protocol for communication. Gossip Protocol is inspired by gossiping in human societies, a highly effective means of spreading information [Jelasity, 2011]. It relies on a network of connected nodes which can send messages to one another. How information spreads across the network is similar to how a virus spreads.

A classic formulation of the Gossip protocol is the Susceptible-Infected-Removed model. A *Susceptible*, S, node has yet to receive the information. An *Infected*, I, node has already received the information. An *Infected* node will continue sending the information to other nodes until it stops, and then it is described as *Removed*, R. Algorithm 1 shows the pseudocode for the protocol.

Each node repeats the gossip cycle, waiting a time, δ, between each cycle. The node is either in a $push$ mode, a $pull$ mode or a $push - pull$ mode. In the $push$ mode, only infected nodes actively send information. All nodes actively pull information in the $pull$ and the $push - pull$ modes. When a node receives information, $update$, it will take one of two possible actions. If it gets the information for the first time, it stores it and moves to the *Infected* state. If it has already received the information, it sends a feedback message to the sending node. When a node receives a feedback message, it moves to the *Removed* state, R, with a probability of $1/k$. Thus, k, represents the duration of infectiousness.

Gossip-based broadcasting of information has a time complexity of $O(logN)$ and message complexity of $O(NlogN)$ [Jelasity, 2011]; the time complexity is how long an

Algorithm 1: Gossip Protocol (Susceptible-Infected-Removed model) [Jelasity, 2011]

```
repeat gossip cycle
    wait(Δ)
    p := random peer
    if push and in state I then
        send update to p
    end
    if pull then
        send update-request to p
    end
until forever

Function inUpdate(m: message)
    if push and in state I then
        send feedback to m.sender
    else
        store m.update
        switch to state I
    end
end

Function inUpdateRequest(m: message)
    if push and in state I then
        send update to m.sender
    end
end

Function onFeedback(m: message)
    switch to state R with prob 1/k
end
```

algorithm will take relative to the size of its input given as N, and the message complexity is the number of messages sent in a distributed system for a service request.

The multicast routing algorithm in Gossip Contracts resembles a flooding algorithm [Obraczka et al., 2001]. In flooding, each incoming message is forwarded to all its neighbours except the one from which it was sent. One flooding algorithm variant, hop-limited flooding, includes a hop count in the messages, and only messages that have taken less than an allowed number of hops are forwarded [Vojnović and Proutiere, 2011].

Recent research in gossip protocol has aimed to extend their applicability to emerging domains, such as Blockchain and Federated Learning. Commonly blockchain implementations use the gossip protocol to broadcast blocks or propagate information among peers. However, it has the shortcoming that as more peers join the blockchain network, duplication and bandwidth use increases [Antwi et al., 2022]. Gossip protocols have been applied in federated machine learning scenarios to facilitate information sharing and model aggregation among distributed nodes [Hegedűs et al., 2016], as described in the next section.

Gossip-based learning Federated Learning or Collaborative Learning is a distributed machine learning technique that enables on-device learning; on mobile phones or wearables. It addresses privacy concerns by keeping user data local. The general principle is to train local models on local datasets and exchange parameters, such as the weights and biases of a deep neural network [Konečnỳ et al., 2016]. Researchers from Google introduced it in 2016 [McMahan et al., 2017]. A recent article in Nature highlighted how privacy concerns and data silos limit the use of medical data for machine learning and how federated learning offers a solution to unlock this [Rieke et al., 2020].

One classification of federated learning algorithms is whether they are coordinated centrally or decentralised [Li et al., 2020]. In the centralised setting, a central server orchestrates and coordinates the different steps of the algorithms during the learning process. This setting leads to significant training latency in real-world scenarios. While in decentralised-federated learning, the nodes can coordinate themselves to obtain the global model; this reduces communication overhead as devices only communicate with their neighbours [Jiang and Hu, 2020].

Hegedűs et al. explored the integration of gossip protocols with federated learning and other distributed learning frameworks to improve convergence speed, communication efficiency, and fault tolerance [Hegedűs et al., 2016, Hegedűs et al., 2019]. They showed that the convergence speed was competitive while not relying on synchronisation and that the technique was robust to extreme and realistic failure scenarios.

PeerSim *PeerSim* [Montresor and Jelasity, 2009] is a popular network simulator commonly used to model and evaluate novel network protocols. It is Open Source and implemented in Java, thus supporting all JVM-based languages. This research uses it to assess the nature-inspired MAS protocol, Gossip Contracts, multi-agent reinforcement learning algorithm, QD(λ) Learning, and framework, Evolved Gossip Contracts, presented to semi-automate the development of decentralised MAS heuristics.

2.2.3 Engineering Multi-agent Systems

The engineering or design of a MAS has long been considered a challenge. Mascardi states that "the systematic engineering of large-scale and open

MAS still poses many challenges" [Mascardi et al., 2019]. González-Briones et al. stated that "because the MAS were conceived as a paradigm to respond to complex problems, the development of a MAS architecture is not easy" [González-Briones et al., 2018]. The most widely used methodologies in the MAS literature are Gaia [Wooldridge and Ciancarini, 2000], Multiagent Systems Engineering (MaSE) [DeLoach et al., 2001] and Tropos [Bresciani et al., 2004].

Gaia [Wooldridge and Ciancarini, 2000] views a multi-agent system as a computational organisation consisting of various interacting roles. The methodology guides a multi-agent system designer from a set of requirements to a detailed and implementable design. First, the designer discovers the organisational structure, a collection of roles agents can have, and how they relate to each other. Roles can form hierarchies, and groups can have similar patterns of interaction.

Each role comprises four attributes: responsibilities, permissions, activities and protocols. Responsibilities are the functionality of a role, while permissions govern the 'rights' or resources available to it. Activities are the computational actions associated with a role which the role owner can independently carry out. Finally, protocols define the means of interacting with other roles. The abstract organisational structure derived is refined until it is at a sufficiently low level of abstraction that it is implementable.

The Multiagent Systems Engineering (MaSE) [DeLoach et al., 2001] methodology uses several graphical models to describe a multi-agent system's goals, behaviours, agent types and agent-communication interfaces. It follows a seven-step process that transforms abstract models into concrete representations. The first three stages make up the *Analysis* phase, which identifies a structured set of goals and use cases that form the basis of a sequence diagram. Then, the roles and associated *tasks* required to satisfy the *goals* are identified. During the four steps of the *Design* phase, the roles and tasks are transformed into agent types and conversations. Users of the MaSE methodology considered it to be "relatively simple to use, yet is flexible enough to allow for a variety of solutions".

The Tropos methodology [Bresciani et al., 2004] augments the software development lifecycle with an agent-oriented view. It has several key concepts: *actor*, *goal* and *plan*. An *actor* is an entity with strategic interests or *goals*. It represents its knowledge of the environment as *beliefs*. An *actor's plan* is a *recipe* for achieving a *goal*. An *actor* can depend on another *actor* to deliver part of its *plan*. The methodology successively develops four distinct models around these concepts: the *actor* model, the *dependency* model, the *goal* model and the *plan* model.

Another technique is Gershenson's general methodology [Gershenson, 2007]. It defines the system as a set of agents working to achieve the "satisfaction" of the system as a whole. This system-level satisfaction is a combination of the weighted satisfaction of the agents. However, *friction* occurs when the increased satisfaction of one agent's goal causes a decrease in the satisfaction of the goal of another agent. Therefore, *mediators* arbitrate among the agents of a system to minimise friction. The methodology includes five steps, one of which involves simulating a solution with different scenarios or mediation strategies and evaluating them. Gershenson mentioned that one could use Reinforcement Learning and Evolutionary Computing techniques to reduce friction. He described them as "blind" because changes to the system are made randomly. But, they are helpful when the relationship between the agents' satisfaction and the system's satisfaction as a whole is obscure.

While these techniques are often cited, the details of whether or how they were used to develop the MAS are not typically available. Also, all of these techniques rely on human designers rather than automated or semi-automated approaches. Trianni et al. describe "the design problem" - given the desired macroscopic collective behaviour, it is not trivial to design the corresponding individual behaviours and interaction rules [Trianni and Nolfi, 2011]. Van Berkel et al. observe that "automatic algorithm generation for large-scale distributed systems is one of the 'holy grails' of artificial intelligence and agent-based modelling. Inventing such an algorithm usually involves a tedious reasoning process for each individual idea" [Van Berkel et al., 2012].

2.3 Bin Packing

In the Bin Packing Problem (BPP), items of different sizes are packed into a finite number of bins of equal volume such that the number of bins used is minimal. The bins can be one or more dimensions. In offline bin packing, the entire list of items and their sizes are known before packing begins. In online bin packing, the items arrive in a particular order and are packed without knowledge of the remaining items. Figure 2.4 shows the unpacked items in the online queue, items already packed in the bins and the blue item being packed into Bin 1.

This section describes two classes of bin packing problems used in the experiments: the bin packing problem (BPP) and the Dynamic Virtual Machine Consolidation Problem (DVMC). These three problems build in complexity, and the most complex, the DVMC problem, is a real-world problem. But first, the section presents a brief introduction to Operations Research and gives examples of using multi-agent systems for solving optimisation problems.

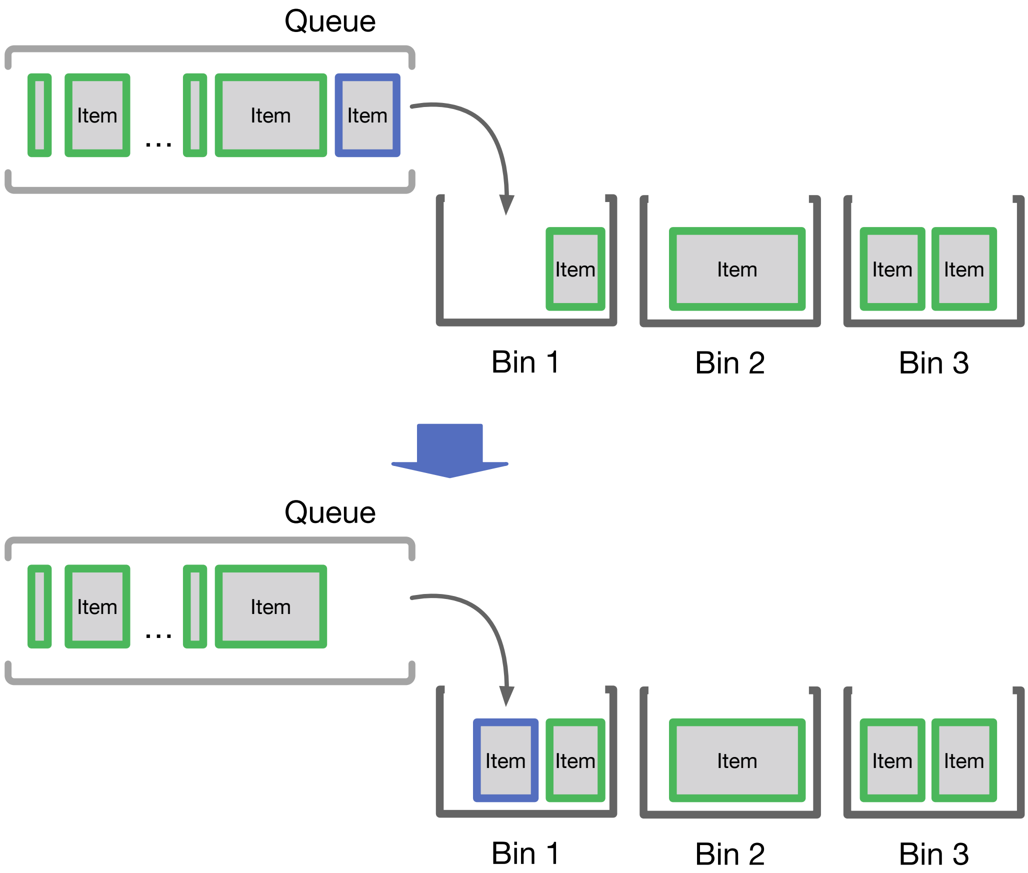


Figure 2.4: The bin packing problem shows the unpacked items in an online queue and the blue item being packed into Bin 1.

2.3.1 Operations Research

The Bin Packing Problem (BPP) is a classical combinatorial optimisation problem from the field of Operations Research (OR). OR is the use of analytical methods for decision-making. The area has roots in defence arising in Britain before World War II. However, OR techniques are now used across many industries, from engineering to finance to logistics [Carter and Price, 2017]. Traditionally these problems are solved using centralised techniques, for example, linear programming and branch and bound [Wolsey, 2020].

Operations research algorithms can be grouped in terms of their computational complexity. Three commonly referenced are P, NP and NP-hard, which are summarised in Table 2.1.

2.3.2 Bin Packing Problem

In the classical one-dimensional bin packing problem, there is a queue or sequence $L = (a_1, a_2, \ldots a_n)$ of items each with size $s(a_i) \leq (0, 1]$ which are packed into m, a minimal number of unit capacity bins, $B_1, B_2, \ldots B_m$, such that:

Table 2.1: Three computational complexity classes are P, NP and NP-hard

P	Deterministic polynomial-time is a complexity class where problems are deterministically solvable in polynomial time, for example, linear search.
NP	Nondeterministic polynomial-time complexity class where problems are not deterministically solvable in polynomial time, but their solutions can be verified in polynomial time, for example, integer factorisation.
NP-hard	Nondeterministic polynomial-time hard is a complexity class where problems are neither deterministically solvable in polynomial, nor can solutions be verified in polynomial time, for example, the subset sum problem.

$$\sum_{a_i \in B_j} s(a_i) \leq 1, 1 \leq j \leq m \tag{2.1}$$

Martello and Toth developed an exact algorithm for the one-dimensional bin packing problem (1D-BPP) called MTP [Toth and Martello, 1990]. While the solution produced by an exact algorithm is perfect, the time taken to make it will increase exponentially as the number of items to be packed increases. Hence, it is impractical for significant problems.

However, several non-optimal but efficient heuristics exist. The *Best Fit* algorithm packs the item into the fullest bin where it fits; ties are broken by favouring the lowest index bin. The *First Fit* algorithm packs each item into the first bin where it fits, possibly opening a new bin if the item cannot fit into any of the currently open bins [Dósa and Sgall, 2013]. To evaluate the new BPP strategies discovered using the EGC framework, they will be benchmarked against these two BPP heuristics.

More recently, research has focused on solving the BPP leveraging nature-inspired algorithms; such as the cuckoo search algorithm [Layeb and Boussalia, 2012, Zendaoui and Layeb, 2016], improved whale optimisation algorithm [Abdel-Basset et al., 2018], modified squirrel search [El-Ashmawi and Abd Elminaam, 2019] and swarm intelligence [Abd Elminaam et al., 2020]. These algorithms use nature-inspired metaheuristics to search for solutions or heuristics for specific bin packing problems. However, the heuristics it finds are centralised and will not have the advantage of no single point of failure as a decentralised heuristic, such as those developed using GC or EGC, will have.

2.3.3 Dynamic Virtual Machine Consolidation Problem

The Dynamic Virtual Machine Consolidation Problem (DVMC) problem is a variant of the vector bin-packing problem from the field of Cloud computing [Prabha et al., 2021, Chekuri and Khanna, 2004]. Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources, such as networks, servers, storage, applications, and services [Sabongari et al., 2020]. Furthermore, the resources can be rapidly provisioned and released with minimal management effort [Mell et al., 2011].

One of the service models in cloud computing is Infrastructure as a service (IaaS), where cloud computing providers supply infrastructure: compute resources, storage resources and network resources, as a utility to consumers to meet their requests to host virtual machines (VMs). A Virtual Machine (VM) is a computing resource that uses software instead of a physical computer to run programs and deploy apps. While many VMs run in one physical machine, or *host*, each runs an operating system and functions independently from the others. IaaS has seen rapid growth since the launch of Amazon EC2 in 2006.

An increasing concern in the field is the energy requirement for cloud computing data centres as it accounts for a significant fraction of the total operating costs [Prabha et al., 2021, Sabongari et al., 2020, Pahlevan et al., 2020, Barroso et al., 2013, Berl et al., 2010]. IaaS services allocate VMs to the hosts based on the maximum resources listed in the VM specification. However, typically VMs will use significantly fewer resources. Consequently, many hosts are underutilised. Greenberg et al. [Greenberg et al., 2008] found that an idle host can consume 70% of the power of a fully utilised machine.

A practical approach to minimise energy consumption is server consolidation or virtual machine consolidation. Virtual machine consolidation migrates VMs away from underutilised hosts, which are put into an energy-saving mode to preserve power consumption [Zhang et al., 2010]. Static virtual machine consolidation starts with empty hosts, and the problem is to pack the VMs efficiently into a minimum number of hosts; the allocation of the VMs can not be changed at runtime. Conversely, Dynamic virtual machine consolidation (DVMC) addresses the problem where the VMs are already allocated to the hosts, and they need to be migrated such that some of the hosts become idle and can be put into sleep mode [Khan et al., 2018]. For example, figure 2.5 shows a set of three hosts before and after consolidation.

Typically, a virtual machine consolidation algorithm has three stages: *Source Host Selection*, *VM Selection*, and *VM Placement* [Beloglazov and Buyya, 2012]. During *Source Host Selection*, it identifies the hosts from which it will migrate the VMs. Then

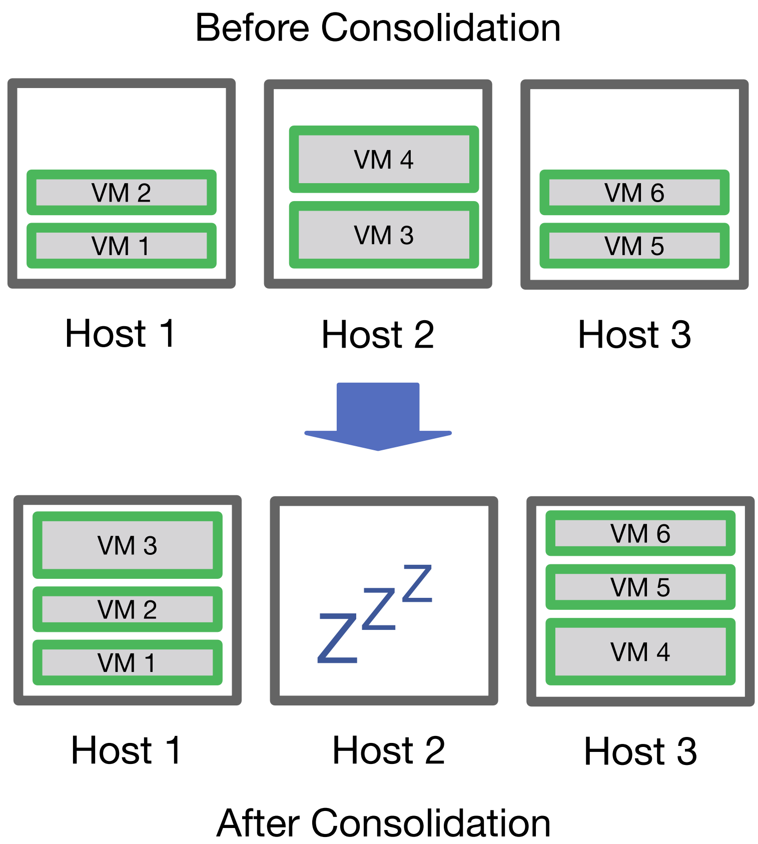


Figure 2.5: Dynamic Virtual Machine Consolidation of three hosts enabling Host 2 to move into an energy-saving mode.

during *VM Selection*, it selects the VM(s) to be relocated. Finally, during *VM Placement*, it decides the destination hosts for each migrating VM.

DVMC algorithms can be classified as either: centralised, where there is a central coordinating controller, or decentralised, where each host holds information and makes decisions in a decentralised manner. Many distributed and decentralised DVMC strategies are proposed in the literature, though this number is relatively small compared to the number of centralised strategies [Khan et al., 2018].

Research into distributed and decentralised DVMC approaches dates back to 2009. However, several DVMC strategies classified as distributed or decentralised still rely on a manager [Mastroianni et al., 2011, Mastroianni et al., 2013] or a global arbiter [Yazir et al., 2010]. Barbagallo et al. proposed a strategy based on the SelfLet Framework [Barbagallo et al., 2009]. Hosts have a scout worker who looks for better hosts for their VMs. The migration probability of a VM is inversely proportional to the Hosts load.

The complexity of the data centre models spans from modelling homogeneous hosts which can accommodate a fixed number of VMs [Marzolla et al., 2011, Sedaghat et al., 2014] to modelling heterogeneous types of hosts with heterogeneous types of VMs, which consider CPU, memory, and network I/O availability and usage [Ferdaus et al., 2014, Wuhib et al., 2015]. Not all approaches consider the workload on the VM. Of those that do, some model it as static [Barbagallo et al., 2009, Sedaghat et al., 2014], others as dynamically, changing over time [Pantazoglou et al., 2015, Masoumzadeh and Hlavacs, 2016]. Several approaches use workload data from actual data centres [Mastroianni et al., 2013, Wuhib et al., 2015]. Some research only presented the results of their strategy without comparison to other published techniques [Wuhib et al., 2015]. The cited research is summarised in Table 2.2.

Table 2.2: List of the cited research highlighting if the approach is fully decentralised, using heterogeneous hosts and virtual machines, and the nature of the virtual machine workload

	Fully Decentralised	Heterogeneous Hosts and VMs	Workloads
Barbagallo-2009 (SelfLets)	✓	Not reported	Static
Yazir-2010	Oracle	–	Dynamic
Reiss-2011 (Sercon)	–	–	Dynamic
Marzolla-2011 (V-MAN)	✓	–	Not modelled
Mastroianni-2013 (ecoCloud)	Manager	–	Real-world, PlanetLab
Sedaghat-2014	✓	–	Static
Wuhib-2015	✓	✓	Real-world, GTD
Pantazoglou-2015	✓	–	Dynamic
Masoumzadeh-2016	✓	✓	Dynamic

A review of distributed DVMC strategies completed by Ashraf et al. in 2016 identified four good classifications: heuristic, metaheuristic, machine learning, and statistical [Ashraf et al., 2018]. They report the most common technique as a local search, distributed or coordinated. Other common approaches are techniques based on Ant Colony Optimisation [Feller et al., 2012, Ferdaus et al., 2014, Farahnakian et al., 2014] and techniques based on gossip protocol [Marzolla et al., 2011, Sedaghat et al., 2014, Wuhib et al., 2015, Khelghatdoust et al., 2016, Masoumzadeh and Hlavacs, 2016].

In Chapter 4.2, the DVMC algorithm, which utilises the Gossip Contracts protocol, will be compared with two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, *ecoCloud*. One of these experiments uses a real-world data set provided by Google, the Google cluster-usage traces (GCT) [Reiss et al., 2011]. Next, *Sercon*, *ecoCloud* and the Google cluster-usage traces (GCT) are presented.

Sercon *Sercon* [Murtazaev and Oh, 2011] is a centralised DVMC algorithm which inherits properties from the well-known heuristic algorithms for bin-packing, *First Fit* and *Best Fit* [Murtazaev and Oh, 2011]. It aims not only to minimise the number of servers needed to host the VMs but also to minimise the number of VM migrations during the consolidation process.

The approach is to sort the hosts based on the current load and select the least loaded host. Then, the VMs are sorted based on load, and the largest VM is selected. Next, the algorithm attempts to allocate the VM to the most loaded server. If it fails, the algorithm assigns the VM to the next most loaded server. This process continues until it allocates all the host's VMs or reaches a threshold of allowed failed attempts. Once *Sercon* has migrated all the Host's VMs, it re-sorts based on load, selects the new least loaded server, and the process starts again.

To sort the servers, *Sercon* allocates a *surrogate weight* to each. It calculates the weight from the server's CPU utilisation and memory usage and the relative load of CPU and memory utilisation in the data centre. It also calculates a similar *surrogate weight* for sorting VMs, based on the CPU utilisation and memory usage of the VMs and the relative load of CPU and memory utilisation within the server. *Sercon* uses a CPU utilisation threshold, typically between 50% and 70%, when determining if a new VM can fit on a host.

Sercon introduces a new measure of DVMC effectiveness called *Migration Efficiency*, which is the number of servers released divided by the number of migrations. For example, if a DVMC algorithm migrates an average of ten VMs to release a server, its *Migration Efficiency* is 10%. Whereas, if it releases a server with each migration, its *Migration Efficiency* is 100%.

Sercon balances packing efficiency and the number of migrations. Murtazaev et al. reported a migration efficiency of 72% to 100%, compared to the First Fit Decreasing algorithm, which was 18% to 25%. However, it does not consider VM workloads that vary over time, so it has no concept of an overloaded server. Consequentially, in experiments with dynamic workloads, the CPU utilisation of the servers can exceed 100%, causing violations of the Service Level Agreement (SLA). Furthermore, when the overall load in the data centre increases, *Sercon* will not wake up sleeping hosts, which causes large numbers of SLA violations. So its packing efficiency becomes less realistic as a comparison .

ecoCloud *ecoCloud* is a distributed DVMC algorithm based on two probabilistic procedures [Mastroianni et al., 2013]. One is to decide if the server should migrate VMs, the *migration procedure*, and the second is to determine where the VM should be migrated to, the *assignment procedure*.

For the *migration procedure*, the server checks if its resource utilisations are between two thresholds, the low migration threshold, T_l^{eco} and the high migration threshold, T_h^{eco}; Typically, $T_l^{eco} = 0.5$ and $T_l^{eco} = 0.95$. Suppose one of the resource utilisation is outside the thresholds. In that case, the server evaluates a migration probability function by performing a Bernoulli trial. The probability of migration increases with the increasing distance of the resource from the threshold. If the trial is successful, the server begins the migration process for one of its VMs by sending a request to a coordinating DC Manager to assign the VM to another server.

The manager is aware of and able to communicate with every server in the data centre. When it is requested to assign a VM, it sends invitations to the servers to host the VM, thus delegating the decision to them. As with the migration procedure, the servers evaluate an assignment probability function by performing a Bernoulli trial where the trial's success is set to the value of the function. The overall assignment function is the product of evaluating separate assignment functions for CPU and memory utilisation. The assignment functions decrease as the utilisation approaches zero, and increases with proximity to a threshold, T^{eco}, above which the utilisation should not exceed. The manager then selects one server at random from the set of servers that offer to host the VM.

ecoCloud was compared against the centralised bin packing algorithm *Best Fit Decreasing* [Mastroianni et al., 2013]. While *Best Fit Decreasing* showed better consolidation, it also had substantially more migrations and a higher likelihood of overloading. *ecoCloud* uses fixed probabilities, so the data centre size influences its effectiveness. While *ecoCloud* is a distributed algorithm, it is not decentralised as it requires a coordinating Manager, which is a single point of failure.

Google cluster-usage traces Google has provided a real-world dataset for evaluating DVMC and other cloud data centre algorithms. The Google cluster-usage traces (GCT) [Reiss et al., 2011] represents 29 days' worth of Google Borg cell information from May 2011 on a cluster of about 12.5k machines. A cell is a set of machines, typically all in a single cluster, that share a cluster management system that allocates the work to the machines. There are ten distinct Machine Types.

Work arrives at a cell in the form of jobs, each composed of one or more tasks. Each task has a set of resource requirements that must be met when scheduling the tasks onto machines. The task resource requests include CPU and memory. At the start of the simulation period, over 130K tasks were already running, comprising 1404 distinct task types. Task resource usage is reported once for each measurement period, typically

5 minutes. CPU usage is average CPU usage during a second period picked uniformly at random from the 5-minute measurement period. It is measured in units of CPU-core seconds per second. If a task uses two cores all the time, it will show as two cores per second. The experiment described in Section 4.2.2 uses the first four hours of this dataset.

2.3.4 Optimisation Problems and Multi-agent Systems

There are many examples of using multi-agent systems for solving optimisation problems. Knotts et al. researched project scheduling with resource constraints, where they presented two techniques, one using reactive agents and the other using deliberative agents [Knotts et al., 2000]. Kwon et al. researched solving supply chain models using multi-agent and case-based reasoning, [Kwon et al., 2005]. Persson et al. developed two techniques, "embedded optimisation" and the "tactical hybrid approach", which are hybrids of classical optimisation and agent-based approaches [Persson et al., 2005]. Xie et al. researched the travelling salesman problem, where they presented Multi-Agent Optimisation System (MAOS), a nature-inspired cooperative search method [Xie and Liu, 2008].

More recently, Hinrichs et al. developed COHDA, Combinatorial Optimisation Heuristic for Distributed Agents, in which a system of self-organising agents cooperate through communication to optimise for a common global objective, and evaluated it with problems from the decentralised energy management space [Hinrichs et al., 2013, Hinrichs et al., 2014]. Rahimian et al. presented a distributed balanced graph partitioning algorithm, called *JA-BE-JA*, both for edge-cut and vertex-cut partitioning [Rahimian et al., 2015]. While *JA-BE-JA* is not a multi-agent system, it is designed for the case where the graph can be fully distributed, possibly with only very few vertices hosted on a single computer. Anders et al. presented TruCAOS+, a trust- and cooperation-based algorithm for open MAS, which they applied to an NP-hard resource allocation problem in power management systems [Persson et al., 2005, Anders et al., 2015].

Also, in the energy space, research by Pournaras et al. focuses on the problem of stabilisation in global resource utilisation. They propose Energy Plan Overlay Self-stabilisation (EPOS), a decentralised agent-based approach to stabilisation of resource utilisation at a global level using local coordination by agents organised into a tree hierarchy [Pournaras et al., 2010, Pournaras, 2013].

EPOS was first enhanced to deliver the Iterative Economic Planning and Optimised Selections (I-EPOS) [Pilgerstorfer and Pournaras, 2017, Pournaras et al., 2018], which no longer needed to perform expensive non-local brute-force operations. It was evaluated on a bicycle-sharing problem. Later, I-EPOS was embedded in EPOS Fog

[Nezami et al., 2021] and assessed on a load-balancing problem from fog computing. These proposals are a valuable contribution, especially given the unsupervised nature of the learning. However, there is a performance risk if there is a communication loss between two agents, as they are arranged as a tree hierarchy which the learning technique leverages.

2.4 Evolutionary Computing

Evolutionary Computing (EC) is a family of algorithms inspired by natural/biological evolution. Typically, EC algorithms are used for solving optimisation problems. They use a trial-and-error problem-solving method where populations of candidate solutions are generated and tested to check their success. Then the candidate solutions evolve, mimicking the mechanisms of natural selection. EC algorithms are meta-heuristics. A meta-heuristic is a search in the heuristic space, while a heuristic is a search in the solution space [Gendreau et al., 2010]. Other examples of metaheuristics include Ant Colony Optimisation, Tabu Search and Simulated Annealing.

EC algorithms encode the potential solutions to a problem in a *genome*. The EC process has several stages, as shown in Figure 2.6. During initialisation, EC randomly generates the individuals of the population. Then, it evaluates these individuals as potential solutions. The evaluation process typically involves running a model for each individual and calculating its fitness to solve a problem. Next, it selects the fitter individuals as the next generation's parents. During mutation and the application of other genetic operators, EC alters the genomes of the selected parents. Mutation involves randomly changing a section of the genome.

Finally, the new offspring are evaluated for fitness, with the fittest selected, mutated, etc., to create the next generation. This process continues for a specified number of generations or until the desired fitness is achieved.

While EC metaheuristics have these essential stages in common, many distinct types of algorithms exist. They differ in how they represent the genome, how their genetic operators manipulate it, and how they translate the genome into an evolved solution.

Research in evolutionary computing started in the 60s in three separate fields, Evolutionary Programming [Fogel, 1998], Evolutionary Strategies [Rozenberg et al., 2012] and Genetic Algorithms [Whitley, 1994]. Later in the 90s, a fourth field, Genetic Programming [Koza, 1992], emerged, and all four disciplines became collectively referred to as Evolutionary Computing (EC). Over the subsequent years, other representations and techniques have emerged.

In 1962, Fogel developed Evolutionary Programming to solve a prediction problem.

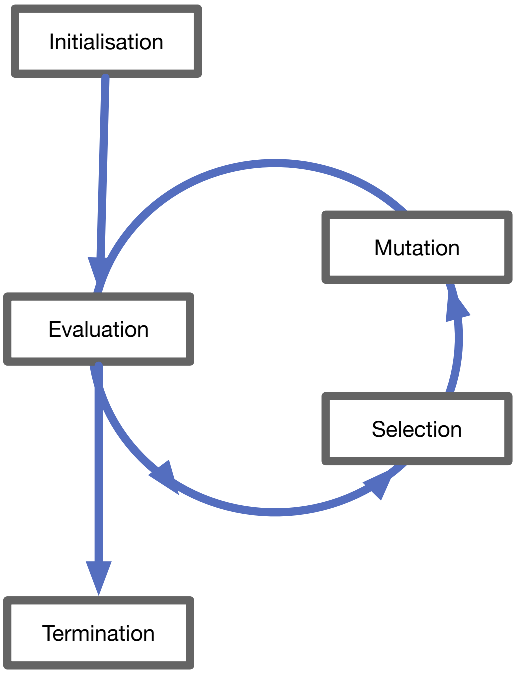


Figure 2.6: The flow through the stages of the evolutionary computing process.

The input to the problem was a sequence of symbols. The output was likewise a symbol. The algorithm has to predict what symbol to return to maximise the benefit. Then, he used it to evolve a finite state machine. The evolution involved adding and removing states and mutating state transitions [Fogel, 1998].

Then in 1964, Rechenberg and Schwefel presented Evolutionary Strategies [Rechenberg, 1965]. They applied it to a fluid dynamics optimisation problem. In Evolutionary Strategies, a possible solution to a problem is represented as a vector, the genome. The algorithm randomly selects some individuals and then mutates them, generating new individuals. All the individuals, original and mutated, are then ranked based on fitness, and the top individuals are selected for the next generation [Rozenberg et al., 2012].

In the 70s, Holland developed Genetic Algorithm (GA) [Holland, 1975, Whitley, 1994]. In GA, the genome is an array of bits, zeros and ones, and like Evolutionary Strategies, it must be decoded to translate it into a solution to the problem. It augments the mutation with a crossover operator, which can swap whole genome sections between parents when forming the next generation.

In Genetic Programming (GP), the genome representation is a computer programme's syntax tree. Typically, a GP tree is a syntax tree consisting of constants, variables, and operators [Koza, 1992]. As GP is the evolutionary computing technique used in EGC, it will be described in detail in the next section.

A powerful EC technique, Grammatical Evolution (GE), was developed by Michael O'Neill in 1998 [Ryan et al., 1998]. In GE, the genome representation is an integer

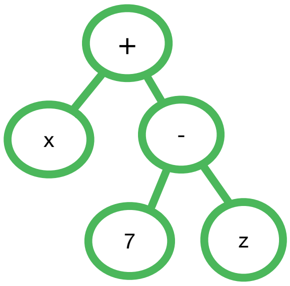


Figure 2.7: An example of a genetic programming tree for the expression $x + (7 - z)$.

vector which can translate into a programme using a predefined user-specified grammar. Typically usually a grammar in Backus-Naur form. The genetic operators used in GE are similar to those used in GA, but the genome encoding to the programme is unique to GE. Several examples of using GE to generate MAS are discussed in Section 2.4.2.

2.4.1 Genetic Programming

Our EGC framework uses Genetic Programming (GP) to search for decentralised algorithms which use the Gossip Contracts (GC) protocol. Genetic Programming (GP) is an evolutionary computing meta heuristic which generates solutions to problems as computer programmes [Koza, 1992]. The genome representation in GP is the programme's syntax tree, generally in LISP S-expression syntax. The tree comprises constants, variables, and operators. An example of a GP tree is shown in Figure 2.7 which represents the expression $x + (7 - z)$. This expression has five nodes, two binary arithmetic operators, $+$ and $-$, two variables, x and z, and one constant, 7.

John Koza, a PhD Student of Prof John Holland, the inventor of GA, developed genetic programming. Early results were published in 1989 [Koza et al., 1989]. GP has been applied to many problems, from facial recognition [Dai et al., 2011] to the Classification of Microarray Datasets [Liu et al., 2014].

The genetic operators typically used in GP are mutation and crossover. There are many variants of these operators' behaviour. Generally, one node, or a whole subtree, is replaced during Mutation with a new randomly generated node or subtree. The node to be mutated is randomly selected. Figure 2.8 shows the mutation of one node of the parent from 7 to a sub-expression $2/x$. During mutation, GP needs to ensure that the child is syntactically correct, like the original parent.

During crossover, two of the selected parents swap nodes or whole sections. Figure 2.9 shows the swap of one node z on *Parent 1* with the subtree $2/x$ in *Parent 1* to form two new children, *Child 1* and *Child 2*. As with mutation, the child tree needs to be syntactically correct.

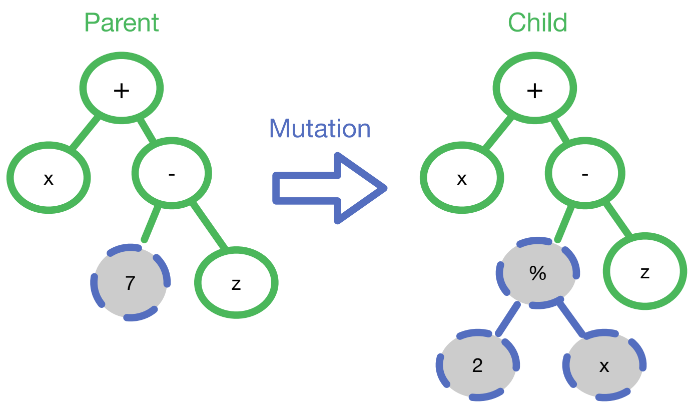


Figure 2.8: The genetic programming mutation of one node of the parent from 7 to a sub-expression $2/x$.

ECJ ECJ [Scott and Luke, 2019] is a long-established and popular evolutionary computing framework. It is Open Source and Java-based. It supports various genome representations, including genetic algorithm, genetic programming and grammatical evolution. In addition, it supports multi-threading and distributed architectures enabling computationally expensive experiments to be executed on supercomputers. The Evolved Gossip Contracts framework presented in Chapter 5 uses ECJ as its genetic programming framework.

2.4.2 Multi-agent Systems and Evolutionary Computing

The idea of using genetic programming to evolve emergent behaviour is as old as GP itself. Koza used GP to develop sets of simple rules to control a group of independent agents to exhibit emergent behaviour [Koza, 1994]. The same year, Mitchell et al. used a genetic algorithm (GA) [Whitley, 1994] to evolve cellular automata that performed one-dimensional density classification [Mitchell et al., 1994]. Manela developed a framework which used GA to find the optimal parameters for a set of architectural features of communicating agents and evaluated it using the pursuit game [Manela, 1994].

The pursuit game was a prevalent evaluation problem for much of the subsequent early work. Iba used it to evaluate his research into evolving cooperative behaviour for the communicating agents using Genetic Programming (GP) [Iba, 1996, Iba et al., 1997, Iba, 1998] as did Haynes et al. in their initial research exploring the concept of “Evolving a team”, leveraging co-evolution. This research focused on how different GP techniques, such as strongly-typed GP [Haynes et al., 1995b] and crossover [Haynes et al., 1995a], impacted effectiveness.

Subsequently, Haynes explored agents in hostile environments containing mines and energy sources. They showed that GP was highly successful at gener-

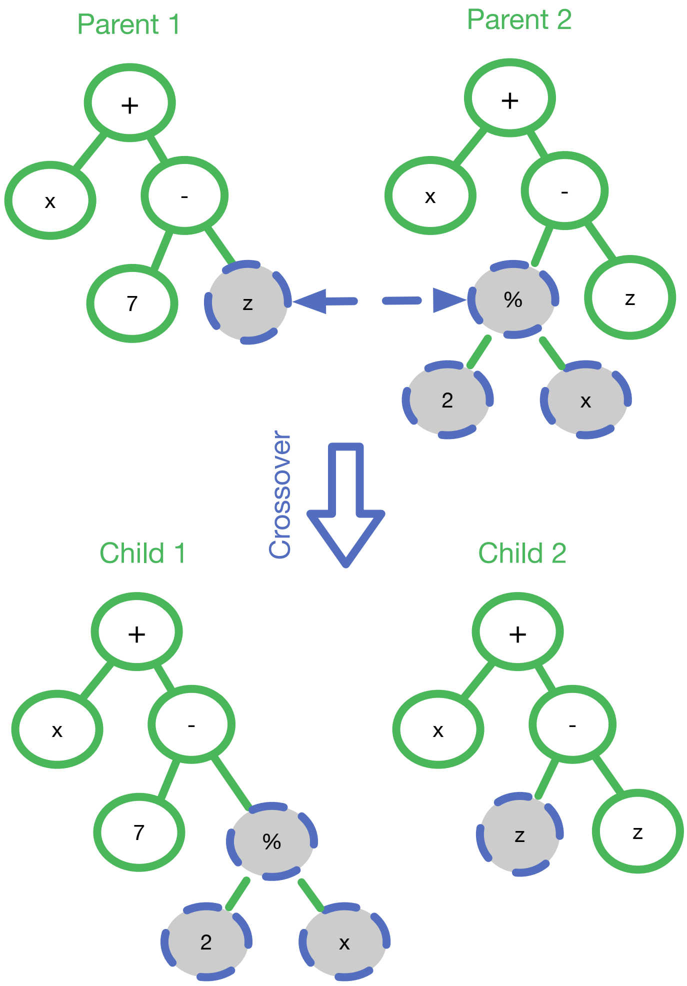


Figure 2.9: The crossover of one node z on *Parent 1* with the subtree $2/x$ in *Parent 1* to form two new children, *Child 1* and *Child 2*.

ating programmes controlling multiple agents cooperating in a fluctuating world [Haynes and Wainwright, 1995]. Luke et al. examined different breeding strategies and coordination mechanisms for evolving teams of agents in the Serengeti world, a predator/prey problem akin to the pursuit game [Luke and Spector, 1996]. Qureshi's research focused on using GP to program agents which communicate and interact to solve a problem [Qureshi, 1996]. He applied it to a simple problem involving two vehicle agents on a two-dimensional grid. He showed that the agents effectively decided what to communicate and when.

Weise et al researched using GP to design distributed algorithms [Weise and Zapf, 2009, Weise and Tang, 2011]. They evaluated the abilities of six different GP techniques on three distributed systems problems: election algorithms, the distributed mutual exclusion at a critical section, and the distributed computation of the greatest common divisor of a set of numbers. They found one of these techniques, Extended Rule-based Genetic Programming (eRBGP), to have performed remarkably better than the others.

Another problem space which saw research was the design of RoboCup agents [Aronsson, 2003, Tavafi and Banzhaf, 2017]. RoboCup is an annual competition between soccer-playing robots; one of the leagues is a simulation league. The RoboCup problem is substantially more complex than the previous problems mentioned.

Luke et al. evolved a team of identical players. The teams shared a strongly-typed genome [Luke et al., 1997]. The genome consisted of two trees, one controlling the kicking behaviour and the other managing the moving behaviour. They evaluated the evolved teams by playing them against each other. The research team reported that they were happy with their results, given the complexity of the problem they were trying to solve.

Tavafi et al. approached the RoboCup problem by breaking the evolutionary process into two phases [Tavafi and Banzhaf, 2017]. In the first phase, the players were evolved by evaluating them on off-game snapshots, situations in a game and the corresponding decisions tagged as correct or wrong. Then, the second phase assessed the best players from the first phase in the complete game environment. They found that their hybrid GP approach improved the overall performance of the final solutions substantially. The percentage of wins across 25 games for their hybrid GP approach was 64% and 100% when played against two top teams from the latest (2016) RoboCup.

The research presented above focuses on proving GP could be an effective strategy for evolving a communicating and cooperating MAS and exploring which GP techniques and features best suit the challenge. The research teams developed bespoke solutions for the unique problem they used to evaluate the MAS. EGC improves upon this as it is a framework designed to be applied to many problem spaces. However, several other frameworks have been proposed, which are described next.

The first of these is Anthill by Babaoglu et al. [Babaoglu et al., 2002]. An Anthill P2P system comprises a network of interconnected *nests* that can perform computations and host resources. User requests take the form of *ants* travelling around the *nest* network to satisfy the request. The ants don't interact directly with one another but by storing information in visited nests; this is akin to stigmergy in real ants. Anthill uses evolutionary techniques, specifically genetic algorithms, to design the ant algorithms. The parameters that define an ant algorithm's behaviour are its "genetic code". As an example, they used Anthill to implement a document-sharing application.

Van Berkel proposed MetaCompiler for Automatic Algorithm Discovery (MAAD), a so-called "Global-to-local compiler", to automatically generate local interaction rules between the constituent system elements starting from a global description of the system behaviour [Van Berkel et al., 2012]. It uses Grammatical Evolution to discover algorithms that fulfil the search goal. They demonstrated success with this technique by re-inventing five well-known distributed algorithms, including Leader election [Attiya and Welch, 2004] and Churn estimation [Hadden et al., 2007].

Ferrante et al. developed GESwarm, a new method for automatically synthesising collective behaviours for swarm robotics using Grammatical Evolution [Ferrante et al., 2013]. They demonstrated that seven out of ten outperformed the hand-coded foraging collective behaviour.

Zhong et al. proposed an evolutionary framework to automatically extract decision rules for agent-based crowd models to reproduce an objective crowd behaviour [Zhong et al., 2014]. The problem of finding optimal decision rules from objective crowd behaviours is formulated as a symbolic regression problem [Augusto and Barbosa, 2000] solved using gene expression programming.

These frameworks are a significant contribution towards semi-automating the search for a MAS which exhibits specific system-level behaviour, and they are evaluated on challenging distributed systems problems. But none were evaluated against an NP-Hard problem as is EGC in this research.

Earlier in 2022, Samarasinghe et al. published several research articles concerning a novel grammar-based evolutionary approach which allows the autonomous emergence of heterogeneity in collective behaviours. The agents must learn coordination, target search, and obstacle avoidance in a 2D world [Samarasinghe et al., 2022a], and an aggregated boid flocking behaviour [Samarasinghe et al., 2022b]. However, these systems do not fit the classification of a MAS, as there is no communication between the agents.

One exception was a proposal by Agogino et al., which demonstrated an evolutionary agent-based solution to the air traffic flow problem [Agogino and Tumer, 2007]. It used a simple evolutionary algorithm to set one parameter within the agent to control its behaviour. Each agent is responsible for one spatial area. The air traffic flow problem is NP-hard. However, as with the research presented by Samarasinghe [Samarasinghe et al., 2022a, Samarasinghe et al., 2022b], it does not fit the classification of a MAS, as there was no communication between the agents.

Based on a review of the literature, EGC is the first EC-based framework developed to design a decentralised heuristic to address an NP-Hard problem. Table 2.3 shows all the cited research and lists whether it uses Evolutionary Computing and whether it addresses an NP-hard problem.

Table 2.3: List of the cited research showing whether it presents a true multi-agent system, uses evolutionary computing and addresses an NP-hard problem.

Reference	Multi-agent Systems	Evolutionary Computing	Problem	NP-hard
Agogino-2007	–	✓	Air traffic flow	✓
Samarasinghe-2022	–	✓	Flocking behaviour	–
Smith-1980 (Contract Net)	✓	–	Resource allocation	✓
Knotts-2000	✓	–	Project scheduling	✓
Kwon-2005	✓	–	Supply chain	✓
Persson-2005	✓	–	Resource allocation	✓
Xie-2008 (MAOS)	✓	–	Travelling salesman	✓
Pournaras-2013 (EPOS)	✓	–	Energy management	✓
Anders-2015 (TruCAOS+)	✓	–	Energy management	✓
Hinrichs-2014 (COHDA)	✓	–	Energy management	✓
Rahimian-2015 (JA-BE-JA)	✓	–	Graph partitioning	✓
Pilgerstorfer-2017 (I-EPOS)	✓	–	Bicycle-sharing problem	✓
Pournaras-2018 (I-EPOS)	✓	–	Bicycle-sharing problem	✓
Nezami-2021 (EPOS Fog)	✓	–	Energy management	✓
Koza-1994	✓	✓	Ant foraging	–
Mitchell-1994	✓	✓	1D density classification	–
Manela-1994	✓	✓	Pursuit Game	–
Iba-1996 1997 1998	✓	✓	Pursuit Game	–
Haynes-1995 1996	✓	✓	Pursuit Game	–
Luke-1996 1997	✓	✓	Pursuit Game, RoboCup	–
Qureshi-1996	✓	✓	2D Grid Vehicles	–
Aronsson-2003	✓	✓	RoboCup	–
Weise-2009 2011	✓	✓	Distributed computing	–
Tavafi-2017	✓	✓	RoboCup	–
Babaoglu-2002 (Anthill)	✓	✓	Document-sharing	–
VanBerkel-2012 (MAAD)	✓	✓	Distributed computing	–
Ferrante-2013 (GESwarm)	✓	✓	Collective foraging	–
Zhong-2014	✓	✓	Symbolic regression	–

2.5 Reinforcement Learning

Reinforcement learning (RL) is a trial-and-error learning framework [Sutton and Barto, 2018]. In RL, an agent exists in an environment that it can observe. It has a goal it aims to achieve and will obtain a reward when it reaches it. With no prior knowledge, the agent wants to learn what action to take next, given the state of the environment at that time, to achieve its goal. The agent tries different actions and assesses their outcome to learn what to do. Its actions may alter the state of the environment. Generally, the agent meets its goal and realises the award at the end of a series of actions. This combination of *trial-and-error search* and *delay reward* is the hallmark of RL algorithms.

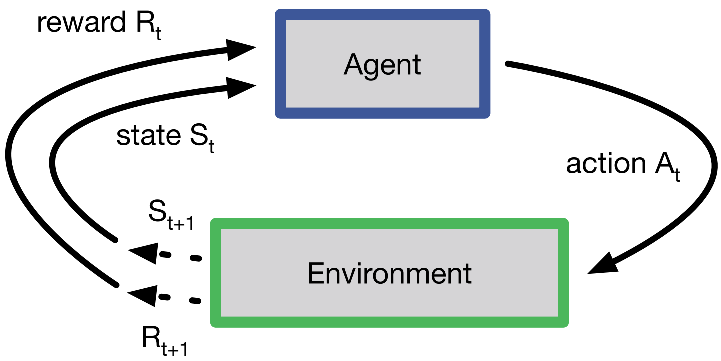


Figure 2.10: An agent perceives the state of the environment and takes an action which may result in a change to the environment and a reward [Sutton and Barto, 2018]

RL has its foundation in two long-standing fields: Animal Learning and Optimal Control. The field of Animal Learning presents the idea of learning by trial and error and rewarding to reinforce desired behaviour. It was Pavlov who first formally used the term "reinforcement" in the context of animal learning [Pavlov, 2010]. Optimal Control gives us the concept of value functions and the dynamic programming technique. Value functions predict the value of an environment in a particular state. Dynamic programming can calculate a value function if its possible to create a model of the environment [Bellman, 1966].

These two threads came together in the late 1980s with the development of Temporal Difference (TD) methods. TD aims to accurately predict a state's value given its future rewards [Sutton, 1984]. At each step, the technique compares the previous predictions to the rewards received and calculates the "temporal difference", known as the TD error. Then, the prediction is updated based on the error.

In RL, the agent's environment is described as a Markov decision process (MDP) [Bellman, 1957]. An MDP is a discrete-time stochastic control process [Puterman, 2014]. The agent uses a policy, π, to decide the best course of action given the state of the environment, s, at time t. The best action is determined based on the reward an agent receives, r, for taking a given action in a specific state and the value of the new state it transitions to, s'. States generally do not generate a reward in an MDP except for the "goal" state(s). Figure 2.10 illustrates this Reinforcement Learning framework.

The simplest formulation of an MDP is a 4-tuple $\langle S, A, T, R \rangle$, where:

- S is the set of environment states;
- A is the set of available actions to the agent;
- $T : S \times A \times S \rightarrow [0, 1]$ is a stochastic state transition function, where the state is transitioned to state $s\prime$ with a probability $0 \leq p \leq 1$ when applying action a in

state s; and

- $R : S \times A \times S \rightarrow R$ is the reward function which gives the reward, r, for the transition from state s to state s' having taken action a.

One class of reinforcement learning algorithms estimates the value of taking each action in a given state, referred to as a state-action pair. The set of values for all the possible state-action pairs is a state-action value function or Q function. $Q_\pi(s, a)$ gives the cumulative expected rewards starting from the state s taking action a and then following policy π. When accumulating the rewards, a discount factor, γ, can be applied, where $\gamma \in [0, 1)$. Discounting reduces the relative importance of future rewards compared to present rewards.

Given an AVF, a policy can be generated by selecting the action with the highest value in a given state; this is referred to as a greedy policy. However, to ensure the agent learns the best policy, it must explore some less valuable actions, which may lead to higher returns. A policy that takes a proportion of exploratory actions is called a soft or ϵ-greedy policy, where ϵ is the probability of taking an exploratory step. Typically $\epsilon = 0.1$, so on average, one in ten actions will be exploratory.

When the stochastic state transition function, T, and the reward function, R, are known, it is possible to create a model of the environment. Then, Dynamic Programming (DP) can be used to solve the state-action value function for an MDP [Bellman, 1966]. Conversely, in learning problems, the state transition function and the reward function are unknown, and $Q_\pi(s, a)$ is calculated from the observation of $\langle S, A, T, R \rangle$ tuples gathered through interactions with the environment. Following each observation and update of the Q function, the agent's policy is greedily optimised for the actions which lead to the best value in the state-action value function.

There are two classifications of tasks an agent can learn with RL: episodic and continuing tasks. Episodic tasks naturally break into subsequences called *episodes*, such as an agent learning to find a route through a maze. While continuing tasks continue with no natural breaks, like an agent learning to balance a pole. Therefore, the discount needs to be set to a non-zero value for continuing tasks so that the values in the action value function don't increase indefinitely.

Q Learning The Q learning algorithm is an offline temporal difference control algorithm [Watkins and Dayan, 1992]. Offline RL algorithms learn one policy while following a separate one, in contrast to online RL algorithms, which have only one policy followed and updated during learning. Temporal difference algorithms update their Q values by learning from experience and rewards and by estimating the value of the next state. Equation 2.2 shows the update rule for the Q function in the Q learning algorithm.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[r + \gamma \max_a Q(s',a) - Q(s,a) \Big] \tag{2.2}$$

The learning rate, α, such that $\alpha > 0$, is a step-size parameter to the Q learning algorithm. It weights the new learned value relative to its original value. It controls the rate at which the algorithm converges. The term, $\max_a Q(s', a)$, represents the maximum state-action value of all the possible actions that the agent can take in the state s'. Due to this term, the algorithm is classified as off-policy. Algorithm 2 shows the full Q Learning algorithm.

Algorithm 2: Q Learning

Algorithm Parameter: discount $\gamma \in [0, 1]$
Algorithm Parameter: step size $\alpha > 0$

Initialise: $Q(s, a) = 0$ for all $s \in S+$, $a \in A(s)$
Initialise: ϵ-greedy policy π randomly
Initialise: s to initial state
Initialise: a to initial action

foreach *learning step* **do**

Take action a and transition to state s'
$r :=$ get reward for transition (s, a, s')

Choose a' from $A(s')$ using π
$a^* := argmax_a\ Q(s', a)$ if a' ties for max then $a^* \leftarrow a'$

$Q(s, a) := Q(s, a) + \alpha\big[r + \gamma Q(s', a^*) - Q(s, a)\big]$

Improve π greedily with respect to Q
$s := s'$ and $a := a'$

end

Watkins Q(λ) Learning Watkins Q(λ) Learning is an extension to his earlier Q Learning algorithm that incorporates the concept of eligibility traces [Watkins, 1989]. An eligibility trace is a temporal record of taking an action in a state, or "passing through" a state-action. In Q(λ) Learning, when we update a state-action's Q value, we also update other recently visited state-actions. As a result, Q(λ) Learning may learn more efficiently.

A parameter to the Q(λ) learning algorithm, the eligibility decay, λ, controls how quickly the update of the visited state-actions decays. The lower the eligibility decay, the faster the update diminishes. When the agent takes action, a', and it is not the best action max_a, an exploratory step has been taken. Then all eligibility traces are reset to 0. The full Q(λ) Learning algorithm is shown in Algorithm 3.

Algorithm 3: Q(λ) learning

Algorithm Parameter: discount $\gamma \in [0, 1]$
Algorithm Parameter: step size $\alpha > 0$
Algorithm Parameter: eligibility trace decay rate $\lambda \in [0, 1]$

Initialise: $Q(s, a) = 0$, $e(s, a) = 0$ for all $s \in S+$, $a \in A(s)$
Initialise: ϵ-greedy policy π randomly
Initialise: s to initial state
Initialise: a to initial action

foreach *learning step* **do**
  Take action a and transition to state s'
  $r :=$ get reward for transition (s, a, s')
  Choose a' from $A(s')$ using π
  $a^* := argmax_a\ Q(s', a)$ if a' ties for max then $a^* \leftarrow a'$
  $\delta := \alpha\big[r + \gamma Q(s', a^*) - Q(s, a)\big]$
  $e(s, a) := 1$
  for *all* (s, a) *where* $e(s, a) > 0$ **do**
    $Q(s, a) := Q(s, a) + e(s, a)\delta$
  end
  for *all* (s, a) **do**
    if $a' = a^*$ **then**
      $e(s, a) := \gamma\lambda e(s, a)$ `// Decay eligibilities`
    else
      $e(s, a) :=$ 0 `// Took exploratory action`
    end
  end
  Improve π greedily with respect to Q
  $s := s'$ and $a := a'$
end

Several variants of Q(λ) learning differ in how they accumulate the eligibility traces, though they are similar in how they decay the eligibility. The variant presented in Algorithm 3 is *accumulating traces*, where it increments the state-action visited by one. *Accumulating traces* will cause the eligibility trace for a repeatedly visited state-action to grow considerably over time. This is especially true for continuing tasks, where the agent will learn to constantly take action to remain in, or return quickly to, a state with a high reward. A better approach for this case is *replacing traces*, which resets the visited eligibility trace to one, as shown in Equation 2.3.

$$e(s, a) = 1 \tag{2.3}$$

A third approach is *dutch traces*, where the original trace value is scaled by $1 - \alpha$ before it is incremented by one, as shown in Equation 2.4.

$$e(s, a) = (1 - \alpha)e(s, a) + 1 \tag{2.4}$$

2.5.1 Multi-agent Reinforcement Learning

Multi-agent Reinforcement Learning (MARL) is a popular research field [Shoham et al., 2003, Zhang et al., 2019]. It extends single-agent RL, studying multiple agents operating in a shared environment. Each agent receives its reward and aims to maximise its long-term return. MARL algorithms can be classified based on whether the agents cooperate or compete with one another. There are three classifications: fully cooperating, fully competing, or a mix of the two. All agents in the fully cooperative MARL setting usually share a common reward function. In contrast, a fully competitive MARL setting is typically modelled as a zero-sum game. This research focuses on decentralised heuristics that are cooperative MAS.

MARL research primarily focuses on stochastic games [Shapley, 1953], which are episodic tasks [Busoniu et al., 2008]. Stochastic games are the extension of an MDP to the multi-agent paradigm. There are two classifications of stochastic games: Markov games and extensive form games. In a Markov game, all agents act simultaneously, while in an extensive form game, the agents take turns to take action.

Another classification of MARL systems relates to whether the agents are controlled centrally or in a decentralised manner. Figure 2.11 shows three MAS learning schemes: centralised controller, fully decentralised and decentralised with networked agents [Zhang et al., 2021].

The central controller receives the observations from all the agents and decides the actions to be taken by each agent; it receives a joint reward. It follows the *centralised-learning-decentralised-execution* paradigm and has recently been widely adopted with multi-agent deep reinforcement learning research [Nguyen et al., 2020]. However, this scheme is only possible when a central controller can exist, which is often not the case.

In the fully decentralised learning scheme, each agent makes its observations, takes

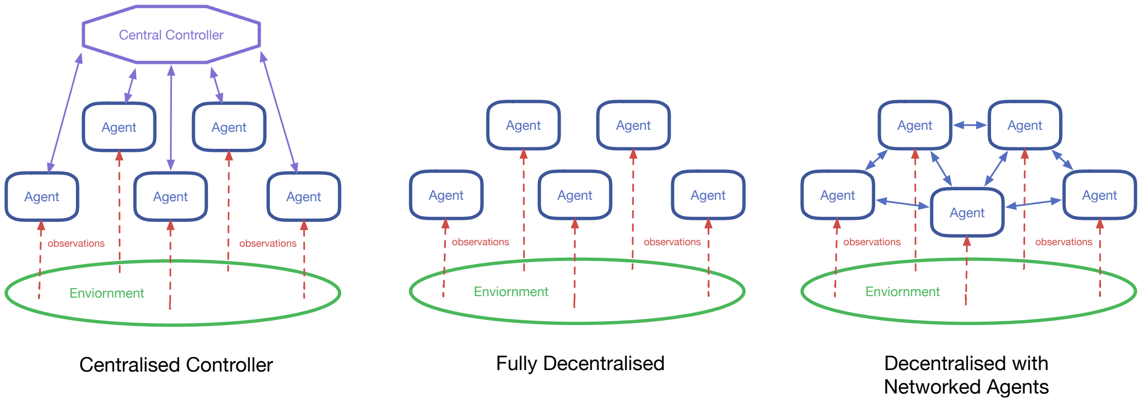


Figure 2.11: Three MAS learning schemes: centralised controller, fully decentralised and decentralised with networked agents [Zhang et al., 2021]

its actions and receives its reward. The agents learn independently. A shortcoming of this scheme is that it suffers from non-convergence [Tan, 1993]. A powerful extension to the fully decentralised learning scheme is where agents can communicate over a network. They exchange information with their neighbouring agents. This scheme facilitates convergence to an optimal decision policy where the performance of the MAS stabilises.

Tan et al. conducted the earliest research in this area [Tan, 1993]. They investigated whether cooperative agents outperform independent agents who do not communicate. They found that while sharing learned policies or episodes among agents speeds up learning, it comes at the cost of communication. For joint tasks, agents engaging in cooperating can significantly outperform independent agents. However, they may learn more slowly in the beginning.

Five years later, in 1998, Claus and Boutilier [Claus and Boutilier, 1998], researching stochastic games, coined the terms *Independent Learners* and *Joint-action Learners* and compared the two. The *Independent Learners* apply classical Q-learning to the MAS — the agents do not communicate or transfer learning. *Joint-action Learners* learning is equivalent to a fully decentralised MARL system. Joint-action Learners learn the Q values for joint actions, equivalent to a centralised controller. One of the challenges is that the state space grows with the number of agents.

In 2000, Lauer et al. proposed Distributed Q Learning, a model-free and coordination-free distributed Q-learning algorithm for the cooperative multi-agent Markov decision process [Lauer and Riedmiller, 2000]. They proved that it would find optimal policies if the environment is deterministic and the reward function is not negative. The algorithm is similar to each agent behaving independently with the Q learning algorithm, where the agent's rewards functions are identical. But, the Q values are only updated if they are greater than the current Q value. It addresses the situation where an agent can be punished for a wrong choice, even if it has chosen an optimal action. The agent should attach less importance to a punishment it received if it chose an action which historically received rewards.

Matignon et al. proposed Hysteretic Q learning in 2007 [Matignon et al., 2007]. It builds on the idea of *optimistic agents* in Distributed Q Learning, but instead of having no update for negative rewards, it has two learning rates, or step sizes, α and β. α is the step size when the error is positive, and β is the step size when the error is negative. If β is 0, this would have the equivalent of Distributed Q Learning. In their experiments, $\alpha \leftarrow 0.1$ and $\beta \leftarrow 0.01$. They found that Hysteretic Q learning performed close to centralised Q learning but did not suffer from the shortcoming of joint-action learners where the state space size depends on the number of agents in the MAS.

Zeng et al. proposed two novel hybrid approaches which combine real-time Q-learning and actor-critic methods [Zeng et al., 2020]. The second approach incorporated eligibility traces. These algorithms learn locally and make decisions independently. Their simulations showed that the convergence rate improved compared with the REINFORCE algorithm on a packet routing problem.

This thesis focuses on the MARL paradigm, where there is no central controller and hence no single point of failure. It explores decentralised learning with networked agents and compares it to a fully decentralised MARL system. Each agent learns the policy for a continuing task, as this type of MARL has many practical applications.

Transfer learning Transfer learning is a machine learning technique which leverages previous knowledge to improve the learning speed or performance [Taylor et al., 2007]. In multi-agent reinforcement learning, this knowledge can come from previous experience or another agent. The former, known as intra-agent transfer, is where an agent applies the learning from one task to a different task. The latter, inter-agent transfer, focuses on reusing knowledge received from another agent. The MARL scheme, decentralised learning of networked agents, is ideally suited to transfer learning.

Multi-agent RL with Networked Agents The earliest research into MARL with networked agents was conducted by Varshavskaya et al. in 2009 [Varshavskaya et al., 2009]. They developed DGAPS, an extension of the policy-based stochastic gradient descent algorithm, known as gradient ascent in policy space (GAPS), to the distributed realm by incorporating agreement algorithms. The agreement algorithm exchanges local rewards and experiences broadcasting to all the agents. One criticism the authors made of their work was that it sends a large amount of information. A preferred approach is to send the required information.

The first value-based MARL algorithm with networked agents was the QD Learning algorithm [Kar et al., 2012]. It is described in detail in a subsequent paragraph as it is foundational to the QD(λ) learning algorithm presented in Chapter 6.

In 2016, Mathkar et al. proposed another value-based distributed reinforcement learning algorithm [Mathkar and Borkar, 2016]. It builds on the classical TD(0) algorithm where the networked agents incorporate updates they received from their neighbouring agents using a gossip-like mechanism, the simple averaging scheme from gossip. They proved that convergence is guaranteed.

Zhang et al. proposed decentralised actor-critic algorithms with value function approximation in 2018. Each agent carried out its actor step, while for the critic step, the agents shared information over the network to reach a consensus. They evaluated a large number of agents, 20, and a large number of states, 20, with two possible actions [Zhang et al., 2018b]. In a follow-up publication later that year, they proposed a multi-agent version of expected policy gradient method, which they applied to continuous spaces [Zhang et al., 2018a]. They proved that convergence is guaranteed if linear functions are used for value function approximation.

QD Learning The QD learning algorithm extends the Q Learning algorithm to the multi-agent field where the agents' learning is decentralised [Kar et al., 2012]. They communicate and share information with agents in their neighbourhood. In QD learning, the update rule for the Q function has two terms, an *innovation* term and a *consensus* term. The innovation term corresponds to the agent's local knowledge of the reward and best next state-action value. The consensus term corresponds to information from the agent's neighbours. Equation 2.5 shows the Q function update rule for agent i. N is the set of agents in the agent i's neighbourhood. The consensus term has a new learning rate, β, where $\beta > 0$.

$$\begin{aligned} Q^i(s,a) \leftarrow Q^i(s,a) + &\underbrace{\alpha \left[r^i + \gamma \max_a Q^i(s',a) - Q^i(s,a)\right]}_{\text{innovation term}} \\ &\underbrace{- \beta \sum_{j \in = N^i} \left[Q^i(s,a) - Q^j(s,a)\right]}_{\text{consensus term}} \end{aligned} \tag{2.5}$$

Given certain conditions on the step sizes, α and β, the algorithm is guaranteed to converge to the optimum Q-function. Algorithm 4 shows the entire QD Learning algorithm.

Algorithm 4: QD learning

Algorithm Parameter: discount $\gamma \in [0, 1]$
Algorithm Parameter: innovation step size $\alpha > 0$
Algorithm Parameter: consensus step size $\beta > 0$

Initialise: $Q(s, a) = 0$ for all $s \in S+$, $a \in A(s)$
Initialise: ϵ-greedy policy π randomly
Initialise: s to initial state
Initialise: a to initial action
Initialise: connect each agent to N other agents randomly

foreach *learning step* **do**

Take action a and transition to state s'
$r :=$ get reward for transition (s, a, s')

Choose a' from $A(s')$ using π
$a^* := argmax_a\ Q(s', a)$ if a' ties for max then $a^* \leftarrow a'$

$\delta_i := \alpha\big[r + \gamma Q(s', a^*) - Q(s, a)\big]$ // Innovation Term
$\delta_c := \beta \sum_{j=N} \big[Q(s, a) - Q^j(s, a)\big]$ // Consensus Term

$Q(s, a) := Q(s, a)\big[\delta_i - \delta_c\big]$

Improve π greedily with respect to Q
$s := s'$ and $a := a'$

end

2.6 Discussion

Multi-agent Systems MAS research is valuable and has many real-world applications. In the mid-1990s, it was established as a research field in its own right, but it builds on other disciplines which date back to the early 80s. It remains popular today. MAS are widely used in telecommunications network management; manufacturing and logistics; crowd, pedestrian, and traffic simulation; and security and surveillance [Herrera et al., 2020, Müller and Fischer, 2014]. While the use of MAS in energy systems is an emerging field.

The application of MAS is anticipated to increase, given recent technological advances. New technologies, such as Edge or Fog computing, the Internet of Things, Smart Cities and Smart Grids, are ideally suited to MAS approaches. Thus, multi-agent system applications are expected to grow significantly [Mascardi and Weyns, 2018, Julian and Botti, 2019].

However, engineering multi-agent systems is challenging. The most popular techniques, GAIA, Multiagent Systems Engineering (MaSE) and Tropos, rely on human designers rather than automated or semi-automated approaches. Trianni coined this challenge "the design problem" [Trianni and Nolfi, 2011]. While Van Berkel said that "automatic algorithm generation for large-scale distributed systems is one of the 'holy grails' of artificial intelligence and agent-based modelling" [Van Berkel et al., 2012]. Thus, techniques to automate, or even semi-automate, the engineering of MAS would be a valuable contribution.

Bin Packing Problem Operations Research (OR) is a field that uses analytical methods for making decisions. It has applications across many industries, from engineering to finance to logistics [Carter and Price, 2017]. One class of OR problems is combinatorial optimisation, where we search for an optimal solution to a problem from a large but finite set of possible solutions. There are many examples of using MAS to solve optimisation problems, such as scheduling [Knotts et al., 2000], supply chain [Kwon et al., 2005], travelling salesman [Xie and Liu, 2008], graph partitioning [Rahimian et al., 2015], resource allocation in energy systems [Anders et al., 2015, Hinrichs et al., 2013, Hinrichs et al., 2014, Pournaras et al., 2010, Pournaras, 2013], bicycle-sharing [Pilgerstorfer and Pournaras, 2017, Pournaras et al., 2018], and most recently, fog computing [Nezami et al., 2021].

One OR problem is the Bin Packing Problem (BPP), a classical combinatorial optimisation problem [Coffman Jr et al., 1996]. A specialisation of the BPP is the Dynamic Virtual Machine Consolidation (DVMC) Problem, a resource allocation problem from cloud computing. Research into decentralised DVMC started as far back as 2009. However, several DVMC strategies classified as distributed or decentralised still rely on a manager [Mastroianni et al., 2011, Mastroianni et al., 2013] or a global arbiter [Yazir et al., 2010].

DVMC research also varies in the complexity of the data centre models and the measures considered when evaluating the proposed decentralised strategies [Prabha et al., 2021]. For some research, the hosts and VMs are modelled as homogenous [Pantazoglou et al., 2015, Sedaghat et al., 2014, Marzolla et al., 2011]. Others either did not model the workload on the VM at all, modelled it as static [Barbagallo et al., 2009, Feller et al., 2012], or as a synthetic random workload [Masoumzadeh and Hlavacs, 2016]. When evaluating the effectiveness of the DVMC strategies, all considered either the packing efficiency or power consumed in the datacentre, but only a few also considered the number of migrations [Murtazaev and Oh, 2011, Yazir et al., 2010], and only one considered the performance degradation due to overloading a host [Masoumzadeh and Hlavacs, 2016].

Thus, developing a genuinely decentralised DVMC strategy and evaluating it in a cloud data centre modelled using a real-world dataset and considering packing efficiency, the number of migrations and SLA violations is valuable. Chapter 4 presents Gossip Contracts (GC), a novel communication protocol for a cooperative MAS. GC is used to engineer a decentralised DVMC strategy evaluated by simulation of a real-world cloud data centre using the Google cluster-usage traces (GCT) [Reiss et al., 2011]. It is evaluated against two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, *ecoCloud*, considering packing efficiency, power consumption, the number of migrations and SLA violations.

Evolutionary Computing Decentralisation is common in natural systems; consider flocking and shoaling. Two nature-inspired techniques that researchers often use in multi-agent systems are evolutionary computing and reinforcement learning. Evolutionary Computing (EC) is a family of algorithms inspired by biological evolution. They are population-based generate-and-test problem solvers.

The idea of using EC to assist in the development of MAS is as old as EC itself [Koza, 1994, Mitchell et al., 1994]. Early research focused on the pursuit game [Manela, 1994, Haynes et al., 1995b, Haynes et al., 1995a, Luke and Spector, 1996, Iba, 1996, Iba et al., 1997, Iba, 1998] or RoboCup agents [Luke et al., 1997, Aronsson, 2003, Tavafi and Banzhaf, 2017]. This research focused on proving EC could be an effective strategy for evolving a communicating and cooperating MAS and exploring which EC techniques and features best suit the challenge. They developed bespoke solutions for unique problems.

A framework which can be applied to many unique problems would be more valuable. A number of these were proposed; namely, Anthill [Babaoglu et al., 2002], MAAD [Van Berkel et al., 2012], and GESwarm [Ferrante et al., 2013]. These frameworks significantly contribute to semi-automating the search for a MAS that exhibits specific system-level behaviour. While they have been evaluated on challenging distributed systems problems, none were assessed on an NP-hard problem.

Thus developing an EC-based framework to design a decentralised heuristic and evaluating it on an NP-hard problem is worthwhile. Chapter 5 presents Evolved Gossip Contracts (EGC), the first EC-based framework developed to design a decentralised heuristic evaluated on an NP-hard problem. EGC uses Genetic Programming (GP) to semi-automate the search for a specific GC protocol. It is evaluated using the NP-hard bin packing problem (BPP). The best BPP strategy developed by EGC is compared to two BPP heuristics, *Best Fit* and *First Fit*, which are popular in the literature.

Reinforcement learning Reinforcement learning (RL) is another nature-inspired technique researchers often use when designing multi-agent systems. RL is a trial-and-error learning framework with delayed rewards [Sutton and Barto, 2018]. Multi-agent Reinforcement Learning (MARL) is a popular research field which extends single-agent RL to the study of multiple agents operating in a shared environment. MARL research primarily focuses on stochastic games [Shapley, 1953], which are episodic tasks [Busoniu et al., 2008].

One classification of MARL systems relates to whether the agents are controlled centrally or in a decentralised manner. The central controller receives the observations from all the agents and decides the actions to be taken by each agent; it gets a joint reward. It follows the *centralised-learning-decentralised-execution* paradigm. This paradigm has recently been widely adopted with multi-agent deep reinforcement learning research [Nguyen et al., 2020]. However, this scheme is only possible when a central controller can exist, which is not always the case.

A MARL system is said to be "fully decentralised" when the agents are fully autonomous, have no central controller, and do not coordinate with one another. Several fully decentralised schemes exist, such as classical Q-learning [Claus and Boutilier, 1998], Distributed Q Learning [Lauer and Riedmiller, 2000], and Hysteretic Q Learning [Matignon et al., 2007]. However, one shortcoming of these approaches is that they suffer from non-convergence [Tan, 1993].

Transfer learning is a machine learning technique which leverages previous knowledge to improve the learning speed or performance [Taylor et al., 2007]. When applied to MARL, this knowledge can come from another agent. This system represents a third MARL classification where the agents are decentralised but use a communication network to share knowledge.

Research into MARL with networked agents began in 2009 with the development of distributed gradient ascent in policy space (DGAPS) [Varshavskaya et al., 2009]. More recent research focused on actor-critic-based approaches [Zhang et al., 2018a, Zhang et al., 2018b], one of which combines real-time Q-learning and actor-critic methods [Zeng et al., 2020]. In addition, two techniques build on value-based methods, QD-Learning [Kar et al., 2012], and an approach which augments the classical TD(0) by incorporating updates received from neighbouring agents by a gossip-like mechanism [Mathkar and Borkar, 2016].

Chapter 6 presents a novel value-based MARL algorithm. It is a hybrid of the RL Q(λ) Learning [Watkins, 1989] algorithm, and the MARL QD Learning algorithm [Kar et al., 2012]. This new MARL algorithm aims to overcome the limitation presented by Tan, that MARL with networked agents "may learn slowly in the beginning" [Tan, 1993]. It also aspires to design a MARL algorithm suited to a sizeable multi-agent system learning a communication protocol, such as the Gossip Contracts protocol. Research into MARL systems learning communication protocols is an emerging area that has seen significant activity in recent years [Foerster et al., 2016, Sukhbaatar et al., 2016, Jiang and Lu, 2018, Das et al., 2019, Mao et al., 2020, Tung et al., 2021].

Figure 2.12 presents the significance and state of the art in multi-agent systems engineering highlighting the contributions of the next three chapters.

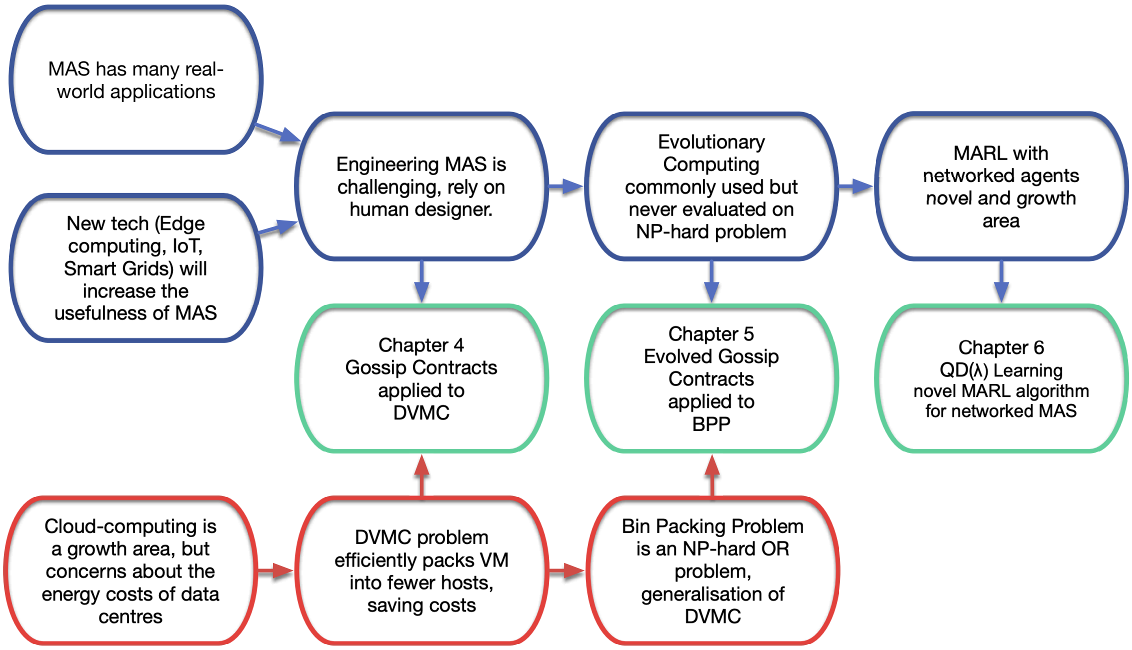


Figure 2.12: Thesis Contributions building on the state of the art.

CHAPTER 3

Computational Architecture

This chapter describes the software packages developed as part of this research. The packages are used to build and evaluate the nature-inspired protocol, algorithm, and framework to semi-automate the development of decentralised MAS heuristics. The packages are available online in support of Open Science [Mc Donnell, 2022].

Three of these packages are the core research output: the Gossip Contracts protocol, Evolved Gossip Contracts framework and QD(λ) Learning algorithm. Two packages are the simulators used for evaluation: the Bin Packing Problem simulator and the DVMC simulator. They leverage two standard technologies: PeerSim [Montresor and Jelasity, 2009], a Java-based network simulator, and ECJ [Scott and Luke, 2019], a Java-based evolutionary computing framework. One final package provides utility features. Figure 3.1 shows the software packages' overall architecture as in a UML diagram.

The software packages are written using Java and Scala. All the code is available online at BitBucket [Mc Donnell, 2022]. In addition, it can process the evaluation simulations' CSV data files using R and Python languages. Table 3.1 shows the versions of the languages and software packages used.

The following section describes some of the critical features of the Util and Simulator packages, leaving the description of the Gossip Contracts, Evolved Gossip Contracts and QD(λ) Learning packages to later in the thesis.

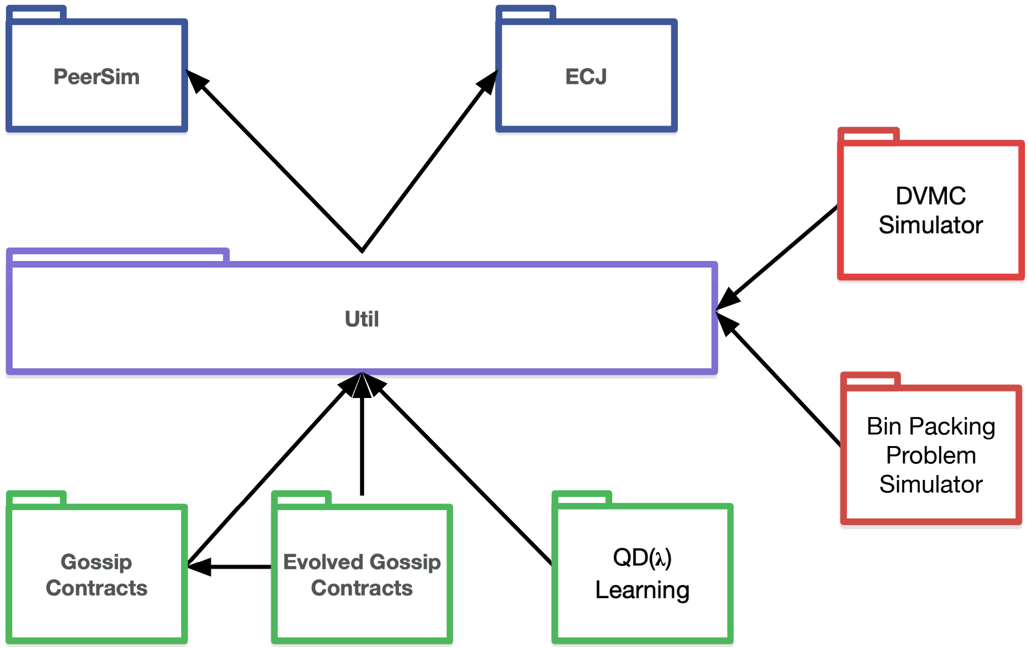


Figure 3.1: A UML (Unified Modelling Language) diagram of the overall architecture shows the software packages with the core research output packages in green, the simulator packages in red, the utility package in purple and the standard technologies packages in blue.

Table 3.1: The versions of the languages and software packages used.

Software	Version
Java	1.8.0_202
Scala	2.12.12
R	3.6.3
Python	3.8.5
PeerSim [Montresor and Jelasity, 2009]	1.0.5
ECJ [Scott and Luke, 2019]	2.7

3.1 The Util Package

The *Util* package provides utility classes and functions to simplify interaction with and across the other packages. It depends upon the PeerSim and ECJ packages, while all other packages depend on it. It builds on PeerSim to provide easy ways to read and write comma-separated value (CSV) files. Building on this, it offers utilities to simplify the observation and logging of the simulations. Finally, it provides an event handling system that allows a block of code to be executed at a future point in the simulation, equivalent to the *Future* concurrent programming concept.

The *Util* package simplifies working with ECJ. It contains a large set of GP operators and terminals. In addition, it has a particular statistics class, *OccamStatistics*, which can output simpler but mathematically equivalent versions of the evolved GP trees. Finally, it has utility classes integrating PeerSim and ECJ – it executes a PeerSim simulation to evaluate each GP individual.

The *Util* package provides tabular RL utilities and integrates them with PeerSim, offering standard RL classes such as Environments, Policies and AVF. Some functions leverage Dynamic Programming to calculate the AVF and hence the greedy policy, given an environment model. It has an implementation of several popular RL algorithms: Q Learning, Q(λ) Learning and QD Learning. It offers utility functions to observers and records the state of the policies and AVFs during the simulations. It has a checkpointing feature that can persist the stats of the Policy and AVFs, enabling simulations to continue from checkpoints.

3.2 The Simulation Packages

The evaluations of the nature-inspired protocol, algorithm and framework use two simulators that were built as part of this research. The first is a Cloud Data Centre simulator to evaluate the GC protocol's ability on the DVMC problem, and the second is a Bin Packing Problem simulator to assess the EGC framework.

The Cloud Data Centre simulator can model the memory, CPU, and the number of cores of a datacentre's hosts and VMs. Furthermore, it supports several VM workloads: static, random, or sinusoidal. In addition, it allows the migration of VMs from one host to another. Finally, it measures the energy efficiency, packing efficiency, number of SLA Violations, and number of Migrations during a simulation.

The Cloud Data Centre simulator integrates with the real-world dataset, Google cluster-usage traces (GCT). GCT represents 29 days' worth of Google Borg cell information from May 2011 on a cluster of about twelve thousand machines [Reiss et al., 2011]. The configuration of the data centres can be driven from the GCT dataset, setting the number and types of hosts and VMs. It can also dictate the arrival and departure of the VMs and even the workload of the VMs.

The Bin Packing Problem simulator can model Online and Offline bin packing problems. It supports bins and items of both one and two dimensions. It can simulate dynamic items that leave their bins after a specific time. Furthermore, it integrates with PeerSim to evaluate decentralised bin packing heuristics where the bins are the agents in a multi-agent system.

While evaluating the nature-inspired protocols, their performance is compared against well-known heuristics from the literature. For example, the performance of the GC-based DVMC heuristic is compared with two DVMC strategies: a centralised strategy, Sercon [Murtazaev and Oh, 2011], and a distributed strategy, ecoCloud [Mastroianni et al., 2013]. And during the evaluation of EGC, the decentralised BPP heuristic developed using EGC is compared to two centralised heuristics, *First Fit* and *Best Fit*.

3.3 Computer Resources

The evaluations of the nature-inspired tools required substantial computing resources to complete. Thus, they were carried out on supercomputers provisioned by the Irish Centre for High-end computing, ICHEC. The evaluations used two machines, Fionn [ICHEC, 2022a] and Kay [ICHEC, 2022b]. Fionn was an SGI ICE-X system of 320 nodes with 2 x 12 core 2.4 GHz Intel Ivy Bridge processors and 64 GiB of RAM. Kay is a cluster of 336 nodes with 2 x 20-core 2.4 GHz Intel Xeon Gold 6148 Skylake processors and 192 GiB of RAM.

CHAPTER 4

Gossip Contracts

Multi-agent systems have many real-world applications. They are widely used in telecommunications network management; manufacturing and logistics; crowd, pedestrian, and traffic simulation; and security and surveillance [Herrera et al., 2020, Müller and Fischer, 2014]. Their application is anticipated to increase, given recent technological advances, such as Edge or Fog computing, the Internet of Things, Smart Cities and Smart Grids. A recent review of agent-based programming for multi-agent systems recommended there be "more application-based papers that can demonstrate features of agent-based programming in the real-world" [Cardoso and Ferrando, 2021].

One technological advance is cloud computing, specifically the dynamic virtual machine consolidation (DVMC) problem. DVMC is a resource allocation problem where virtual machines are packed into as few hosts as possible to minimise energy use in the data centre. It is a variant of the bin packing problem (BPP), a classical combinatorial optimisation problem [Coffman Jr et al., 1996] from the field of operations research.

There are many examples of using MAS to solve optimisation problems, including developing decentralised DVMC heuristics. The advantage of decentralised DVMC strategies is that they scale well to large data centres and are resilient to coordinator node failures.

However, several DVMC strategies classified as distributed or decentralised still rely on a manager [Mastroianni et al., 2011, Mastroianni et al., 2013] or a global arbiter [Yazir et al., 2010]. Others oversimplify the cloud data centre, assuming that the hosts and VMs are homogeneous. Or they neglect to model the VMs' workload or simply model it with synthetic random workloads. When evaluating the DVMC strategies, they only consider power consumption. They do not consider other critical real-world characteristics, such as the number of VM migrations or service level agreement (SLA) violations.

This chapter presents Gossip Contracts (GC), a novel communication protocol for networked multi-agent systems to facilitate decentralised cooperation strategies.

- ❖ Section 4.1.1 describes the protocol and how it differs from Contract Net, from which it is inspired. For example, while Contract Net assumes that the manager and contractor of the system are different agents, GC agents are both managers and contractors. Furthermore, the Contract Net managers only send task announcement messages to their immediate neighbourhood, while GC leverages a Gossip-like multi-cast protocol to send tender messages much further across the network.

- ❖ Section 4.1.2 shows how to use Gossip Contracts to develop a fully-decentralised DVMC strategy. Each host in the cloud data centre is a GC agent in the MAS. At regular intervals, each host can send a tender message announcing its free resources to the other hosts if it is not overloaded. The probability with which it does this is inversely proportional to the number of available resources. Thus, the fewer resources, the more likely it is to tender. Other hosts can bid to migrate some, or all, of their VMs to a tendering host, but only if all the VMs they aim to migrate can fit into the offered resources. A host will bid when its SLA is being violated or overloaded. But it might also bid when it is balanced; the fewer VMs it has, the more likely it will bid.

- ❖ Section 4.1.3 describes the cloud data centre model used for the experiments. To evaluate the GC-based DVMC strategy, we created a cloud data centre simulator. It supports heterogeneous hosts and VM modelling, CPU and memory, n-point linear power models, and static and dynamic VM workloads. It also supports defining the data centre configuration, hosts and VMs and workloads using data from an actual data centre using the Google cluster-usage traces (GCT).

- ❖ Section 4.2 describes the experimental set-up and configuration. The GC-based DVMC strategy is compared with two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, ecoCloud. Three performance metrics are considered when comparing these strategies: power consumption, number of migrations and host availability; availability is the percentage of hosts delivering the desired service level.

❖ Section 4.2.2 described the configuration and results for each of the four experiments. The experiments increase in size and complexity. Exp1 represents a small data centre and demonstrates that GC can consolidate the VMs in a data centre when the VM workload is static over time. Exp 2 is a more extensive data centre and shows GC can consolidate VMs in a data centre when the VM workload is dynamic. Exp 3 represents a massive data centre and demonstrates that GC scales well and can consolidate the VMs in a massive data centre with dynamic VM workloads. Lastly, Exp 4 represents an actual data centre, a simulation of a Google Borg cell in May 2011. It demonstrates that GC can consolidate the VMs in a massive and realistic data centre with real VM workloads.

❖ Finally, Section 4.3 discusses two interesting observations of this research. The first considers DVMC as a multi-objective optimisation problem. It presents radar charts of the energy consumption, the number of SLA violations (availability) and the number of migrations for Exp3 and Exp4. The second topic explores how the GC protocol behaved during the experiments and describes several early attempts which did not work or worked but were not competitive with *Sercon* and *ecoCloud*.

4.1 Methodology

4.1.1 Gossip Contracts

Gossip Contracts (GC) is a communication protocol for networked multi-agent systems to facilitate decentralised cooperation strategies. It is inspired by the Contract Net [Smith, 1980] and Gossip protocols [Jelasity et al., 2007] described in Section 2.2.1 and Section 2.2.2, respectively. However, Gossip Contracts differs from Contract Net in that managers are also contractors, therefore more akin to the definition of agents. In Contract Net, the *Task Announcements* are typically sent only to the managers' neighbourhood. In GC, they are referred to as tender messages, and they use a counter-based gossip protocol to travel much further across the multi-agent system, so the agents have a higher number of tenders from which they can choose.

Each agent connects to k other agents, its neighbourhood, through a randomised P2P network overlay. At the heart of GC are the messages sent between the agents. There are three types of messages: a tender message, a bid message and an award message. Every message has a sender, a receiver, an expiration time and a contract identifier (ID); a unique ID for each specific tender.

Tender messages have a task to be completed and a guide price indicating the tender's relative importance to the tendering agent. In addition, they have a gossip

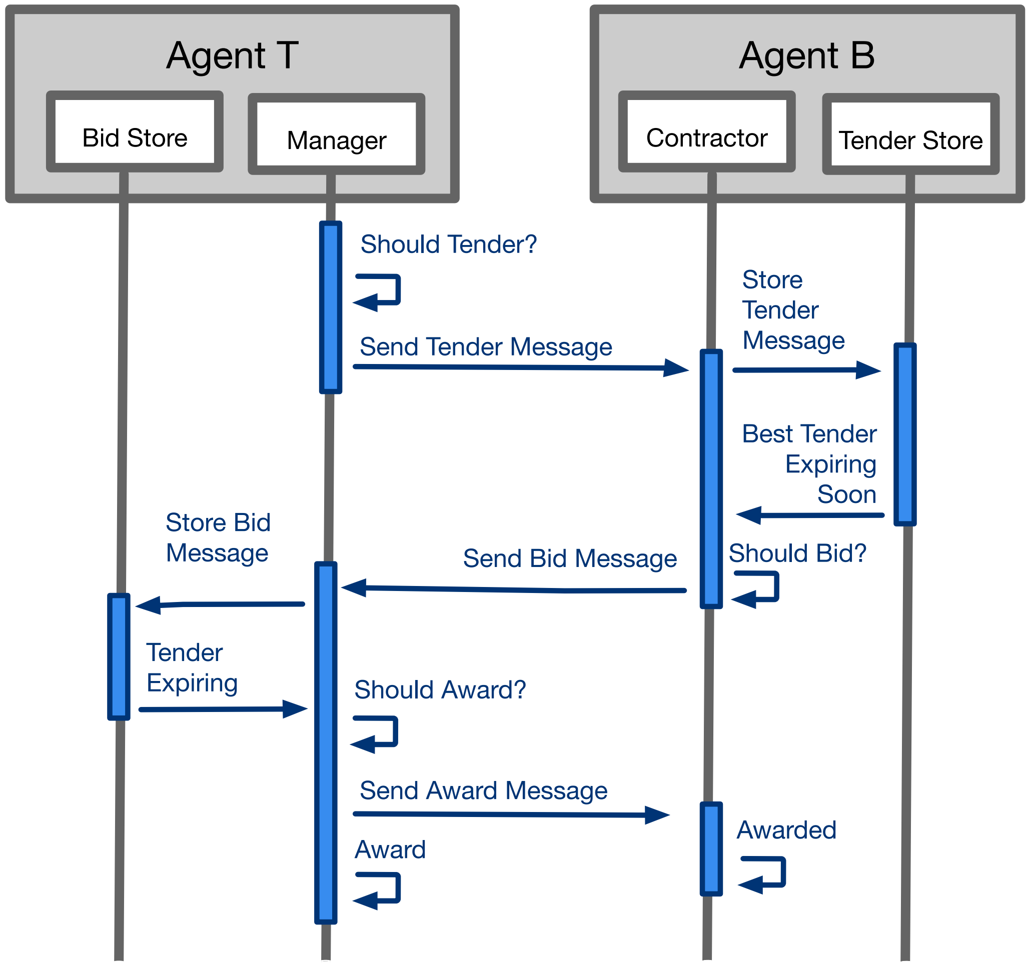


Figure 4.1: The flow of messages within Gossip Contracts

counter. Tender messages are forwarded to the receiving agents' neighbours if their counter exceeds zero. GC decrements the counter each time it forwards a message. This forwarding mechanism allows tender messages to travel further across the network than their immediate neighbourhood. A bid message has a proposal for the completion of a task and an offer price which indicates the relative importance of this bid to the bidding agent. Finally, the tendering agent sends an award message to the agent that it is awarding the contract.

Figure 4.1 shows the flow of messages between two Gossip Contract agents. In the example, Agent T will tender, and Agent B will bid.

1. Agent T decides whether or not to tender.
2. As it decides to tender, it creates a tender message. This message contains the task to be tendered for and its guide price. It also sets a timer for each which ends at the tender expiration time. Then, Agent T sends the message to its immediate neighbours.

3. Agent B is a neighbour of Agent T and receives the tender message. Agent B stores the message in its tender store. Agent B then sets a timer that ends just before the tender message expires. It then checks if the gossip counter of the tender message is greater than zero, and if it is, it decrements the gossip counter and forwards the tender message to all its neighbours. Agent B receives many tender messages. It stores them, starts a timer, and then forwards them to the agents in its neighbourhood.
4. As each timer ends, Agent B evaluates if the expiring tender message is the best tender message it has. To decide which is best, Agent B has a tender comparator which can order tenders. When one or more tenders have the same score as the best tender, GC randomly selects one.
5. If the best tender is the tender whose timer is ending, Agent B decides whether to bid or not.
6. If Agent B decides to bid, it sends a bid message to Agent T. This message contains its proposal and its offer price.
7. Agent T receives the bid message and stores it. Additionally, agent T will receive bids from other agents that it also stores.
8. When the timer set by Agent T at the start of the tender ends, it evaluates all the bids it has received and selects the best one. To decide which bid is best, Agent T uses a bid comparator that orders the bids. If more than one bids has the same score as the best bid, it chooses one randomly. Agent T then decides if it should award the tender to this bid.
9. If Agent T decides to award, it sends an award message to Agent B and carries out any steps needed to complete the award.
10. If Agent B receives an award message, it completes any steps needed to complete the award.

To use a GC multi-agent system to solve a specific problem, from a computational perspective, the MAS designer extends from the generic *GCAgent* and implements the methods listed in Table 4.1. First, they define when an agent should tender, *shouldTender()*, when it should bid for a tender, *shouldBid()*, and when it should award a bid, *shouldAward()*. Then, they define the task, *getTask()*, and the proposal, *getProposal()*, which form the subject of the tender messages and bid messages, respectively; it is typical for the task type and the proposal type to be the same. Next, they return the guide, *getGuide()*, used during the creation of a tender message and the offer, *getOffer()*, used during the creation of a bid message. They provide a means of mapping tender messages and bid messages to doubles so that GC can sort them to identify the best ones; *tenderToDouble()* and *bidToDouble()*. And finally, they set out the steps to be completed when a tender is awarded on both the tendering and the bidding agents, with the *award()* and *awarded()* methods, respectively.

Table 4.1: The methods of a *GCAgent* which are extended to implement a decentralised co-operation strategy to address a specific problem.

Name	Description
shouldTender()	To decide if an agent should tender.
getTask()	The task to be completed
getGuide()	To generate a double representing the value of the tender to the tendering agent.
tenderToDouble()	To convert a tender to a double so that all the tenders an agent has received can be ordered to select the best tender; lower scoring tenders are considered better.
shouldBid()	To decide if an agent should bid for the best tender.
getProposal()	The proposal to be offered
getOffer()	To generate a double representing the value of the bid to the bidding agent.
bidToDouble()	To convert a bid to a double so that all the bids an agent has received for a tender can be ordered to select the best bid; lower scoring bids are considered better.
shouldAward()	To decide if a tendering agent should award to the best bid.
award()	How the tendering agent awards a task
awarded()	How a task is awarded to the bidding agent

Algorithm 5 shows the algorithm executed by an agent when it is tendering, and Algorithm 6 shows the algorithm executed when an agent is bidding. As bids are binding, at any time, a GC agent can either tender or bid, but not both; they cannot simultaneously bid on several different tenders. The agent's *isIdle()* method will return true if an agent is neither tendering, bidding, nor awarding. The GC protocols have several parameters listed in Table 4.2, which can be used to tune the behaviour of the agent system.

4.1.2 Dynamic Virtual Machine Consolidation with Gossip Contracts

This section describes how to develop a fully-decentralised DVMC protocol with Gossip Contracts. Each host in the cloud data centre is a Gossip Contract agent. There is no controlling or coordinating agent. The cloud data centre model considers CPU and

Table 4.2: The Gossip Contracts parameters which can be used to tune the behaviour of the agent system.

Name	Description
neighbourhood size, k	The number of agents to which tender messages are sent
tender gossip counter, g	The number of times a tender message is gossiped or forwarded
tender timeout, t_t	The time before a tender expires
bid buffer, b_b	The time window before a tender is due to expire when a bid is sent
award timeout, t_a	The amount of time required to complete the awarding and awarded steps

Algorithm 5: Tendering Algorithm

```
t := tendering agent
if t.isIdle() and t.shouldTender() then
    tenderMessage := createTenderMessage()
    tenderMessage.guide := t.getGuide()
    tenderMessage.tenderGossipCounter := g
    sendTenderMessageToNeighbours(tenderMessage)
    scheduleCallback(tenderMessage.getTimeout(),
      evaluateBids(tenderMessage))
end
Function evaluateBids(tenderMessage):
    bids := t.bidsForTenderMessage(tenderMessage)
    order bids using bidToDouble()
    bestBid := selectBestBid(bids)
    if t.shouldAward(bestBid) then
        awardMessage := createAwardMessage()
        sendAwardMessage(bestBid.sender, awardMessage)
    end
```

memory utilisation. In GC-based DVMC, each host can be in one of several mutually-exclusive states defined by an *overloaded threshold*, T_o; the concept of being overloaded only applies to CPU utilisation. The states are:

- *Balanced*, where either the CPU utilisation is less than or equal to the overloaded threshold and the memory utilisation is less than or equal to 100%;
- *Overloaded*, where the CPU utilisation is greater than the overloaded threshold; and
- *Violating*, where the CPU or memory utilisation is greater than 100%.

In the model, the tendering agents offer CPU and memory resources on their host.

Algorithm 6: Bidding Algorithm

```
b := bidding agent
tenderMessage := tender message received by b
tenders := list of other tender messages received by b
if !tenderMessage.hasExpired and !tenders.contains(tenderMessage) then
    tenders := tenders + tenderMessage
    if tenderMessage.tenderGossipCounter > 0 then
        tenderMessage.tenderGossipCounter -= 1
        sendTenderMessageToNeighbours(tenderMessage)
    end
    scheduleCallback(tenderMessage.getTimeout() - bidBuffer,
     b.evaluateTenders())
end
Function evaluateTenders():
    order tenders using tenderToDouble()
    bestTender := selectBestTender(tenders)
    if b.isIdle() and bestTender about to expire and b.shouldBid(bestTender)
     then
        bidMessage := createBidMessage()
        bidMessage.offer := b.getOffer()
        sendBidMessage(bestTender.sender, bidMessage)
    end
```

The bidding agents propose a list of VMs to be migrated to the tendering host to avail of those resources. Thus, the task type is the available resources, CPU and RAM, and the bid proposal type is a list of VMs.

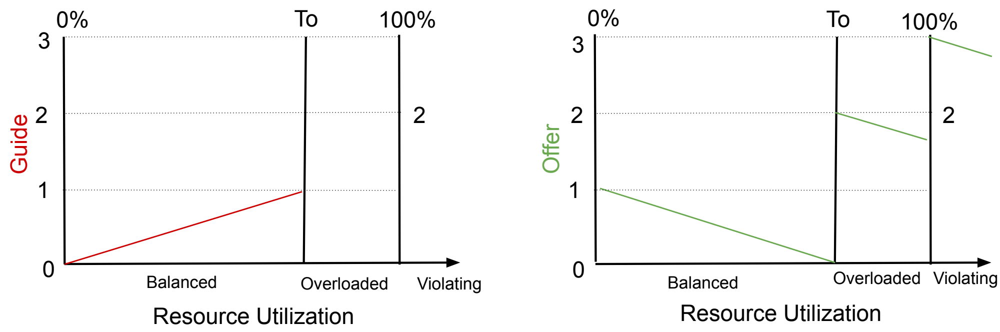
The VMs selected depends on the state of the host. If the host is balanced, all the VMs on the host are proposed. While if the host is overloaded or violating, the VMs are sorted by CPU utilisation and selected for proposal one by one, with the smallest selected first until enough are chosen such that if they were migrated, the host would be balanced.

Figure 4.2 shows the guide and offer functions. The guide, $f_g(x, T_o)$, and offer, $f_o(x, T_o)$, are numbers between 0 and 1. The function calculates the guide is expressed as:

$$f_g(x, T_o) = x/T_o \qquad 0 < x <= T_o \tag{4.1a}$$

where x is the highest resource utilisation, and T_o is the overloading utilisation threshold. While, the offer is expressed as:

$$f_o(x, T_o) = -(x/T_o) + 1 \qquad 0 < x <= T_o \tag{4.2a}$$
$$f_o(x, T_o) = -(x/T_o) + 2 \qquad T_o < x <= 1 \tag{4.2b}$$
$$f_o(x, T_o) = -(x/T_o) + 3 \qquad x > 1 \tag{4.2c}$$



(a) Guide function

(b) Offer function

Figure 4.2: The Guide and Offer functions

An agent is more likely to tender under the following scenarios: if it is idle, not already tendering, bidding, awarding or being awarded; if it is balanced and if it passes a Bernoulli trial; a test which returns a boolean outcome. The probability of returning true is p and false q, where $q = 1 - p$. Here *p* is set to the value of the guide function. An agent will decide to bid:

- if it is idle;
- if the offered available resources can fit the VMs that the agent wants to migrate away; and
- if, when the host is balanced, it passes a Bernoulli trial whose probability of success is set to the value of the offer function.

An agent will decide to award if the proposed VMs fit into the available resources.

4.1.3 Cloud Data Centre Simulator

To evaluate the GC-based DVMC strategy, a cloud data centre simulator was built with the PeerSim network simulator [Montresor and Jelasity, 2009]. Figure 4.3 shows the architecture of the Cloud Data Centre simulator. The cloud data centre comprises several heterogeneous *Hosts*, N_h and heterogeneous *VMs*, N_{vm}, provisioned on those hosts. A *Host* extends from the PeerSim *GeneralNode*. Each *Host* can process a certain number of instructions per second, measured in millions of instructions per second (MIPS). It has an amount of available memory, measured in GB. A *VM* requests an amount of CPU capacity measured in MIPS and an amount of memory measured in GB.

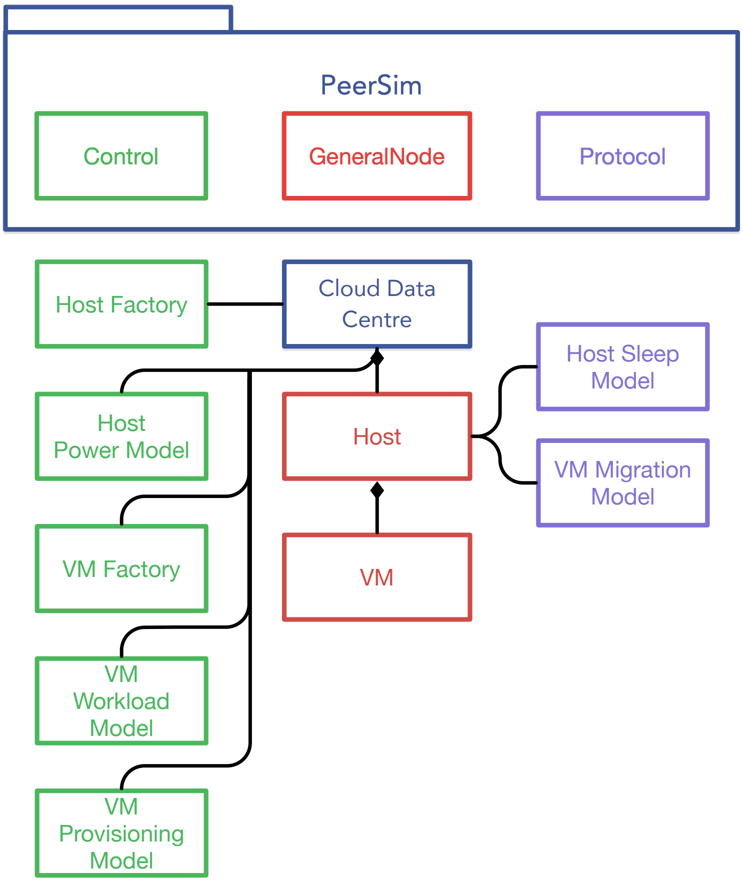


Figure 4.3: The architecture of the cloud data centre simulator

VMs are initially allocated to hosts by the *VMProvisioningModel*. A *First Fit* initial allocation will pack the VMs into the hosts using the First Fit bin packing method. The *HostPowerModel* calculates the power of a host using an n-point linear model. For example, a 2-point model would provide the idle and the max utilisation power consumption and calculate the power for utilisation as the linear interpolation between the idle and max utilisation.

The *VMWorkloadModel* models how the VMs′ CPU workload varies over time. It is modelled with a sinusoidal function, a commonly used test input function for simulating dynamical systems. Sterman et al. state that oscillation is one of the fundamental modes of behaviour observed in dynamic systems [Sterman, 2000]. The initial is a value set at random. The amplitude of the sine function is the maximum value possible while ensuring that the sine function will never evaluate to be less than zero or greater than the VM's requested CPU. The *shift* of the sine function is randomly selected up to a maximum shift value, max_shift, which is configurable. In addition, the sine function's period, P, is configurable.

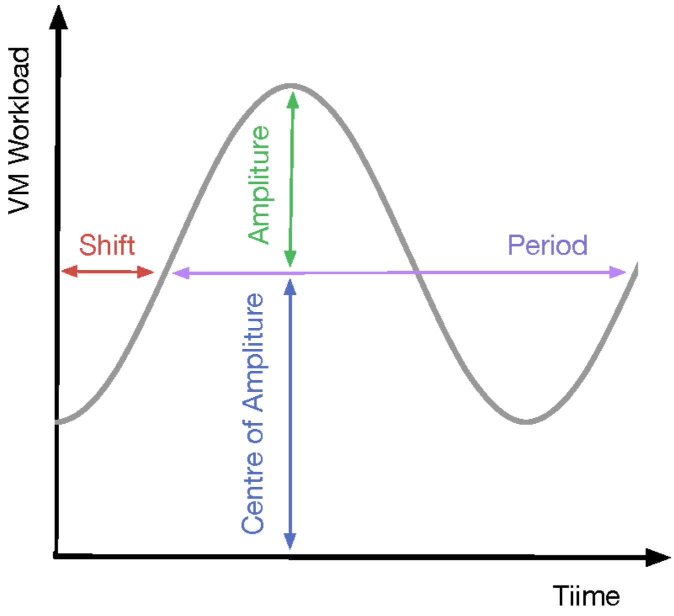


Figure 4.4: Sinusoidal virtual machine workload model

The *HostSleepModel* models the sleep functionality of the hosts. The *HostSleepModel* utilises PeerSim's *GeneralNode*'s *FailState* feature. It puts a Host into a sleep state by setting the Host's *FailState* to *DOWN* and wakes a sleeping Host by setting the Host's *FailState* back to *OK*. While in a *DOWN* state, PeerSim nodes will not process messages or run next-cycle events. However, in PeerSim version 1.0.5, no next-cycle events are rescheduled when a node moves back from the *DOWN* state to the *OK* state. As this is not desired behaviour, PeerSim was modified to ensure that all nodes have next-cycle events scheduled, though they are only executed when the node is in the *OK* state.

The *HostSleepModel* can also find a sleeping Host by searching in its neighbourhood. If this initial search fails, one of the neighbours is randomly selected, and the investigation continues in that host's neighbourhood. The search continues, hopping from one neighbourhood to the next, up to a maximum number of hops, *max hops*; this was 10 for the experiments.

The *VMMigrationModel* models a VM migrating from one host to another; the model instantaneously migrates VMs. This simplification was adopted as the research focuses on the VM Placement aspect of the DVMC strategy.

The *VMProvisioningModel*, *HostPowerModel*, and *VMWorkloadModel* are all PeerSim *Controls*, along with the *HostFactory* and *VMFactory*.In addition, thee *HostSleepModel*, and *VMMigrationModel* implement the PeerSim *Protocol* interface.

The hosts connect through a static network overlay initialised at the start of the simulation. Each host has 50 neighbours. The network is reliable; messages are neither delayed nor lost. The network overlay is not reconfigured when a host goes into a sleep state. Note neither *Sercon* nor *ecoCloud* send peer-to-peer (P2P) messages. However, GC-based DVMC does rely on the P2P network; it sends *TenderMessages* to *k* awake hosts in its neighbourhood.

4.2 Experimental Results

The evaluation of the GC-based DVMC strategy compared it with two established DVMC strategies, Sercon [Murtazaev and Oh, 2011], which is a centralised strategy, and ecoCloud [Mastroianni et al., 2013], a distributed strategy. These strategies were re-implemented using PeerSim and the cloud data centre model.

Sercon *Sercon* has a threshold parameter, T^{ser}, which sets the upper limit of CPU utilisation of a host that prevents a node's CPU from reaching 100% utilisation. In addition, it has two parameters which control the number of migrations: the maximum allowed migrations, m_0^{ser}, and the minimum migration efficiency, ME_0^{ser}. m_0^{ser} is an upper bound of the number of migrations while ME_0^{ser} is the minimum allowed percentage of migration efficiency. It has another parameter, the allowed number of unsuccessful migration attempts, N_{UMA}^{ser}. This parameter prevents the algorithm from entering an infinite loop. The authors of [Murtazaev and Oh, 2011] reported using the following values for their experiments: $T^{ser} = 0.7$ and $ME_0^{ser} = 0.1$. They did not report values for m_0^{ser} or N_{UMA}^{ser}. Table 4.3 shows the parameters used in the experiments unless otherwise stated.

Table 4.3: The Sercon parameters used

T^{ser}	m_0^{ser}	ME_0^{ser}	N_{UMA}^{ser}
0.7	not set	0.1	10

ecoCloud The *ecoCloud* [Mastroianni et al., 2013] DVMC strategy uses three thresholds: the assignment threshold, T^{eco}, the low threshold, T_l^{eco}, and the high threshold, T_h^{eco}. The assignment function has a shape parameter, p, which modulates the shape of the function. Similarly, the migration procedures have two shape parameters: α^{eco} to shape the low migration procedure, and β^{eco}, to shape the high migration procedure. The *ecoCloud* authors set the thresholds and parameters to: $T^{eco} = 0.8$, $p^{eco} = 3$, $T_l^{eco} = 0.5$, $\alpha^{eco} = 0.25$, $T_h^{eco} = 0.9$ and $\beta^{eco} = 0.25$, for their experiments on a real data centre. They also found no advantage to sending *assign* invitations to more than 100 hosts [Mastroianni et al., 2013]. Table 4.4 shows the parameters used in the experiments unless otherwise stated.

Table 4.4: The ecoCloud parameters used

T^{eco}	p^{eco}	T_l^{eco}	α^{eco}	T_h^{eco}	β^{eco}
0.8	3	0.5	0.25	0.95	0.25

GC-based strategy Table 4.5 lists the parameters used in the experiments for the GC-based DVMC strategy unless otherwise stated. There parameters are described in Table 4.2.

Table 4.5: The Gossip Contracts-based dynamic virtual machine consolidation algorithms parameters

T_o	k	g	t_t	b_b	t_a
0.9	3	3	3	1	1

Cloud Data Centre Configuration The hosts used in the first three experiments (*Exp1, Exp2 and Exp3*) were HP ProLiant ML110 G4 (Intel Xeon 3040, 2 cores x 1860 MHz, 4GB) [CNET, 2018] and HP ProLiant ML110 G5 (Intel Xeon 3075, 4 cores x 2660 MHz, 4GB) [Amazon, 2018]. The frequency of the server CPUs maps to a MIPS rating. The HP ProLiant ML110 G4 maps to 1860 MIPS per core, and the HP ProLiant ML110 G5 maps to 2660 MIPS per core. Table 4.6 shows the power consumption by the servers at different load levels in Watts [Beloglazov and Buyya, 2012].

Table 4.6: The power consumption in Watts of the HP ProLiant G4 and HP ProLiant G5 at various CPU utilisation used to create a 10-point linear power model [Beloglazov and Buyya, 2012].

Server	0%	10%	20%	30%	40%	50%
HP ProLiant G4	86	89.4	92.6	96	99.5	102
HP ProLiant G5	93.7	97	101	105	110	116
Server	**60%**	**70%**	**80%**	**90%**	**100%**	
HP ProLiant G4	106	108	112	114	117	
HP ProLiant G5	121	125	129	133	135	

The VMs used in the first three experiments (*Exp1, Exp2 and Exp3*) correspond to Amazon EC2 [Amazon, 2018] as defined by Beloglazov et al. [Beloglazov and Buyya, 2012] and used by Masoumzadeh et al. [Masoumzadeh and Hlavacs, 2016]. In their research, Beloglazov and Masoumzadeh imposed the limitation that the VMs were single-core. In Beloglazov's case, the workload data they used for the simulations was from single-core VMs from PlanetLab. Thus, the amount of RAM is divided by the number of cores in each VM type. Table 4.7 shows the VM flavours

Table 4.7: VM instance types

Instance Type	MIPS	RAM
Micro	500	613 MB
Small	1000	1.7 GB
Extra Large	2000	3.75 GB
High-CPU Medium	2500	0.85 GB

The first three experiments had increasing numbers of hosts and VMs. *Exp 1* had 800 VM allocated to 250 hosts, *Exp 2* had 8,000 VMs allocated to 2,500 hosts; and *Exp 3* had 80,000 VMs allocated to 25,000 hosts. In addition, for *Exp 3* the Sinusoidal VM workload model has $max\_shift = 2\pi$. While the VMs workload is increasing and decreasing in a sinusoidal fashion, the sum of all the VMs workloads is relatively static. Due to the size of *Exp 3*, it was necessary to set $m_0^{ser} = 500$ to ensure *Sercon* completed in a timely fashion. The length of the experiments was three simulation hours; every 5 minutes, it executed the DVMC algorithms.

Exp 4 used the Google cluster-usage traces (GCT) [Reiss et al., 2011] as an alternative to the sinusoidal workload model used in *Exp 2* and *Exp 3*. In the GCT dataset, the CPU and memory are normalised. The maximum value of each is one. Note they are normalised independently. To map the normalised CPU and memory values to the DVMC model, the CPU and memory of one map to those of the largest machine; the HP ProLiant ML110 G5 (Intel Xeon 3075, 4 cores x 2.66 MHz, 4GB). One MHz is considered to be 1,000 MIPS per the other experiments. Thus, the normalised CPU measure of *1* becomes four cores with 2660 MIPS each or 10,640 MIPS in total, and the normalised memory measure of *1* becomes 4GB of RAM. For the Machine Power Consumption model, a 2-point linear model is used, where the idle power (power consumption 0%) is 70% of the fully utilised machine [Greenberg et al., 2008].

For *Exp 4*, 450,146 records were read from the first of the *task_events* files and filtered for SCHEDULE events at time 0. Thus 132,982 tasks were running at the start of the simulation. A task is mapped to a DVMC model VM with the normalised CPU request mapping to the VMs MIPS by multiplying by 10,640; this is equivalent to the machine CPU mapping described above. The VMs were not allocated to the hosts as they were in the Google Trace Dataset. Instead, they were assigned using the FirstFit algorithm. A VM is added to a host if, after being added, the sum of the workloads is less than 70% of capacity.

To generate the VMs workload data, we read 7,355,725 records from the first three *task_usage* files. The DVMC VM workloads were derived from the GCT dataset task CPU rate. Each VM workload adjustment is assigned to the time step halfway between the start and end time of the record. One exception is when the start time signifies the beginning of the simulation and so the time step is set to 0. The CPU rate is converted to MIPS using the same conversion formula used for the Host CPU capacity and the VMs MIPS specification.

4.2.1 Performance Metrics

The DVMC problem can be viewed as a multi-objective problem. Deb et al. define a multi-objective problem as one that simultaneously considers several conflicting objectives [Deb, 2005] . One objective is to minimise the number of awake hosts, by migrating VMs away from underloaded hosts to put them into sleep mode. But during the migration of a VM between hosts, the VMs performance will be degraded. These migrations may breach the agreed SLA for a customer. Thus the second objective is to minimise the number of migrations.

SLA violations may also occur if the VMs are packed too tightly into the hosts. Thus, a third objective is to minimise the number of SLA violations, or increase the availability. Availability is the percentage of hosts which deliver desired service. An overloaded host is considered unavailable. Typically, availability targets are described as the number of nines. For example, one nine is 90% availability, two is 99%, and so on up to the ultimate of six nines or 99.9999% availability. Hence, the experiments evaluate the power consumption of the cloud data centre, the total number of migrations, and host availability.

4.2.2 Experiments

Each experiment was repeated 20 times and the results presented in the tables and figures are the averages of these runs. The p-values were calculated by comparing the GC-based DVMC strategy to the *Sercon* and *ecoCloud* strategies.

Exp 1 The first experiment, *Exp 1*, highlights how the three models compare at repacking the VMs into fewer hosts. It simulates a small data centre, 250 hosts and 800 VMs,

Table 4.8: Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 1

Power (MW)	Max	Min	Mean	p-Value
GC	25.9	18.8	19.9 ± 1.2	
Sercon	25.8	18.0	19.3 ± 1.2	2e-25
ecoCloud	25.8	20.2	21.7 ± 1.3	3e-142
Migrations	Max	Min	Mean	p-Value
GC	176	140	160 ± 9	
Sercon	187	131	157 ± 14	0.5
ecoCloud	310	216	264 ± 28	2e-14
Availability	Max	Min	Mean	p-Value
GC	100.0	100.0	100.0 ± 0.0	
Sercon	100.0	100.0	100.0 ± 0.0	NA
ecoCloud	100.0	96.7	99.6 ± 0.7	2e-63

and runs for 90 minutes of simulation time. Initially, the VMs were randomly allocated to the hosts, and the VM's workload remained static for the experiment.

The CPU demand, power consumption, number of migrations and availability for *Exp 1* are in Table 4.8 and Figure 4.5. The CPU demand is the sum of the CPU workloads across all VMs in Terahertz (THz). The power consumption measure is the total power consumed by the data centre, measured in Megawatts (MW). The number of migrations is the accumulative count of migrations of VMs from one host to another. The availability is the percentage of hosts delivering the desired service; this means the CPU and memory utilisation are equal to or below what the Host machine can provide.

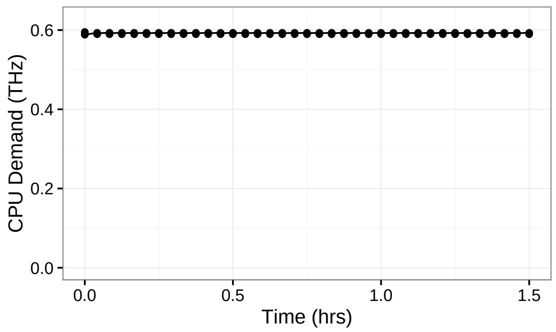
Exp 1 demonstrates that GC can consolidate the VMs in a data centre when the VM workload is static over time. *Sercon* had the lowest power consumption, while GC had the least number of migrations; neither had availability less than 100%. *Sercon* re-packs the VMs in one iteration of the algorithm, with GC converging much more rapidly than *ecoCloud* to its steady state. Towards the start of the experiment, *ecoCloud* had lower availability. This likely arises due to *ecoCloud*'s assignment function only considering the current utilisation of the server and not what the utilisation would be were its invitation accepted by the assigning manager.

Exp 2 The second experiment, *Exp 2*, represents a more realistic data centre with the overall load modulating slowly in a sinusoidal pattern over time. It has 2,500 hosts and 8,000 VMs, initially allocated based on the First Fit algorithm. The VM workloads were dynamic, increasing and decreasing sinusoidally over 1 hour. The experiment ran for a

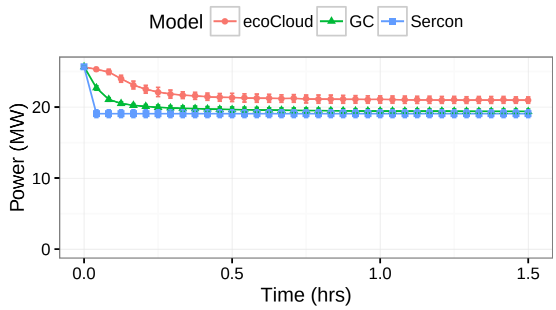
Table 4.9: Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 2

Power (MW)	Max	Min	Mean	p-Value
GC	258.1	160.6	203.6 ± 29.5	
Sercon	258.1	155.3	187.4 ± 24.1	9e-30
ecoCloud	273.3	184.1	223.6 ± 27.9	1e-38
Migrations	**Max**	**Min**	**Mean**	**p-Value**
GC	13278	11964	12605 ± 401	
Sercon	2209	1083	1728 ± 460	3e-43
ecoCloud	13837	12612	13300 ± 325	6e-07
Availability	**Max**	**Min**	**Mean**	**p-Value**
GC	100.0	94.4	99.3 ± 1.3	
Sercon	100.0	84.3	94.6 ± 5.7	4e-82
ecoCloud	100.0	83.7	95.4 ± 5.0	3e-76

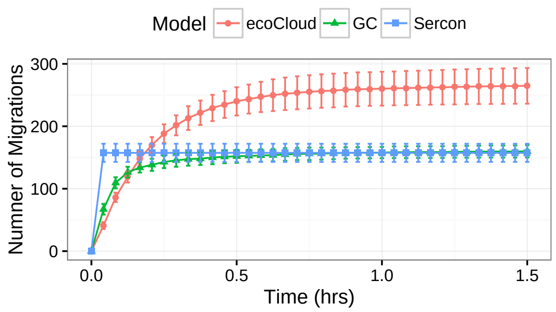
total of 3 hours. Table 4.9 and Figure 4.6 show the CPU demand, power consumption, number of migrations and availability for *Exp 2*.



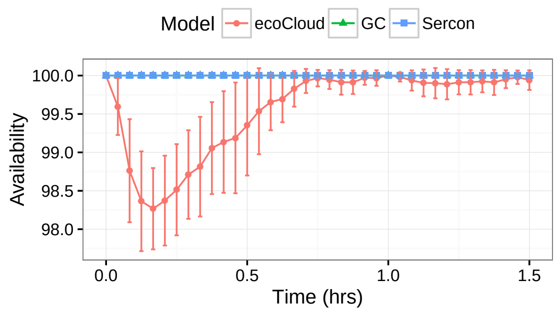
(a) CPU Demand



(b) Power Consumption



(c) Migrations



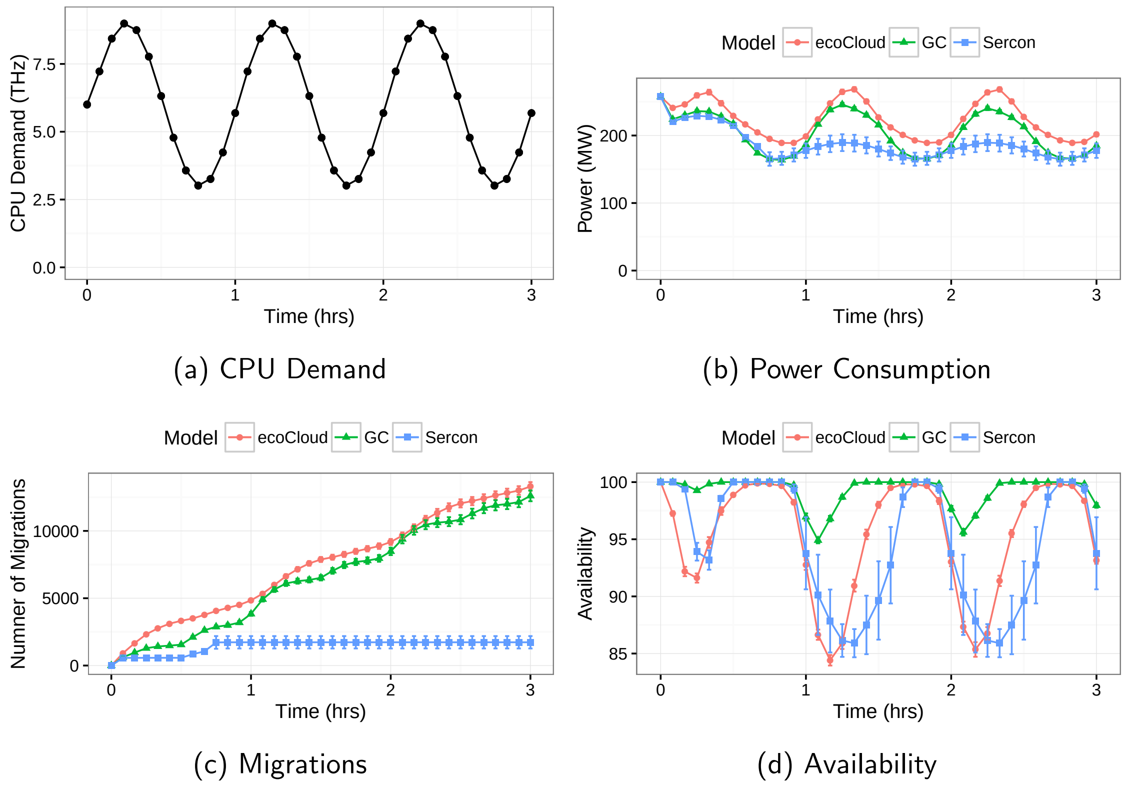
(d) Availability

Figure 4.5: CPU Demand, Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 1

Exp 2 demonstrates that GC can consolidate the VMs in a data centre when the VM workload is dynamic, changing over time. GC power consumption is consistently lower than *ecoCloud*, with a similar number of migrations and substantially higher availability. As with *Exp 1*, *Sercon* has the lowest power consumption. However, it also has poor availability, as dynamic workloads were not considered when it was designed. It reached a steady state at 45 minutes, coinciding with the lowest VM workload demand for the data centre. It cannot detect if a host is violating its SLA, nor can it wake up a sleeping host when the capacity in the existing awake hosts is exhausted. Consequently, once it has packed the VMs as tightly as possible, it gets trapped in a state with low availability.

Exp 3 The third experiment, *Exp 3*, represents a massive data centre with a relatively constant overall load. However, the load of the individual VMs modulates slowly over a one-hour period. It has 25,000 hosts with 80,000 VMs initially allocated based on the First Fit algorithm, and it ran for 3 hours.

As the complexity of *Sercon* is $O(n^4)$, the maximum allowed migrations, m_0^{ser}, is 500 to ensure it completes in a timely fashion. To offset this, its execution frequency is increased from once every 5 minutes to once every 30 seconds. Table 4.10 and Figure 4.7 show the CPU demand, power consumption, number of migrations and availability



(a) CPU Demand (b) Power Consumption

(c) Migrations (d) Availability

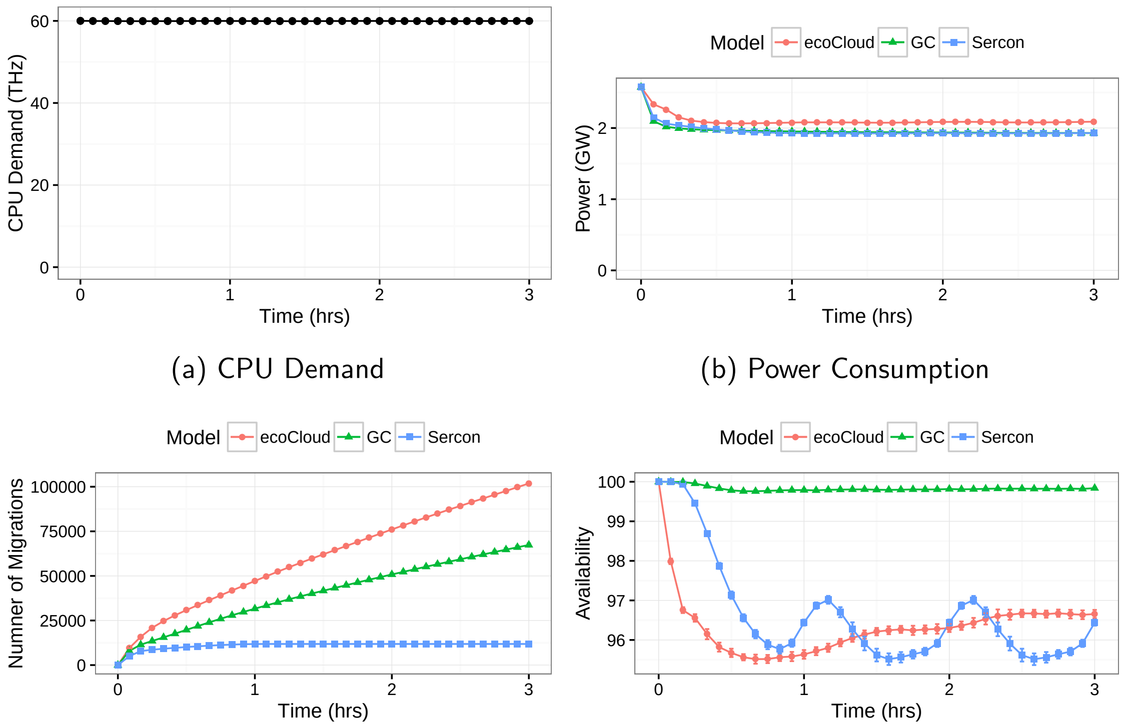
Figure 4.6: CPU Demand, Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 2

Table 4.10: Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 3

Power (MW)	Max	Min	Mean	p-Value
GC	2575.3	1921.0	1969.0 ± 105.6	
Sercon	2575.7	1907.2	1965.5 ± 111.8	0.5
ecoCloud	2577.0	2054.2	2107.9 ± 95.4	2e-124
Migrations	**Max**	**Min**	**Mean**	**p-Value**
GC	68689	65774	67345 ± 864	
Sercon	12106	11438	11837 ± 171	3e-38
ecoCloud	103454	100022	101844 ± 948	2e-48
Availability	**Max**	**Min**	**Mean**	**p-Value**
GC	100.0	99.7	99.8 ± 0.1	
Sercon	100.0	95.2	96.7 ± 1.3	4e-312
ecoCloud	100.0	95.2	96.4 ± 0.8	0

for *Exp 3*.

Exp 3 demonstrates that GC scales well and can consolidate the VMs in a massive



(a) CPU Demand (b) Power Consumption

(c) Migrations (d) Availability

Figure 4.7: CPU Demand, Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 3

Table 4.11: Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 4

Power (MW)	Max	Min	Mean	p-Value
GC	716.9	694.8	707.2 ± 4.1	
Sercon	716.9	677.4	684.9 ± 5.9	¡ 2.2e-16
ecoCloud	716.9	700.8	707.7 ± 3.9	0.004
Migrations	**Max**	**Min**	**Mean**	**p-Value**
GC	7842	6614	7287 ± 324	
Sercon	174	160	161 ± 4	3e-27
ecoCloud	5147	4520	4824 ± 189	6e-24
Availability	**Max**	**Min**	**Mean**	**p-Value**
GC	100.0	96.9	98.9 ± 0.8	
Sercon	100.0	95.3	97.8 ± 1.0	6e-130
ecoCloud	100.0	95.8	97.9 ± 0.9	7e-115

data centre with dynamic VM workloads. As with *Exp2*, *Sercon* exhibits the lowest power consumption of the three strategies and substantially lower numbers of migrations, but it has a low availability compared to GC. GC quickly reached a steady state power consumption which was only marginally higher than *Sercon*'s eventual power consumption with significantly higher availability.

Exp 4 The fourth experiment, *Exp 4*, represents an actual data centre. It is a simulation of a Google Borg cell in May 2011 [Reiss et al., 2011]; A Borg cell comprises many thousands of machines managed as a single unit. *Exp 4* includes 12,555 hosts and 132,982 VMs initially allocated based on the First Fit algorithm with a threshold of 0.7; it ran for 4 hours.

The GC configuration had several changes for *Exp 4*. Firstly, the *neighbourhood size*, k, was increased from 3 to 4. Also, *tender gossip counter*, g, was increased from 3 to 4. These configuration changes were necessary as the experiment had over 12 thousand hosts. And because almost all the hosts needed to be awake in the data centre to handle the workload.

As with *Exp 3*, the maximum allowed migrations for *Sercon*, m_0^{ser}, is 500 to ensure it completes in a timely fashion and consequentially increases its execution frequency to once every 30 seconds. Table 4.11 and Figure 4.8 show the CPU demand, power consumption, number of migrations and availability for *Exp 4*.

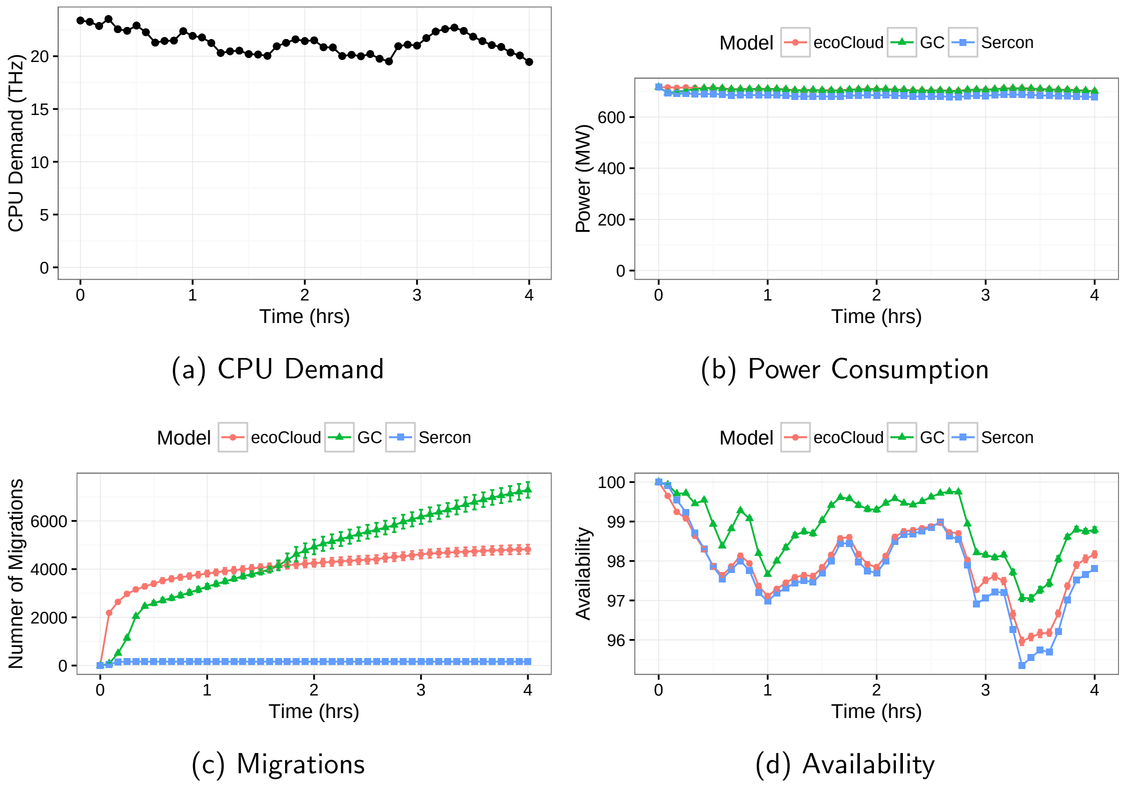
Exp 4 demonstrates that GC can consolidate the VMs in a massive and realistic data centre with real VM workloads. *Sercon* exhibits the lowest power consumption of the three strategies and substantially lower numbers of migrations, but it has a low availability compared to GC. GC quickly reached a steady state power consumption which was only marginally higher than *Sercon*'s eventual power consumption with significantly higher availability. The power consumption for all three models is not statistically significantly different. GC has better availability but also a higher number of migrations.

4.3 Discussion

4.3.1 DVMC as a Multi-objective Optimisation Problem

Deb et al. described a multi-objective optimisation problem as one that considers several conflicting objectives simultaneously. There is usually no single optimal solution but a set of alternatives with different trade-offs [Deb, 2005]. The DVMC problem can be considered as a multi-objective problem trying to satisfy three constraints. On the one hand, a strategy must minimise the data centre energy consumption while on the other minimise the number of SLA violations and migrations.

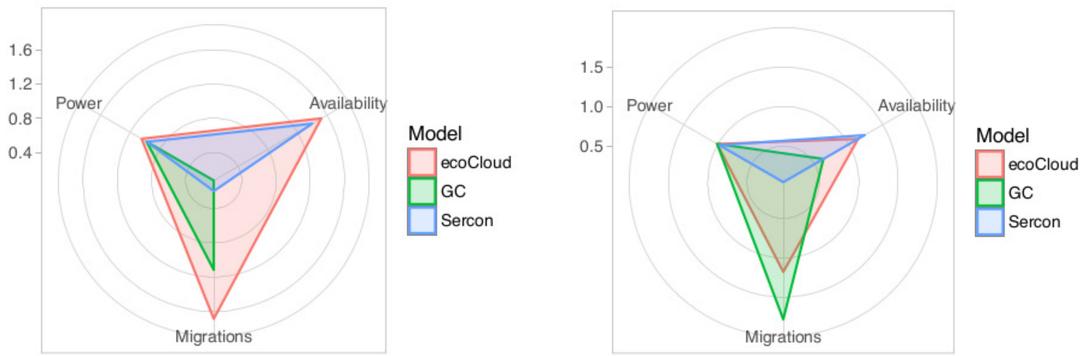
An effective way to visualise this is with a radar chart. Figure 4.9 shows the radar charts for *Exp3* and *Exp4* with energy consumption, the number of SLA violations



(a) CPU Demand
(b) Power Consumption
(c) Migrations
(d) Availability

Figure 4.8: CPU Demand, Power Consumption, Migrations and Availability of Sercon, ecoCloud and GC for Exp 4

(availability) and the number of migrations represented on the three radii. They were divided by the measures' mean values to normalise the radii. A value of 1 means the model achieved the mean value for that measure. While a value of 1.5 means the model reached 1.5 times the mean value for that measure. Smaller triangles represent better models.



(a) Exp3

(b) Exp4

Figure 4.9: Radar charts for *Exp3* and *Exp4* with energy consumption, the number of SLA violations (availability) and the number of migrations represented on the three radii.

Figure 4.9 shows that all three models perform similarly for power consumption for both experiments. However, GC performs best regarding availability, and Sercon performs best regarding the number of migrations. For *Exp 3*, GC outperforms ecoCloud for the number of migrations, while for *Exp 4*, ecoCloud outperforms GC for the number of migrations.

The multi-objective nature of the problem makes it difficult to definitively state that one of the three DVMC strategies is superior to the others, as none was best in all three constraints. One option to address this would be to map each constraint to a standard measure, for example, cost. Energy consumption costs money, as do SLA violations. Also, migrations are undesirable as they impact customer experience and can be related to SLA violations. However, the constraint mapping to cost will vary per data centre, so it could not definitively state which DVMC strategy is best even with a standard measure.

A preferred approach would be to evaluate the DVMC strategies as multi-objective problems. A multi-objective optimisation problem is one where more than one objective function is to be optimised simultaneously [Deb, 2014]. While a single-objective optimisation will search for the best solution to a problem, a multi-objective optimisation will search for a set of the best solutions.

An optimal solution is known as a Pareto-optimal solution. By definition, they are solutions that cannot be improved in any of the objectives without degrading at least one of the other objectives. The set of Pareto optimal solutions is often called the Pareto front. A technique, such as calculating the hypervolume from evolutionary multi-objective optimisation, can be used to compare heuristics across multiple objectives [Zitzler and Thiele, 1999]. However, a multi-objective comparison of the DVMC strategies is outside the scope of this thesis. Another approach would be to make the behaviour of the strategy configurable, allowing the data centre administrators to select the relative weights for each constraint for their data centre.

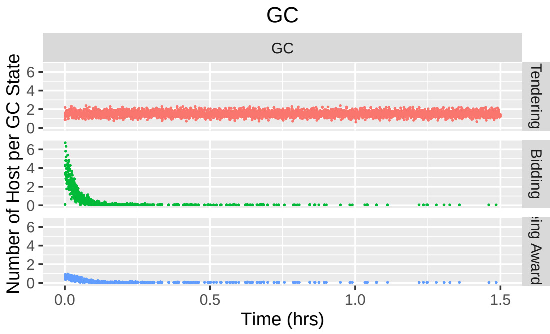
4.3.2 GC Strategy

To better understand how the GC strategy works, let's explore the state of the GC agents over time. Figure 4.10 shows the state of the GC agents for *Exp 1*, *Exp 2*, *Exp 3* and *Exp 4*. The number of tendering hosts is relatively constant over time, around 1 or 2 out of the 250 hosts. However, the number of bidding hosts is initially high, reaching eight, and has an exponential decay with time. Similarly, the number of hosts awarded is relatively low and decaying. This pattern carries through for *Exp 3*, though at a much larger scale, with about 125 of the 80,000 hosts tendering. For *Exp 2*, the tendering, and more notably bidding, increases when the rate of change of the overall data centre load is at its highest. The behaviour for *Exp 4* is similar to that of *Exp 3*.

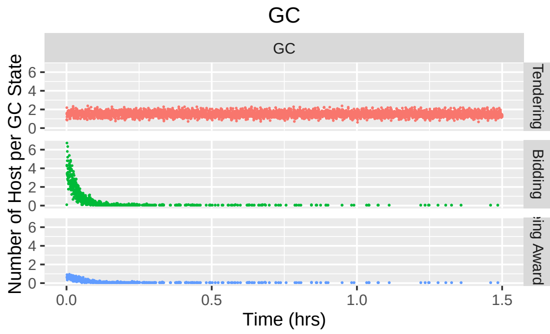
The GC Tender Message Guide and Bid Message Offer values are essential to the strategy. They control the tendering and bidding probability and which tenders and bids are deemed best. The higher the guide and offer, the more likely the tender or bid will happen, and the higher it is rated against other tenders and bids; see Figure 4.2.

At first glance, they appear to be the opposite of what one might expect. A host whose capacity is close to the overloaded threshold and with only a few resources available will have a higher offer than a less utilised host with more resources available. Similarly, a host with a lower resource utilisation will have a higher offer and be more likely to be awarded over a host with higher resources but which can fit better into the resources offered in the tender. Why is this a good strategy? In *Sercon*, the hosts are sorted, and the least utilised host has its VMs migrated to the fullest host. The GC strategy does the same. A fuller host will be more likely to tender, have a higher guide, and accept the highest offer from the least utilised bidding host.

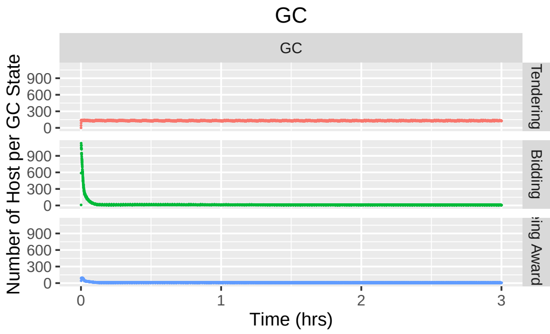
An earlier version of the GC-based DVMC strategy did not tender for available resources; instead, it tendered for VMs to be migrated. It had the concept of underloading, similar to *ecoCloud*. An underloaded host tendered to offer VMs, one VM per tender. The strategy was to migrate them away to put the host into sleep mode. One of the challenges with this approach was that a tendering host would often get bids for some,



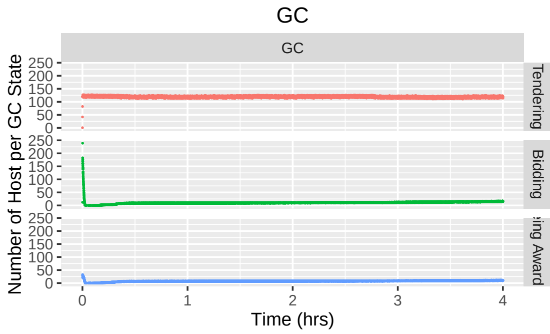
(a) GC states for *Exp 1*



(b) GC states for *Exp 2*



(c) GC states for *Exp 3*



(d) GC states for *Exp 4*

Figure 4.10: The GC agents for *Exp 1*, *Exp 2*, *Exp 3* and *Exp 4*

but not all, of its VMs. If it awarded the VMs it had bid for, it would reduce the number of VMs it had, but as some remained, it could not enter sleep mode. Later it might bid for other tenders and, if awarded, would receive new VMs, which would take it further away from its objective of entering sleep mode. So after many migrations, it might have a utilisation similar to the utilisation with which it started.

Several approaches to address this were tried. For example, a strategy with a congestion detector that favours hosts with the least VMs. Or a strategy that only migrates a VM if it can migrate all the VMs. The latter was an improvement but did not compare to the performance of *Sercon* or *ecoCloud*.

4.4 Conclusion

As noted in the Literature Review chapter, Section 2.3.3, there is novelty and value in developing a genuinely decentralised DVMC strategy, which is evaluated in a cloud data centre modelled using a real-world dataset, which considers packing efficiency, the number of migrations and SLA violations and comparing it to other published techniques.

This chapter presented a decentralised DVMC strategy and evaluated it in a cloud data centre modelled using a real-world dataset, the Google cluster-usage traces (GCT)

[Reiss et al., 2011]. This DVMC strategy was developed using a novel communication protocol for a cooperative MAS called Gossip Contracts (GC). It was evaluated against two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, *ecoCloud*. The evaluation considered packing efficiency, power consumption, the number of migrations and SLA violations. The GC strategy outperformed the other strategies concerning minimising SLA violations. However, it had similar power consumption to *Sercon*, which was lower than *ecoCloud*. In addition, it had equivalent or fewer migrations to *ecoCloud* but a higher number of migrations than *Sercon*. The research in this chapter was published in [Mc Donnell et al., 2020a], and at the time of writing this thesis, it received twelve citations (Scopus).

As decentralisation is prevalent in natural systems, could leveraging nature-inspired techniques speed up this of designing the GC-based heuristic? Evolutionary computing is a family of algorithms inspired by biological evolution. They are population-based generate-and-test problem solvers. The next chapter presents evolved gossip contracts (EGC), a framework for semi-automating a GC-based heuristic design which leverages evolutionary computing.

CHAPTER 5

Evolved Gossip Contracts

5.1 Introduction

Decentralised agent-based approaches offer significant advantages over centralised methods for certain types of problems [Davidsson et al., 2007]. For example, systems composed of large numbers of independent modules that are rapidly changing. Moreover, several critical emerging technologies fit these systems, including Edge computing and the Internet of Things [Mascardi and Weyns, 2018, Julian and Botti, 2019, Kong et al., 2022]. However, the engineering of multi-agent systems is challenging. Mascardi states that "the systematic engineering of large-scale and open MAS still poses many challenges" [Mascardi et al., 2019]

A framework which can aid the design of a decentralised heuristic would be valuable to keep pace with the rising demand for these new technologies. A number of these have been proposed; namely, Anthill [Babaoglu et al., 2002], MetaCompiler for Automatic Algorithm Discovery (MAAD) [Van Berkel et al., 2012], and GESwarm [Ferrante et al., 2013]. These frameworks significantly contribute to semi-automating the search for a MAS that exhibits specific system-level behaviour. While they were evaluated on challenging distributed systems problems, none were assessed on an NP-hard problem. Thus developing an evolutionary computing-based framework to design a decentralised heuristic and evaluating it on an NP-hard problem is worthwhile.

This chapter presents a novel framework used to semi-automate the development of a decentralised heuristic which is robust enough to solve an NP-hard combinatorial optimisation problem. The Evolved Gossip Contracts (EGC) framework is the first EC-based framework developed to design a decentralised heuristic that was evaluated on an NP-hard problem. EGC uses Genetic Programming (GP) to semi-automate the search for a specific GC protocol. It is evaluated using the NP-hard bin packing problem (BPP). The best BPP strategy developed by EGC is compared to two BPP heuristics, *Best Fit* and *First Fit*, which are well established in the literature.

- ❖ Section 5.2.1 presents the Evolved Gossip Contracts (EGC) framework, which uses Genetic Programming to automate the discovery or design of GC-based decentralised protocols. It explains the four-step process of developing a multi-agent system with a desired macro behaviour. The first three comprise the learning phase, and the fourth is the production phase. The MAS designer implements Steps 1 and 3, while Step 2 is automated with GP. Once the learning phase completes, the newly learnt GC-based protocol can be embedded into a production system.

- ❖ Section 5.2.2 explains how a MAS designer uses the Evolved Gossip Contracts framework to discover a decentralised bin packing heuristic. The bins, referred to as *SmartBins*, are GCAgents, the agents in the MAS.

- ❖ Section 5.3.1 presents the results of experiments evaluating the EGC framework. It compares the evolved *SmartBin* heuristics with two popular bin packing heuristics: First Fit and Best Fit. It lists the experimental setup, the ECJ, the PeerSim and the Gossip Contracts configuration.

- ❖ Section 5.3.2 presents the datasets used, namely the Triplets dataset, a long-established 1DBPP dataset defined by Falkenauer et al. [Falkenauer, 1996].

- ❖ Section 5.3.3 describes the individual experiments and their results. The first demonstrates that a MAS designer can use the EGC framework to discover decentralised heuristics to solve an NP-hard optimisation problem, specifically the 1D-BPP. The second shows that GP adds a significant but small value to the EGC technique. The third shows that a larger population size is better at evolving effective heuristics than a larger number of generations. Then, the best-evolved heuristics across all the experiments are further evaluated on larger datasets. Lastly, the fourth experiment explores the effect of increasing the convergence speed weight, and the messages sent weight using multi-criteria fitness.

- Section 5.4 interprets the experimental results and discusses several limitations of the EGC framework.

5.2 Methodology

The methodology section explains how the Evolved Gossip Contracts framework functions and how it is applied to develop a decentralised bin packing heuristic. It comprises two parts. The first presents the Evolved Gossip Contracts framework, which uses Genetic Programming to automate the discovery or design of GC-based decentralised protocols. The second explains how a MAS designer uses the Evolved Gossip Contracts framework to discover a decentralised bin packing heuristic.

5.2.1 Evolving Gossip Contracts

Section 4.1.1 presented the Gossip Contracts (GC) protocol on which the Evolved Gossip Contracts (EGC) framework builds. Section 4.1.2 shows how GC was used to define a decentralised co-operation heuristic by implementing methods which specialise the behaviour of a *GCAgent* to solve the DVMC problem. The EGC framework uses Genetic Programming to search for the implementations of these GC methods.

Note EGC is not carrying out a hyper-parameter optimisation, which would be a search for the optimal GC parameters. Instead, the GC parameters, such as number of neighbours or tender timeout, are fixed. EGC searches for the implementation of the *GCAgent's* methods. These methods control how the individual *GCAgents* behave. The collective behaviour of all the agents together generates the macro behaviour of the multi-agent systems as a whole.

There are four steps when using the EGC framework to develop a Multi-agent System with a desired macro behaviour. Figure 5.1 shows these steps. The first three comprise the learning phase, and the fourth is the production phase. While a person or MAS designer implements Steps 1 and 3, Step 2 is automated with genetic programming (GP); GP is described in Subsection 2.4.1. Once the learning phase completes, the newly learnt GC-based protocol is embedded into a production system; this is Step 4. Note that GP is not used during the production phase; only the GC protocol implemented as the best heuristic evolved during Step 2.

In Step 1, the MAS designer prepares four computational artefacts, which EGC uses in Step 2. They are an implementation of some of the GCAgent's methods to tailor it to the problem being solved; the set of GP functions and terminals EGC can use in the evolved and semi-evolved method; a simulation environment where the evolved methods are evaluated; and a fitness function for evaluating the GP individuals.

1. Firstly, they develop a model for the problem space, which EGC will use as the simulation environment to evaluate the evolved methods of the *GCAgent*. Then, they develop a fitness function which EGC will use to calculate the fitness of the

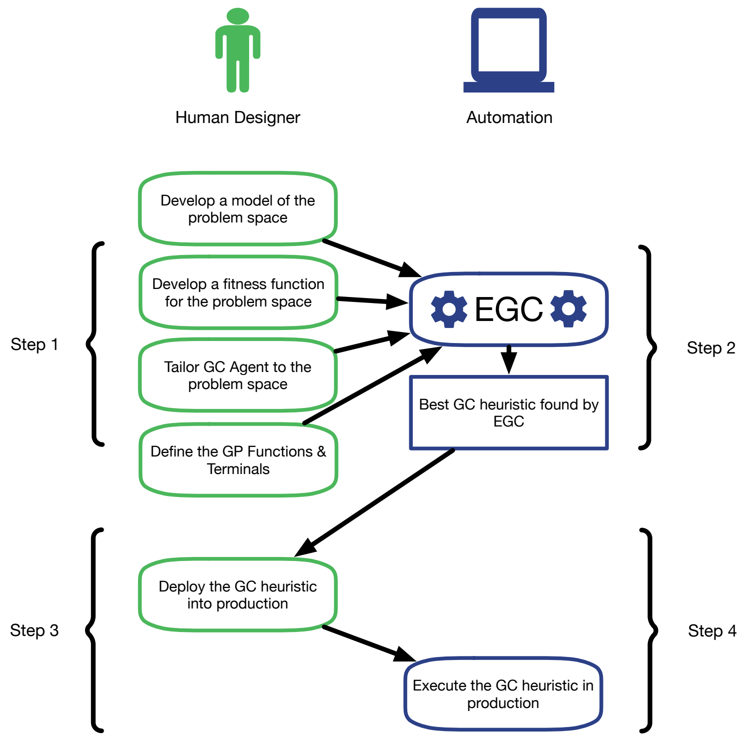


Figure 5.1: The four-step process to develop a multi-agent system with a desired macro behaviour using Evolved Gossip Contracts.

evolved methods of the *GCAgent*. The fitter a *GCAgent*, the more likely EGC will select it for the next GP generation.

2. To tailor the *GCAgent's* to a particular problem, some of its methods must be implemented by the MAS designer, thus narrowing the search space for the GP process. While it might be desirable to evolve all the methods and limit the work required by the MAS designer, it would also cause the heuristic search space to be massive, effectively making the search impractical. Hence, the *getTask()* and *getProposal()* are human-implemented. This way, the MAS designer specifies what can be tendered for and what can be offered. Generally, they also implement the *award()* and *awarded()* methods and thus control how a successful or failed tender process is implemented.

3. The *GCAgent's* methods which the MAS designer does not implement, will be evolved using GP as part of Step 2; these are referred to as evolved methods. Typically, the *shouldTender()*, *getGuide()*, *tenderToDouble()*, *shouldBid()*, *getOffer()*, *bidToDouble()* and *shouldAward()* are evolved. It is possible to partially implement and partially evolve a *GCAgent's* method; these are known as semi-evolved methods.

4. Next, the MAS designer defines the set of GP functions and terminals that EGC

is allowed to use in the evolved and semi-evolved methods. Table 5.1 shows the default set of functions and terminals for EGC. They are basic arithmetic, relational and logical operators, three arithmetic functions and several constants. The guide and offer double variables, parameters to several GCAgent methods, are included. Typically, this set will be augmented by variables unique to the problem space.

Table 5.1: The functions and terminals with a description of the values they return.

Name	Description
$true$	The constant true
0	The constant zero
$+$	Add two inputs
$-$	Subtract two inputs
$*$	Multiply two inputs
div	Protected divide function, divides the first input by the second
$==$	Return true if the two inputs are equal
$<$	Return true if the first input is less than the second input
$>$	Return true if the first input is greater than the second input
!	Return true if the input is false, and false if the input is true.
$\&\&$	Return true if and only if the two inputs are true
$\vert\vert$	Return true if one of the two inputs are true
min	Return the smaller of the two inputs
max	Return the larger of the two inputs
abs	Return the absolute value of the input
$-$	Return the negative of the input
guide	The guide price in a tender message
offer	The offer price in a bid message

Lastly, the MAS designer defines a fitness function for evaluating an individual solution. EGC executes simulations of the individuals' evolved GC protocol and measures how effectively it meets the system's objective. EGC uses the fitness function to assess each individual's fitness.

Along with the evaluation, there are two other valuable fitness criteria: the convergence speed and the number of messages sent. The convergence speed, c, measures how long the MAS executed before it converged. If it does not converge, this will be the simulation end time. Convergence speed is measured by checking for convergence at regular time intervals during the evaluation. The number of messages sent, m, is the total number of tender, bid and award messages sent. Distributed algorithms are criticised for the amount of message passing required, as it increases network overhead [Khan et al., 2018]. The sum of the weights is one, as shown in equation 5.1

$$w_e + w_c + w_m = 1 \tag{5.1}$$

EGC can have different weights for each of these three criteria. The evaluation weight, w_e, measures how important it is for the MAS to effectively solve the problem. The convergence speed weight, w_c, measures how important it is for the multi-agent system to converge quickly. Finally, the message sent weight, w_m, measures how important it is for the multi-agent system to minimise the number of messages sent.

The multi-criteria fitness value, q, is calculated as the weighted sum of the three criteria, as shown in equation 5.2. Consequentially, the three criteria are normalised, thus ensuring the fitness is not skewed to the favour of one. To do this, divide each criterion by the largest value found across all first-generation individuals. For example, if the largest fitness found among all the individuals evaluated in the first generation was 20, then all the fitness values for all individuals for all generations are normalised by dividing them by 20.

$$q = w_e e + w_c c + w_m m \tag{5.2}$$

In Step 2, the EGC framework uses the prepared artefacts to develop a GC-based decentralised heuristic with the best fitness. Figure 5.2 shows the computational flow of the EGC framework. The Evolved Gossip Contracts (EGC) framework is implemented in Scala (version 2.12 with JDK version 1.8). The *GCAgent* uses the *PeerSim* network simulator (version 1.0.5) [Montresor and Jelasity, 2009] for the multi-agent simulations. The *EGCAgent* extends from the *GCAgent* and uses *ECJ* (version 2.7), a Java-based evolutionary computation research system [Scott and Luke, 2019], as the GP framework.

Figure 5.2 shows the computational flow of EGC starting with the initialisation of the GP system, *ECJ*, using the *EGCAgent.params* file. It creates the first generation of individuals to be evaluated and begins the GP process.

EGC evaluates the individuals using the *PeerSim* simulator (see Subsection 2.2.2). *PeerSim* initiates using the *GCAgent.properties* config file. This creates the *EGCAgents* in the multi-agent system. As the simulation progresses, the methods of the *EGCAgents* are called. When these called methods are evolved or semi-evolved, the return value is calculated by evaluating the GP tree for that method. Finally, when the multi-agent system simulation completes, EGC evaluates the individual, calculating their fitness.

When EGC has evaluated all individuals, it uses GP to create the next generation. It

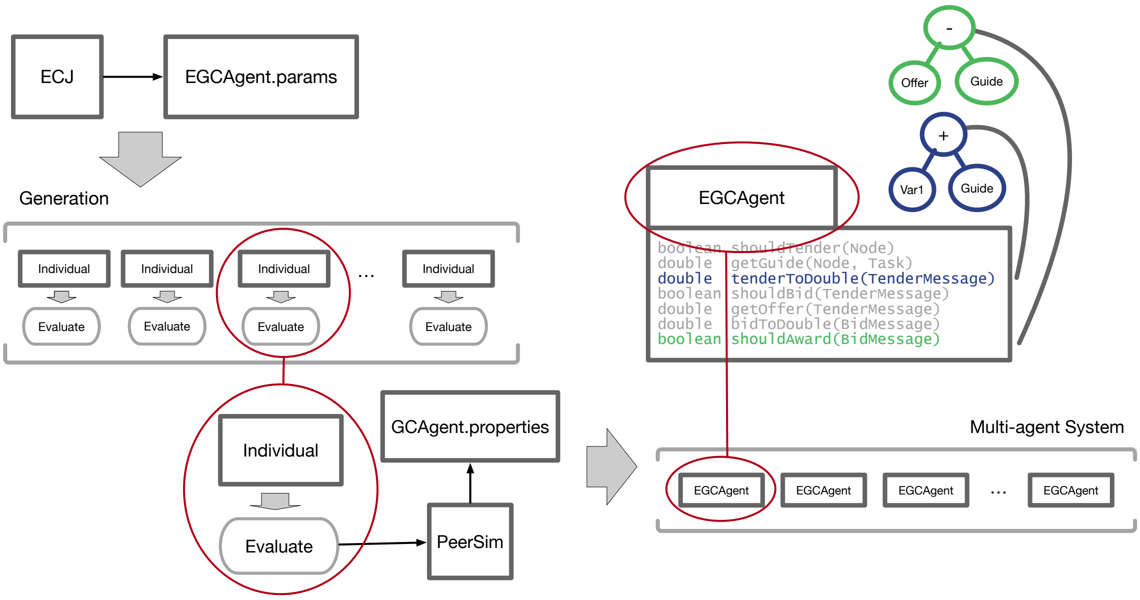


Figure 5.2: The Architecture of an Evolved Gossip Contracts system showing how it evolves a Gossip Contracts-based multi-agent system.

then evaluates these new individuals. This process continues until EGC evaluates a fixed number of generations. Algorithm 7 shows the pseudocode for the EGC algorithm. Note that the method *evaluateSimulation()* is not defined because its definition is unique to each use of the EGC framework. While Algorithm 8 shows pseudo-code for the EGC algorithm when the multi-criteria fitness is used.

At the end of Step 2, one of the individuals will have higher overall fitness. Then, in Step 3, the MAS designer copies this individual's evolved methods into a lightweight *GCAgent* which is used in the production phase, Step 4.

5.2.2 SmartBin

This section describes the preparation of the four computational artefacts and their use by the EGC framework to develop a GC-based decentralised bin packing heuristic. As part of Step 1, the model was developed using the PeerSim simulator. In this formulation of a multi-agent strategy for the *1D-BPP*, the bins, which are referred to as *SmartBins*, are *GCAgents*; they are the agents in the MAS. The *SmartBin* can access the *queue* where the items to be packed are available. Each simulation comprises several cycles. Once per cycle, each *SmartBin* checks the queue to see if there are items to be packed. If so, it proceeds to pack the item at the head of the queue.

Figure 5.2.2 shows a schematic of the model with the *Queue* and three *SmartBins*. *SmartBin 1* checks the *Queue* where one Item is ready to be packed. It stores the item as *SmartBin 1* can fit it. Then, it starts the tender bid cycle, offering the item to other

Algorithm 7: Evolved Gossip Contracts Algorithm

```
ecj := initialise ECJ with the EGCAgent.params file
G := numberOfGenerations
N := numberOfIndividuals
R := numberOfRuns
individuals := ecj.createFirstGeneration()
a/* Initialise an array to store the fitnesses of the
   individuals.                                          */
fitnesses := empty array of size N
for g := 1 to G do
    for i := 1 to N do
        fitnesses[i] := evaluateIndividual(individuals[i], R)
    end
    individuals := ecj.createNextGeneration(individuals, fitnesses)
end
Function evaluateIndividual(individual, R):
    peersim := initialise PeerSim with the GCAgent.properties file
    totalEvaluations, evaluation := 0
    for r := 1 to R do
        peersim.runSimulation()
        evaluation := evaluateSimulation()
        totalEvaluations += evaluation
    end
    return totalEvaluations/R
return
```

bins. It sends a Tender message to *SmartBin 2*, which in turn gossips the message to *SmartBin 3*. *SmartBin 2* can not fit the item and so does not bid, while *SmartBin 3* can. *SmartBin 3* sends a bid message to *SmartBin 1*. When the tender timeout is reached, *SmartBin 1* decides to accept the bid from *SmartBin 3* and responds with an Award message. The item is then moved from *SmartBin 1* to *SmartBin 3*. The figure shows one possible outcome for the three *SmartBins*, but the evolved methods dictate their actual behaviour.

Next, the fitness function was developed to evaluate the fitness of the evolved methods of the *GCAgent*. Several PeerSim simulations were executed for each evaluation, each with a different queue. At the end of each simulation, the number of *SmartBins* containing items is counted. The fitness is calculated as the mean number of *SmartBins* needed to pack all the items across these simulations; the lower the number of *SmartBins* needed, the fitter the evolved GC protocol.

Then, the *SmartBin's* methods are implemented to tailor it to solve bin packing problems. Each *SmartBin* executes the GC tender algorithm once per cycle and executes

Algorithm 8: Evolved Gossip Contracts with multi-criteria fitness function Algorithm

```
ecj := initialise ECJ with the EGCAgent.params file
G := numberOfGenerations
N := numberOfIndividuals
R := numberOfRuns
individuals := ecj.createFirstGeneration()
/* Initialise arrays to store the evaluations, convergence
   times, number of messages sent and fitnesses of the
   individuals.                                            */
e, c, m, fitnesses := empty array of size N
for g := 1 to G do
    for i := 1 to N do
        (e[i], c[i], m[i]) := evaluateIndividual(individuals[i], R)
    end
    /* first generation store max values across all individuals
       for normalisation                                    */
    if g = 1 then
        e_max := max value in e
        c_max := max value in c
        m_max := max value in m
    end
    for i := 1 to N do
        fitnesses[i] := (w_e e[i]/e_max) + (w_c c[i]/c_max) + (w_m m[i]/m_max)
    end
    individuals := ecj.createNextGeneration(individuals, fitnesses)
end
Function evaluateIndividual(individual, R):
    peersim := initialise PeerSim with the GCAgent.properties file
    totalEvaluations, totalConvergenceTime, totalMessagesSent := 0
    for r := 1 to R do
        totalEvaluations += peersim.runSimulation()
        totalConvergenceTime += peersim.getSimulationEndTime()
        totalMessagesSent += peersim.getNumberOfMessagesSent()
    end
    return (totalEvaluations/R, totalConvergenceTime/R,
     totalMessagesSent/R)
return
```

the GC bid algorithm each time it receives a tender message. They communicate via the GC protocol to pack the items from the queue into as few *SmartBins* as possible.

As a *SmartBin* starts executing the GC tender algorithm, it checks the queue to see if an item is to be packed. If there is, and if the *SmartBin* can fit the item given its free capacity, it then pops the item from the queue and stores it. Then, it executes the

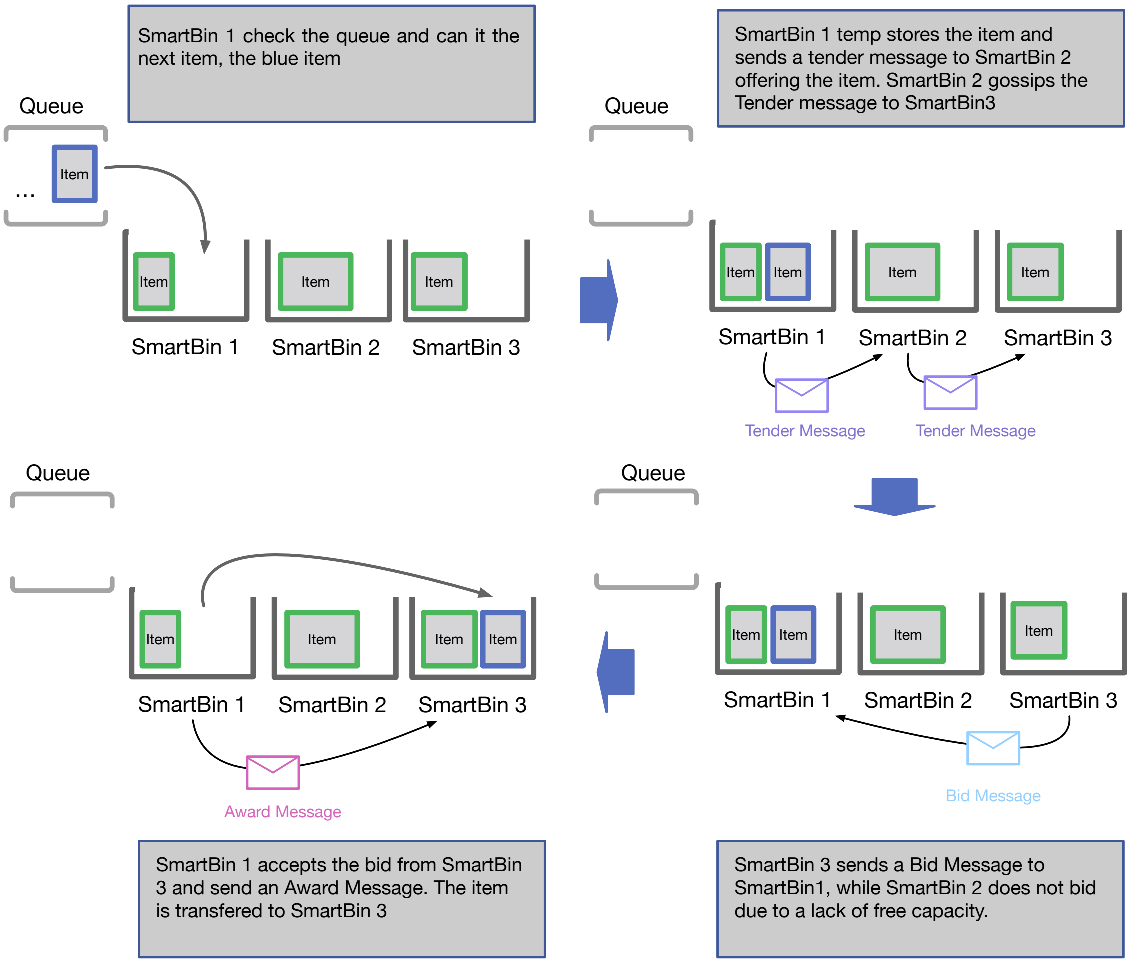


Figure 5.3: The model with three *SmartBins* and the *Queue*

GC tender algorithm offering the item to other *SmartBins*. Thus *getTask()* is human-implemented and returns the stored item. The methods *award()* and *awarded()* are also human-implemented. When the tender is successful, the item is packed into the winning *SmartBin*, and when not, into its own *SmartBin*.

Two semi-evolved methods are also implemented, *shouldTender()* and *shouldBid()*.The *shouldTender()* method will return true when a task is stored for packing, and its GP tree returns true. The *shouldBid()* method will return true when it can fit the item, and its GP tree returns true. The remaining five methods *getGuide()*, *tenderToDouble()*, *getOffer()*, *bidToDouble()* and *shouldAward()* are all evolved. Algorithm 9 shows the pseudo-code for a *SmartBin*.

Lastly, the GP functions and terminals used in the evolved and semi-evolved methods are defined. For the *SmartBins*, they are the default set of GP nodes described in Table 5.1. They are augmented by two new ones, *freeCapacity* and *itemSize* described in Table 5.2.

Algorithm 9: SmartBin Algorithm

```
/* Human-implemented methods are highlighted in teal          */
/* evolved and semi-evolved methods are highlighted in violet.
   */
individual := initialise the GP individual needed to access the GP trees
queue := initialise to the queue where the items become available
itemToPack := initialise to Null
Function preTenderAwardCycle():
    if queue.hasItem() and canFit(item) then
        itemToPack := item
    end
Function shouldTender():
    return itemToPack != Null and
     individual.shouldTenderTree.evaluate()
Function getTask():
    return itemToPack
Function getGuide():
    return individual.getGuideTree.evaluate()
Function tendetoDouble(tenderMessage):
    return individual.tenderoDoubleTree.evaluate(tenderMessage)
Function shouldBid(tenderMessage):
    return itemToPack != Null and canFit(tenderMessage.item) and
     individual.shouldBidTree.evaluate(tenderMessage)
Function getProposal(tenderMessage):
    return Null
Function getOffer(tenderMessage):
    return individual.getOfferTree.evaluate(tenderMessage)
Function bidToDouble(bidMessage):
    return individual.bidToDoubleTree.evaluate(bidMessage)
Function shouldAward(bidMessage):
    return individual.shoulAwardTree.evaluate(bidMessage)
Function award(awardMessage):
    itemToPack := Null
Function awarded(awardMessage):
    pack awardMessage.task
Function postTenderAwardCycle():
    if itemToPack != Null then
        pack itemToPack
        itemToPack := Null
    end
```

5.3 Experimental Results

This section will consider the effectiveness of the EGC framework by evaluating the *SmartBin* heuristics in comparison with two popular bin packing heuristics, namely, *First*

Table 5.2: A description of the *SmartBin* extra genetic programming variables.

Name	Description
freeCapacity	The remaining free capacity in the *SmartBin*
itemSize	The size of the item to be packed

Fit and *Best Fit*. First, it will describe the experimental setup and the BPP test cases. Then it will present the experiments and their results.

5.3.1 Experimental Setup

The EGC configuration has three aspects: the ECJ/GP configuration, the PeerSim configuration and the Gossip Contracts configuration. Table 5.3 shows the configuration for the experiments unless otherwise stated.

The ECJ/GP Configuration is based on Koza's classic configuration included with *ECJ* [Scott and Luke, 2019]. For example, the crossover operation is performed on about 90% of the individuals in the population and the mutation operation is performed on about 1%. The main differences with Koza's classic configuration are that the minimum depth of the trees is one instead of two, and parsimony pressure is used during selection. Parsimony pressure is a selection operator which favour smaller individuals, a technique used to limit the growth rate of the GP trees. Also, note the number of Generations, G and Population size, N, vary in the three experiments. Finally, in the PeerSim configuration, the number of cycles is the number of times in a simulation that each *SmartBin* can execute its tendering algorithm. Refer to Table 5.3 for an explanation of these parameters.

5.3.2 Test Cases

The experiments use a long-established 1DBPP dataset defined by Falkenauer et al. called *Triplets* [Falkenauer, 1996]. The name, *Triplets*, comes from the fact that each set of three items is defined so that they fit exactly into one bin. Thus, the minimum number of bins needed to pack the items is a third of the number of items.

For these problems, each bin has a capacity of 1000, c. The first item size, s_1, is generated by drawing uniformly from the integer range $[380, 490]$, leaving a space S, between 510 and 620. The second item size, s_2, is drawn uniformly from $[250, S/2)$, and the third item, s_3 is selected by equation 5.3.

$$c = s_1 + s_2 + s_3 \tag{5.3}$$

Table 5.3: The genetic programming, PeerSim and Gossip Contracts configuration

EGC/GP Configuration	
Generations, G	20
Population size, N	240
Number of runs, R	20
Crossover probability	0.9
Mutation probability	0.1
Tree initialisation	Ramped half-and-half
Tree initialisation minimum depth	1
Tree initialisation maximum depth	6
Selection	Tournament selection with parsimony pressure
Selection tournament size	7
PeerSim Configuration	
Number of cycles	15
Time steps per cycle	300
Number of *SmartBins*	160
GC Configuration	
Neighbourhood size, k	4
Tender gossip counter, g	4
Tender timeout, t_t	4
Bid buffer, b_b	1
Award timeout, t	1

A *Triplet* dataset is described by four numbers, for example, *249-380-490-250*, as illustrated in Figure 5.4. The first number, *249*, represents the number of items to be packed. Note the number of items must be a factor of three. The second and third numbers, *380* and *490* are the min and max of the integer range from which the first item size is taken. The fourth number, *250*, is the min of the integer range from which the second item size is taken. The experiments use datasets which only differ in the numbers of items: 120, 249, 501 and 999. The first item size is between 380 and 490, and the second is larger than 250.

The experiments use this dataset in two ways; in the learning phase as part of the EGC's evaluation process and in the production phase to evaluate how the evolved *SmartBins* heuristics compare to the popular bin packing heuristics. During the EGC's evaluation process, each GP individual is evaluated using 20 unique queues created from the *249-380-490-250* dataset. However, when comparing the evolved *SmartBins* heuristics with the popular bin packing heuristics, the experiment uses a larger number of unique queues, 100, and queues of different types: *249-380-490-250*, *501-380-490-250*

Figure 5.4: A *Triplet* dataset is described by four numbers

and *999-380-490-250*.

Table 5.4 shows the median and interquartile range (IQR) for *Best Fit* and *First Fit* for the three datasets. A Shapiro–Wilk test of normality [Shapiro and Wilk, 1965] showed that the number of open bins across the experimental executions does not follow a normal distribution; thus, non-parametric statistical measures are used. For example, median and IQR are used instead of mean and standard deviation, and the Wilcoxon Signed-rank test is used instead is Student's T-test.

Table 5.4: The median and interquartile range (IQR) for *Best Fit* and *First Fit* for the *249-380-490-250*, *501-380-490-250* and *999-380-490-250* datasets.

Dataset	Heuristic	Median	IQR
249-380-490-250	*Best Fit*	93	1
	First Fit	93	1
501-380-490-250	*Best Fit*	186	1
	First Fit	186	1
999-380-490-250	*Best Fit*	372	1
	First Fit	372	1

5.3.3 Experiments

The first experiment, *Exp1*, demonstrates that the EGC framework can be used to discover decentralised heuristics to solve an NP-hard optimisation problem, specifically the 1D-BPP. They are better than the popular *Best Fit* or *First Fit* heuristics. The next experiment, $Exp1_{seed=19}$ explores the best evolved *SmartBin* in *Exp 1*, the strategy evolved with a seed of 19. This experiment evaluates how $Exp1_{seed=19}$ performed against the *Best Fit* and *First Fit* heuristics.

The second experiment, *Exp2*, shows that GP adds significant value to the EGC technique. Finally, the third experiment, *Exp3*, explores the impact of population size and the number of generations on evolving effective heuristics. Finally, the fourth experiment, *Exp4*, evaluates multi-criteria fitness to increase the convergence speed and reduce the messages sent. The experiments are summarised in Table 5.5.

Table 5.5: Summary of the experiments

Experiment	Description
Exp 1	The EGC framework can discover a decentralised heuristic to solve the 1D-BPP.
$Exp1_{seed=19}$	The performance of the next evolved strategy against the *Best Fit* and *First Fit* heuristics.
Exp 2	Genetic programming adds significant value to the EGC technique.
Exp 3	A grid parameter sweep across the number of generations and the number of individuals in the population parameters.
Exp 4	Use of multi-criteria fitness to increase the convergence speed and to reduce the messages sent.

Unless otherwise stated, these experiments used the *SmartBin* formulation described in Section 5.2.2 and the experimental setup described in Section 5.3.1. Each experiment was repeated 20 times with different random seeds to ensure the results were statistically significant. Table 5.6 shows a summary of the results, including the number of generations, population size, and the median and IQR of the number of open bins, across the 20 random seeds. Interestingly, the IQR for *Exp2* is smaller than that of the other experiments. This indicates that during the random search for a heuristic, where EC is not used, the results are highly consistent across the 20 runs with different random seeds. For experiment *Exp1*, there were 4,800 individuals evaluated, each with 20 unique datasets, and repeated for 20 different random seeds; this gives a total of 1,920,000 evaluations. *Exp2* had a similar number of evaluations, while *Exp3* had 36 times that of *Exp1*. The computing resources used were provided by the Irish Centre for High-End Computing, ICHEC. The experiments were executed on nodes with 2x20-core 2.4 GHz Intel Xeon Gold 6148 (Skylake) processors and 192 GiB of RAM.

Table 5.6: The number of generations and population size for *Exp1* and *Exp2*, along with the median and IQR of the number of open bins across the 20 random seeds.

	Exp1	*Exp2*
Generations, G	20	1
Population size, N	240	4800
Open Bins Median (IQR)	92.975 (1.4125)	93.35 (0.0750)

Exp 1 The first experiment evaluated the effectiveness of EGC in discovering decentralised heuristics. It used the *SmartBin* formulation executing for 20 generations with

a population size of 240. The *Exp1* section of Table 5.7 shows the number of open bins for the 20 executions of *249-380-490-250* for the individual with the best fitness across the 20 seeds.

The Shapiro–Wilk normality test showed that the distribution is not normal, as evident from the box plot in Figure 5.5. The median value is 92.975 and has an IQR of 1.4125. A one-sample Wilcoxon Signed test [Wilcoxon, 1992] show that the evolved *SmartBins* population median, 92.975, is statistically significantly less than the medians of *Best Fit* or *First Fit*, both 93; the p-value was 0.0212. Thus, there is high confidence that the evolved *SmartBins* population median is lower than the medians of *Best Fit* or *First Fit*.

Table 5.7: The minimum number of open bins for the best individual in each of the 20 executions of *Exp1* and *Exp2*.

Exp1	Seed	0	1	2	3	4	5	6
	Open Bins	93.05	93.0	91.65	93.1	93.0	93.05	92.1
	Seed	7	8	9	10	11	12	13
	Open Bins	92.95	93.05	91.45	93.0	91.6	91.55	93.1
	Seed	14	15	16	17	18	19	
	Open Bins	93.0	92.85	93.05	92.95	91.5	90.6	
Exp2	Seed	0	1	2	3	4	5	6
	Open Bins	93.35	93.4	93.45	93.3	93.55	93.5	93.4
	Seed	7	8	9	10	11	12	13
	Open Bins	93.2	93.4	93.35	93.3	93.3	93.25	93.45
	Seed	14	15	16	17	18	19	
	Open Bins	93.4	93.35	93.55	93.35	93.35	93.35	

Exp $1_{seed=19}$ The following section will explore the best evolved *SmartBin* in *Exp1*, which is the execution where the seed is set to 19 in *Exp1*. As it performed very well against *Best Fit* and *First Fit*, it will be analysed in more detail. Figure 5.6a shows box plots of the number of open bins for each generation for the individuals. Note that while even in the first generation, there is an individual with a good score of 93.65, 19 generations later, the best score has reduced considerably, by 3.05, to 90.60. This can be seen clearly in Figure 5.6b, which shows the number of open bins for the best individual in each generation.

The *SmartBin* evolved in $Exp1_{seed=19}$ was compared with the popular bin packing heuristics, *Best Fit* and *First Fit*. This evaluation used a larger number of queues, 100, and queues of different types. Figures 5.6c, 5.6d and 5.6e show the box plots of these

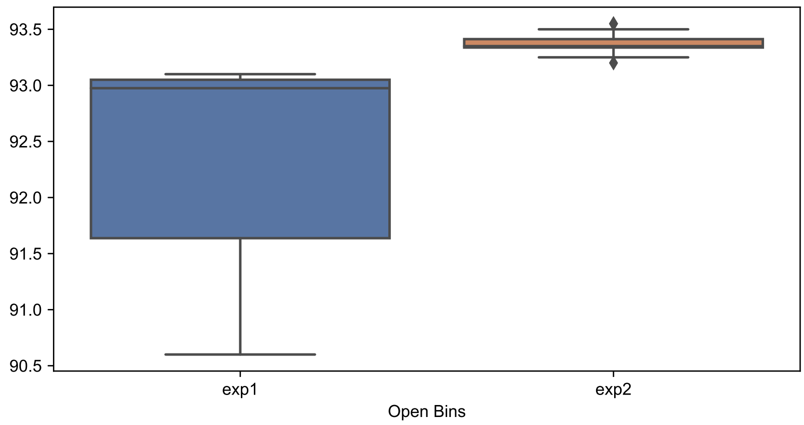


Figure 5.5: Box plots of the minimum number of open bins for the best individual in each of the 20 executions of *Exp1* and *Exp2*.

experiments for the datasets *249-380-490-250*, *501-380-490-250* and *999-380-490-250*, respectively. As the number of items to be packed increases, from 249 in Figure 5.6c to 999 in Figure 5.6e, the improvement in the performance of the *SmartBin* evolved in $Exp1_{seed=19}$ becomes more pronounced. Its packing effectiveness is amplified with larger numbers of items.

The datasets used so far in the comparison are the same type as those used by EGC. The *SmartBin* evolved in $Exp1_{seed=19}$ was also evaluated using previously unseen datasets. These evaluations use another popular dataset from the literature, the *Uniform* dataset; this dataset proved most difficult for Martello, S. and Toth [Toth and Martello, 1990]. It has items of integer size uniformly distributed from 20 to 100, and the bin capacity is 150.

$Exp1_{seed=19}$ was evaluated with three of these datasets *250-20-100*, *500-20-100* and *1000-20-100* where the number of items is: 250, 500 and 1000, respectively. Figures 5.6f, 5.6g and 5.6h shows the box plots of these experiments for the datasets *250-20-100*, *500-20-100* and *1000-20-100*, respectively. The improved performance of $Exp1_{seed=19}$ on the *Triplets* dataset is no longer evident. This may be due to it being over-fitted to the *Triplet* dataset on which it was trained.

Algorithm 10 shows the simplified evolved algorithm for $Exp1_{seed=19}$. The algorithm is simplified and rewritten to aid readability, though the solutions presented are functionally equivalent to those output by the GP process. Only methods which EGC evolved are included.

All methods in the evolved *SmartBin* algorithm comprise a single term, with one exception; the method *tenderToDouble()*. The evolved *SmartBin* will always tender, bid and award. Effectively, the algorithm does not use the guide. Instead, the bids are sorted based on the offer, which is the free capacity of the bidding agent. Thus, the smaller the free capacity of the bidding agent, the better the bid. This mechanism for sorting the bids will select the bidding agent where the item best fits.

The method, *tenderToDouble()*, sorts the tender messages a bidding agent receives using two parameters. They are the free capacity, f, and the size of the item to be packed, i. Equation 5.4 shows the *tenderToDouble()* method.

$$\left| \min\left(4i^2 - 2if + f, -i^2 + if\right) \right| \tag{5.4}$$

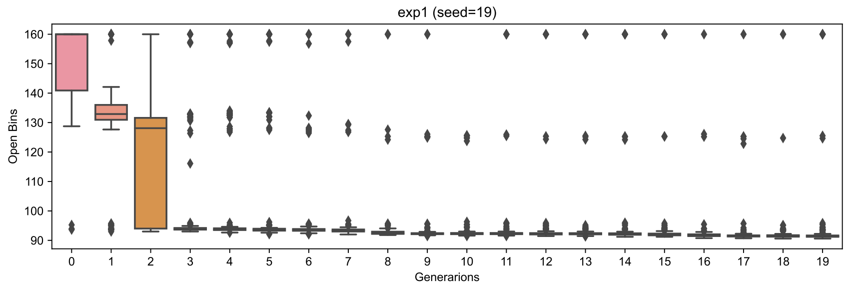
Figure 5.7 visualises as a heatmap the *tenderToDouble()* method; the heat scale is in millions. A lower heat value represents a higher-valued tender message. Thus, the pale blue represents the best items to bid for, given their size and the free space in the *SmartBin*. There are two distinct pale blue lines on the heatmap. The first, with a slope of one, is expected; this line represents the situation where the item fits precisely in the *SmartBin*. The second line has a slope of two and represents the case where the free capacity is twice the size of the item.

Algorithm 10: Evolved *SmartBin* Algorithm *Exp1* (seed=19)

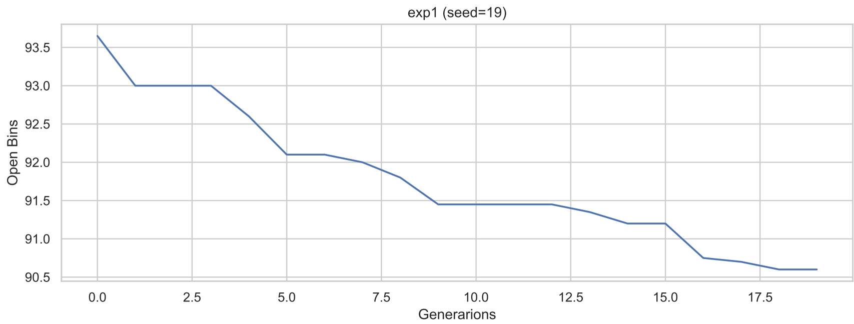
```
itemToPack := initialise to Null
Function shouldTender():
    return itemToPack != Null and true
Function getGuide():
    return 0
Function tendetoDouble(tenderMessage):
    f := freeCapacity
    i := tenderMessage.item.size
    return abs(min(4i^2 − 2if + f, −i^2 + if))
Function shouldBid(tenderMessage):
    return itemToPack != Null and canFit(tenderMessage.item) and true
Function getOffer(tenderMessage):
    return freeCapacity
Function bidToDouble(bidMessage):
    return bidMessage.offer
Function shoulAward(bidMessage):
    return true
```

Exp 2 The second experiment investigates whether the GP adds significant value to the EGC framework. It compares the set of best evolved *SmartBins* in *Exp1* to those evolved where the number of generations is reduced to one.

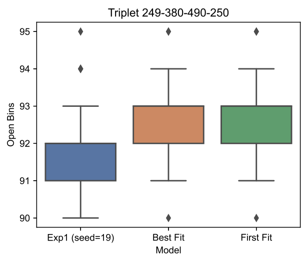
Exp2 experiment should have an equivalent number of evaluations to *Exp1*, to ensure a fair comparison. Thus the number of individuals in *Exp2* is increased to 4800. During



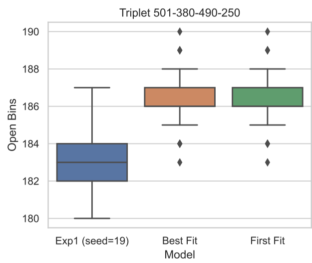
(a) Box plots of the number of open bins for each generation for the individuals in the experiment of *Exp1* with a seed of 19.



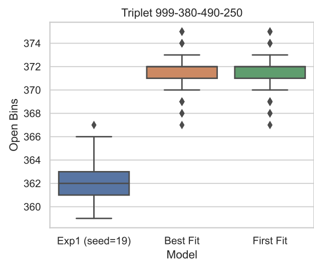
(b) Plots of the number of open bins for the best individual for each generation in the experiment of *Exp1* with a seed of 19.



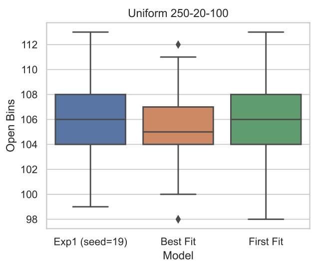
(c)



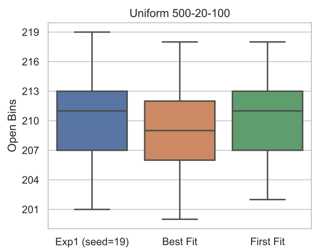
(d)



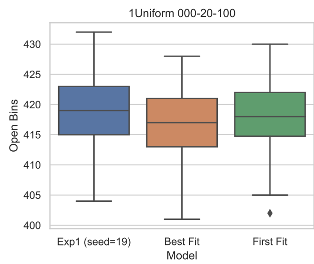
(e)



(f)



(g)



(h)

Figure 5.6: Box plots showing *Exp1* with a seed of 19 compared to the popular bin packing heuristics, *Best Fit* and *First Fit*, using 100 queues of different types *Triplet 249-380-490-250* (c), *Triplet 501-380-490-250* (d), *Triplet 999-380-490-250* (e), *Uniform 250-20-100* (f), *Uniform 500-20-100* (g) and *Uniform 1000-20-100* (h).

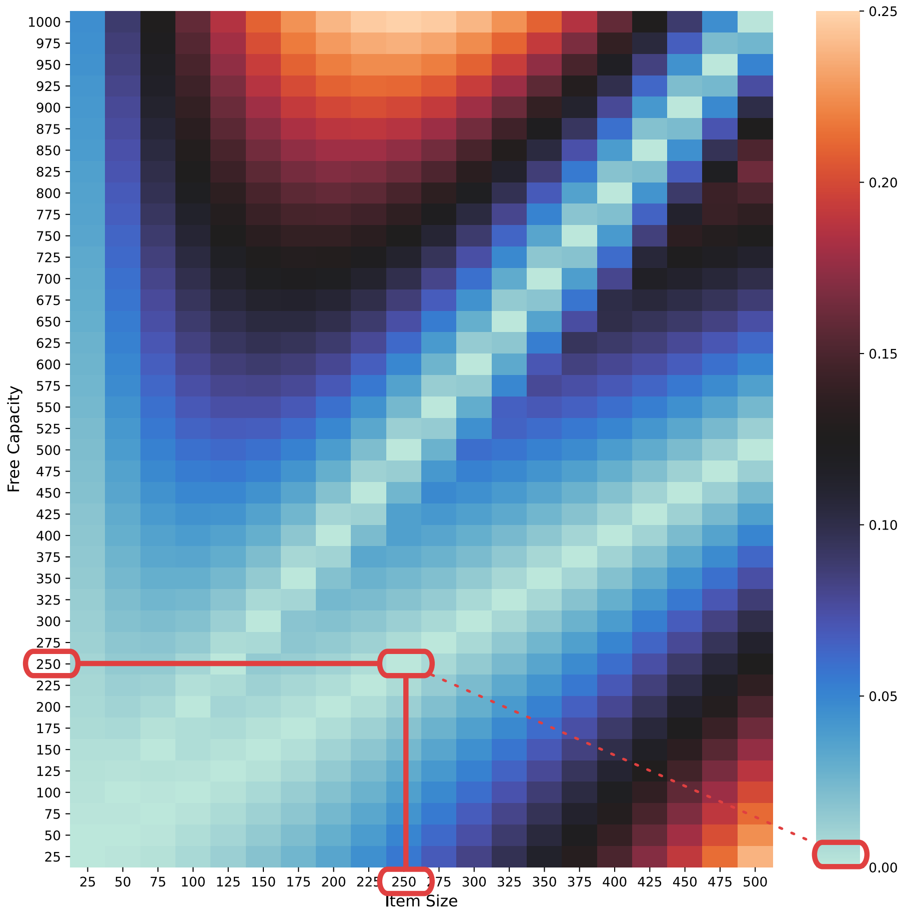


Figure 5.7: Heatmap of the return value for the *tenderToDouble()* method for $Exp1_{seed=19}$ given the size of the item to be packed, i, and free capacity of the bidding *SmartBin*, f; the heatmap scale is in millions and the lower the score the better the tender. For example, for $i = 250$ and $f = 250$ the value is 0, the ideal score.

Exp1, there were 4,800 evaluations; each of the 240 individuals was evaluated 20 times, once per generation. While during *Exp2*, there were also 4,800 evaluations; each of the 4,800 individuals was evaluated once as there is only one generation.

The *Exp2* section of Table 5.7 shows the number of open bins for the 20 executions of *249-380-490-250* for the individual with the best fitness across the 20 seeds. The median value is 93.35, and an IQR of 0.0750. A two-sample Wilcoxon Signed test was used to compare the results of *Exp2* with *Exp1*; this statistical test compares two paired sample sets that do not have a normal distribution. It shows that there is a minor decrease between the evolved SmartBin from *Exp1* compared to that of *Exp2*, but that it is statistically significant; the p-value was $3.8e^{-06}$. These results show that GP does add value to the EGC technique.

Exp 3 The third experiment investigates the performance impact of two parameters. It completes a grid parameter sweep across the number of generations – 10, 20, 40 and 120 – and the number of individuals in the population – 40, 120, 240 and 480. The distribution of the number of open bins across the 20 random seeds for all these experiments was not normal when evaluated using the Shapiro–Wilk normality test; thus, non-parametric statistical measures and tests are used for the analysis. Table 5.8 shows the median and IQR of the number of open bins across the 20 seeds, while Figure 5.8 shows the boxplots for each of the configurations; outliers are not shown.

Table 5.8: The median and IQR of the number of open bins across the 20 seeds for *Exp3* — a grid search with 10, 20, 40 and 120 generations and 40, 120, 240 and 480 individuals in the population.

	Number of Generations			
Population Size	10	20	40	120
40	94.35 (9.78)	94.35 (8.69)	94.35 (1.35)	94.35 (1.35)
120	93.83 (0.39)	93.78 (0.38)	93.83 (0.40)	93.75 (0.43)
240	93.53 (1.75)	92.98 (1.41)	93.18 (1.85)	92.75 (1.83)
480	93.28 (1.39)	93.03 (1.59)	92.75 (1.69)	93.08 (1.69)

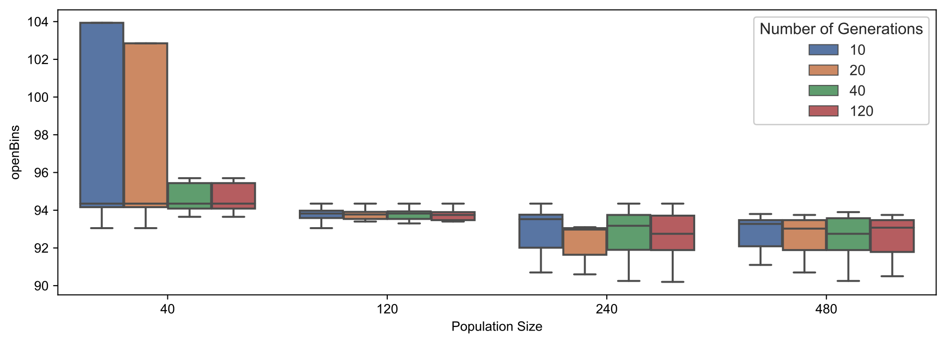


Figure 5.8: Box plots of the minimum number of open bins for the best individual in each of the 20 executions of grid search; outliers are not shown.

The lowest median values were for the population size of 240 and 120 generations and the population size of 480 and 40 generations. The distribution of these two data sets shows they are not distinct using a two-sided Wilcoxon signed-rank test; the p-value is 0.86. The configuration used in *Exp1* has the next lowest median value; it was a population size of 240 and 20 generations. Again, statistical tests showed that the difference of 0.23 between the distributions was not statistically significant; the p-value was 0.38. In fact, there is no difference of statistical significance between the six distributions from experiments with a population of size 240 and 480, and 20, 40 and 120 generations. From this, I conclude that there is no improvement by adding extra individuals or generations over a population of 240 and 20 generations; this is the configuration used in *Exp1*.

Next is an evaluation of whether the EGC framework performs better or worse, having a smaller number of individuals in the populations but a higher number of generations keeping the number of evaluations consistent with *Exp1*. Table 5.9 shows the median and IQR of the number of open bins across the 20 seeds, while Figure 5.9 shows the boxplots for each of the configurations; outliers have been removed. The lowest median is for a population of 240 and 20 generations, which is the configuration used in *Exp1*. The difference of the medians of adjacent experiments is only significant in three cases: P=240 G=20 (exp1) and P=160 G=30 (p-value is 0.012); P=160 G=30 and P=120 G=40 (p-value is 0.019); and P=120 G=40 and P=80 G=60 (p-value is 0.047). From this, I conclude that a larger population with smaller generations is better than a smaller population with larger generations, given a minimum number of generations, 20.

Table 5.9: The median and IQR of the number of open bins across the 20 seeds for experiments with the number of evaluations equivalent to *Exp1*, but with varying population size and the number of generations.

Population Size	Number of Generations	Median (IQR)
4800	1	93.35 (0.08)
480	10	93.28 (1.39)
240	20	92.98 (1.41)
120	40	93.83 (0.40)
160	30	93.40 (1.33)
80	60	94.10 (0.63)
40	120	94.35 (1.35)

Exp 4 The fourth experiment explores using multi-criteria fitness and increasing the convergence speed weight and the messages sent weight. *Exp 4a* only weights the evaluation, thus has a similar weighting to *Exp 1, 2 and 3*. It is the benchmark for *Exp 4b* and *Exp 4c*. *Exp 4b* increases the convergence speed weight to 0.3. This weighting should find decentralised BPP heuristics which converge more quickly. In contrast, *Exp 4b* increases the messages sent weight to 0.3. Again, this weighting should find decentralised BPP heuristics which send fewer messages. Table 5.10 shows the experiment's criteria weights.

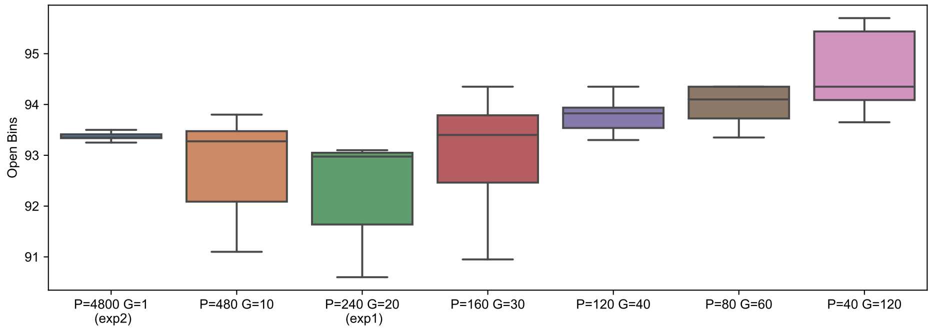


Figure 5.9: Box plots of the minimum number of open bins for the best individual in each of the 20 executions of seven experiments with the number of evaluations equivalent to *Exp1*, but with varying populations size (P) and the number of generations (G); outliers are not shown

Table 5.10: The criteria weights for *Exp 4a*, *Exp 4b* and *Exp 4c*

Criteria weight	*Exp 4a*	*Exp 4b*	*Exp 4c*
Evaluation, w_e	1.0	0.7	0.7
Convergence speed, w_c	0.0	0.3	0.0
Messages sent, w_m	0.0	0.0	0.3

The EGC/GP configuration differs in *Exp 4*. The number of generations and runs was reduced from 20 to 12, while the population size was 250, slightly larger than the 240 used in *Exp 1*. The PeerSim and GC configuration did not alter. Table 5.11 shows the EGC/GP configuration that differs. The datasets used during EGC's learning phase were also different. It used four Uniform datasets; they were *120-20-100*, *250-20-100*, *500-20-100* and *1000-20-100*. It used three unique queues from each dataset to make up the twelve runs.

Table 5.11: The configuration for *Exp 4*

EGC/GP Configuration	
Generations, G	12
Population size, N	250
Number of runs, R	12

Table 5.12 shows the mean (standard deviation), across the 20 runs, of the fitness, number of open bins, convergence time and the messages sent for the last generation for *Exp 4a*, *Exp 4b* and *Exp 4c*.

Figure 5.10, 5.11, and 5.12 show the results of the GP analysis for the experiments across the 20 runs. The "Fitness by generation" charts show how the mean fitness of

Table 5.12: The mean (standard deviation) of fitness, number of open bins, convergence time in cycles and messages sent in millions for the final generation for *Exp 4a*, *Exp 4b* and *Exp 4c*

Exp	Fitness	Open Bins	Convergence Cycles	Messages Sent
Exp 4a	0.952 (0.003)	194.238 (0.548)	6.170 (0.378)	6.158 (0.032)
Exp 4b	0.966 (0.003)	194.121 (0.856)	6.106 (0.429)	6.156 (0.026)
Exp 4c	0.751 (0.002)	195.000 (0.654)	5.687 (0.331)	6.066 (0.065)

individuals changed for each generation. In comparison, the "Open bins by generation" charts show the mean number of bins opened. The "Convergence time by generation" graphs show the mean convergence time in cycles, and the "Messages sent by generation" charts show the mean number of messages sent in millions.

Analysing the graphs in Figure 5.10, one can see the GP making steady progress in finding ever-fitter individuals as the number of open bins decreases to 194.238; a fitness of 1 represents an ideal solution. The convergence time increases slightly with the generations to 6.170, and the number of messages sent increases to 6.158 million. These results are the benchmark for *Exp 4b* and *Exp 4c*. The best BPP strategy found by EGC from all 20 runs had a fitness of 0.961; it was the run with a seed of 2. The number of open bins was 192.3333, the convergence time in cycles was 5.917, and the number of messages sent was 6.157 million.

The aim of *Exp 4b* is to explore if the EGC framework can develop a decentralised strategy which converges as fast as possible. The mean number of open bins is 193.941 with a standard deviation of 5.389, while lower than for *Exp 4a*, it is not statistically significantly lower. The mean convergence time in cycles is 6.106 with a standard deviation of 0.429, which is lower than for *Exp 4a*; however, this is not statistically significant. The best BPP strategy found by EGC in *Exp 4b* from all 20 runs has a fitness of 0.974; it was the run with seed 2. The number of open bins was slightly lower than *Exp 4a* at 192.0833, the convergence time in cycles was 6, which is higher than *Exp 4a*, and the number of messages sent was 6.167 million, again higher than *Exp 4a*.

The aim of *Exp 4c* is to evaluate if the EGC framework can develop a decentralised strategy using fewer messages. The mean number of open bins is 195.000 with a standard deviation of 0.654, which is statistically significantly higher than *Exp 4a*. On the other hand, the mean number of messages sent is 6.066 million with a standard deviation of 0.065, which is statistically significantly lower than for *Exp 4a*, which was 6.158 with a standard deviation of 0.032; the p-value was 2.6782×10^{-6}. The best BPP strategy found by EGC in *Exp 4c* from all 20 runs has a fitness of 0.754; it was the run with seed 8. The number of open bins was higher than *Exp 4a* and *Exp 4b* at 195.1667, and the convergence time in cycles was 5.75, which is lower than *Exp 4a* and *Exp 4b*, and the number of messages sent was 5.910822 million again lower than *Exp 4a* and *Exp 4b*.

In summary, the results of *Exp 4b* do not show a statistically significantly faster convergence time. The author believes this is due to the design of the *SmartBin* strategy with the EGC Framework. At each cycle, agents can check the queue to see if an item is ready to be packed. If there is, and it can fit the item, it embarks on the tender-bid-award cycle. It does not recheck the queue until the next cycle. Thus the *SmartBins* can process at most one item per cycle. Addressing this would require the EGC framework to execute several tender-bid-award cycles per cycle. The results of *Exp 4c* showed a statistically significantly lower number of messages when the messages sent criterion is weighted. Minimising the number of messages is beneficial as distributed algorithms are criticised for requiring large numbers of messages [Khan et al., 2018].

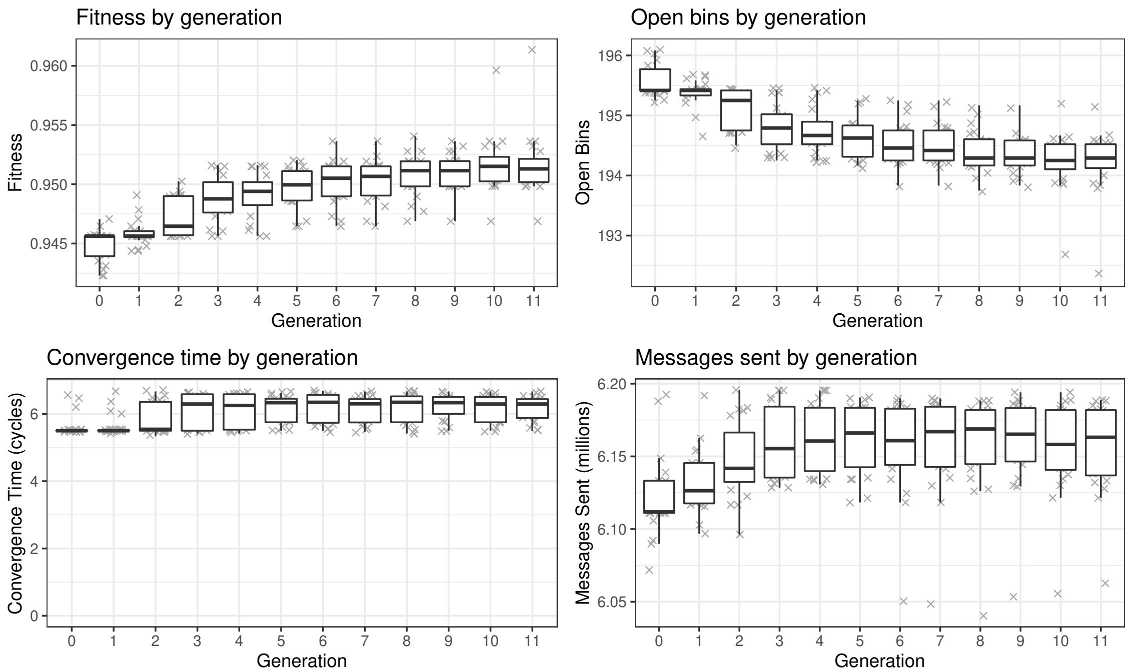


Figure 5.10: The results for *Exp 4a* showing the fitness, number of open bins, convergence time and number of messages sent for each generation.

5.4 Discussion

The experimental results show that EGC has been used to develop a decentralised heuristic to solve the 1D-BPP. And that, given a suitable configuration, the decentralised heuristic was better than the two well-known heuristics, *Best Fit* and *First Fit*. However, these findings cannot be extrapolated to conclude that EGC will be equally successful at tackling a different NP-hard optimisation problem.

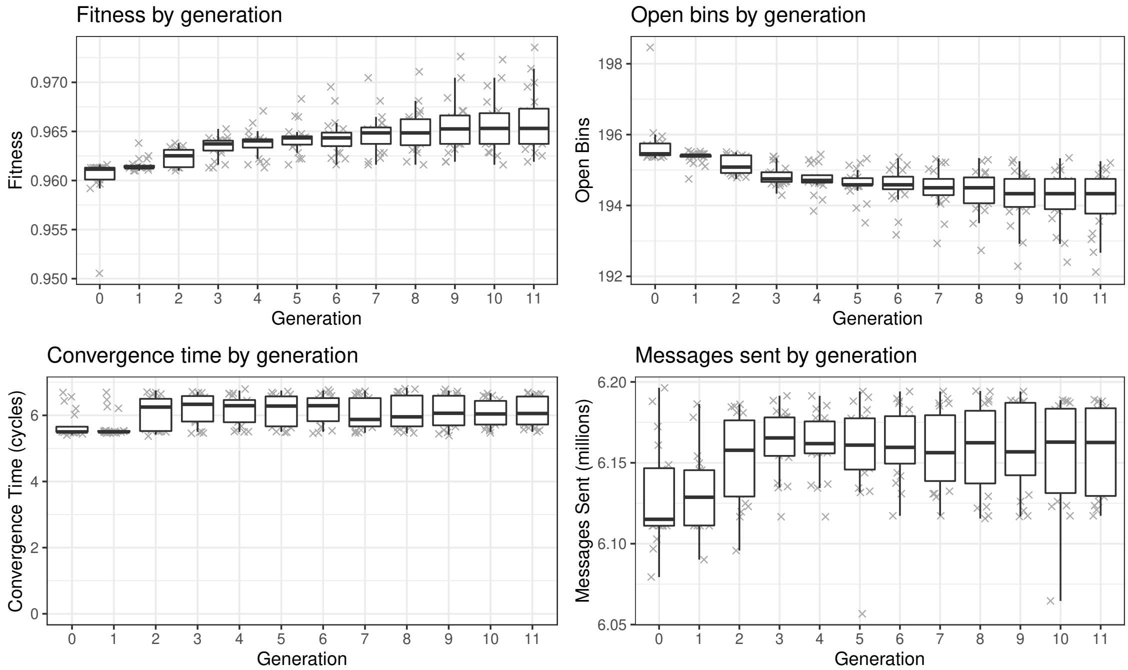


Figure 5.11: The results for *Exp 4b* showing the fitness, number of open bins, convergence time and number of messages sent for each generation.

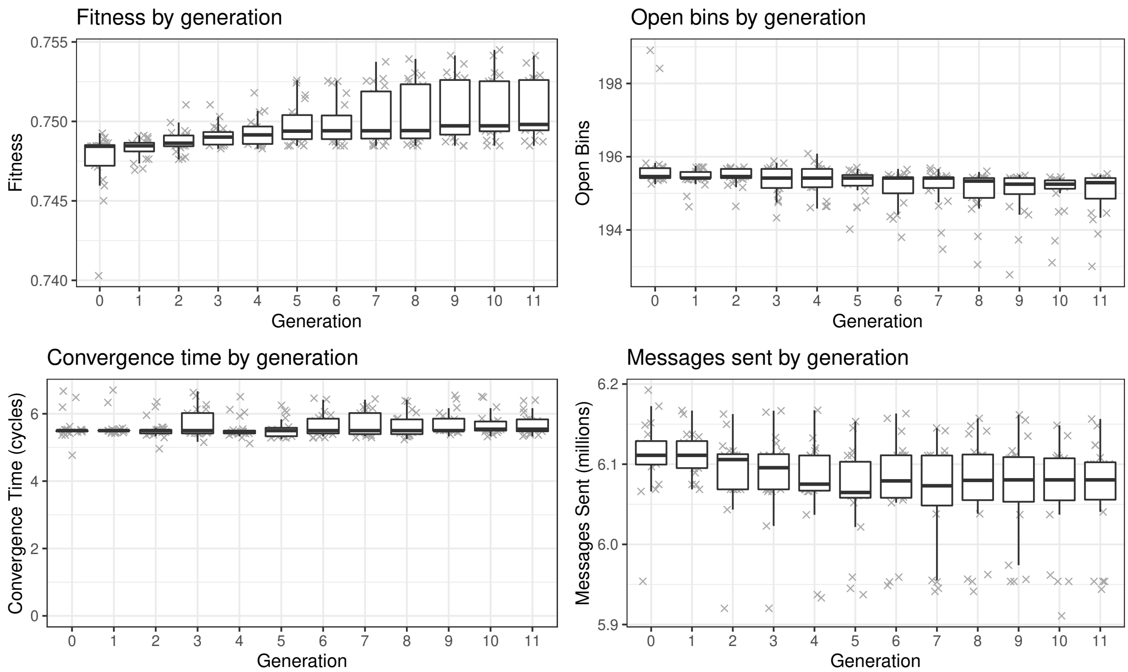


Figure 5.12: The results for *Exp 4c* showing the fitness, number of open bins, convergence time and number of messages sent for each generation.

One area worthy of discussion is the configuration, in particular, the choice of the number of generations, individuals and evaluation executions. The configuration used in the first experiment, *Exp1*, was selected starting with the default configuration of 20 generations, 40 individuals and 20 evaluation executions. Then trial and error adjustments were made to the number of individuals until the results were suitable. However, increasing both the number of generations and individuals may also be a sound strategy.

Exp2 shows that having no generations at all, essentially a random search for effective strategies, performs worse than the configuration in *Exp1*. While *Exp3* shows that the results observed were consistently better with a higher number of individuals than with higher numbers of generations, given a minimum number of generations, 20. This finding is consistent with the findings of Vrajitoru et al., whose research focused on the search for the optimal balance between the population size and the number of generations [Vrajitoru, 2000]. Their general conclusion was that larger populations have a significantly better chance of improving effectiveness.

The amount of evaluations needed to ensure EGC is effective at finding a decentralised strategy depends on many factors. These factors include the grammar's size and the problem space's fitness landscape. While it's easy to infer that a more extensive grammar will require more evaluations, the shape of the fitness landscape is generally not known. Knowing the optimal configuration for EGC for one problem will not inform what it will likely be for another.

Is the evolved *SmartBin* heuristic from $Exp1_{seed=19}$ overfitted to the dataset on which it was trained? When it was evaluated on an unseen dataset, it no longer performed better, suggesting it is. The risk of overfitting reinforces the importance of selecting a representative dataset for the learning phase. If the dataset is representative, the impact of overfitting may be minimised.

For example, consider the virtual machine (VM) allocation problem from cloud computing, which is a multidimensional vector BPP [Chekuri and Khanna, 2004]. The packing dimensions are CPU, memory, disc space and network bandwidth required by the VM as it is packed into an available host. In cloud data centres, there is typically a relatively small number of possible VM types allowed. Thus a heuristic which packed precisely to these VM types would be better, but if the VM types changed, a new heuristic would need to be learned.

5.5 Conclusion

Implications The contribution of this chapter is a novel technique used to semi-automate the development of a decentralised heuristic. This technique is robust enough to solve an NP-hard combinatorial optimisation problem. This contribution is valuable because decentralised agent-based approaches offer significant advantages over centralised methods for certain types of problem domains – rapidly changing systems composed of large numbers of independent modules [Davidsson et al., 2007]. Moreover, several key rising technologies fit these systems, including Edge computing and the Internet of Things [Mascardi and Weyns, 2018, Julian and Botti, 2019]. Therefore, EGC, as a semi-automated approach, can rapidly generate new heuristics to keep pace with the rising demand.

Limitations However, the EGC framework has several limitations. Firstly, its use is limited to certain problems, problems where the system is composed of distinct entities that can be agents in the multi-agent system. Secondly, the problem needs to be modelled in a simulation environment, with test cases representing the real problem, for the GP evaluation of the individuals. There must be a means of evaluating the success of each simulation; this is used to calculate the fitness. Designing a fitness function may be non-trivial and becomes even more so if the problem is a multi-objective problem, as are many real-world problems. And there must be sufficient computing resources available to complete these evaluations a vast number of times, typically 10 to 100 thousand times. Lastly, software engineering effort is needed to implement the GC methods that are not evolved and to determine the grammar and variables for the GC methods that are.

The alternative would be to human-implement all the methods, which requires significant trial and error to ensure the desired result. Of course, either of these options will take longer than simply implementing one of the classical heuristics, *Best Fit* or *First Fit*. But the benefits of the decentralised heuristic, scalability and no single-point-of-failure, are lost.

Recommendations Further research should investigate how EGC might address other optimisation problems, such as a multi-dimensional BPP or the Dynamic Virtual Machine Consolidation (DVMC) problem. Scheduling problems and problems from supply chain and decentralised energy management may also fit within the limitations of EGC.

Once the EGC framework has engineered a GC protocol for a MAS, the heuristic cannot adapt to changes in the problem environment. However, reinforcement learning (RL) could be use to overcome this shortcoming. Thus, the focus of the next chapter is to develop a suitable multi-agent RL algorithm that can learn a communication protocol.

CHAPTER 6

QD(λ) Learning

6.1 Introduction

Reinforcement learning (RL) is a trial-and-error learning framework with delayed rewards [Sutton and Barto, 2018]. It is a nature-inspired technique; influenced by learning in biological systems. With no prior knowledge, an agent learns what action to take next, given the state of the environment at that time, to achieve its goal. Multi-agent Reinforcement Learning (MARL) extends single-agent RL to operate in an environment shared by multiple agents. Researchers commonly use MARL for designing multi-agent systems.

One classification of MARL systems relates to whether the agents are controlled centrally or in a decentralised manner. A MARL system is said to be "fully decentralised" when the agents are fully autonomous, have no central controller, and do not coordinate with one another. Several fully decentralised schemes exist, such as classical Q-learning [Claus and Boutilier, 1998]. One shortcoming of these approaches is that they suffer from non-convergence [Tan, 1993].

Transfer learning is a machine learning technique which leverages previous knowledge to improve the learning speed or performance [Taylor et al., 2007, Da Silva and Costa, 2019]. When applied to MARL, this knowledge can come from another agent. This system represents the MARL classification, where the agents are decentralised but use a communication network to share knowledge. For example, in one value-based method, QD learning [Kar et al., 2012], the update rule for the Q function has a second term corresponding to information from the agent's neighbours.

However, research on MARL takes the view that while "agents engaging in partnership can significantly outperform independent agents ... they may learn slowly in the beginning" [Tan, 1993]. This research aims to develop a MARL algorithm which overcomes this limitation. Additionally, it aspires to design a MARL algorithm suited to a large multi-agent system learning a communication protocol, such as the Gossip Contracts protocol. Research into MARL systems learning communication protocols is an emerging area that has seen notable activity in recent years [Foerster et al., 2016, Sukhbaatar et al., 2016, Jiang and Lu, 2018, Das et al., 2019, Mao et al., 2020, Tung et al., 2021]

- Section 6.2 describes the QD(λ) learning algorithm, which is the main contribution of this chapter. For its evaluation, test reinforcement learning environments were developed, which require the agents to transition through several states before receiving a reward. These environments emulate the environment of an agent learning a protocol, where the agent takes a series of actions, sending and receiving messages, before receiving a reward. Then, dynamic programming (DP) is used to estimate the test environment's actual action-value function (AVF). The accuracy of an RL algorithm can be calculated as the root-mean-squared error (RMSE) from the estimated AVF. While the accumulative root-mean-squared error (ARMSE) measures the algorithm's convergence speed. Lastly, a brief review of the three RL algorithms used in the evaluation explores how their parameters impact their convergence accuracy and speed.

- Section 6.3 presents the two experiments and their results. The first explores the parameters of the QD(λ) learning algorithm and how they impact the convergence speed and quality. The second compares the QD(λ) learning algorithm to three other RL algorithms assessing its effectiveness in different test environments.

- Section 6.4 analyses the experimental evidence and proposes a general hypothesis — the QD(λ) learning algorithm can learn faster than the other RL algorithms for environments where many steps delay the reward. Then, it highlights several considerations, including that the hypothesis has not been proven. Two observations from the experimental results are consistent with previous findings. Firstly, the eligibility traces of Q(λ) learning will offer faster learning when many steps delay the rewards, as in the larger test environments. Secondly, MARL algorithms which share information across agents may initially learn more slowly.

6.2 Methodology

This section describes the QD(λ) learning algorithm, which is the main contribution of the chapter. Following this, it presents the reinforcement learning environments developed for the RL algorithms' evaluation. The environment's true action-value function (action-value function) is estimated using dynamic programming (DP). The effectiveness of an RL algorithm is evaluated by calculating the difference, or error, between the AVF it learns and the DP-estimated AVF. The root-mean-squared error (RMSE) compares the accuracy of the RL algorithm once converged, while the accumulative root-mean-squared error (ARMSE) compares the algorithm's convergence speed. Finally, a brief review of the three RL algorithms used in the evaluation explores how their parameters impact their convergence accuracy and speed.

6.2.1 QD(λ) Learning

The QD(λ) learning algorithm is a novel hybrid of the RL Q(λ) learning algorithm [Watkins, 1989], and the MARL QD learning algorithm [Kar et al., 2012]. These algorithms are described in detail in Section 2.5 and Section 2.5.1, respectively. The QD(λ) learning algorithm combines the eligibility traces of the Q (λ) learning algorithm with the consensus term — information from the agents' neighbourhoods — from QD learning, as shown in Figure 6.1.

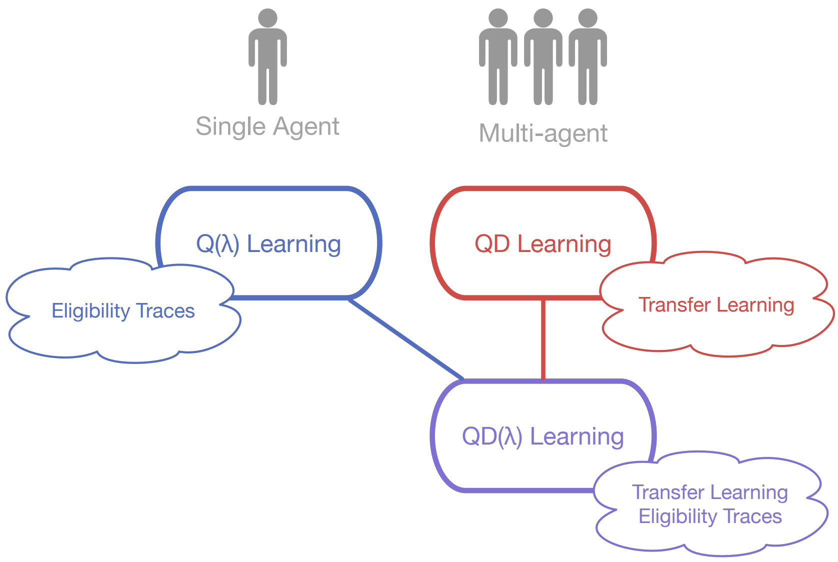


Figure 6.1: The relationship between the Q(λ) learning, QD learning and QD(λ) learning algorithms.

The parameters in QD(λ) learning, described in Table 6.1, are a combination of those from the Q(λ) learning and QD learning algorithms. The discount rate, γ, determines the present value of future rewards. While the innovation step size, α, which is equivalent to the step size in the Q learning algorithm, controls the rate at which an algorithm converges to a solution. The higher the innovation step size, the faster it moves towards convergence, but the less accurate the final converged value. The consensus step size, β, controls the weight of learning information received from the agent's neighbourhood. While the number of neighbours, N, is the number of agents from whom the agent receives learning information. Lastly, The eligibility decay, λ, controls the relative updates to recently visited state actions.

Table 6.1: The parameters of QD(λ) learning algorithms.

Parameter		Description
Discount factor	γ	Present value of future reward
Innovation step size	α	Rate at which an algorithm converges to a solution
Consensus step size	β	Weight of learning information received from the agent's neighbourhood
Number of neighbours	N	Number of neighbours from whom the agent receives learning information
Eligibility decay	λ	Relative updates to recently visited state actions

The QD(λ) learning algorithm is shown in Algorithm 11. At the initialisation of the learning process, the algorithm sets the AVF, $Q(s,a)$, and the eligibility traces, $e(s,a)$, to zero and randomly initialises the ϵ-greedy policy, π. Next, it sets the initial state, s, and the initial action, a. Then, it randomly connects each agent to N other agents. The agents use this network to transfer learning information, specifically the $Q(s,a)$ values that comprise the consensus term of the update rule. The body of the algorithm is repeated for each learning step.

First, the algorithm takes action a and transitions to state s'. It gets the reward, r, for the transition (s,a,s'). Then, it chooses the next action, a', from the policy, where the set of all possible actions for the state is $A(s')$. The algorithm will differ when it takes a non-optimal action — an exploratory action with a lower value in the AVF than the policy's greedy action. It records the policy's greedy action, a^*, to ascertain this later. However, when a' has the same values in the AVF as a^*, it sets a^* as the action to be taken.

Then, the algorithm sets the eligibility trace for the previous state action, $e(s,a)$, to one. The AVF update is a combination of the *innovation* term, $\alpha\big[r+\gamma Q(s',a^*)-Q(s,a)\big]$ and the *consensus* term, $\beta\sum_{j=N}\big[Q(s,a)-Q^j(s,a)\big]$. For every state action in the AVF with an eligibility trace greater than zero, the value is updated with the eligibility trace for the state actions scaling the update. Equation 6.1 shows the full update rule.

Algorithm 11: QD(λ) learning

Algorithm Parameter: discount $\gamma \in [0, 1]$
Algorithm Parameter: innovation step size $\alpha > 0$
Algorithm Parameter: consensus step size $\beta > 0$
Algorithm Parameter: eligibity trace decay rate $\lambda \in [0, 1]$

Initialise: $Q(s, a) = 0$, $e(s, a) = 0$ for all $s \in S+$, $a \in A(s)$
Initialise: ϵ-greedy policy π randomly
Initialise: s to initial state
Initialise: a to initial action
Initialise: connect each agent to N other agents randomly

foreach *learning step* **do**

  Take action a and transition to state s'
  $r :=$ get reward for transition (s, a, s')

  Choose a' from $A(s')$ using π
  $a^* := argmax_a\ Q(s', a)$ if a' ties for max then $a^* \leftarrow a'$

  $\delta_i := \alpha\big[r + \gamma Q(s', a^*) - Q(s, a)\big]$ `// Innovation Term`
  $\delta_c := \beta \sum_{j=N} \big[Q(s, a) - Q^j(s, a)\big]$ `// Consensus Term`

  $e(s, a) := 1$
  for *all* (s, a) *where* $e(s, a) > 0$ **do**
    $Q(s, a) := Q(s, a) + e(s, a)\big[\delta_i - \delta_c\big]$
  end

  for *all* (s, a) **do**
    if $a' = a^*$ **then**
      $e(s, a) := \gamma\lambda e(s, a)$ `// Decay eligibilities`
    else
      $e(s, a) := 0$ `// Took exploratory action`
    end
  end

  Improve π greedily with respect to Q
  $s := s'$ and $a := a'$
end

$$Q^i(s,a) \leftarrow Q^i(s,a) + e^i(s,a)\Bigg[\underbrace{\alpha\left[r + \gamma \max_a Q^i(s',a) - Q^i(s,a)\right]}_{\text{innovation term}} \\ \underbrace{- \beta \sum_{j \in = N^i} \left[Q^i(s,a) - Q^j(s,a)\right]}_{\text{consensus term}}\Bigg] \tag{6.1}$$

Under certain conditions, the QD(λ) update rule collapses to the other three RL algorithms' update rules. For example, if the consensus stepsize parameter is zero, $\beta = 0$, the update rule is equivalent to that of Q(λ) learning. Or, if the eligibility decay is zero, $\lambda = 0$, the update rule is equivalent to that of QD learning. And, should both the consensus stepsize parameter and eligibility decay be zero, then the update rule is equivalent to that of Q learning.

After the algorithm updates the AVF, it updates the eligibilities. When it takes an exploratory step, it resets all the eligibility traces to zero. Otherwise, when $a' = a^*$, it decays the eligibilities for all state actions. It does this, as with the Q(λ) learning algorithm, because steps before the exploratory steps no longer have the required relationship to the greedy policy. Finally, the algorithm improves the greedy policy using the newly-updated action-value function. It sets the state-action, (s,a), to the new state-action, (s',a'), and continues to the next learning step.

6.2.2 Reinforcement learning Test Environments

The experiments that evaluate the learning algorithms use randomly generated reinforcement learning environments. They emulate the environment of an agent learning a protocol, where the agent takes a series of actions, such as constructing, sending, receiving or processing messages, before obtaining a reward. The sets of environments vary in the number of states used; they have 4, 8, 16 and 32 states, each having two actions. There are no terminal states, so the experiments are continuing tasks. The reward for all states is zero, except for one, selected randomly, which returns a reward of one.

From any state action, the probability of transitioning to any other state is non-zero. These state transitions are generated randomly, but for one state, the transition probability is increased fifty-fold. Then the probabilities are normalised. This fifty-fold increase creates an environment where the agent has to pass through a series of states in order to get a reward. Figure 6.2 illustrates this journey. It shows a heatmap of the transition probabilities for an environment with eight states and two actions in each

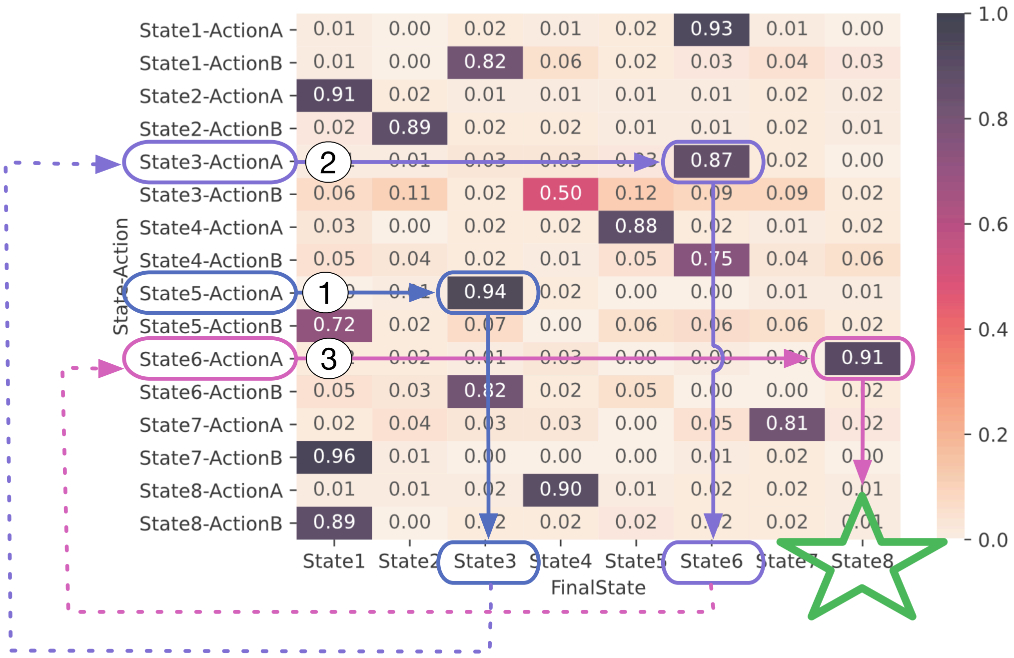


Figure 6.2: Heatmap of the transition function probabilities for the environment with eight states and seed set to zero.

state. It is the transition function generated for $seed = 0$. In Figure 6.2, the reward state was *State 8*, as indicated with the green star. Overlaid on the heatmap is an example of a route from the starting state, *State 5*. The agent takes *Action A*, which transitions the environment to *State 3*. Then, the agent retakes *Action A*, which transitions the environment to *State 6*. Lastly, the agent takes *Action A* for a third time, and the environment transitions to *State 8*, and the agent receives the reward. As this is a continuous task, the agent carries on taking actions until the experiment ends.

Dynamic programming (DP) [Bellman, 1966] can generate estimates of the true AVF when an environment's state transition function and reward function are known. Figure 6.3 shows the AVF calculated using DP for an environment with eight states and $seed = 0$. The DP algorithm has two parameters: a discount factor, γ, and an allowed difference threshold, θ. The experiments set these to 0.9 and 0.001, respectively.

The evaluation of an RL algorithm compares the algorithm's AVF with the estimate of the AVF generated using DP. The root-mean-squared error (RMSE) is the root of the mean of the squared errors of each state-action value for each agent. Figure 6.4 shows *Agent 0* state action's value and squared error for the environment with eight states and a seed of zero.

A second measure, the accumulative root-mean-squared error (ARMSE), measures the convergence speed of an RL algorithm. It is the accumulation of the RMSE at each episode. Figure 6.5 shows the AVF RMSE and ARMSE for an RL algorithm. The RL

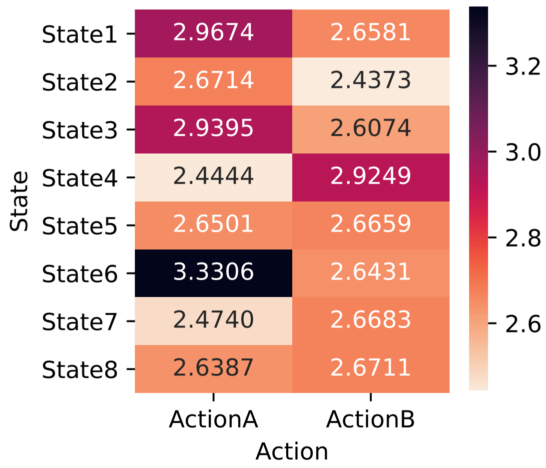


Figure 6.3: Heatmap of the action value function calculated using dynamic programming for the environment with eight states and seed set to zero.

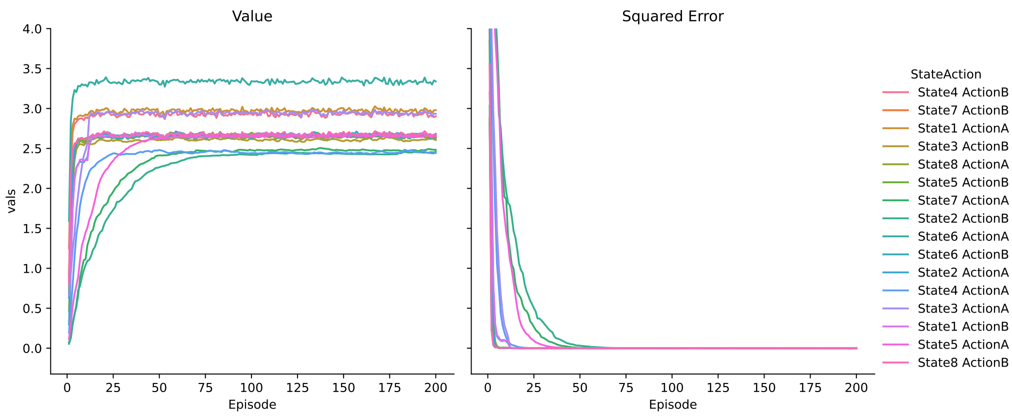


Figure 6.4: The value and squared error of each state action for *Agent 0* for the environment with eight states and a seed of zero.

algorithm RMSE falls rapidly before converging, while the ARMSE rises rapidly before steadying to grow at a constant rate. RMSE and ARMSE are used in the next section to evaluate and compare the RL algorithms.

6.2.3 Reinforcement Learning Algorithms

This subsection presents a brief review of the three benchmark RL algorithms used in the evaluation of the QD(λ) learning. The benchmark algorithm are Q learning, Q(λ) learning and QD learning. It explores how their parameters impact their convergence accuracy and speed.

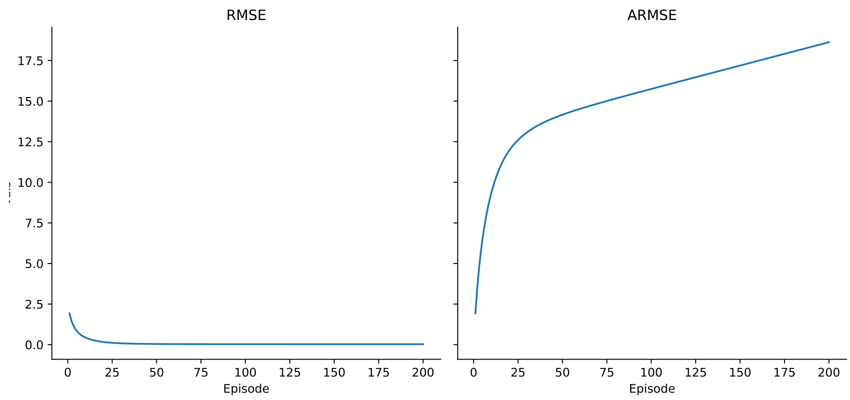


Figure 6.5: The action-value function root-mean-squared error (RMSE) and accumulated root-mean-squared error (ARMSE) for an RL algorithm.

Q Learning The Q learning algorithm is a temporal difference algorithm that updates its Q values by learning from experience and rewards and estimating the value of the next state. Equation 6.2 shows the update rule for the AVF in the Q learning algorithm.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a} Q(s',a) - Q(s,a) \right] \tag{6.2}$$

The learning rate, α, such that $\alpha > 0$, is the step-size parameter of the Q learning algorithm. It weights the newly learned value relative to its original value. It controls the rate at which the algorithm converges. Figure 6.6 shows the median AVF RMSE across 40 random seeds of the Q learning algorithm for environments with eight states. It highlights the impact of the step size parameter on the convergence speed and value. A larger step size, s, causes the algorithm to converge faster but with less accuracy.

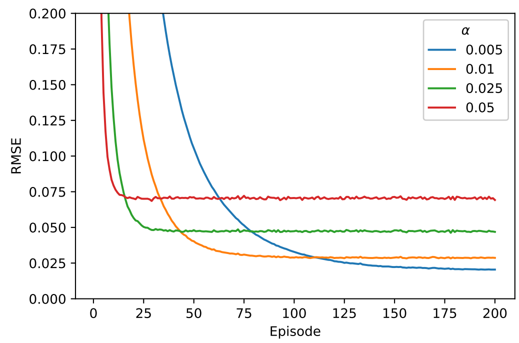


Figure 6.6: The action-value function root-mean-squared error (RMSE) of the Q learning algorithm for the environment with eight states highlighting the impact of the step size parameter, α, on the convergence speed and value

Q(λ) Learning The Watkins Q(λ) learning algorithm is an extension to his earlier Q learning algorithm that incorporates the concept of eligibility traces [Watkins, 1989]. In Q(λ) learning, when the algorithm updates a state-action's Q value, it also updates other recently visited state-actions. As a result, Q(λ) learning may learn more efficiently. One parameter of the Q(λ) learning algorithm is the eligibility decay, λ, which controls how quickly the update of the visited state actions decays. The higher the eligibility decay, the slower the update diminishes. Q(λ) learning learns faster than Q Learning.

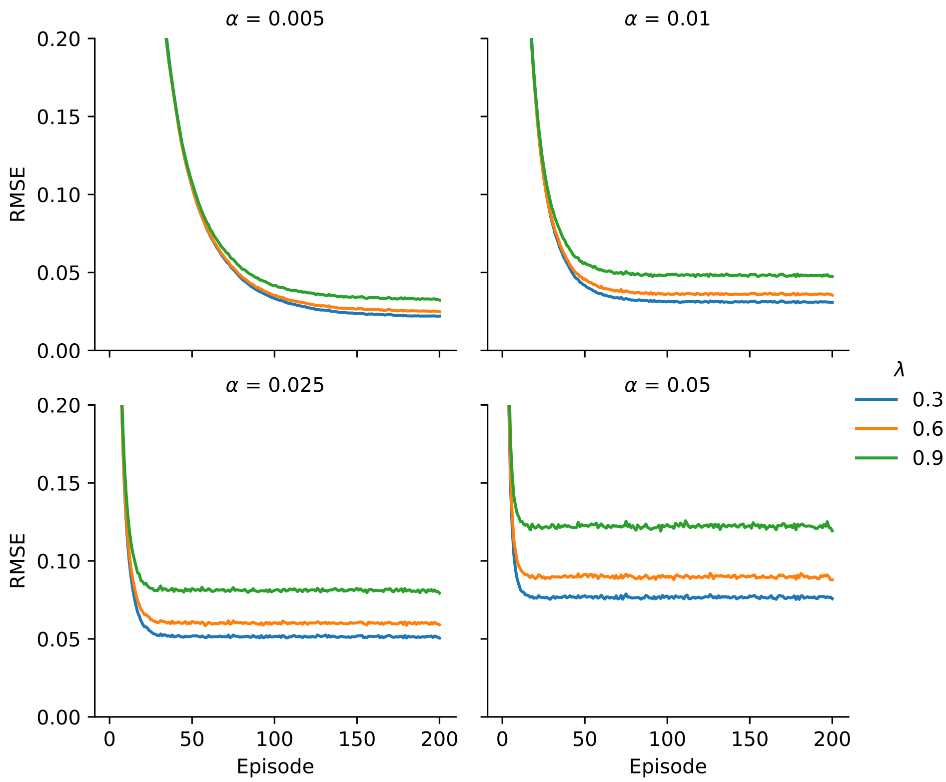


Figure 6.7: The action-value function root-mean-squared error (RMSE) of the Q(λ) learning algorithm for the environment with eight states highlighting the impact of the step size parameter, α, and the eligibility decay parameter, λ, on the convergence speed and value

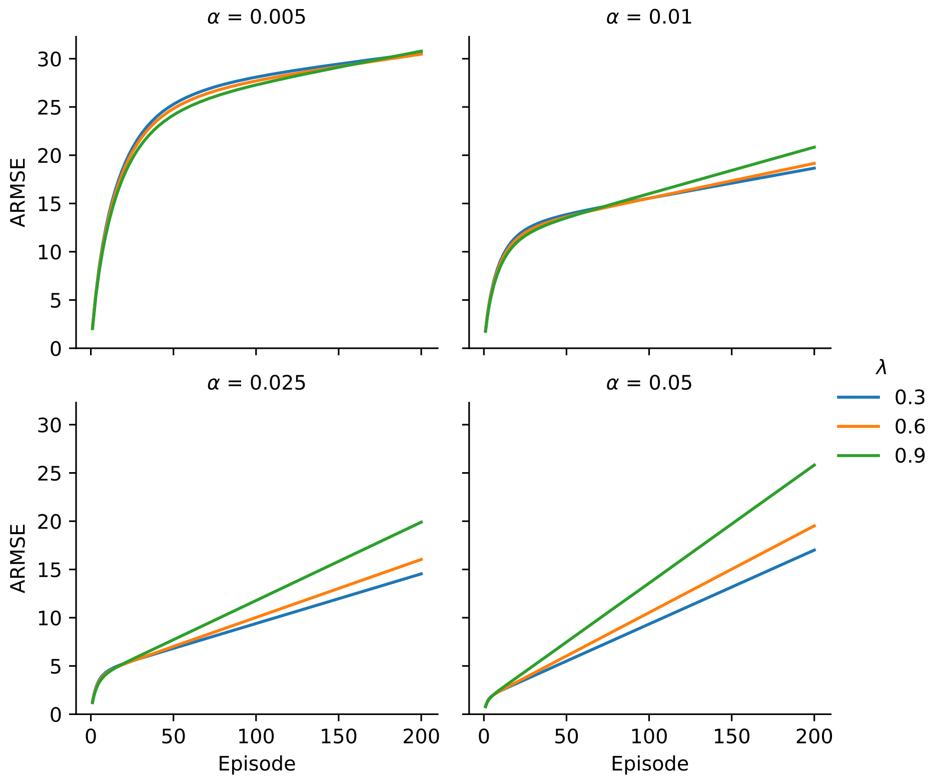


Figure 6.8: The action-value function accumulative root-mean-squared error (ARMSE) of the Q(λ) learning algorithm for the environment with eight states highlighting the impact of the step size parameter, α, and the eligibility decay parameter, λ, on the convergence speed and value

Figure 6.7 and Figure 6.8 show the AVF RMSE and the ARMSE of the Q(λ) learning algorithm for environments with eight states highlighting the impact of the step size parameter, α, and the eligibility decay parameter, λ, on the convergence speed and value. As with the Q learning algorithm, a larger step size, α, causes the algorithm to converge faster but with less accuracy. A higher eligibility decay, λ, also causes the algorithm to converge quicker but with less accuracy; this faster convergence is more clearly seen in Figure 6.8.

QD Learning The QD learning algorithm extends the Q learning algorithm to the multi-agent field where the agents communicate and share information [Kar et al., 2012]. Thus, the update rule for the AVF has one more term than the Q learning update rule. The Q learning term is referred to as the *innovation* term, and the new QD learning term is the *consensus* term; the *consensus* term corresponds to information from the agent's neighbours. Equation 6.3 shows the AVF update rule for *Agent* i. N is the set of agents in the *Agent* i*'s* neighbourhood. The *consensus* term has a new learning rate,

β, where $\beta > 0$.

$$Q^i(s,a) \leftarrow Q^i(s,a) + \underbrace{\alpha \left[r + \gamma \max_a Q^i(s',a) - Q^i(s,a) \right]}_{\text{innovation term}} \underbrace{- \beta \sum_{j \in = N^i} \left[Q^i(s,a) - Q^j(s,a) \right]}_{\text{consensus term}} \tag{6.3}$$

Figure 6.9 and Figure 6.10 shows the AVF RMSE of the QD learning algorithm for the environment with eight states highlighting the impact of the innovation step size parameter, α, and the consensus step size parameter, β. As with the previous learning algorithm, a larger innovation step size causes the algorithm to converge faster but with less accuracy. Conversely, a larger consensus step size causes the algorithm to converge both faster and more accurately. This improvement is more pronounced when the innovation step size is larger.

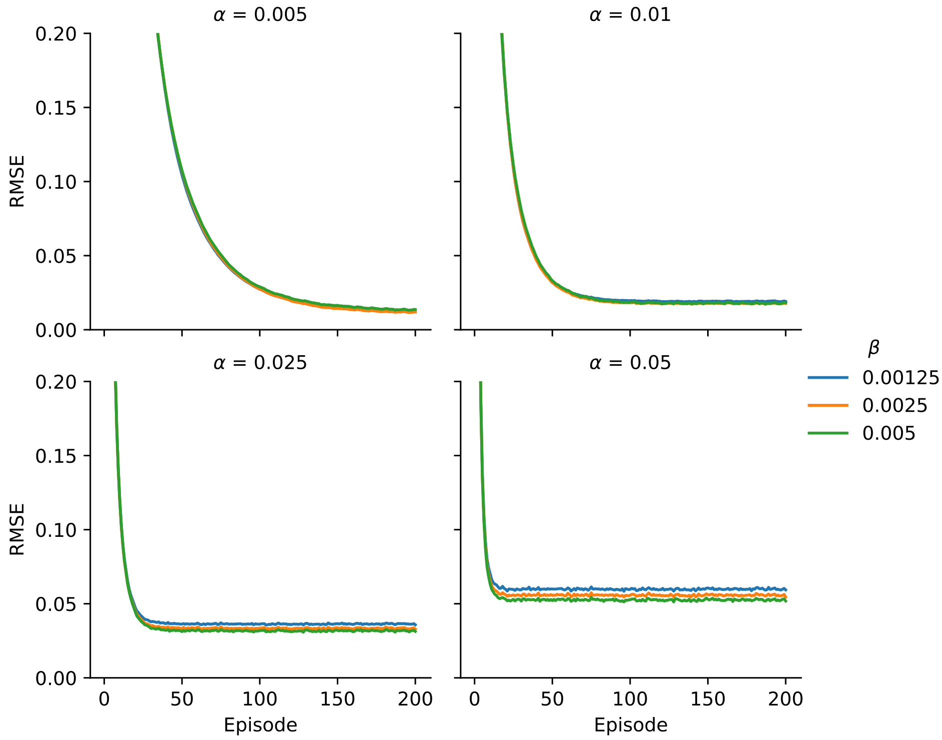


Figure 6.9: The action-value function root-mean-squared error (RMSE) of the QD learning algorithm for the environment with eight states highlighting the impact of the innovation step size parameter, α, and the consensus step size parameter, β, on the convergence speed and value.

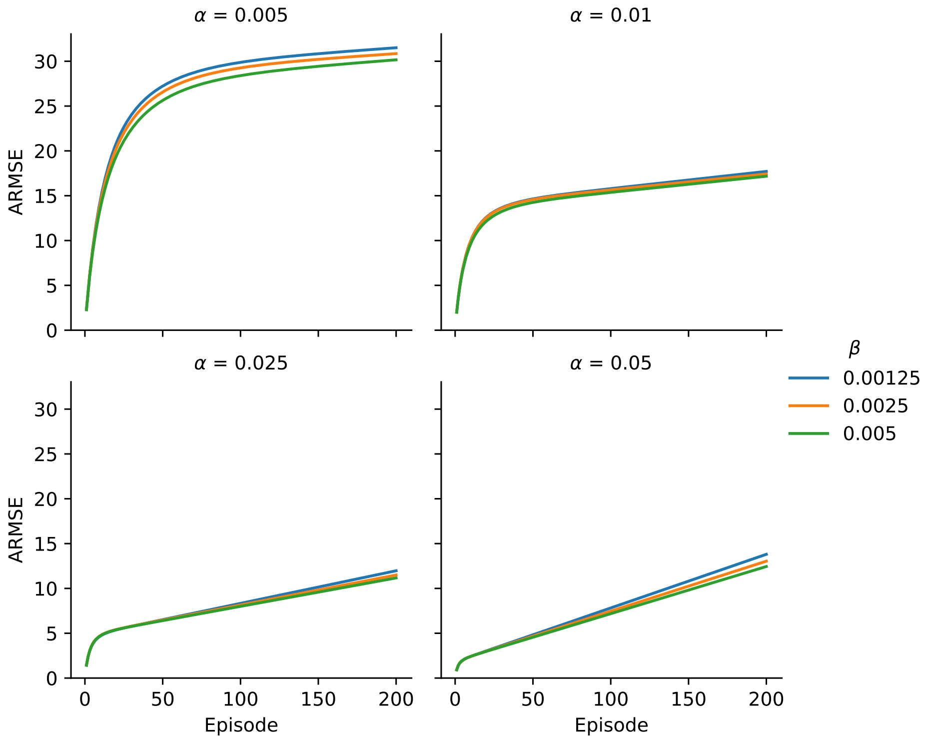


Figure 6.10: The action-value function accumulative root-mean-squared error (ARMSE) of the QD learning algorithm for the environment with eight states highlighting the impact of the innovation step size parameter, α, and the consensus step size parameter, β, on the convergence speed and value.

6.3 Experimental Results

This section presents the experimental setup and the RL algorithms' parameters and the two experiments. The first experiment explores the parameters of the QD(λ) learning algorithm and how they impact the convergence speed and quality. The second compares the QD(λ) learning algorithm to three other RL algorithms assessing its effectiveness in different test environments.

6.3.1 Experimental Setup

The experiments were multi-agent systems with 50 agents. The agents had a soft policy with the probability of taking a random, rather than greedy, action, ϵ, set to 0.1. Each agent took an action once per cycle. Every episode, 10,000 cycles, the experiments recorded the state action values and their error from the estimated AVF; note the problem was not reset to an initial state at the start of an episode. Each experiment comprised

200 episodes. Table 6.2 summarises this experimental configuration.

Table 6.2: The configuration used for the experiments.

Parameter	Value
Number of Agents, N	50
Probability of taking a random action, ϵ	0.1
Learning cycles per episode	10,000
Number of episodes	200

RL algorithms are configured by one or more parameters. Changes to the parameters will impact the performance of a learning algorithm in an environment. There is no optimal configuration of the parameters for all environments, though there is an optimal configuration for any one environment. Thus, the experiments evaluated the RL algorithms across several different configurations, as shown in Table 6.3. Note that every combination of the parameters listed is evaluated when there is more than one parameter. For example, for the Q(λ) learning, there were 12 different parameter combinations evaluated.

Table 6.3: The parameters used for each learning algorithm.

Algorithm	Parameter	Values
	Discount factor, γ	0.9
Q Learning	Step size, α	0.05, 0.025, 0.01, 0.005
Q(λ) Learning	Step size, α	0.05, 0.025, 0.01, 0.005
	Eligibility decay, λ	0.3, 0.6, 0.9
QD Learning	Innovation step size, α	0.05, 0.025, 0.01, 0.005
	Consensus step size, β	0.005, 0.0025, 0.00125
QD(λ) Learning	Innovation step size, α	0.05, 0.025, 0.01, 0.005
	Consensus step size, β	0.005, 0.0025, 0.00125
	Eligibility decay, λ	0.3, 0.6, 0.9

6.3.2 Experiments

Each experiment was repeated 40 times with different random seeds. A Shapiro–Wilk test of normality [Shapiro and Wilk, 1965] showed that the results did not follow a normal distribution; thus, non-parametric statistical measures are used. For example, the results of two different RL algorithms were compared using a two-sample Wilcoxon Signed test [Wilcoxon, 1992].

Exp1 The first experiment explores the impact of the parameters of the QD(λ) learning algorithm. It compares the 36 configurations of QD(λ) learning listed in Table 6.3 in an environment with eight states. Figure 6.11 shows the AVF RMSE for the 36 configurations of the QD(λ) learning algorithm. Table 6.4 shows the AVF RMSE for episodes 50 and 200.

The configuration of $\alpha = 0.005, \beta = 0.0025, \lambda = 0.3$ was the best configuration at the 200th episode. The statistical significance of the difference between a configuration and the best configuration is indicated by the p-values. The difference was statistically significant for all configurations except for $\alpha = 0.005, \beta = 0.0005, \lambda = 0.3$.

Exp2 The second experiment evaluates the performance of the QD(λ) learning algorithm against three well-known value-based learning algorithms. Two of these algorithms, Q learning and Q(λ) learning, employ the fully decentralised scheme, while one, QD learning, uses the networked agent's scheme. This experiment uses four different environments with 4, 8, 16 and 32 states.

The experiment compares the four learning algorithms with all the configurations listed in Table 6.3 in all four environments, repeated 40 times. Figure 6.12 shows the best configurations of each of the four learning algorithms for the different sizes of environments, while Table 6.5 shows the median AVF RMSE across the 40 random seeds for episodes 50 and 200 for the best configurations. The p-values indicate the statistical significance of the difference between an algorithm and the best algorithm.

The results show that QD learning is best across all four environments in the 200th episode, and the differences are statistically significant. The best configuration of the QD learning algorithm differs across the four environments. However, QD(λ) learning is best across all four environments in the 50th episode, and again, the differences are statistically significant. This result supports the hypothesis that the QD(λ) learning algorithm learns faster than the other three RL algorithms.

Further support of this hypothesis is evident in Table 6.6 and Figure 6.13, which show the median AVF accumulative RMSE for the best configurations of each of the four learning algorithms for the four different types of environments. These results indicate that QD(λ) learning has the lowest accumulated RMSE in the 200th episode for environments with eight or more states; the differences are statistically significant. From this, one can hypothesise that for larger state spaces, while QD learning learns a more accurate AVF, QD(λ), learning learns faster. For the small environment with four states, the QD(λ) learning algorithm is best at episode 200; but this difference was only significant compared to the Q learning algorithm.

Effectively, an RL algorithm aims not to learn the AVF but rather to learn the optimal

Table 6.4: The action-value function median root-mean-squared error (RMSE) across the 40 random seeds for episodes 50 and 200 for the QD(λ) learning algorithm for the 36 configurations listed in Table 6.3

			Episode = 50		Episode = 200	
α	β	λ	RMSE	p-value	RMSE	p-value
0.005	0.00125	0.3	0.0433	$< 10^{-4}$	0.0093	0.1039
0.005	0.00125	0.6	0.0440	$< 10^{-4}$	0.0107	0.0129
0.005	0.00125	0.9	0.0447	$< 10^{-4}$	0.0160	0.0002
0.005	0.0025	0.3	0.0420	$< 10^{-4}$	0.0077	0.8931
0.005	0.0025	0.6	0.0443	$< 10^{-4}$	0.0087	0.0275
0.005	0.0025	0.9	0.0421	$< 10^{-4}$	0.0136	0.0025
0.005	0.005	0.3	0.0446	-	0.0074	-
0.005	0.005	0.6	0.0436	$< 10^{-4}$	0.0081	0.0099
0.005	0.005	0.9	0.0453	$< 10^{-4}$	0.0154	0.0006
0.01	0.00125	0.3	0.0211	$< 10^{-4}$	0.0157	$< 10^{-4}$
0.01	0.00125	0.6	0.0226	$< 10^{-4}$	0.0188	$< 10^{-4}$
0.01	0.00125	0.9	0.0309	$< 10^{-4}$	0.0283	$< 10^{-4}$
0.01	0.0025	0.3	0.0187	$< 10^{-4}$	0.0132	$< 10^{-4}$
0.01	0.0025	0.6	0.0201	$< 10^{-4}$	0.0161	$< 10^{-4}$
0.01	0.0025	0.9	0.0256	$< 10^{-4}$	0.0242	$< 10^{-4}$
0.01	0.005	0.3	0.0184	$< 10^{-4}$	0.0112	$< 10^{-4}$
0.01	0.005	0.6	0.0189	$< 10^{-4}$	0.0125	$< 10^{-4}$
0.01	0.005	0.9	0.0252	$< 10^{-4}$	0.0221	$< 10^{-4}$
0.025	0.00125	0.3	0.0325	$< 10^{-4}$	0.0329	$< 10^{-4}$
0.025	0.00125	0.6	0.0402	$< 10^{-4}$	0.0393	$< 10^{-4}$
0.025	0.00125	0.9	0.0574	$< 10^{-4}$	0.0571	$< 10^{-4}$
0.025	0.0025	0.3	0.0286	$< 10^{-4}$	0.0283	$< 10^{-4}$
0.025	0.0025	0.6	0.0351	$< 10^{-4}$	0.0343	$< 10^{-4}$
0.025	0.0025	0.9	0.0519	$< 10^{-4}$	0.0513	$< 10^{-4}$
0.025	0.005	0.3	0.0245	$< 10^{-4}$	0.0239	$< 10^{-4}$
0.025	0.005	0.6	0.0296	$< 10^{-4}$	0.0310	$< 10^{-4}$
0.025	0.005	0.9	0.0448	$< 10^{-4}$	0.0470	$< 10^{-4}$
0.05	0.00125	0.3	0.0558	$< 10^{-4}$	0.0555	$< 10^{-4}$
0.05	0.00125	0.6	0.0684	$< 10^{-4}$	0.0691	$< 10^{-4}$
0.05	0.00125	0.9	0.0952	$< 10^{-4}$	0.0969	$< 10^{-4}$
0.05	0.0025	0.3	0.0493	$< 10^{-4}$	0.0488	$< 10^{-4}$
0.05	0.0025	0.6	0.0598	$< 10^{-4}$	0.0631	$< 10^{-4}$
0.05	0.0025	0.9	0.0861	$< 10^{-4}$	0.0933	$< 10^{-4}$
0.05	0.005	0.3	0.0424	$< 10^{-4}$	0.0420	$< 10^{-4}$
0.05	0.005	0.6	0.0541	$< 10^{-4}$	0.0570	$< 10^{-4}$
0.05	0.005	0.9	0.0781	$< 10^{-4}$	0.0801	$< 10^{-4}$

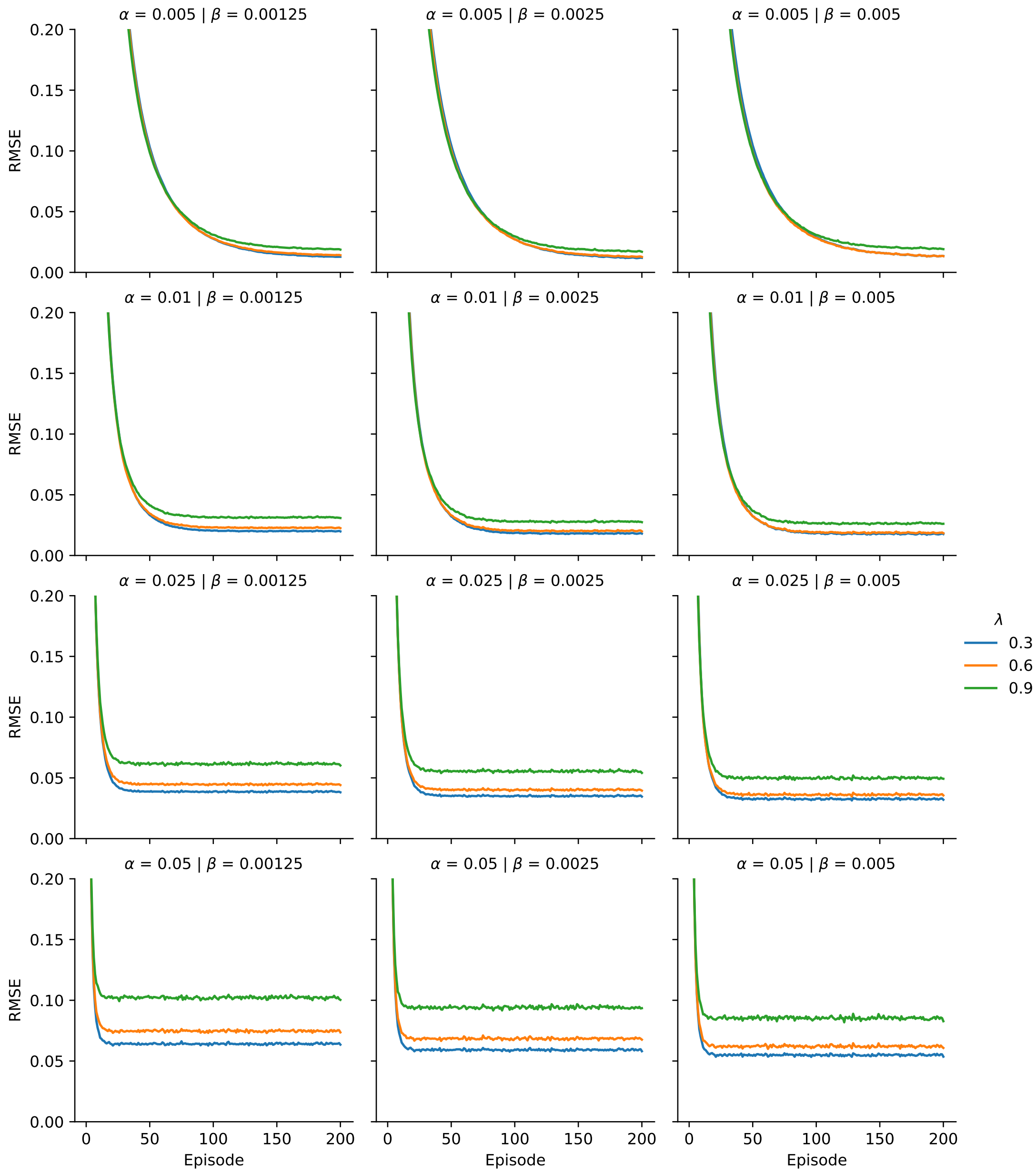


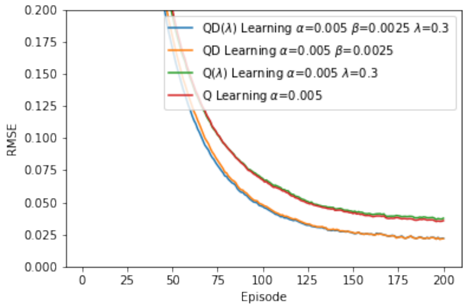
Figure 6.11: The action-value function root-mean-squared error (RMSE) for the 36 configurations of the QD(λ) learning algorithm as listed in Table 6.3 highlighting the impact of the innovation step size parameter, α, the consensus step size parameter, β, and the eligibility decay, λ, on the convergence speed and value.

policy. The estimated AVF generated using DP can be used to create a policy to evaluate the performance of an RL algorithm in learning the optimal policy. With this policy, the number of differences between the policy learned by the RL algorithm and the estimated optimal policy generated using DP is accumulated. As the number of differences is always positive, the evaluation used the mean error (ME) rather than the root-mean-squared error. The mean error is the average differences across the agents' policies. The accumulative mean error (AME) is the accumulated mean error for an RL algorithm across the episodes.

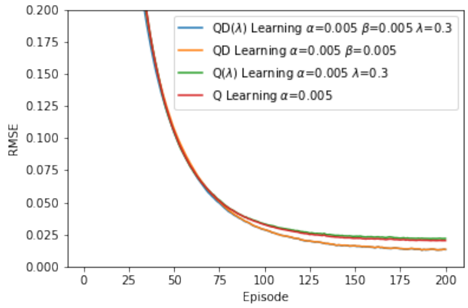
Table 6.7 and Figure 6.14 show the median accumulative mean error (AME) across 40 random seeds for the best configurations of each of the four learning algorithms for the four different types of environments. The results show that QD(λ) learning got the lowest Policy AME at the 200th episode for the three environments where the number of states is eight or more. But, this difference is only statistically significant for the environments with 32 states. QD(λ) learning had the lowest AME at episode 50 for all four environments, and the differences are statistically significant for the environments with 16 and 32 states.

Table 6.5: The action-value function median root-mean-squared error (RMSE) across 40 random seeds for episodes 50 and 200 for the best configurations of the four learning algorithms for four different types of environments

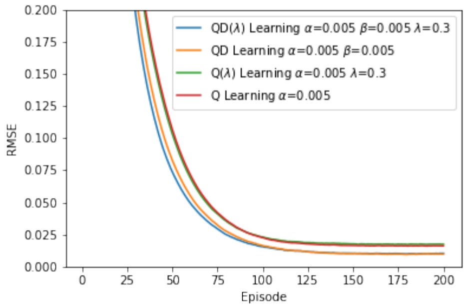
		Episode = 50		Episode = 200	
Number of states = 4	Configuration	RMSE	p-value	RMSE	p-value
QD(λ) Learning	α=0.005 β=0.0025 λ=0.3	0.0615	-	0.0143	-
QD Learning	α=0.005 β=0.0025	0.0587	$< 10^{-4}$	0.0153	0.3750
Q Learning	α=0.005	0.1037	$< 10^{-4}$	0.0269	$< 10^{-4}$
Q(λ) Learning	α=0.005 λ=0.3	0.1070	$< 10^{-4}$	0.0279	$< 10^{-4}$
Number of states = 8	Configuration	RMSE	p-value	RMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	0.0446	$< 10^{-4}$	0.0074	0.0139
QD Learning	α=0.005 β=0.005	0.0461	-	0.0070	-
Q Learning	α=0.005	0.0524	$< 10^{-4}$	0.0159	$< 10^{-4}$
Q(λ) Learning	α=0.005 λ=0.3	0.0534	$< 10^{-4}$	0.0173	$< 10^{-4}$
Number of states = 16	Configuration	RMSE	p-value	RMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	0.0553	$< 10^{-4}$	0.0064	0.0011
QD Learning	α=0.005 β=0.005	0.0594	-	0.0059	-
Q(λ) Learning	α=0.005 λ=0.3	0.0808	$< 10^{-4}$	0.0151	$< 10^{-4}$
Q Learning	α=0.005	0.0956	$< 10^{-4}$	0.0135	$< 10^{-4}$
Number of states = 32	Configuration	RMSE	p-value	RMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	0.0996	$< 10^{-4}$	0.0049	0.0315
QD Learning	α=0.005 β=0.005	0.1111	-	0.0048	-
Q(λ) Learning	α=0.005 λ=0.3	0.1398	$< 10^{-4}$	0.0116	$< 10^{-4}$
Q Learning	α=0.005	0.1498	$< 10^{-4}$	0.0110	$< 10^{-4}$



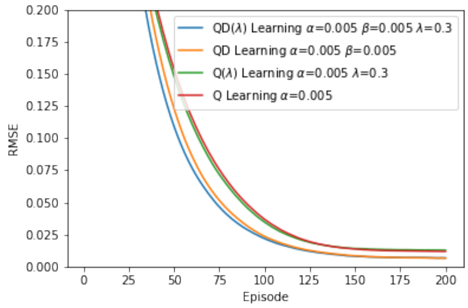
(a) Number of states = 4



(b) Number of states = 8



(c) Number of states = 16



(d) Number of states = 32

Figure 6.12: The action-value function root-mean-squared error (RMSE) for the best configurations of each of the four learning algorithms for four different types of environments

6.4 Discussion

The main contribution of this chapter is the QD(λ) learning algorithm, a value-based MARL algorithm that leverages networked agents. It is a hybrid of the RL Q(λ) learning algorithm [Watkins, 1989], and the MARL QD learning algorithm [Kar et al., 2012]. It combines the advantages of transfer learning, from QD learning, with faster learning in environments where rewards are delayed, from Q(λ) learning.

The experiments did not show that the QD(λ) learning algorithm has the lowest overall median AVF root-mean-squared error (RMSE). Instead, they showed that QD learning has a lower RMSE and that the difference is statistically significant when the environment has 16 or more states. However, they did show that the QD(λ) learning algorithm gets the lowest median AVF accumulative root-mean-squared error (ARMSE) when there are eight or more states in the environment. This experimental evidence supports the theory that the QD(λ) learning algorithm can learn faster than the other RL algorithms for environments where many steps delay the reward. However, while the results of *Exp2* are consistent with this proposal, it has not been proven.

The experiments used parameter sweeps to evaluate the model across different config-

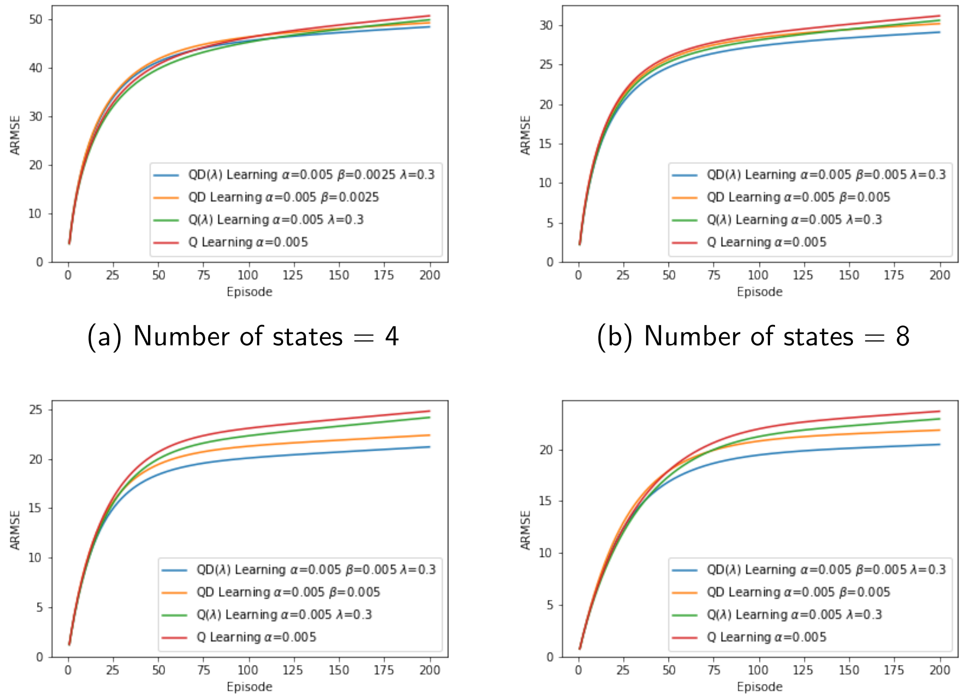
Table 6.6: The action-value function median accumulative root-mean-squared error (ARMSE) across 40 random seeds for episodes 50 and 200 for the best configurations of each of the four learning algorithms for four different types of environments

		Episode = 50		Episode = 200	
Number of states = 4	Configuration	ARMSE	p-value	ARMSE	p-value
QD(λ) Learning	α=0.005 β=0.0025 λ=0.3	35.1261	$< 10^{-4}$	39.8139	0.0039
QD Learning	α=0.005 β=0.0025	35.8691	$< 10^{-4}$	42.0724	0.0581
Q(λ) Learning	α=0.005 λ=0.3	33.2289	$< 10^{-4}$	38.6751	0.4597
Q Learning	α=0.005	33.9514	-	38.1652	-
Number of states = 8	Configuration	ARMSE	p-value	ARMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	19.9183	-	22.3943	-
QD Learning	α=0.005 β=0.005	20.3954	$< 10^{-4}$	23.6829	0.0001
Q(λ) Learning	α=0.005 λ=0.3	20.9529	0.0201	25.5197	0.0075
Q Learning	α=0.005	21.8806	0.0738	26.5480	0.0007
Number of states = 16	Configuration	ARMSE	p-value	ARMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	18.2678	-	20.3043	-
QD Learning	α=0.005 β=0.005	19.2181	$< 10^{-4}$	20.7923	$< 10^{-4}$
Q(λ) Learning	α=0.005 λ=0.3	20.8282	0.0294	23.9693	$< 10^{-4}$
Q Learning	α=0.005	22.3586	0.5453	25.9672	$< 10^{-4}$
Number of states = 32	Configuration	ARMSE	p-value	ARMSE	p-value
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	15.8704	-	18.9138	-
QD Learning	α=0.005 β=0.005	16.5807	$< 10^{-4}$	19.6555	$< 10^{-4}$
Q(λ) Learning	α=0.005 λ=0.3	16.4933	$< 10^{-4}$	21.3227	$< 10^{-4}$
Q Learning	α=0.005	17.1567	$< 10^{-4}$	22.3191	$< 10^{-4}$

urations. In total, the evaluations used 64 configurations across the four RL algorithms. Ideally, the parameter sweep would ensure that the parameter values of the best configurations were not at the limits of the sweep. For example, the parameter sweep of step size, α, ranges from 0.05 to 0.005, with the best configuration for all four RL algorithms being $\alpha = 0.005$; this is at the limit of the parameter sweep.

Would $\alpha = 0.0025$ have achieved better results for any RL algorithm? While this is likely the case, as a smaller step size results in a lower RMSE, it may not have achieved better results within 200 episodes, as there was insufficient time for the algorithm to converge. But extra episodes may result in the configuration of $\alpha = 0.0025$ being the best. As the best step size is again at the limit of the parameter sweep, the earlier question remains valid — would an even smaller step size get even better results for one of the RL algorithms? A similar cyclical argument applies to the eligibility decay parameter, λ and the consensus step size, β, which are also at the limit of the parameter sweep.

Exp1 presents the AVF RMSE for a QD(λ) parameter sweep in environments with eight states. The best configuration was $\alpha = 0.005$, $\beta = 0.005$ and $\lambda = 0.3$. *Exp2* compares the best configuration of all the RL algorithms across the four different environ-



(a) Number of states = 4

(b) Number of states = 8

(c) Number of states = 16

(d) Number of states = 32

Figure 6.13: The action-value function accumulative root-mean-squared error (ARMSE) for the best configurations of each of the four learning algorithms for four different types of environments

ments. It shows that for the test cases with eight states or more, the best configuration for QD(λ) learning was also $\alpha = 0.005$, $\beta = 0.005$ and $\lambda = 0.3$. But, these results should not be extrapolated to other environments with different transition functions, rewards or discounts (γ).

Lastly, caution should be applied when considering the significance of the results of the policy accumulated mean error (AME). An ideal policy will be generated from an AVF, which is far from ideal, so long as the state action values for each state have the ideal action as the highest value. This makes learning the best policy even more dependent on the specifics of the environment. Consequentially, the AVF RMSE results are more valuable for inferring general statements.

Two observations made from the experimental results are consistent with previous findings. The first concerns Q(λ) learning. Sutton et al. reported *"methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps"* [Sutton et al., 1998].

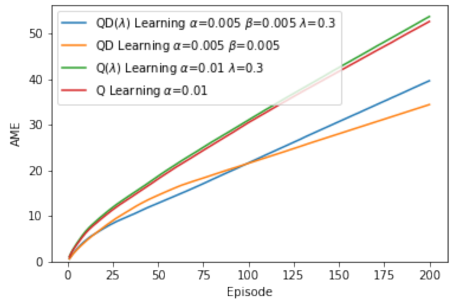
Table 6.8 shows the median AVF RMSE across the 40 random seeds for episodes of the best configurations of Q learning and Q(λ) learning for the four environments. It

Table 6.7: The Policy accumulative mean error (AME) across 40 random seeds for episodes 50 and 200 for the best configurations of the four learning algorithms for four different types of environments.

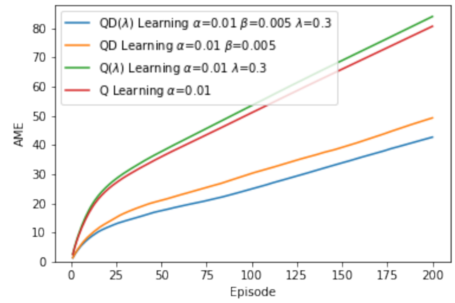
		Episode = 50		Episode = 200	
Number of states = 4	Configuration	AME	p-value	AME	p-value
QD Learning	α=0.005 β=0.005	$< 10^{-4}$	-	$< 10^{-4}$	-
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	$< 10^{-4}$	0.2845	$< 10^{-4}$	0.2600
Q(λ) Learning	α=0.01 λ=0.3	0.3346	$< 10^{-4}$	0.2449	0.0001
Q Learning	α=0.01	0.3702	0.0001	0.2581	$< 10^{-4}$
Number of states = 8	Configuration	AME	p-value	AME	p-value
QD Learning	α=0.005 β=0.005	$< 10^{-4}$	-	$< 10^{-4}$	-
QD(λ) Learning	α=0.005 β=0.005 λ=0.3	$< 10^{-4}$	0.2845	$< 10^{-4}$	0.2600
Q(λ) Learning	α=0.01 λ=0.3	0.3346	$< 10^{-4}$	0.2449	0.0001
Q Learning	α=0.01	0.3702	0.0001	0.2581	$< 10^{-4}$
Number of states = 16	Configuration	AME	p-value	AME	p-value
QD(λ) Learning	α=0.01 β=0.005 λ=0.6	61.3700	-	96.2600	-
QD Learning	α=0.01 β=0.005	67.3400	0.0001	124.6100	0.4846
Q Learning	α=0.01	126.8800	0.3678	241.2900	$< 10^{-4}$
Q(λ) Learning	α=0.01 λ=0.3	130.5700	0.4123	250.1000	$< 10^{-4}$
Number of states = 32	Configuration	AME	p-value	AME	p-value
QD(λ) Learning	α=0.01 β=0.005 λ=0.6	189.9700	-	324.2300	-
QD Learning	α=0.01 β=0.005	203.3400	$< 10^{-4}$	357.9300	0.0208
Q Learning	α=0.025	249.3500	$< 10^{-4}$	666.7100	$< 10^{-4}$
Q(λ) Learning	α=0.025 λ=0.3	257.0900	0.0002	685.2300	$< 10^{-4}$

shows that the median for Q(λ) learning is lower in all cases. However, the difference is only statistically significant in the larger environments with 16 and 32 states. These results are consistent with the assertion by Sutton et al. that the eligibility traces of Q(λ) learning will offer faster learning when many steps delay the rewards, as is the case in the larger test environments.

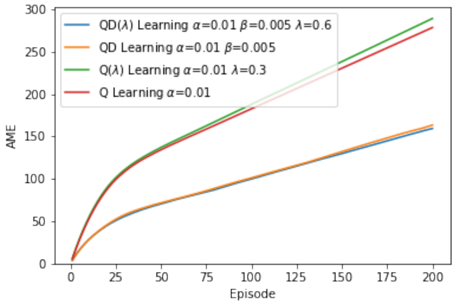
The second observation consistent with previous findings concerns Q(λ) learning; Tan stated that "agents engaging in partnership can significantly outperform independent agents although they may learn slowly in the beginning" [Tan, 1993]. Table 6.8 shows the median AVF RMSE across the 40 random seeds for the 3rd and 50th episodes of the best configurations of Q learning and QD learning for the four environments. It shows that, in the 3rd episode, the Q learning algorithm has a lower RMSE than QD learning for the environments with 16 or more states; the difference is statistically significant. By the 50th episode, the reverse is true. These findings are consistent wit Tan's assertion.



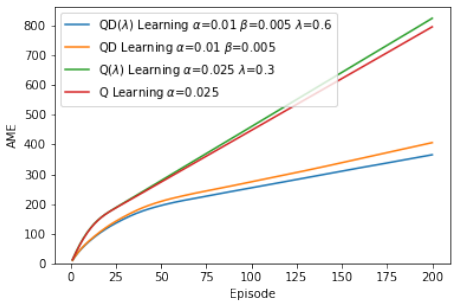
(a) Number of states = 4



(b) Number of states = 8



(c) Number of states = 16



(d) Number of states = 32

Figure 6.14: The Policy accumulative mean error (AME) for the best configurations of each of the four learning algorithms for four different types of environments

Table 6.8: The action-value function median root-mean-squared error (RMSE) across the 40 random seeds for episodes of the best configurations of Q learning and Q(λ) learning for the four environments.

		Episode = 50	
Number of states = 4	Configuration	RMSE	p-value
Q Learning	α=0.005	0.1037	-
Q(λ) Learning	α=0.005 λ=0.3	0.1070	0.8297
Number of states = 8	Configuration	RMSE	p-value
Q Learning	α=0.005	0.0524	-
Q(λ) Learning	α=0.005 λ=0.3	0.0534	0.9571
Number of states = 16	Configuration	RMSE	p-value
Q(λ) Learning	α=0.005 λ=0.3	0.0808	-
Q Learning	α=0.005	0.0956	0.0022
Number of states = 32	Configuration	RMSE	p-value
Q(λ) Learning	α=0.005 λ=0.3	0.1398	-
Q Learning	α=0.005	0.1498	$< 10^{-4}$

6.5 Conclusion

Implications The contribution of this chapter is the QD(λ) learning algorithm, a value-based MARL algorithm that leverages networked agents. QD(λ) learning applies

Table 6.9: The action-value function median root-mean-squared error (RMSE) across the 50 random seeds for the 5th and 200th episodes of the best configurations of Q learning and QD learning for the four environments.

		Episode = 3		Episode = 50	
Number of states = 4	Configuration	RMSE	p-value	RMSE	p-value
Q Learning	α=0.005	2.0355	-	0.1037	-
QD Learning	α=0.005 β=0.0025	2.4811	0.1010	0.0587	$< 10^{-4}$
Number of states = 8	Configuration	RMSE	p-value	RMSE	p-value
Q Learning	α=0.005	1.6447	0.7881	0.0524	$< 10^{-4}$
QD Learning	α=0.005 β=0.005	1.6288	-	0.0461	-
Number of states = 16	Configuration	RMSE	p-value	RMSE	p-value
Q Learning	α=0.005	1.1320	-	0.0956	-
QD Learning	α=0.005 β=0.005	1.1326	0.0348	0.0594	$< 10^{-4}$
Number of states = 32	Configuration	RMSE	p-value	RMSE	p-value
Q Learning	α=0.005	0.6614	-	0.1498	-
QD Learning	α=0.005 β=0.005	0.6793	$< 10^{-4}$	0.1111	$< 10^{-4}$

transfer learning to share information between agents, similar to the MARL QD learning algorithm [Kar et al., 2012]. It also uses eligibility traces when updating the action value function akin to the RL Q(λ) learning algorithm [Watkins, 1989]. Eligibility traces offer faster learning when rewards are received after many steps.

The RL environments used in the experiments emulate the environment of an agent learning a protocol, where the agent takes a series of actions, such as constructing, sending, receiving and processing messages, before getting a reward. The results support the theory that the combination of transfer learning and eligibility traces in the QD(λ) learning algorithm results in faster learning. This finding suggests that QD(λ) learning would be the preferred MARL algorithm for complex and dynamic environments. For example, a cooperative multi-agent system learning a coordination protocol like Gossip Contracts.

Limitations However, the QD(λ) learning algorithm has several limitations. While there is no extra network traffic over the QD learning algorithm, additional computation is required. Though QD(λ) learns faster than the other RL algorithms for the test environments with eight or more states, the difference is slight. Is the extra computation worth the effort for a minor improvement in learning speed?

Recommendations The general hypothesis proposed in this chapter is that QD(λ) learning learns faster than other RL algorithms for environments where the agent must transition through many states before a reward is received. However, while the experimentation supports the theory, it has not been proven. A mathematical proof of the hypothesis is outside the scope of this research.

In further work, the author aims to use the QD(λ) learning algorithm with Gossip Contracts to create the Learned Gossip Contracts (LGC) framework. LGC can leverage QD(λ) learning to augment the GC protocol and continually learn the best protocol to address a specific problem. This leveraging of a MARL algorithm in LGC is similar to how EGC uses genetic programming to search for the implementation of the GC methods to tailor it to a problem. QD(λ) learning is valuable when the problem LGC is addressing changes over time, as the algorithm will learn the new optimal policy.

CHAPTER 7

Conclusion

The central objective of this research is to *enhance the use of nature-inspired techniques that semi-automate the development of decentralised heuristics for addressing challenges in complex systems*. It focuses on various nature-inspired techniques and problem domains, outlining three specific objectives that the research literature has not sufficiently explored. They are to:

1. Develop a genuinely decentralised DVMC strategy and evaluate it in a cloud data centre modelled using a real-world dataset, considering packing efficiency, the number of migrations and SLA violations.
2. Develop an EC-based framework to design decentralised heuristics evaluated on an NP-hard problem.
3. Develop a value-based MARL algorithm that combines the advantages of transfer learning while minimising the initial loss of learning speed.

7.1 Summary of Research Contributions

In addressing the above research objectives, this thesis makes several contributions to the advancement of contemporary literature. The subsequent subsections summarise the four key contributions of this thesis.

7.1.1 Gossip Contracts

Gossip Contracts (GC) is a novel communication protocol for networked multi-agent systems to facilitate decentralised cooperation strategies. GC is inspired by Contract Net, a communication protocol for a system of distributed problem solvers to facilitate allocating tasks across the system's agents [Smith, 1980]. GC leverages a nature-inspired gossip-like multi-cast protocol to send tender messages further across the network. GC was evaluated by using it as the communication protocol for a novel decentralised DVMC strategy described in the next subsection. The work outlined in this contribution has been published in the following journal paper:

- Nicola Mc Donnell, Enda Howley, and Jim Duggan. *"Dynamic virtual machine consolidation using a multi-agent system to optimise energy efficiency in cloud computing."* Future Generation Computer Systems 108 (2020): 288-301. `https://doi.org/10.1016/j.future.2020.02.036`

7.1.2 Gossip Contracts-based DVMC strategy

The second contribution of this thesis is a novel decentralised DVMC strategy built using GC. It was evaluated using a cloud data centre model derived from a real-world dataset, the Google cluster-usage traces (GCT) [Reiss et al., 2011]. The assessment considered packing efficiency, power consumption, the number of migrations and SLA violations. It was compared to two well-known DVMC strategies: a centralised strategy, *Sercon* and a distributed strategy, *ecoCloud*. The GC strategy outperformed the other strategies with respect to minimising SLA violations. However, it had similar power consumption to *Sercon*, which was lower than *ecoCloud*. In addition, it had equivalent or fewer migrations to *ecoCloud* but a higher number of migrations than *Sercon*. The work outlined in this contribution has been published in the following journal paper:

- Nicola Mc Donnell, Enda Howley, and Jim Duggan. *"Dynamic virtual machine consolidation using a multi-agent system to optimise energy efficiency in cloud computing."* Future Generation Computer Systems 108 (2020): 288-301. `https://doi.org/10.1016/j.future.2020.02.036`

7.1.3 Evolved Gossip Contracts

The Evolved Gossip Contracts (EGC) framework is a novel EC-based framework to design decentralised heuristics that leverage the GC protocol. EGC uses genetic programming to semi-automate the search for a specific GC protocol. The idea of using evolutionary computing (EC) to assist in developing MAS has a long history. Early research focused on proving EC could be an effective strategy for evolving a MAS and exploring which

EC techniques and features best suit the challenge. However, frameworks which a MAS designer can apply to many problems would be more valuable. While researchers have proposed several such frameworks, which are a significant contribution, based on my review of the literature none were evaluated on an NP-hard problem.

The EGC framework was used to develop GC-based decentralised heuristics for the NP-hard bin packing problem (BPP). The best BPP strategy developed by EGC is shown to be better than two BPP heuristics popular in the literature, *Best Fit* and *First Fit*, for the datasets on which it was trained. The work outlined in this contribution has been published in the following conference and journal papers:

- Nicola Mc Donnell, Enda Howley, and Jim Duggan. *"Evolved Gossip Contracts-A Framework for Designing Multi-agent Systems."* Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I 16. Springer International Publishing, 2020. `https://doi.org/10.1007/978-3-030-58112-1_44`
- Nicola Mc Donnell, Enda Howley, and Jim Duggan. *"A Genetic Programming-based Framework for Semi-automated Multi-agent Systems Engineering."* ACM Transactions on Autonomous and Adaptive Systems (TAAS) (2023) `https://doi.org/10.1145/3584731`

7.1.4 QD(λ) Learning

The fourth contribution is the QD(λ) learning algorithm. It is a novel value-based MARL algorithm that leverages networked agents. It incorporates the advantages of transfer learning from the MARL QD learning [Kar et al., 2012] algorithm and the faster learning when rewards are delayed from the RL Q(λ) Learning algorithm [Watkins, 1989]. The QD(λ) learning algorithm was evaluated in specially designed RL environments that require agents to transition through several states before receiving a reward. The QD(λ) learning algorithm achieved the lowest accumulative root-mean-squared error when the environment has eight or more states, thus overcoming the finding by Tan that "agents engaging in partnership ... may learn slowly in the beginning" [Tan, 1993]. The work outlined in this contribution is under review in the following conference workshop paper:

- Nicola Mc Donnell, Enda Howley, and Jim Duggan. *"QD(λ) Learning: Multi-agent Reinforcement Learning algorithm with eligibility decay"* Neural Computing and Applications (2023)

7.1.5 Open Source Software Packages

The fifth contribution is a set of Open Source software packages. These packages were developed as part of this research and are available online in support of Open Science [Mc Donnell, 2022]. They include the packages of the three research contributions: the GC protocol, the EGC framework and the QD(λ) learning algorithm, and the two packages of the simulators for evaluations: the Bin Packing Problem simulator and the DVMC simulator. As these packages are easily accessible, this research is easier to reproduce, critique and extend.

7.2 Impact

The contributions outlined in the previous subsection can influence future research in several distinct ways. Firstly, MAS designers can use the Gossip Contracts protocol proposed in Chapter 4 to develop large cooperative multi-agent systems. It may be used in place of the Contract Net protocol, especially where the MAS comprises many cooperating agents. Furthermore, cloud data centre-provisioning technology developers can use the DVMC strategy developed in Chapter 4 using Gossip Contracts. It is especially beneficial where minimising SLA violations is of paramount concern to the data centre managers.

MAS designers can use the Evolved Gossip Contracts framework proposed in Chapter 5 to semi-automate the design of decentralised heuristics. Demand for these algorithms is growing with innovative technologies such as Edge computing and the Internet of Things. The empirical results in Chapter 5 showed that a MAS designer could use the EGC framework to develop decentralised heuristics for challenging problems, such as those from the NP-hard complexity class.

Finally, the empirical results in Chapter 6 show that the QD(λ) learning algorithm learns more quickly, than the other three RL algorithms, in large environments that require agents to pass through several states before receiving the reward. Thus, QD(λ) learning is suited to a large multi-agent system learning a communication protocol. This is a emerging research area has seen significant activity in the last few years [Foerster et al., 2016, Sukhbaatar et al., 2016, Jiang and Lu, 2018, Das et al., 2019, Mao et al., 2020, Tung et al., 2021].

7.3 Limitations

The research contributions have several limitations outlined in the next paragraphs.

Gossip Contracts During the development of the GC-based DVMC strategy presented in Chapter 4, the author spent significant effort to discover one which was competitive with *Sercon* and *ecoCloud*. Thus, using GC, a MAS designer has no guarantees that they can develop an effective heuristic or that they can develop it within an acceptable time limit. It was this limitation of GC which inspired the development of the Evolved Gossip Contracts framework.

Evolved Gossip Contracts The use of the Evolved Gossip Contracts framework is restricted to specific problems where the system is composed of distinct entities that can be agents in a multi-agent system. There must be a means of calculating the fitness of the evolved programs or candidate GC protocols. Typically, a MAS designer will assess each candidate GC protocol in a simulation environment with a series of test cases. These assessments have several limitations. The test cases must be representative of the real problem. Additionally, it may be non-trivial to define the fitness function, especially for multi-objective problems. Furthermore, the evaluation may be computationally expensive.

QD(λ) Learning The most notable limitation of the QD(λ) learning algorithm is that, though it learned faster than the other RL algorithms for the larger test environments, the difference is slight. Is the extra computation worth the effort for a minor improvement in learning speed? Besides, while the QD(λ) learning algorithm learned faster, it did so at the cost of learning effectiveness; the QD learning algorithm had a lower root-mean-squared error for environments with a higher number of states.

7.4 Future Work

Arising out of this thesis there are a number of promising avenues for future research.

Gossip Contracts There are several parameters which control the behaviour of the Gossip Contracts protocol, such as the number of agents in an agent's immediate neighbourhood or the number of times the tender messages are forwarded. An analysis of these parameters to understand their impact on the MAS would be valuable. Ideally, best practices on configuring GC protocol would guide MAS designers who are uptaking the Gossip Contracts protocol

Gossip Contracts-based DVMC strategy The cloud data centre model described in the experiments in Chapter 4 utilised a real-world dataset, the Google cluster-usage traces

[Reiss et al., 2011]. The evaluation considered packing efficiency, power consumption, the number of migrations and SLA violations. However, a more in-depth study of the Gossip Contracts-based DVMC strategy could consider network load, message delivery issues and significant network failure. It could explore the criticism of gossip protocol for its heavy use of network bandwidth. Furthermore, it could investigate reports in the literature that decentralised algorithms are resilient to network delays and failures and perform well in dynamic environments where nodes arrive and leave the network over time.

Evolved Gossip Contracts The empirical results presented in Chapter 5 evaluated the EGC framework's effectiveness at developing a decentralised heuristic for one NP-hard optimisation problem, the bin packing problem. Using the EGC framework to develop decentralised heuristics for other NP-hard problems could strengthen the assertion made in the chapter. The finding of this further research may make it possible to create a more general statement about the usefulness of EGC.

Learned Gossip Contract The author's aim in developing the QD(λ) learning algorithm is to use it with Gossip Contracts to create the Learned Gossip Contracts (LGC) framework. LGC can leverage QD(λ) learning to augment the GC protocol and continually learn the best protocol to address a specific problem. This leveraging of a MARL algorithm in LGC is similar to how EGC uses genetic programming to search for the implementation of the GC methods to tailor it to a problem. QD(λ) learning is valuable when the problem LGC addresses changes over time, as the algorithm will learn the new optimal policy more quickly than QD learning.

7.5 Final Remarks

Multi-agent systems are already widely used, and their value is growing with new technologies, such as edge computing and the Internet of things. However, they are non-trivial to develop, with the most commonly employed methods being process-based implementations by MAS designers. Self-organisation is pervasive in natural systems, so some MAS designers use nature-inspired techniques to aid them. This thesis explored a series of frontiers in these areas, presenting a novel communication protocol for cooperating MAS, a framework to semi-automate in the design of MAS, and a MARL algorithm that leverages networked agents. I hope this thesis has enhanced the use of nature-inspired techniques that semi-automate the development of decentralised heuristics for resolving complex systems.

Bibliography

[Abd Elminaam et al., 2020] Abd Elminaam, D. S., Al-Mutairi, W. M., Abd El Fattah, M., and El-Ashmawi, W. H. (2020). An adaptive fitness-dependent optimizer for the one-dimensional bin packing problem. *IEEE Access*.

[Abdel-Basset et al., 2018] Abdel-Basset, M., Manogaran, G., Abdel-Fatah, L., and Mirjalili, S. (2018). An improved nature inspired meta-heuristic algorithm for 1-d bin packing problems. *Personal and Ubiquitous Computing*, 22(5-6):1117–1132.

[Agogino and Tumer, 2007] Agogino, A. and Tumer, K. (2007). Evolving distributed agents for managing air traffic. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1888–1895.

[Albino et al., 2015] Albino, V., Berardi, U., and Dangelico, R. M. (2015). Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of urban technology*, 22(1):3–21.

[Amazon, 2018] Amazon (2018). Amazon ec2 instance types. Available online at: https://aws.amazon.com/ec2/instance-types/, Accessed on 13.05.2018.

[Anders et al., 2015] Anders, G., Schiendorfer, A., Siefert, F., Steghöfer, J.-P., and Reif, W. (2015). Cooperative resource allocation in open systems of systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(2):1–44.

[Antwi et al., 2022] Antwi, R., Gadze, J. D., Tchao, E. T., Sikora, A., Nunoo-Mensah, H., Agbemenu, A. S., Obour Agyekum, K. O.-B., Agyemang, J. O., Welte, D., and Keelson, E. (2022). A survey on network optimization techniques for blockchain systems. *Algorithms*, 15(6):193.

[Aronsson, 2003] Aronsson, J. (2003). *Genetic programming of multi-agent system in the robocup domain*. Univ.

[Ashraf et al., 2018] Ashraf, A., Byholm, B., and Porres, I. (2018). Distributed virtual machine consolidation: A systematic mapping study. *Computer Science Review*, 28:118–130.

[Attiya and Welch, 2004] Attiya, H. and Welch, J. (2004). *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons.

[Atzori et al., 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer networks*, 54(15):2787–2805.

[Augusto and Barbosa, 2000] Augusto, D. A. and Barbosa, H. J. (2000). Symbolic regression via genetic programming. In *Proceedings. Vol. 1. Sixth Brazilian Symposium on Neural Networks*, pages 173–178. IEEE.

[Babaoglu et al., 2002] Babaoglu, O., Meling, H., and Montresor, A. (2002). Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 15–22. IEEE.

[Balaji and Srinivasan, 2010] Balaji, P. G. and Srinivasan, D. (2010). An introduction to multi-agent systems. In *Innovations in multi-agent systems and applications-1*, pages 1–27. Springer.

[Barbagallo et al., 2009] Barbagallo, D., Di Nitto, E., Dubois, D. J., and Mirandola, R. (2009). A bio-inspired algorithm for energy optimization in a self-organizing data center. In *International Workshop on Self-Organizing Architectures*, pages 127–151. Springer.

[Barroso et al., 2013] Barroso, L. A., Clidaras, J., and Hölzle, U. (2013). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154.

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684.

[Bellman, 1966] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37.

[Beloglazov and Buyya, 2012] Beloglazov, A. and Buyya, R. (2012). Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420.

[Berl et al., 2010] Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., and Pentikousis, K. (2010). Energy-efficient cloud computing. *The computer journal*, 53(7):1045–1051.

[Bonomi et al., 2012] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16.

[Bratman, 1987] Bratman, M. (1987). Intention, plans, and practical reason.

[Bresciani et al., 2004] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

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

[Cardoso and Ferrando, 2021] Cardoso, R. C. and Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, 10(2):16.

[Carter and Price, 2017] Carter, M. W. and Price, C. C. (2017). *Operations research: a practical introduction*. Crc Press.

[Chekuri and Khanna, 2004] Chekuri, C. and Khanna, S. (2004). On multidimensional packing problems. *SIAM journal on computing*, 33(4):837–851.

[Claus and Boutilier, 1998] Claus, C. and Boutilier, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998(746-752):2.

[CNET, 2018] CNET (2018). Hp proliant ml110 g4 specs. Available online at: `https://www.cnet.com/products/hp-proliant-ml110-g4-xeon-3040-1-86-ghz-monitor-none-series/specs/`, Accessed on 13.05.2018.

[Coffman Jr et al., 1996] Coffman Jr, E. G., Garey, M. R., and Johnson, D. S. (1996). Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems*, pages 46–93.

[Da Silva and Costa, 2019] Da Silva, F. L. and Costa, A. H. R. (2019). A survey on transfer learning for multiagent reinforcement learning systems. *Journal of Artificial Intelligence Research*, 64:645–703.

[Dai et al., 2011] Dai, F., Kushida, N., Shang, L., and Sugisaka, M. (2011). A survey of genetic algorithm-based face recognition. *Artificial Life and Robotics*, 16(2):271–274.

[Das et al., 2019] Das, A., Gervet, T., Romoff, J., Batra, D., Parikh, D., Rabbat, M., and Pineau, J. (2019). Tarmac: Targeted multi-agent communication. In *International Conference on Machine Learning*, pages 1538–1546. PMLR.

[Davidsson et al., 2007] Davidsson, P., Persson, J. A., and Holmgren, J. (2007). On the integration of agent-based and mathematical optimization techniques. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pages 1–10. Springer.

[Deb, 2005] Deb, K. (2005). Multi-objective optimization. In *Search methodologies*, pages 273–316. Springer.

[Deb, 2014] Deb, K. (2014). Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer.

[DeLoach et al., 2001] DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258.

[Di Marzo Serugendo et al., 2005] Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A. (2005). Self-organization in multi-agent systems. *Knowledge Engineering Review*, 20(2):165–189.

[Dósa and Sgall, 2013] Dósa, G. and Sgall, J. (2013). First fit bin packing: A tight analysis. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[El-Ashmawi and Abd Elminaam, 2019] El-Ashmawi, W. H. and Abd Elminaam, D. S. (2019). A modified squirrel search algorithm based on improved best fit heuristic and operator strategy for bin packing problem. *Applied Soft Computing*, 82:105565.

[Falkenauer, 1996] Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30.

[Farahnakian et al., 2014] Farahnakian, F., Ashraf, A., Pahikkala, T., Liljeberg, P., Plosila, J., Porres, I., and Tenhunen, H. (2014). Using ant colony system to consolidate vms for green cloud computing. *IEEE Transactions on Services Computing*, 8(2):187–198.

[Feller et al., 2012] Feller, E., Morin, C., and Esnault, A. (2012). *A case for fully decentralized dynamic VM consolidation in clouds*. PhD thesis, INRIA.

[Ferdaus et al., 2014] Ferdaus, M. H., Murshed, M., Calheiros, R. N., and Buyya, R. (2014). Virtual machine consolidation in cloud data centers using aco metaheuristic. In *European conference on parallel processing*, pages 306–317. Springer.

[Ferrante et al., 2013] Ferrante, E., Duéñez-Guzmán, E., Turgut, A. E., and Wenseleers, T. (2013). Geswarm: Grammatical evolution for the automatic synthesis of collective behaviors in swarm robotics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 17–24.

[Foerster et al., 2016] Foerster, J., Assael, I. A., De Freitas, N., and Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29.

[Fogel, 1998] Fogel, D. B. (1998). *Artificial intelligence through simulated evolution*. Wiley-IEEE Press.

[Gendreau et al., 2010] Gendreau, M., Potvin, J.-Y., et al. (2010). *Handbook of metaheuristics*, volume 2. Springer.

[Georgeff et al., 1998] Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1998). The belief-desire-intention model of agency. In *International workshop on agent theories, architectures, and languages*, pages 1–10. Springer.

[Gershenson, 2007] Gershenson, C. (2007). *Design and control of self-organizing systems*.

[González-Briones et al., 2018] González-Briones, A., De La Prieta, F., Mohamad, M. S., Omatu, S., and Corchado, J. M. (2018). Multi-agent systems applications in energy optimization problems: A state-of-the-art review. *Energies*, 11(8):1928.

[Greenberg et al., 2008] Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P. (2008). The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73.

[Hadden et al., 2007] Hadden, J., Tiwari, A., Roy, R., and Ruta, D. (2007). Computer assisted customer churn management: State-of-the-art and future trends. *Computers & Operations Research*, 34(10):2902–2917.

[Haynes et al., 1995a] Haynes, T., Sen, S., Schoenefeld, D., and Wainwright, R. (1995a). Evolving a team. In *Working notes for the AAAI symposium on genetic programming*, pages 23–30. MIT Cambridge.

[Haynes et al., 1995b] Haynes, T., Wainwright, R. L., Sen, S., and Schoenefeld, D. A. (1995b). Strongly typed genetic programming in evolving cooperation strategies. In *Proceedings of The Sixth International Conference on Genetic Algorithms*, volume 95, pages 271–278.

[Haynes and Wainwright, 1995] Haynes, T. D. and Wainwright, R. L. (1995). A simulation of adaptive agents in a hostile environment. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 318–323.

[Hegedűs et al., 2016] Hegedűs, I., Berta, Á., Kocsis, L., Benczúr, A. A., and Jelasity, M. (2016). Robust decentralized low-rank matrix decomposition. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 7(4):1–24.

[Hegedűs et al., 2019] Hegedűs, I., Danner, G., and Jelasity, M. (2019). Gossip learning as a decentralized alternative to federated learning. In *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19*, pages 74–90. Springer.

[Herrera et al., 2020] Herrera, M., Pérez-Hernández, M., Kumar Parlikad, A., and Izquierdo, J. (2020). Multi-agent systems and complex networks: Review and applications in systems engineering. *Processes*, 8(3):312.

[Hinrichs et al., 2013] Hinrichs, C., Lehnhoff, S., and Sonnenschein, M. (2013). Cohda: A combinatorial optimization heuristic for distributed agents. In *International Conference on Agents and Artificial Intelligence*, pages 23–39. Springer.

[Hinrichs et al., 2014] Hinrichs, C., Lehnhoff, S., and Sonnenschein, M. (2014). A decentralized heuristic for multiple-choice combinatorial optimization problems. In *Operations Research Proceedings 2012*, pages 297–302. Springer.

[Holland, 1975] Holland, J. H. (1975). Adaptation in natural and artificial systems.

[Iba, 1996] Iba, H. (1996). Emergent cooperation for multiple agents using genetic programming. In *International conference on parallel problem solving from nature*, pages 32–41. Springer.

[Iba, 1998] Iba, H. (1998). Evolutionary learning of communicating agents. *Information Sciences*, 108(1-4):181–205.

[Iba et al., 1997] Iba, H., Nozoe, T., and Ueda, K. (1997). Evolving communicating agents based on genetic programming. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pages 297–302. IEEE.

[ICHEC, 2022a] ICHEC (2022a). Fionn. Available online at: `https://www.ichec.ie/about/infrastructure/fionn`, last accessed on 21.09.2022.

[ICHEC, 2022b] ICHEC (2022b). Kay. Available online at: `https://www.ichec.ie/about/infrastructure/kay`, last accessed on 21.09.2022.

[Jelasity, 2011] Jelasity, M. (2011). Gossip. In *Self-organising software*, pages 139–162. Springer.

[Jelasity et al., 2007] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Van Steen, M. (2007). Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8.

[Jiang and Hu, 2020] Jiang, J. and Hu, L. (2020). Decentralised federated learning with adaptive partial gradient aggregation. *CAAI Transactions on Intelligence Technology*, 5(3):230–236.

[Jiang and Lu, 2018] Jiang, J. and Lu, Z. (2018). Learning attentional communication for multi-agent cooperation. *Advances in neural information processing systems*, 31.

[Julian and Botti, 2019] Julian, V. and Botti, V. (2019). Multi-agent systems. *Applied Sciences*, 9(7).

[Kanchana et al., 2022] Kanchana, V., Nath, S., and Singh, M. K. (2022). A study of internet of things oriented smart medical systems. *Materials Today: Proceedings*, 51:961–964.

[Kar et al., 2012] Kar, S., Moura, J. M., and Poor, H. V. (2012). Qd-learning: A collaborative distributed strategy for multi-agent reinforcement learning through consensus. *arXiv preprint arXiv:1205.0047*.

[Khan et al., 2018] Khan, M. A., Paplinski, A., Khan, A. M., Murshed, M., and Buyya, R. (2018). Dynamic virtual machine consolidation algorithms for energy-efficient cloud resource management: A review. In *Sustainable Cloud and Energy Services*, pages 135–165. Springer.

[Khelghatdoust et al., 2016] Khelghatdoust, M., Gramoli, V., and Sun, D. (2016). Glap: distributed dynamic workload consolidation through gossip-based learning. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 80–89. IEEE.

[Knotts et al., 2000] Knotts, G., Dror, M., and Hartman, B. C. (2000). Agent-based project scheduling. *Iie Transactions*, 32(5):387–401.

[Konečnỳ et al., 2016] Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

[Kong et al., 2022] Kong, L., Tan, J., Huang, J., Chen, G., Wang, S., Jin, X., Zeng, P., Khan, M. K., and Das, S. K. (2022). Edge-computing-driven internet of things: A survey. *ACM Computing Surveys (CSUR)*.

[Koza, 1992] Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

[Koza, 1994] Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112.

[Koza et al., 1989] Koza, J. R. et al. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, volume 89, pages 768–774.

[Kwon et al., 2005] Kwon, O., Im, G., and Lee, K. C. (2005). Mace-scm: An effective supply chain decision making approach based on multi-agent and case-based reasoning. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 82b–82b. IEEE.

[Lauer and Riedmiller, 2000] Lauer, M. and Riedmiller, M. (2000). An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer.

[Layeb and Boussalia, 2012] Layeb, A. and Boussalia, S. R. (2012). A novel quantum inspired cuckoo search algorithm for bin packing problem. *International Journal of Information Technology and Computer Science*, 4(5):58–67.

[Li et al., 2010] Li, J., Wei, H., and Xia, X. (2010). The multi-agent model and method for energy-saving generation dispatching system. In *2010 International Conference on Power System Technology*, pages 1–8. IEEE.

[Li et al., 2020] Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. (2020). Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60.

[Liu et al., 2014] Liu, K., Tong, M., Xie, S., and Zeng, Z. (2014). Fusing decision trees based on genetic programming for classification of microarray datasets. In *International Conference on Intelligent Computing*, pages 126–134. Springer.

[Luke et al., 1997] Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. (1997). Co-evolving soccer softbot team coordination with genetic programming. In *Robot Soccer World Cup*, pages 398–411. Springer.

[Luke and Spector, 1996] Luke, S. and Spector, L. (1996). Evolving teamwork and coordination with genetic programming. *Genetic Programming*, 96:150–56.

[Luo et al., 2022] Luo, Y., Jiang, X., Zhong, S., Ji, Y., and Sun, G. (2022). Multi-satellite task allocation and orbit planning for asteroid terminal defence. *Aerospace*, 9(7):364.

[Manela, 1994] Manela, M. (1994). *Contributions to the Theory and Applications of Genetic Algorithms*. University of London, University College London (United Kingdom).

[Mao et al., 2020] Mao, H., Zhang, Z., Xiao, Z., Gong, Z., and Ni, Y. (2020). Learning multi-agent communication with double attentional deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 34:1–34.

[Marzolla et al., 2011] Marzolla, M., Babaoglu, O., and Panzieri, F. (2011). Server consolidation in clouds through gossiping.

[Mascardi and Weyns, 2018] Mascardi, V. and Weyns, D. (2018). Engineering multi-agent systems anno 2025. In *International Workshop on Engineering Multi-Agent Systems*, pages 3–16. Springer.

[Mascardi et al., 2019] Mascardi, V., Weyns, D., Ricci, A., Earle, C. B., Casals, A., Challenger, M., Chopra, A., Ciortea, A., Dennis, L. A., Díaz, Á. F., et al. (2019). Engineering multi-agent systems: State of affairs and the road ahead. *ACM SIGSOFT Software Engineering Notes*, 44(1):18–28.

[Masoumzadeh and Hlavacs, 2016] Masoumzadeh, S. S. and Hlavacs, H. (2016). A gossip-based dynamic virtual machine consolidation strategy for large-scale cloud data centers. In *Proceedings of the Third International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 28–34. ACM.

[Mastroianni et al., 2011] Mastroianni, C., Meo, M., and Papuzzo, G. (2011). Self-economy in cloud data centers: Statistical assignment and migration of virtual machines. In *European Conference on Parallel Processing*, pages 407–418. Springer.

[Mastroianni et al., 2013] Mastroianni, C., Meo, M., and Papuzzo, G. (2013). Probabilistic consolidation of virtual machines in self-organizing cloud data centers. *IEEE Transactions on Cloud Computing*, 1(2):215–228.

[Mathkar and Borkar, 2016] Mathkar, A. and Borkar, V. S. (2016). Distributed reinforcement learning via gossip. *IEEE Transactions on Automatic Control*, 62(3):1465–1470.

[Matignon et al., 2007] Matignon, L., Laurent, G. J., and Le Fort-Piat, N. (2007). Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 64–69. IEEE.

[Mc Donnell, 2022] Mc Donnell, N. (2022). Phd repository. Available online at: `https://bitbucket.org/nixsphd/gaia/src/master/`, last accessed on 21.09.2022.

[Mc Donnell et al., 2020a] Mc Donnell, N., Howley, E., and Duggan, J. (2020a). Dynamic virtual machine consolidation using a multi-agent system to optimise energy efficiency in cloud computing. *Future Generation Computer Systems*, 108:288–301.

[Mc Donnell et al., 2020b] Mc Donnell, N., Howley, E., and Duggan, J. (2020b). Evolved gossip contracts-a framework for designing multi-agent systems. In *Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I 16*, pages 637–649. Springer.

[McMahan et al., 2017] McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR.

[Mell et al., 2011] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.

[Mitchell et al., 1994] Mitchell, M., Crutchfield, J. P., and Hraber, P. T. (1994). Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75(1-3):361–391.

[Montresor and Jelasity, 2009] Montresor, A. and Jelasity, M. (2009). Peersim: A scalable p2p simulator. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 99–100. IEEE.

[Müller and Fischer, 2014] Müller, J. P. and Fischer, K. (2014). Application impact of multi-agent systems and technologies: A survey. In *Agent-oriented software engineering*, pages 27–53. Springer.

[Murtazaev and Oh, 2011] Murtazaev, A. and Oh, S. (2011). Sercon: Server consolidation algorithm using live migration of virtual machines for green computing. *IETE Technical Review*, 28(3):212–231.

[Nezami et al., 2021] Nezami, Z., Zamanifar, K., Djemame, K., and Pournaras, E. (2021). Decentralized edge-to-cloud load balancing: Service placement for the internet of things. *IEEE Access*, 9:64983–65000.

[Nguyen et al., 2020] Nguyen, T. T., Nguyen, N. D., and Nahavandi, S. (2020). Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE transactions on cybernetics*, 50(9):3826–3839.

[Obraczka et al., 2001] Obraczka, K., Viswanath, K., and Tsudik, G. (2001). Flooding for reliable multicast in multi-hop ad hoc networks. volume 7, pages 627–634. Springer.

[Pahlevan et al., 2020] Pahlevan, A., Zapater, M., Coskun, A. K., and Atienza, D. (2020). Ecogreen: Electricity cost optimization for green datacenters in emerging power markets. *IEEE Transactions on Sustainable Computing*, 6(2):289–305.

[Pantazoglou et al., 2015] Pantazoglou, M., Tzortzakis, G., and Delis, A. (2015). Decentralized and energy-efficient workload management in enterprise clouds. *IEEE transactions on cloud computing*, 4(2):196–209.

[Parunak, 1996] Parunak, H. V. D. (1996). Applications of distributed artificial intelligence in industry. *Foundations of distributed artificial intelligence*, 2(1):18.

[Pavlov, 2010] Pavlov, P. I. (2010). Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex. *Annals of neurosciences*, 17(3):136.

[Persson et al., 2005] Persson, J. A., Davidsson, P., Johansson, S. J., and Wernstedt, F. (2005). Combining agent-based approaches and classical optimization techniques. In *Third European Workshop on Multi-Agent Systems*.

[Pilgerstorfer and Pournaras, 2017] Pilgerstorfer, P. and Pournaras, E. (2017). Self-adaptive learning in decentralized combinatorial optimization-a design paradigm for sharing economies. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 54–64. IEEE.

[Pournaras, 2013] Pournaras, E. (2013). *Multi-level reconfigurable self-organization in overlay services*. PhD thesis.

[Pournaras et al., 2018] Pournaras, E., Pilgerstorfer, P., and Asikis, T. (2018). Decentralized collective learning for self-managed sharing economies. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(2):1–33.

[Pournaras et al., 2010] Pournaras, E., Warnier, M., and Brazier, F. M. (2010). Local agent-based self-stabilisation in global resource utilisation. *International Journal of Autonomic Computing*, 1(4):350–373.

[Prabha et al., 2021] Prabha, B., Ramesh, K., and Renjith, P. (2021). A review on dynamic virtual machine consolidation approaches for energy-efficient cloud data centers. *Data Intelligence and Cognitive Informatics: Proceedings of ICDICI 2020*, pages 761–780.

[Puterman, 2014] Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[Qureshi, 1996] Qureshi, A. (1996). Evolving agents. *Genetic Programming*, pages 369–374.

[Rahimian et al., 2015] Rahimian, F., Payberah, A. H., Girdzijauskas, S., Jelasity, M., and Haridi, S. (2015). A distributed algorithm for large-scale graph partitioning. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(2):1–24.

[Răileanu and Borangiu, 2023] Răileanu, S. and Borangiu, T. (2023). A review of multi-agent systems used in industrial applications. *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future: Proceedings of SOHOMA 2022*, pages 3–22.

[Rechenberg, 1965] Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment Library Translation 1122*.

[Reiss et al., 2011] Reiss, C., Wilkes, J., and Hellerstein, J. L. (2011). Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14.

[Rejeb et al., 2022] Rejeb, A., Rejeb, K., Simske, S., Treiblmaier, H., and Zailani, S. (2022). The big picture on the internet of things and the smart city: a review of what we know and what we need to know. *Internet of Things*, 19:100565.

[Richa et al., 2023] Richa, A., Hanachi, C., and Stolf, P. (2023). Simulating and analyzing crowdsourcing impacts in flood management: A geo-spatial agent-based approach. In *International Conference on Advanced Information Systems Engineering*, pages 107–122. Springer.

[Rieke et al., 2020] Rieke, N., Hancox, J., Li, W., Milletari, F., Roth, H. R., Albarqouni, S., Bakas, S., Galtier, M. N., Landman, B. A., Maier-Hein, K., et al. (2020). The future of digital health with federated learning. *NPJ digital medicine*, 3(1):119.

[Rosenschein and Zlotkin, 1994] Rosenschein, J. S. and Zlotkin, G. (1994). *Rules of encounter: designing conventions for automated negotiation among computers*. MIT press.

[Rozenberg et al., 2012] Rozenberg, G., Bäck, T., and Kok, J. N. (2012). *Evolutionary Strategies*, pages 673–698. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Ryan et al., 1998] Ryan, C., Collins, J. J., and Neill, M. O. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer.

[Sabongari et al., 2020] Sabongari, N. A., Gital, A. Y., Boukari, S., Ja'afaru, B., Ahmed, M. A., and Chiroma, H. (2020). A review on virtual machine positioning and consolidation strategies for energy efficiency in cloud data centers. *International Journal of Advanced Computer Science and Applications*, 11(6).

[Samarasinghe et al., 2022a] Samarasinghe, D., Barlow, M., Lakshika, E., and Kasmarik, K. (2022a). Grammar-based autonomous discovery of abstractions for evolution of complex multi-agent behaviours. *Swarm and Evolutionary Computation*, page 101106.

[Samarasinghe et al., 2022b] Samarasinghe, D., Barlow, M., Lakshika, E., and Kasmarik, K. (2022b). Grammar-based cooperative learning for evolving collective behaviours in multi-agent systems. *Swarm and Evolutionary Computation*, 69:101017.

[Scott and Luke, 2019] Scott, E. O. and Luke, S. (2019). Ecj at 20: toward a general metaheuristics toolkit. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1391–1398.

[Sedaghat et al., 2014] Sedaghat, M., Hernández-Rodriguez, F., and Elmroth, E. (2014). Autonomic resource allocation for cloud data centers: A peer to peer approach. In *2014 International Conference on Cloud and Autonomic Computing*, pages 131–140. IEEE.

[Shapiro and Wilk, 1965] Shapiro, S. S. and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611.

[Shapley, 1953] Shapley, L. S. (1953). Stochastic games. *Proceedings of the national academy of sciences*, 39(10):1095–1100.

[Shoham et al., 2003] Shoham, Y., Powers, R., and Grenager, T. (2003). Multi-agent reinforcement learning: a critical survey. Technical report, Technical report, Stanford University.

[Smith, 1980] Smith, R. G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, (12):1104–1113.

[Sterman, 2000] Sterman, J. (2000). *Business dynamics*. Irwin/McGraw-Hill c2000..

[Subhan et al., 2023] Subhan, F., Mirza, A., Su'ud, M. B. M., Alam, M. M., Nisar, S., Habib, U., and Iqbal, M. Z. (2023). Ai-enabled wearable medical internet of things in healthcare system: A survey. *Applied Sciences*, 13(3):1394.

[Sukhbaatar et al., 2016] Sukhbaatar, S., Fergus, R., et al. (2016). Learning multiagent communication with backpropagation. *Advances in neural information processing systems*, 29.

[Sutton, 1984] Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[Sutton et al., 1998] Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

[Tan, 1993] Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337.

[Tavafi and Banzhaf, 2017] Tavafi, A. and Banzhaf, W. (2017). A hybrid genetic programming decision making system for robocup soccer simulation. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1025–1032.

[Taylor et al., 2007] Taylor, M. E., Stone, P., and Liu, Y. (2007). Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(9).

[Toth and Martello, 1990] Toth, P. and Martello, S. (1990). *Knapsack problems: Algorithms and computer implementations*. Wiley.

[Trianni and Nolfi, 2011] Trianni, V. and Nolfi, S. (2011). Engineering the evolution of self-organizing behaviors in swarm robotics: A case study. *Artificial life*, 17(3):183–202.

[Tung et al., 2021] Tung, T.-Y., Pujol, J. R., Kobus, S., and Gunduz, D. (2021). A joint learning and communication framework for multi-agent reinforcement learning over noisy channels. *arXiv preprint arXiv:2101.10369*.

[Van Berkel et al., 2012] Van Berkel, S., Turi, D., Pruteanu, A., and Dulman, S. (2012). Automatic discovery of algorithms for multi-agent systems. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 337–344.

[van Moergestel et al., 2013] van Moergestel, L., Puik, E., Telgen, D., and Meyer, J.-J. (2013). Embedded autonomous agents in products supporting repair and recycling. In *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 1–8. IEEE.

[Varshavskaya et al., 2009] Varshavskaya, P., Kaelbling, L. P., and Rus, D. (2009). Efficient distributed reinforcement learning through agreement. In *Distributed Autonomous Robotic Systems 8*, pages 367–378. Springer.

[Vojnović and Proutiere, 2011] Vojnović, M. and Proutiere, A. (2011). Hop limited flooding over dynamic networks. In *2011 Proceedings IEEE INFOCOM*, pages 685–693. IEEE.

[Vrajitoru, 2000] Vrajitoru, D. (2000). Large population or many generations for genetic algorithms? implications in information retrieval. In *Soft Computing in Information Retrieval*, pages 199–222. Springer.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

[Watkins, 1989] Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, King's College, Cambridge United Kingdom.

[Weise and Tang, 2011] Weise, T. and Tang, K. (2011). Evolving distributed algorithms with genetic programming. *IEEE Transactions on Evolutionary Computation*, 16(2):242–265.

[Weise and Zapf, 2009] Weise, T. and Zapf, M. (2009). Evolving distributed algorithms with genetic programming: Election. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 577–584. ACM New York, NY, USA.

[Weiss, 1999] Weiss, G. (1999). *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press.

[Whitley, 1994] Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85.

[Wilcoxon, 1992] Wilcoxon, F. (1992). Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer.

[Wolsey, 2020] Wolsey, L. A. (2020). *Integer programming*. John Wiley & Sons.

[Wooldridge, 2009] Wooldridge, M. (2009). *An introduction to multiagent systems*. John wiley & sons.

[Wooldridge and Ciancarini, 2000] Wooldridge, M. and Ciancarini, P. (2000). Agent-oriented software engineering: The state of the art. In *International workshop on agent-oriented software engineering*, pages 1–28. Springer.

[Wuhib et al., 2015] Wuhib, F., Yanggratoke, R., and Stadler, R. (2015). Allocating compute and network resources under management objectives in large-scale clouds. *Journal of Network and Systems Management*, 23(1):111–136.

[Xie and Liu, 2008] Xie, X.-F. and Liu, J. (2008). Multiagent optimization system for solving the traveling salesman problem (tsp). *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):489–502.

[Yazir et al., 2010] Yazir, Y. O., Matthews, C., Farahbod, R., Neville, S., Guitouni, A., Ganti, S., and Coady, Y. (2010). Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 91–98. Ieee.

[Zendaoui and Layeb, 2016] Zendaoui, Z. and Layeb, A. (2016). Adaptive cuckoo search algorithm for the bin packing problem. In *Modelling and Implementation of Complex Systems*, pages 107–120. Springer.

[Zeng et al., 2020] Zeng, S., Xu, X., and Chen, Y. (2020). Multi-agent reinforcement learning for adaptive routing: A hybrid method using eligibility traces. In *2020 IEEE*

16th International Conference on Control & Automation (ICCA), pages 1332–1339. IEEE.

[Zhang et al., 2019] Zhang, J., Wang, G., and Song, Y. (2019). Task assignment of the improved contract net protocol under a multi-agent system. *Algorithms*, 12(4):70.

[Zhang et al., 2018a] Zhang, K., Yang, Z., and Basar, T. (2018a). Networked multi-agent reinforcement learning in continuous spaces. In *2018 IEEE conference on decision and control (CDC)*, pages 2771–2776. IEEE.

[Zhang et al., 2021] Zhang, K., Yang, Z., and Başar, T. (2021). Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, pages 321–384.

[Zhang et al., 2018b] Zhang, K., Yang, Z., Liu, H., Zhang, T., and Basar, T. (2018b). Fully decentralized multi-agent reinforcement learning with networked agents. In *International Conference on Machine Learning*, pages 5872–5881. PMLR.

[Zhang et al., 2010] Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.

[Zhang et al., 2022] Zhang, Z., Liu, H., and Wu, G. (2022). A dynamic task scheduling method for multiple uavs based on contract net protocol. *Sensors*, 22(12):4486.

[Zhong et al., 2014] Zhong, J., Luo, L., Cai, W., and Lees, M. (2014). Automatic rule identification for agent-based crowd models through gene expression programming. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1125–1132.

[Zitzler and Thiele, 1999] Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257–271.