Multi-Policy Optimization in Decentralized Autonomic Systems

Ivana Dusparic

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

June 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work. I agree that Trinity College Library may lend or copy this thesis upon request.

Ivana Dusparic

Dated: May 26, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ivana Dusparic

Dated: May 26, 2010

Acknowledgements

First and foremost I'd like to thank my supervisor, Prof. Vinny Cahill, for his guidance, expertise, encouragement, and endless patience during my Ph.D. studies.

The other main acknowledgement goes to As'ad Salkham for all his help throughout this process; for providing RL libraries, for numerous fruitful discussions on RL and UTC, and for sharing both joys and frustrations of traffic simulation. I'd also like to express my gratitude to Vinny Reynolds, the author of Dublin traffic simulator, and everyone else who has worked on extending it over the years: Raymond Cunningham, Anurag Garg, Sylvain Cabrol and Mikhail Volkov. A big thank you to Mark Gleeson and Andrew Jackson for the help with computing resources.

Thanks to all past and present members of DSG for making it such a great group to work in and for setting great examples for me to follow.

Thanks to Lero, the Irish Software Engineering Research Centre, for sponsoring my research.

I'd also like to thank all of my friends for encouraging me, each in their own way, from close and afar. A special thank you to John for putting up with me over the past few months, and to Elizabeth Daly for her feedback on this thesis.

And finally, thank you to my family, for all their support, for never questioning me and for always being there.

Ivana Dusparic

University of Dublin, Trinity College

June 2010

Abstract

Autonomic computing systems are those that are capable of managing themselves based only on high-level objectives given by humans. In such systems the details of how to meet their objectives, even in the face of changing operating conditions, are left to the systems themselves. Therefore, autonomic systems are required to be able to self-optimize, self-heal, self-protect, and self-configure. Enabling autonomic behaviour is particularly challenging in decentralized autonomic systems, where central control is not tractable or even possible, due to the large number and geographical dispersion of the entities involved. In such systems, entities only have local views of their immediate environments and no global view of the system exists. Decentralized autonomic systems can be implemented as multi-agent systems, in which each entity is modelled as an intelligent agent. These agents can self-organize based only on local actions and interactions, so that the global behaviour of the system, required to meet its objectives, emerges from the agents' local behaviours.

This thesis addresses self-optimization in decentralized autonomic systems. Examples of techniques used to self-optimize autonomic systems include ant-colony optimization, evolutionary algorithms, neural networks, and reinforcement learning (RL). RL is considered particularly suitable for use in large-scale autonomic systems, as it does not require a predefined model of the environment. However, most applications of RL in decentralized autonomic computing address systems that optimize their behaviours towards only a single policy, while in reality management of most autonomic systems requires optimization towards multiple, often conflicting policies. These policies can be heterogeneous (i.e., implemented on different sets of agents, be active at different times and have different levels of priority), leading to the heterogeneity of the agents of which the system is composed. The cooperation required for self-optimization is particularly challenging in such heterogeneous multi-agent environments, as agents might not be aware of other agents' policies and their relative priority for the system. Additionally, since agents operate in the same shared environment, dependencies can arise between their performance and therefore between policy implementations as well.

To address self-optimization in such decentralized autonomic systems in the presence of agent

heterogeneity, policy dependency and lack of global knowledge, this thesis proposes Distributed W-Learning (DWL). DWL is an RL-based algorithm for agent-based self-optimization that enables collaboration between heterogeneous agents in order to simultaneously satisfy multiple heterogeneous system policies. DWL learns and exploits the dependencies between neighbouring agents and between policies to improve performance while respecting the relative priorities of the policies. Instead of always executing the locally-best action, DWL agents take into account how their actions affect their immediate neighbours; if a neighbouring agent has a very strong preference on an agent's local action, that agent will defer to it and execute the action nominated by that neighbour. In particular, suggestions by neighbouring agents will be executed if their importance exceeds the importance of the local action when scaled using a cooperation coefficient, which can be predefined, or can be learnt to maximize the reward received in the immediate neighbourhood. By selecting actions in this manner, DWL does not require central control or a global view, as it relies solely on local actions and interactions with one-hop neighbours.

We have evaluated the DWL algorithm in a simulation of an urban traffic control (UTC) system, a canonical example of the class of decentralized autonomic systems that we are addressing. We show that DWL is a suitable technique for optimization in UTC, as it outperforms the currently most-widely deployed fixed-time and simple adaptive controllers in our simulation. Collaborative DWL scenarios can outperform non-collaborative scenarios, depending on the level of collaboration, when the cooperation coefficient is fixed. When DWL agents learn the level of cooperation individually, the learnt scenarios outperform all non-collaborative scenarios and either outperform or perform as well as with predefined collaboration coefficients. We also show that addressing two policies simultaneously using DWL can, based on policy dependencies, improve the performance of both policies over corresponding single-policy implementations. These results hold for a variety of traffic loads and patterns and therefore show DWL's wide applicability in UTC, as well as suggesting that DWL might be suitable for optimization in other large-scale autonomic systems with similar characteristics.

Publications Related to this Ph.D.

- Ivana Dusparic and Vinny Cahill. Research issues in multiple policy optimization using collaborative reinforcement learning. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Washington, DC, USA, May 2007. IEEE Computer Society.

- Ivana Dusparic and Vinny Cahill. Autonomic management of large-scale critical infrastructures. In *HotAC III: 3rd Workshop on Hot Topics in Autonomic Computing, in conjunction with the IEEE International Conference on Autonomic Computing ICAC '08,* Chicago, IL, USA, June 2008.

- Ivana Dusparic and Vinny Cahill. Multi-policy optimization in decentralized autonomic systems (Extended abstract). In Decker, Sichman, Sierra, Castelfranchi, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09),* Budapest, Hungary, May 2009.

- Ivana Dusparic and Vinny Cahill. Distributed W-Learning: an algorithm for multi-policy optimization in decentralized autonomic systems (Poster). In *Proceedings of the 6th IEEE International Conference on Autonomic Computing (ICAC '09),* Barcelona, Spain, June 2009.

- Ivana Dusparic and Vinny Cahill. Using reinforcement learning for multi-policy optimization in decentralized autonomic systems - an experimental evaluation. In *Proceedings of the 6th International Conference on Autonomic and Trusted Computing (ATC '09),* Brisbane, Australia, July 2009. Lecture Notes In Computer Science,. Springer-Verlag, Berlin, Heidelberg.

- Ivana Dusparic and Vinny Cahill. Distributed W-Learning: Multi-Policy Optimization in Self-Organizing Systems. In *Proceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '09)*, San Francisco, California, September 2009.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"*When it is obvious that the goals cannot be reached, don't adjust the goals, adjust the action steps.*"

- Confucius

This thesis addresses multi-policy self-optimization in large-scale heterogeneous autonomic systems. It proposes Distributed W-Learning (DWL), an algorithm based on Reinforcement Learning (RL) that enables simultaneous self-optimization of autonomic system behaviour towards multiple policies. DWL learns and exploits the dependencies between multiple policies and between neighbouring agents comprising an autonomic system to select optimal actions, while respecting relative priority of policies. DWL does not require global knowledge or centralized system control as all actions and interactions are performed locally at the agent-level, so that global system behaviour emerges from these interactions. We evaluate DWL in a simulation of an urban traffic control (UTC) system, a canonical example of large-scale decentralized autonomic systems. This chapter motivates the work, introduces multi-policy multi-agent optimization and DWL, outlines the main contributions of this work, and presents a roadmap for the remainder of the thesis[1].

1.1 Autonomic Systems

Autonomic computing systems are systems that are capable of managing themselves based only on high-level goals given by humans (Kephart & Chess, 2003). In such systems the details of how to

[1] Parts of this chapter are based on material appearing in (Dusparic & Cahill, 2009a,b)

meet their goals, even in the face of changing operating conditions, are left to the systems themselves. Therefore, autonomic systems are required to be able to self-optimize, self-heal, self-protect, and self-configure in order to meet the goals. Enabling autonomic behaviour is particularly challenging in decentralized autonomic systems (Wolf & Holvoet, 2007), where central control is not tractable, due to the large number and geographical dispersion of the entities involved. Important examples of such systems that could benefit from autonomic management include large-scale critical infrastructures (transportation networks, electricity, gas, and water supply), as well as numerous other applications that involve, for example, scheduling, task allocation, routing, or load balancing, such as global supply chains (OLeary, 2008). A decentralized autonomic system can be implemented as a group of agents (Tesauro et al., 2004), or so-called autonomic elements (Kephart & Chess, 2003). These agents can self-organize based only on local actions and interactions, so that the global behaviour of the system, required to meet its goals, emerges from the agents' local behaviours. A number of techniques that support self-organization and emergence have been used to implement autonomic characteristics. Examples include ant-colony optimization in load balancing (Montresor et al., 2002), particle swarm optimization in wireless network routing (Kadrovach & Lamont, 2002) and digital evolution in autonomous robot navigation (Goldsby et al., 2008). RL is considered particularly suitable for self-optimization in large-scale heterogeneous environments, as it does not require a model of the environment, which, due to the scale and complexity of such systems, is time-consuming and complex to construct (Abdulhai et al., 2003; Tesauro, 2007). Additionally, RL is capable of taking into account the long-term consequences of the actions selected, enabling the system to learn not only the immediate payoffs of its actions, but also the best actions for the long-term performance of the system (Tesauro, 2007). RL has already been successfully used in self-organizing autonomic systems (e.g., for load balancing (Dowling, 2005) and resource allocation (Tesauro et al., 2006)), however, these implementations focus mostly on a single system policy (or multiple policies expressed using a single learning process and a single reward signal), while most multi-policy RL techniques have so far been implemented only on a single agent (e.g., (Cuayahuitl et al., 2006; Humphrys, 1996a)). In this thesis we argue that, in order to be more widely utilized as a technique for self-optimization of heterogeneous decentralized autonomic systems, RL needs to be capable of addressing optimization in environments where both multiple policies and multiple agents coexist simultaneously. Such environments pose a number of challenges that multi-policy multi-agent technique needs to address. We outline these challenges in section 1.3.

1.2 Policies, Goals, Objectives and Criteria

Before we move onto issues associated with multi-policy optimization in autonomic system, we need to define the term "*policy*" as it will be used in this thesis. As autonomic systems need to self-optimize their behaviour towards high-level goals, a means of expressing those high-level goals is required. This is achieved by the use of policies (White et al., 2004). In policy-based management, a policy has been defined as "a representation, in a standard external form, of desired behaviors or constraints on behavior" (White et al., 2004) or as "a definite goal, course or method of action to guide and determine present and future decisions" (Westerinen et al., 2001). In this thesis, we adopt a broader, more lax definition of a policy used in autonomic computing, as defined in (Kephart & Walsh, 2004):

> "a policy is any type of formal behavioural guide".

This definition is sufficient for the purposes of this thesis, as we do not address issues related to policy specification (Ganek & Corbi, 2003), but focus on balancing multiple goals expressed as policies. Therefore, a policy is a formal expression of a system goal, while we use the word "*goal*" more informally to denote "an aim or desired result" (Oxford, 2000) of some system's behaviour.

Examples of policies that we use in the evaluation of DWL in UTC and that satisfy the above definition are "optimize global vehicle waiting time" and "prioritize public transport vehicles".

However, as this thesis uses RL on an agent-level to learn how to meet the high-level goals of autonomic systems, we will also need to refer to an agent's policy Π as defined in RL (Suton & Barto, 1998), i.e.,

> "a mapping from the states to probabilities of selecting each possible action".

Therefore, the meaning of the word "*policy*" in this thesis is two-fold, and refers to both a formal expression of a high-level system goal, and to an agent's local state-action mappings used to meet that high-level goal on an agent level, depending on the context.

In Chapter 2, when discussing work related to this thesis, we retain the terminology that the original work uses and therefore also use terms multi-*objective* (as used in, e.g., (Angus & Woodward, 2009; Tan et al., 2005)), multi-*goal* (as used in, e.g., (Cuayahuitl et al., 2006; Gadanho & Hallam, 2001; Karlsson, 1997)), and multi-*criteria* optimization (as used in, e.g., (Hiraoka et al., 2008; Natarajan & Tadepalli, 2005)) to refer to the problem of meeting multiple system goals.

1.3 Issues in Multi-Policy Multi-Agent Self-Optimization

In this section we introduce and analyze properties of policies that could be present in autonomic systems. We also investigate the impact that different policy characteristics and different environment characteristics can have on agents comprising an autonomic system. Specifically, we discuss issues of policy and agent heterogeneity, policy and agent dependency, as well as consequences that heterogeneity and dependency might have on agent collaboration.

1.3.1 Policy Heterogeneity

The policies that large-scale decentralized systems are required to implement can have different characteristics. For example, policies can be deployed on different sets of agents (i.e., have different spatial scope), can be active at different times (i.e., have different temporal scope), and can have different levels of importance to the system (i.e., have different priorities). In terms of spatial scope, policies can be local (deployed on a single agent), regional (deployed on only a subset of the agents in the system) or global (deployed on all of the agents in the system), as illustrated in Figure 1.1.

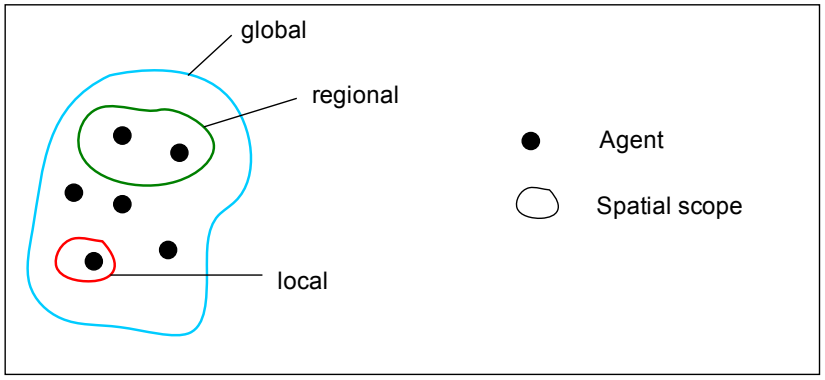


Fig. 1.1: Policy spatial scope

In terms of their temporal scope policies can be sporadic (i.e., active only at certain times during the system operation) or continuous (i.e., active during the whole system operation). Their priority can range from low to high, where the hierarchy of priorities might not be known to agents implementing the policies, and can change over time. For example, in UTC, a public transport vehicle could be given a higher priority than usual if it is running behind a schedule and if it is carrying a large number of passengers.

Additionally, policies can have different relationships to one another even if their characteristics are otherwise the same. For example, two policies might both be regional, continuous policies of the

same priority. However, due to their regional scope, the sets of agents on which they are deployed can be the same, can overlap, or can be disjoint, as shown in Figure 1.2.

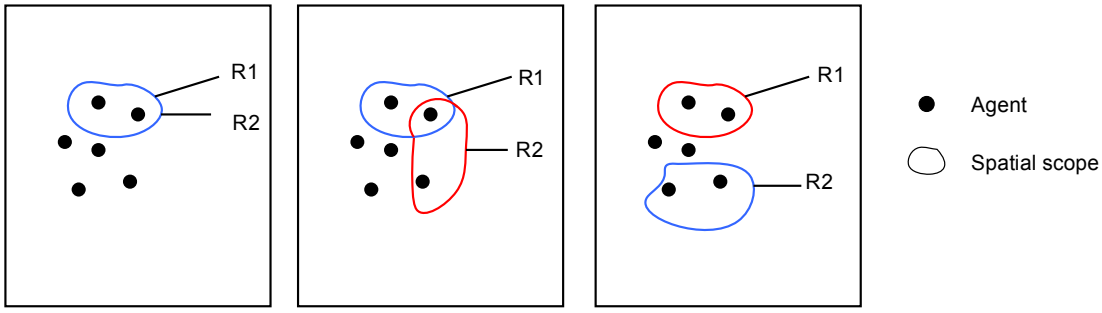


Fig. 1.2: Policy relationships

Heterogeneity of policies and their characteristics leads to heterogeneity of agents while different policy relationships can lead to different levels of dependency between policies and between agents. Therefore, multi-policy optimization techniques in multi-agent environments need to not only explicitly address meeting multiple system policies, but also address the implications of the presence of multiple policies, namely agent dependency and heterogeneity, as well as policy dependency. We discuss these implications in the following sections.

1.3.2 Agent Dependency

The performance of an agent in a multi-agent system can be influenced, directly or indirectly, and both positively and negatively, by other agents' actions (Rosenschein & Zlotkin, 1994). For example, in a UTC system, the performance of one junction can be affected by some or all of its upstream and/or downstream neighbours.

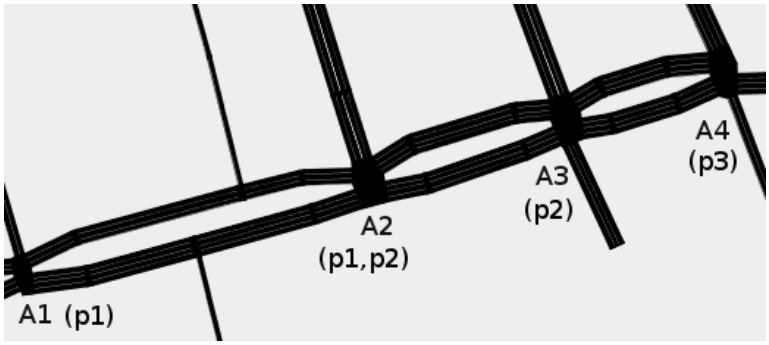


Fig. 1.3: Agent and policy heterogeneity

For an example, consider Figure 1.3 which shows several linked junctions controlled by agents. If the junction controlled by agent A2 is oversaturated, traffic can come to a standstill and queues at that junction can spill over upstream to block the junction controlled by A3. No traffic will then be able to go through this junction regardless of the actions of A3, as there will be no space available on the downstream road. Likewise, the performance of A1, A3, and any agents at other junctions that feed traffic to A2 can cause oversaturation at A2 if they are letting through more traffic than A2 is clearing. The dependency can also extend to the agents further upstream from A2, such as A4, as that agent influences the performance of A3, which in turn influences the performance of A2, causing a potential dependency between non-neighbouring agents A2 and A4. As a result, there is a potential dependency between all of the agents in a UTC system. Due to this dependency, agents that control junctions should, when selecting an action to execute, consider not just their own direct benefit, but also the influence of that action on other agents, particularly their immediate neighbours.

1.3.3 Agent Heterogeneity

It is particularly difficult for agents to take the needs of other agents into account when agents are heterogeneous and implement different policies. The first source of heterogeneity is the difference in the agents' operating environment and capabilities. For example, in a UTC scenario, intersections can have different layouts, i.e., the number of incoming and outgoing approaches and the allowed traffic maneuvers can be different. In RL, this maps to agents having different state spaces and different action sets, since the combinations of traffic signal settings (i.e., traffic-light phases) available to agents controlling junctions of different layout differ. This results in agents not having a common interpretation of the meaning of particular states or actions.

Another source of heterogeneity is a result of different policies being deployed in the system. For example, agents in a UTC system might be required to optimize global traffic flow, but also to prioritize emergency vehicles and public transport, or to deal with pedestrian crossing requests. Due to the different spatial and temporal scope of these policies, agents in the system may be required to implement different sets of policies at a given point in time. For example, consider again Figure 1. Agent A2 is required to implement policies p1 and p2, while one of its neighbours, agent A1, is only implementing p1, and the other neighbouring, agent A3, only p2. The further downstream neighbour A4, implements only its own local policy p3. Therefore, agents can be heterogeneous both in terms of their capabilities, and in terms of policies they implement, but they should nevertheless consider the influence of their actions on their neighbours, regardless of neighbours' capabilities or policies they implement.

1.3.4 Policy Dependency

Agent dependency and agent heterogeneity, as discussed in previous sections, can cause dependencies between policies as well. For example, we discussed how in Figure 1 the actions of A3 can influence the performance of agents A2 and A4. This implies that the implementation of the policy p2 (implemented by A3) can influence the performance of policies p3 (implemented by A4) and p1 (implemented by A2). Therefore, when an agent makes an action selection, it should not only consider actions that are optimal for the implementation of its own policies, but also the effect of those actions not only on other agents, but also on other policies, particularly on those that its immediate neighbours are implementing.

1.3.5 Cooperation in Heterogeneous Environments

Due to policy and agent dependency in large-scale agent-based systems, optimal performance of the system as a whole might not be as simple as optimizing the performance of all entities individually, but agents might be required to cooperate with each other. Dependencies between agents need to be properly managed to avoid harmful interactions and account for non-local effects of local agent's actions (Sycara, 1998). It has been established that various forms of cooperation between agents are able to improve the performance of the overall system over performance of independent agents (Tan, 1993; Vlassis, 2007).

However, in order to implement cooperation, a way to motivate an agent to cooperate is needed, as, in terms of local reward received, it might be better for an agent to act selfishly. This is particularly challenging if agents implement different policies; we need to motivate an agent to sacrifice its local reward and sacrifice performance towards its own policies, to help other agents meet their policies. It is particularly important for an agent to be willing to engage in cooperative behaviour when other agents' policies have a higher priority for the system than the policies that the local agent is implementing. Even when agents are motivated to engage in cooperation with their neighbours, we need to address the issue of how they should cooperate, i.e., what information should agents exchange and how will that information be interpreted. Heterogeneity of environments and policies makes this particularly difficult. For example, if agents only exchange their latest rewards (Tan, 1993), a receiving agent might not know why a sending agent received that reward, for which policy on which agent was it received, for which particular state, or for which action taken. Heterogeneous agents are not able to exchange learnt experiences either, as their state spaces and actions differ, so the knowledge acquired at one agent is not applicable to other agents with different state-action pairs.

Once an agent is motivated to cooperate, and has a means to do so, it needs to determine with

whom to cooperate. For example, it might need to cooperate only with other agents implementing the same policies as its local ones, or with all agents regardless of their policies. However, if agents only have local views they might not be aware of which other agents, if any, are implementing the same policies as they are. Additionally, agents implementing a low-priority policy might need to help agents that implement higher-priority policies. Due to the lack of a global view, agents might not know the relative priorities of their local policies and other agents' policies. As the levels of dependency between agents might differ, an agent might only need to collaborate with other agents whose actions it is influenced by, and agents that are influenced by its local actions. Also, once agents know which other agents to collaborate with, they need to determine the extent and frequency of that collaboration; for example, should they always collaborate or only in certain situations, and should they collaborate fully, honoring all the collaboration requests, or only those that satisfy certain criteria.

1.3.6 Summary

In this section we have analyzed the characteristics of policies that large-scale multi-agent systems might be required to implement, as well as characteristics of the environment that agents are situated in. We conclude that, due to the shared policy deployment environment and shared agent operating environment, techniques for multi-policy multi-agent optimization need to enable collaboration between policies as well as between agents, as the performance of one policy/agent can affect performance of other policies/agents. Due to different policy and environment characteristics, both policies and agents within the system can be heterogeneous. Therefore, the required collaboration needs to be enabled not just between homogeneous policies and agents, but between all policies and agents regardless of policy characteristics or agent capabilities. We expand on these observations in Chapter 3 and Chapter 4 and use them to motivate the design of DWL.

1.4 Thesis Aims and Objectives

The general objective of the thesis is to address the gap in RL-based self-optimization techniques, whereby existing techniques address either multiple policies on a single agent, or address collaborative optimization in multi-agent systems, but towards only a single policy. We believe that, if RL-based techniques are to be fully utilized as self-organizing techniques for optimization in decentralized autonomic systems, they need to be capable of optimizing towards multiple heterogeneous policies on multiple heterogeneous agents simultaneously. Such an approach could improve the performance of the overall system towards all of its policies, as policies can learn in each other's presence and learn

the impact that they have one on another, i.e., the dependencies between them. Additionally, such a technique needs to enable collaboration between heterogeneous agents, as heterogeneous policies of different scope introduce agent heterogeneity. This thesis argues that RL is a suitable basis for such a technique, due its ability to learn suitable behaviours without requiring a model of the environment, and ability to incorporate long-term effects of actions selected into its learning process. This thesis analyzes requirements for such an RL-based multi-agent multi-policy technique for optimization in large-scale heterogeneous environments, presents the design and implementation of a suitable technique, and evaluates it in the simulation of UTC.

1.5 Distributed W-Learning

This thesis introduces Distributed W-Learning (DWL), an RL-based multi-policy optimization technique that enables collaboration between heterogeneous agents to meet system-wide policies. DWL uses W-learning, a technique for multi-policy action selection on a single agent (Humphrys, 1996b). In W-learning, each policy is implemented as a separate Q-Learning (Watkins & Dayan, 1992) process with its own state space. Agents learn Q-values (Watkins & Dayan, 1992) for state-action pairs for each policy and, at every time step, each policy nominates an action based on these Q-values. Using W-Learning, agents also learn, for each of the states of each of their policies, what happens, in terms of reward received, if the action nominated by that policy is not obeyed. This is expressed as a W-value (Humphrys, 1996b).

In DWL, as well as Q-values and W-values for all of their local policies, all agents also learn Q-values and W-values for all of the policies that their immediate neighbours implement (so-called remote policies), i.e., they learn how their local actions affect their neighbours' states. At each time step, each agent considers the W-values for the current state of each of its local and remote policies, as shown in Figure 1.4. If any of the immediate neighbours' policies has a higher W-value than the agent's local W-values, the action suggested by that neighbour can be executed. Neighbours' W-values can be multiplied by a cooperation coefficient C, to enable a local agent to give a varying degree of importance to the neighbours' action suggestions. C can range from a fully non-cooperative value, C=0, where an agent does not consider neighbours' suggestions at all, to a fully cooperative, C=1, where neighbours' suggestions matter as much as local ones. C can be predefined or can be learnt at runtime, so as to maximize the rewards received on all agent's local and remote policies. In this way DWL enables collaboration (by enabling action suggestions by neighbours' policies) between heterogeneous agents regardless of the policies that they implement. It is decentralized and self-organizing as it requires

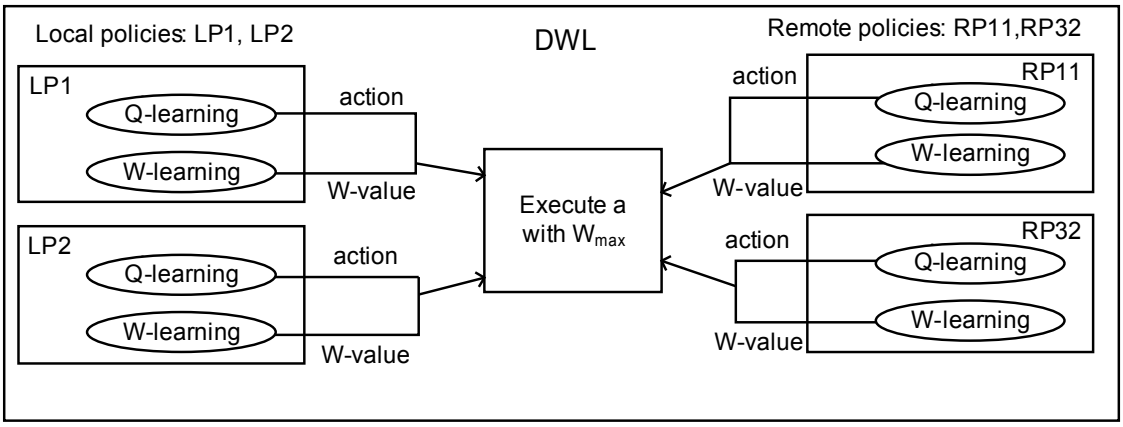


Fig. 1.4: DWL action selection on a single agent

only local actions and interactions with immediate neighbours, and as such is, we believe, suitable for implementation in large-scale decentralized autonomic systems.

1.6 Urban Traffic Control

We have evaluated the performance of DWL in a simulation of a decentralized autonomic UTC system. Existing centralized UTC optimization techniques are failing to deal with the pressure of high traffic loads and new decentralized adaptive learning techniques are being investigated to deal with increasing traffic congestion and the resulting cost and pollution (Salkham et al., 2008). UTC systems are large-scale (i.e., consisting of hundreds of traffic lights), highly dynamic (i.e., traffic conditions can unexpectedly change due to accidents, road closures, traffic light failures), need to operate under variety of contexts (i.e., peak hour traffic, surge of traffic from/to particular areas due to major events etc) and are heterogeneous (i.e., traffic lights control junctions of different layouts with different traffic-control signal settings available to them based on those layouts). We believe UTC systems can be improved by being implemented as decentralized autonomic systems, with traffic lights, or groups of traffic lights controlling a single junction, represented as autonomic elements. A traffic light should be able to observe its environment using sensors (e.g., traffic cameras, inductive loops, GPS information, and car-to-car and car-to-vehicle communication), analyze the traffic conditions, and learn, select and execute the most appropriate action (i.e., traffic-signal setting) for the given environmental conditions.

UTC also makes a suitable domain for the evaluation of multi-policy optimization techniques, as UTC systems need to meet multiple, heterogeneous, and often conflicting, policies during their operation. For example, UTC systems need to optimize general traffic flow, while also prioritizing

public transport and emergency vehicles. These policies are heterogeneous in that they have different levels of priority (i.e., emergency vehicles have higher priority than private vehicles), different temporal scope (i.e., they occur with different frequencies, with emergency vehicles being present in the system less often than private vehicles), and different spatial scope (i.e., they are present in different parts of the system, with public transport vehicles being present only on certain routes, while private vehicles are spread throughout the system). However, existing research on decentralized learning techniques in UTC focuses primarily on improving general traffic flow, without accommodating simultaneous learning for other policies, such as those required by pedestrians, emergency vehicles, and public transport vehicles (see Chapter 2 for more details). We believe that implementing UTC systems using multi-policy multi-agent RL-based technique, i.e., addressing requirements of all traffic participants simultaneously, can improve the performance and adaptivity of UTC systems.

1.7 Thesis Assumptions

In designing and evaluating DWL, this thesis makes a number of assumptions about the environment in which DWL is to be deployed. DWL design assumptions limit the scope of the thesis by limiting the number of issues that DWL addresses, while the evaluation environment assumptions are imposed by the capabilities and limitations of the UTC simulation testbed used.

In this thesis, agents are assumed to be stationary, i.e., their locations are fixed and they do not move through the environment. This is the case in our evaluation area, as agents are associated with traffic lights, which are stationary. Agents are also assumed to be failure-free, and hence issues arising from agents not being able to contact their neighbours, or agents receiving incomplete or inaccurate information are not addressed. In the simulations in this thesis, all agents are bootstrapped with a list of their one-hop neighbours, which remains constant throughout the experiment as agents are stationary and failure-free. This thesis, therefore, does not address neighbour-discovery mechanisms.

System time on all agents in the system is synchronized, and the agents are assumed to make decisions simultaneously at fixed time intervals (or multiples of a minimum set interval). This thesis does not investigate how asynchronous decision making would impact on the design and performance of DWL.

Rewards received by all of the evaluated policies are assumed to be comparable and reflect their priority. If the rewards were not comparable, DWL would need to be extended with mechanisms to scale the received rewards in a manner that reflects their relative priorities.

In the simulations performed in this thesis, the behaviour of traffic lights is determined by the

simulated control mechanism (DWL or the baselines), while the behaviour of cars, i.e., their starting position, route, and destination, are predefined. A consequence of this characteristic of the simulation environment is that, if there is no available road space for cars to join the simulation at the junction specified as their starting position, they are not inserted into the simulation. This behaviour is discussed further in Chapter 3 and Chapter 6 when discussing evaluation metrics.

1.8 Thesis Contribution

This thesis identifies and motivates the need for an RL-based multi-agent multi-policy self-optimization technique. It presents the challenges of multi-policy optimization, and based on them proposes the requirements for such a technique. The main contribution of the thesis is the design, implementation and evaluation of DWL, an algorithm for multi-policy multi-agent self-optimization in large-scale decentralized autonomic systems, that satisfies these requirements. Unlike existing RL-based techniques, DWL enables simultaneous optimization towards the multiple policies that a system is required to implement. This is crucial for wider adoption of RL-based techniques for self-optimization in autonomic systems, as most of such systems have multiple performance goals and policies. DWL enables collaboration between agents regardless of the policies they implement, enabling optimization in heterogeneous environments. DWL learns and takes advantage of the dependencies between policies, to improve their performance, while respecting policy priorities. DWL enables agents to engage in different levels of collaboration, where that level can be predefined or learnt to maximize the rewards received by a group of agents engaging in collaboration. Furthermore, DWL requires only local actions and interactions with immediate agent neighbours, enabling decentralized self-optimization in large-scale systems without a global view, central control, or predefined environment model. This thesis evaluates DWL in the simulation of UTC. The evaluation shows that DWL is suitable for application in UTC, as it outperforms existing UTC techniques. DWL is suitable for collaboration in heterogeneous multi-policy environments, as collaborative deployments using a predefined value of C can outperform non-collaborative ones (depending on the value of C), and DWL collaborative deployments that learn suitable values of C can outperform non-collaborative deployments, as well as outperform, or perform as well as, deployments with predefined values of C. Addressing multiple policies simultaneously using DWL improves the overall performance of the system, by either improving the performance of both policies when compared to their corresponding single-policy deployments, or by having a small negative effect on one policy to ensure that the performance of the other policy is not neglected. We also show that, even though it is primarily designed for multi-policy optimization, DWL can also be

applied in single-policy systems to improve their performance by enabling collaboration between the agents.

1.9 Roadmap

The structure of the remainder of the thesis is as follows. Chapter 2 presents background material and related research in the field. It introduces autonomic computing systems and existing self-organizing techniques utilized in their implementation. It focuses on RL-based algorithms, providing background on the techniques and a survey of multi-agent and multi-policy techniques. It introduces our evaluation domain, UTC, providing a glossary of the UTC terminology used in the evaluation, as well as reviews of existing UTC techniques. Chapter 3 presents a case-study on non-collaborative multi-policy optimization using existing RL-based multi-policy techniques, based on which we have derived requirements for DWL. Chapter 4 motivates and describes the design of DWL. Chapter 5 presents the implementation of DWL as an extension of an existing RL-based optimization framework. Chapter 6 describes the evaluation of DWL as a multi-agent multi-policy self-optimization technique in heterogeneous large-scale autonomic environments and analyzes the findings. Chapter 7 concludes this thesis with the summary of the work and outlines the issues that remain open for future work.

Chapter 2

Autonomic Computing

"*Everything that can be invented has been invented.*"

- falsely attributed to Charles H. Duell,
United States Patent and Trademark Office Commissioner (1899)

In this chapter we introduce the autonomic computing domain and its characteristics. We focus on decentralized autonomic systems and some of the techniques used to implement them, in particular focusing on multi-agent systems and self-organizing algorithms. We discuss reinforcement learning (RL) in detail, in order to provide the necessary background for understanding Distributed W-Learning (DWL), the multi-agent multi-policy RL-based optimization algorithm that this thesis proposes, as well as to position DWL in relation to existing work. Finally, we discuss the use of RL in our evaluation domain, urban traffic control (UTC).

2.1 Introduction to Autonomic Computing

The management of computing systems is becoming increasingly difficult due to their size, geographic dispersion, heterogeneity, and inter-connectivity (Kephart & Chess, 2003). To address installation, configuration, optimization and maintenance of such complex large-scale systems, IBM has proposed the concept of autonomic computing (Kephart & Chess, 2003). In autonomic systems, the need for human intervention is removed, as such systems have the capability to self-manage and adapt their behaviour according to high-level goals, even in dynamic environments (IBM, 2005). The concept of autonomic computing is modelled on the human autonomic nervous system, which governs body

temperature, breathing, and heart rate, therefore freeing the conscious brain from dealing with those low-level functions. Analogously, autonomic computing systems only require the specification of high-level goals from their users and administrators, while the details of how to accomplish those goals are left to the systems themselves (Kephart, 2005).

The ability of an autonomic system to self-manage is based on several so-called self-* (self-star) properties: self-optimization, self-configuration, self-healing, and self-protection (Ganek & Corbi, 2003). The property of autonomic systems on which this thesis focuses is self-optimization. Autonomic systems may need to continually self-optimize towards some high-level goals and learn the optimal behaviours that meet those goals for any given set of operating conditions. Self-configuration refers to a capability of autonomic systems to configure themselves, including installation and configuration of new components, automatically. Self-healing refers to autonomic systems being able to detect, diagnose and repair any operating problems, and self-protection to the ability of autonomic systems to protect themselves from malicious attacks or cascading failures uncorrected by the self-healing measures (Kephart & Chess, 2003).

Autonomic systems can be controlled in a centralized or decentralized manner. The decentralized approach is increasingly being utilized due to the large-scale and geographical dispersion of elements of autonomic systems rendering traditional centralized or hierarchical management arhictectures infeasible and intractable (Tesauro et al., 2004). Decentralized autonomic systems are defined as systems constructed as a group of locally interacting autonomous entities that cooperate to maintain desired system-wide behaviour and properties without central control (Wolf & Holvoet, 2007). Such autonomous entities are called autonomic elements (IBM, 2005). Autonomic elements consist of a managed element and an autonomic manager (see Figure 2.1). The autonomic manager has four main functions: monitor the environment, analyze the current performance of the managed element as well as the current environmental conditions and predict future system behaviours, plan actions required to meet the goals given the current conditions, and execute those actions. By performing these actions in an automated way, an autonomic element forms an intelligent control loop (IBM, 2005).

In the next section we discuss the techniques and algorithms that are used to implement autonomic elements and to build decentralized autonomic computing systems.

2.2 Engineering Decentralized Autonomic Computing Systems

In order to enable the implementation of self-* behaviours, the autonomic computing field draws on and extends research and technology from multiple scientific disciplines such as artificial intelligence,

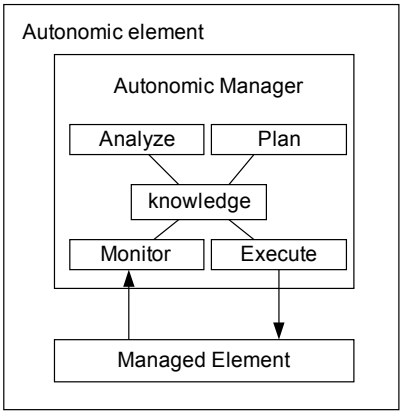


Fig. 2.1: Autonomic element (IBM, 2005)

distributed systems, network management, fault-tolerant computing, requirements engineering, statistical modelling, trusted computing and others (Bustard & Sterritt, 2007; McCann & Huebscher, 2004; Sterritt, 2005; Sterritt et al., 2005). In particular, concepts from control theory (Abdelwahed & Kandasamy, 2007) and multi-agent systems (MAS) (Tesauro et al., 2004) have been used in the design of autonomic elements and the interaction between them.

Modelling an autonomic manager as a closed control loop is inspired by control theory (Diao et al., 2005), where a controller observes the performance of the entity that it is controlling and determines control signals that optimize a given performance criterion (Kirk, 2004). Often, performance needs to be optimized for multiple criteria, requiring the use of multi-objective optimization techniques. Traditionally, multi-objective optimization techniques in control theory include vector optimization, nonlinear multi-objective programming, goal programming, fuzzy multi-objective linear programming, and evolutionary algorithms (Ehrgott & Gandibleux, 2002).

However, use of control theory relies on the existence of precise and correct models of a system and accurate estimates of the effects of its operating environment (Abdelwahed & Kandasamy, 2007), which are complex and time-consuming to develop (Tesauro, 2007). To overcome this difficulty, techniques and ideas developed in the MAS community, which do not rely on complex models for their performance, such as self-organization and emergent behaviour, multi-agent learning, and agent coordination, are being utilized in the engineering of decentralized autonomic systems (Tesauro et al., 2004).

In the remainder of this section we introduce MAS and outline some of their characteristics so that we can position our multi-agent approach to multi-policy optimization using DWL. We also discuss several self-organizing MAS techniques that have been used to implement self-optimization capabilities

and as such can be applied in autonomic computing.

2.2.1 Multi-Agent Systems

A MAS is defined as "a loosely coupled network of agents that interact to solve problems that are beyond the individual capabilities or knowledge of each agent" (Sycara, 1998), where an agent, the main building block of a MAS, is defined as "a computational entity that is perceiving and acting upon its environment, and that is autonomous in that its behaviour, at least partially, depends on its own experience" (Weiss, 1999). An intelligent agent is such an agent that acts towards pursing its goals (Weiss, 1999). Additionally, agents can have the capability to learn how to achieve their goals, i.e., they can be learning agents (Russell & Norvig, 2003). The learning process carried out by an agent refers to all activities executed with the intention of meeting the particular goal (Sen & Weiss, 1999). As part of this process, agents can learn from other agents, or learn about other agents, if that information can be used to enable better local decision making (Sen & Weiss, 1999).

Various aspects of MAS have been the subject of extensive research in artificial intelligence, in particular distributed artificial intelligence (Vlassis, 2007) and game theory (Parsons & Wooldridge, 2002). MAS can differ in a number of characteristics that influence the techniques used to implement them. Agents can be cooperative (i.e., work towards a common goal) or self-interested (i.e., work only towards meeting their own individual goals) (Vlassis, 2007). Agents in a MAS can learn independently or interactively (Sen & Weiss, 1999), and interactive agents can differ in the type of information they exchange (Tan, 1993). The purpose of interaction could be to ensure locally optimal behaviours are also globally good (e.g., in (Dowling, 2005)) or to improve the quality or speed of the local learning process (e.g., in (Tan, 1993)). Information exchanged between learning agents can include current environment sensations, the result of individual learning episodes, or full learnt experiences (Tan, 1993). Agents in a MAS can be homogeneous (i.e., identical in their abilities and goals) or heterogeneous (i.e., have different goals, have different actions available to them, and have different environment models) (Stone & Veloso, 2000). If agents are heterogeneous with respect to their goals, those goals can have one of a number of relationships to each other. Goals can be compatible (i.e., completion of one does not prevent completion of the other), complementary (i.e., completion of one contributes towards completion of the other) or conflicting (i.e., completion of one prevents the full completion of the other) (Weiss, 1999). Goals can also have "side effects", where those side effects can be positive (i.e., an agent can unintentionally achieve another agent's goals) or negative (i.e., an agent, by meeting its own goal, can prevent other agents from meeting theirs) for other agents (Rosenschein & Zlotkin, 1994). Moreover, an agent might depend on some other agent to meet its goal, leading to

different levels of dependence between the agents (Wooldridge, 2002). Dependence between agents can be unilateral (i.e., one agent depends on another for meeting its goal but not vice versa), reciprocal (i.e., agents depend on one another but not necessarily with respect to the same goal), mutual (i.e., agents depend on one another with respect to the same goal), or agents can be independent.

One of the main characteristics of MAS-based techniques in which we are particularly interested is that they can exhibit self-organizing behaviour, i.e., MAS do not require a central management component or a global view of the system. All learning, actions and interactions can be performed locally by the agents, while global behaviour emerges from these actions and interactions. We discuss the concepts of self-organization and emergence in more detail in the next section.

2.2.2 Emergence and Self-Organization in Multi-Agent Systems

Emergence and self-organization are the phenomena that occur in complex adaptive systems (Flake, 2000) and can be engineered into decentralized MAS and autonomic systems. However, depending on the literature, their definitions and the distinctions given between them differ. The following does not aim to provide a new definition of emergence and self-organization, but present working definitions of both for the purposes of this thesis.

In (Wolf & Holvoet, 2004), emergence and self-organization are rigorously defined as clearly distinct phenomena, with the authors acknowledging that they frequently occur together, but also arguing that they can occur separately. The main distinction between this work and others is that here self-organization is not defined as a result of local agent interactions, but refers to any occurrence of system behaviour and structure arising without external control. Emergence, however, is defined as a result of local interactions, as a process where global behaviour or structure arises from local interactions, and where global behaviour is novel with respect to the individual agents' behaviours (Wolf & Holvoet, 2004). In (Serugendo et al., 2003), emergence is used as a part of the description of self-organizing behaviour, without discussing one occurring without the other. In (Flake, 2000; Gershenson & Heylighen, 2003), self-organization is defined as a process where global order is created from local interactions, giving it characteristics associated with emergence in (Wolf & Holvoet, 2004). (Anthony et al., 2007) distinguishes between different levels of emergence, i.e., first-order and second-order emergence. First-order emergence refers to the form of emergent behaviour considered in other literature, while second-order emergence refers to the ability of an emergent system to evolve through learning.

For the purposes of this thesis, we will refer to self-organization as a property of the system that allows it to function without external control, and emergence as a property of the system that enables

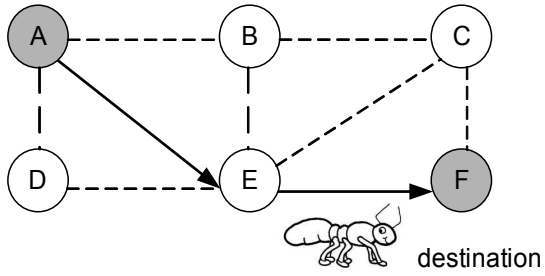


Fig. 2.2: An ant searching for the shortest path (Dorigo & Di Caro, 1999)

global behaviour to arise from purely local actions and interactions. These definitions are the closest to the views on self-organization and emergence expressed by (Wolf & Holvoet, 2004). It is important to note that, defined like this, emergence can be seen as one way for self-organizing behaviour to arise, but that self-organizing behaviour can also be engineered into the system rather than be a result of local actions and interactions. This view is shared by Babaoglu et al. (2005), who argue that autonomic systems can be implemented using self-organization which is a result of emergence, rather than explicitly programmed behaviours.

In the following sections we review some of the most commonly-used algorithms that exhibit self-organizing and/or emergent behaviours: ant colony optimization, evolutionary algorithms, particle swarms and neural networks.

2.2.2.1 Ant Colony Optimization

Ant Colony Optimization (ACO) refers to a family of optimization algorithms inspired by the behaviour of ants in an ant colony. When searching for a food source, ants in a colony converge to moving over the shortest path, among different available paths, when moving between their nest and the food source (see Figure 2.2).

This behaviour is enabled by stigmergy, a form of indirect communication between ants through the environment, realized by depositing a substance called a pheromone on the path. Trips over shorter paths get completed more quickly, causing more trips to be made on those routes and therefore more pheromone to be deposited on them. Stronger pheromone trails attract more ants, further increasing the number of ants following the route and increasing the amount of pheromone deposited on it (Dorigo & Di Caro, 1999; Maniezzo et al., 2004). Figure 2.2 illustrates this process. Ants following route A $\rightarrow$ E $\rightarrow$ F will make a round trip start$\rightarrow$ destination$\rightarrow$ start quicker than ants following A$\rightarrow$ B$\rightarrow$ C$\rightarrow$ F route, therefore biasing the path selection of further ants leaving the starting position, by depositing pheromone on the route more quickly than ants that are taking a longer trip.

In agent-based systems ants are mapped to individual agents searching for the solution to an optimization problem. Agents leave feedback about the effectiveness of their solution for other agents similar to a pheromone trail, where the amount of pheromone deposited is proportional to the quality of the solution.

ACO can also be applied to multi-objective optimization problems (Angus & Woodward, 2009). In Multi-Objective ACO (MOACO), multiple objectives are assigned different weights, either a priori, or dynamically during the search, and solutions are constructed using the specified weightings. MOACO algorithms can use a single colony of ants with a single matrix storing pheromone values, or can use a separate colony of ants for each objective and update multiple pheromone matrices. A single pheromone matrix is generally used if the relative weights of the objectives can be specified a priori, as that weighting is used to combine the quality of the solution towards multiple objectives into a single pheromone value to be stored. If the relative weights of solutions change dynamically, a separate colony of ants is assigned to each objective, and separate pheromone matrices are maintained. Information from the matrices is combined at solution construction time using the objective weightings at that particular time. If separate ant colonies are used for each objective, each colony will find the optimal solution for its own objective, leaving compromise solutions undiscovered. To address this issue, "spy" ants can be introduced (Doerner et al., 2003), that occasionally combine information gathered by all of the colonies to find trade-off solutions.

The most common applications of ACO today in autonomic systems are primarily in systems that need to converge towards the shortest (lowest cost) path and have terminating states (i.e., arriving at the food source). Examples of the use of single-policy ACO include routing in wired (Di Caro & Dorigo, 1998) and wireless networks (Di Caro et al., 2005), vehicle routing (Hoar et al., 2002), file sharing (Babaoglu et al., 2002) and load balancing applications (Montresor et al., 2002). Multi-policy ACO has been applied in vehicle routing to minimize the number of vehicles involved in delivery, total travel time, and total delivery time (Gambardella et al., 1999; Baran & Schaerer, 2003) and in goods transportation to minimize the cost of operation of pick up and delivery fleets (Doerner et al., 2003). ACO implementations are fully distributed, as no central agent dictates or knows the behavior of all ants, are adaptive to changes (as pheromones fade with time, allowing for new routes to emerge), and robust against individual agent failures. ACO implementations allow for combining exploitation of existing paths with new path sampling, i.e., exploration. This ensures that ants do not converge to a suboptimal path but to the shortest path, and can discover new paths should a shorter one become available.

2.2.2.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) (Eiben & Smith, 2003) are a family of optimization algorithms inspired by biological evolution. The initial population of solutions is created randomly, and through the evolutionary processes of selection, crossover and mutation, the most suitable solutions are found after a number of generations. A fitness function is used to quantify the quality of the solution against the required optimization objective, and to select high-quality solutions that should pass on their genetic material to the next generation. Selected solutions crossover to create offspring, which contains the genetic material of both parents. Additionally, mutations, or random changes to the genetic material, also occasionally happen, to generate solutions that differ from any other solution currently present in the population. If the new mutation is beneficial, it will survive through selection and crossover, otherwise it will die out.

By mimicking the biological evolutionary process, EAs are able to self-adapt, i.e., they can evolve and tune their own parameters, suggesting their suitability for the implementation of self-organizing and autonomic systems (Eiben, 2005).

Applications of EAs in such multi-agent systems include robot soccer, where the actions of each player are optimized using EAs (Lekavy, 2005), large-scale multi-server web services, where an EA are used to optimize session handling strategies (Eiben, 2005), and traffic control, where EAs are used to tune the time sequences for traffic signals (Hoar et al., 2002).

EAs are also extensively used in multi-objective optimization (Tan et al., 2005), due to their ability to find multiple solutions representing multiple trade-off points (Van Veldhuizen & Lamont, 2000) in a single run (Coello, 1999). EAs make use of a number of approaches to combine multiple objectives to be optimized. A weighted sum approach combines the objectives into a single function, using weighting coefficients that reflect the importance of the objectives. A goal programming approach requires a designer to specify target goals for each of the objectives, and an EA is used to minimize the absolute deviations from the targets specified. Using the ϵ-Constraint method, a single objective optimization is carried out for the most important objective, while other objective functions are considered as constraints bound by an allowable levels ϵ (where values of ϵ specify values of the objective functions which should not be exceeded) (Coello, 1999).

Evolutionary optimization is also often used in combination with other optimization techniques to select the most suitable combinations of input parameters. For example, in applications of W-learning (Humphrys, 1996b) (further discussed in Section 2.3.2.2), EA was used to find optimal combinations of RL parameters.

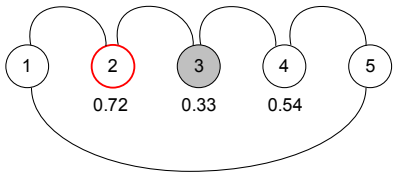


Fig. 2.3: Particle swarm optimization (Kennedy & Russell, 2001)

2.2.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) (Kennedy & Russell, 2001) is a self-organizing optimization technique inspired by the flocking behaviour of birds. Each particle (a bird) in a swarm (a flock or a population) represents a potential solution, and moves through the multidimensional problem search space (possible set of solutions) seeking an optimal solution. Solutions are evaluated against a fitness function that represents the optimization objective. Particles broadcast their current position (i.e., the quality of their current solution) to their neighbours. Each particle then accelerates its movement towards a function of the best position it has found so far and the best position found by its neighbours (Blum & Merkle, 2008). For example, consider Figure 2.3, depicting a five-particle PSO population. During the optimization process, particle 3 receives broadcasts from its neighbours 2 and 4. As particle 2 currently has the best position in 3's neighbourhood, 3 will accelerate towards a function of its own previous best position and 2's current position. As the swarm iterates, the fitness of the solutions increases, and particles focus on the area of the search space with high-quality solutions. Therefore, optimal solutions to the optimization problem arise in an emergent self-organizing way, where each particle only needs to communicate with its neighbours.

The existing applications of PSO-based optimization techniques include self-organizing document clustering, where PSO has been used to minimize the intra-cluster distances and maximize the distance between clusters by using swarm particles to represent possible solutions and evaluating them against a similarity metric (Cui et al., 2005), self-organizing networked sensor systems, where PSO has been used to optimize connectivity and minimize sensor overlap (Kadrovach & Lamont, 2002) and robotic learning, where PSO has been used for unsupervised learning of obstacle avoidance (Pugh et al., 2005).

PSO can also be applied to multi-objective optimization problems, by combining all objectives into a single weighted function (Parsopoulos & Vrahatis, 2002), by using importance ordering of objectives specified by users, by assigning a population of swarms to each objective and recombining solutions, or by considering particle's performance towards multiple objectives when selecting a leader towards which to accelerate (Reyes-Sierra & Coello, 2006).

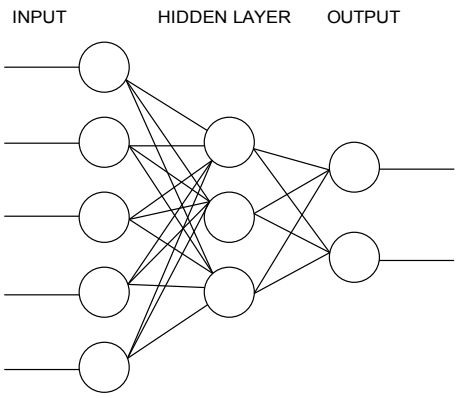


Fig. 2.4: Neural network (Weijters & Hoppenbrouwers, 1995)

2.2.2.4 Artificial Neural Networks

Artificial neural networks (ANNs) are adaptive learning systems based on biological neural networks. They consist of an interconnected group of artificial neurons whose structure changes during the learning process. An ANN is provided with a set of inputs, which, through a series of hidden layers, lead to one of the outputs, as shown in Figure 2.4. Each of the circles in Figure 2.4 represents a neuron, while lines connecting them represent connections with associated weights. During the training phase, connections between neurons strengthen each time a set of inputs generates a given output. In such a way, an ANN learns to, based on a given input, select the output that has the highest probability (connection strength) of being an optimal solution based on experience so far, i.e., it learns how to map an input vector to an output. Due to the number of connections between neurons and multiple hidden layers, ANNs can produce complex global behaviours, that emerge from the simple processing capabilities of neurons and the connections between them.

Examples of ANN application areas include traffic signal control, where traffic conditions are given as input and traffic signal controller settings were output (Srinivasan et al., 2006) and in the simulation of autonomous robots, where neural networks were used for trajectory planning by robot manipulators (Ramdane-Cherif, 2007).

ANNs are also often used in combination with other self-organizing and learning techniques. For example, ANNs have been used in combination with RL to map RL states to actions (Tesauro, 1999), or in combination with PSO, where PSO has been used to evolve neural network weights (Kennedy & Russell, 2001).

2.2.3 Multi-Agent Systems Summary and Conclusions

In this section we have introduced MAS as an engineering technique for building decentralized autonomic systems. We believe the capability of MAS to self-organize based only on local actions and interactions make them particularly suitable for the implementation of self-* properties of autonomic systems. Moreover, the characteristics of an intelligent agent, i.e., its ability to observe and act upon its environment, map to desired capabilities of an autonomic element, enabling utilization of agent techniques in the implementation of autonomic elements. After introducing agents and MAS characteristics, we have described some of the most commonly used self-organizing MAS algorithms, e.g., ACO, PSO, EAs, and ANNs. Based on the different characteristics of these algorithms, they could be utilized in implementation of different self-* properties in autonomic systems. For example, ANNs could be applied in diagnostics problems due to their ability to map vectors of input data (symptoms) to an output (cause of the problem) and as such can be used in the engineering of self-healing behaviours. The ACO family of algorithms is most commonly used in the type of optimization problems that involve finding the shortest/most optimal path between two points (e.g., routing), while PSO and EAs are used for problems that require parameter tuning to find optimal their combinations, and as such could be used for self-optimization.

In the next section we introduce RL, a learning algorithm which, even though it is not self-organizing in its original form, is widely used in self-organizing multi-agent systems and autonomic computing.

2.3 Reinforcement Learning

In this section we introduce the basics of reinforcement learning (RL) required for an understanding of DWL, review existing RL-based algorithms for multi-policy and multi-agent optimization, and present existing applications of RL in autonomic computing.

2.3.1 Introduction to Reinforcement Learning

RL (Suton & Barto, 1998) is a learning technique based on trial and error that has been researched and applied in control theory, machine learning and artificial intelligence problems, as well as non-computer science domains such as psychology. RL is a single-agent, unsupervised learning technique, whereby an agent learns how to maximize the rewards received for its actions, based on its interaction with the environment. However, an agent does not only need to learn from immediate rewards, but also from delayed reinforcements. An agent might need to take a sequence of low-reward actions to

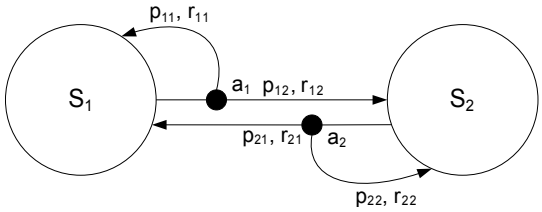


Fig. 2.5: State transition diagram (Suton & Barto, 1998)

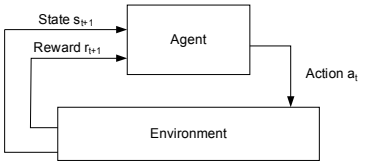


Fig. 2.6: Reinforcement learning process (Suton & Barto, 1998)

reach a state with a high reward value.

Such delayed-reinforcement learning problems are often modelled as Markov Decision Processes (MDPs). An MDP consists of:

- a set of states S = $\{s_1, s_2, ..., s_n\}$
- a set of actions A = $\{a_1, a_2, ..., a_m\}$
- a reward function R: S x A→ R, where R (s_t, s_{t+1}, a_t) is the expected numerical value of the next reward, received after taking an action a_t in state s_t, and transitioning to state s_{t+1}
- state transition function T: S x A → P, where P (s_t, s_{t+1}, a_t) is the probability of an agent transitioning to state s_{t+1} after taking an action a_t in state s_t.

For example, consider Figure 2.5, showing a state transition graph for an agent with two states, S = $\{s_1, s_2\}$ and two actions, A = $\{a_1, a_2\}$. The large open circles represent states, and the small solid circles represent actions. If an agent is in state s_1, and executes an action a_1, it will stay in state s_1 with probability p_{11} and receive reward r_{11}, or it will transition to state s_2, with probability p_{12} and receive reward r_{12}. Analogously, if an agent is in state s_2, and executes an action a_2, it will stay in state s_2 with probability p_{22} and receive reward r_{22}, or it will transition to state s_1, with probability p_{21} and receive reward r_{21}.

An RL agent's interaction with the environment (where environment is modelled as an MDP) is depicted in Figure 2.6 and consists of the following steps:

- the agent observes the current state of the environment $s_t \in$ S

- the agent takes an action $a_t \in \mathrm{A}$
- the agent observes the subsequent state $s_{t+1} \in \mathrm{S}$
- the agent receives the reward r_{t+1} for the action a_t taken in s_t from the environment based on the desirability of the next state, s_{t+1}. If the reward has a negative value, it is often referred to as a cost, or penalty.

An agent's task is to, based on the rewards received, estimate *value functions* for its state-action pairs. A value function describes how good it is, in terms of future expected return, for an agent to perform a given action in a given state. When estimating a value function, an agent does not only take into account immediate reward, but also takes into account the long-term desirability of the states, taking into account states that are likely to follow, and the rewards available in those states.

Based on experience (i.e., interaction with the environment as described above), at each time step an agent updates its *policy,* Π. Π provides a mapping from each state, $s \in \mathrm{S}$, and action, $a \in \mathrm{A}$, to the probability $\Pi(s, a)$ of taking action a when in state s. The optimal policy is the one that maximizes the total discounted expected reward of an agent.

There are three major assumptions that must hold in order to enable modelling an environment as an MDP:

1. The state transitions of an MDP must possess the Markov property. Given that the state of the MDP at time t is known, transition probabilities to the state at time $t + 1$ are independent of all previous states or actions, and depend only on the state and action at t. Therefore, the current state is assumed to hold all relevant information from past experiences. However, even if the environment is non-Markov, the use of MDPs is possible if the state space is constructed so that the Markov property holds as nearly as possible (Suton & Barto, 1998).

2. MDPs assume full environment observability, i.e., the state is known at the time when an action is to be taken. If this is not the case, the environment can be modelled as a Partially Observable MDP (POMDP), where at each time step, each state is associated with the probability of the environment currently being in that state (Kaelbling et al., 1996).

3. MDPs assume that the environment is stationary, i.e., the probabilities of state transitions or of receiving a specific reward do not change over time. If the environment is slow-changing, RL algorithms can still be applied as older experiences are discounted when new experiences are gathered. Non-stationary environments can also be modelled as multiple distinctive MDPs (Choi et al., 2002), one for each set of environment conditions.

Problems modelled as MDPs can be solved using *model-free* or *model-based* RL strategies, where a model of the environment consists of a state transition function and a reward function (Kaelbling et al., 1996). Model-free strategies do not require a model of the environment and update the value function and policy directly based on the experience. Model-based strategies first learn the model of the environment by exploring the environment and maintaining statistics about the results of each action, and then use the learnt model to calculate the optimal policy. The applicability of model-free and model-based strategies depends on the environment characteristics. Model-free approaches perform better in situations that require complex, potentially multiple, models of the environment that need to account for a variety of environment conditions (e.g., UTC (Abdulhai et al., 2003)). Model-based approaches have slower execution times and greater storage costs, as they need to store a model, but are suitable in situations where acquiring real-world experience is expensive as they can benefit more from the experience they do acquire (Dowling, 2005).

In this review, we particularly focus on Q-learning, a popular model-free RL algorithm, as it is the basis of DWL.

2.3.1.1 Q-Learning

Q-learning (Watkins & Dayan, 1992) is a model-free algorithm that learns from delayed reinforcement. Using Q-learning, an agent learns to associate actions with the expected long-term reward of taking that action in a particular state, represented by a Q-value. Q-values for state-action pairs, $Q(s, a)$, are updated using the Formula 2.1:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \underset{a'}{max} Q(s',a')) \tag{2.1}$$

where:

- s is the current state, s' is the possible next state, a_i is the current action, and a_i' is the possible next action
- α is a learning rate, $0 \leq \alpha \leq 1$, which determines the weight that new experiences have in the Q-value calculation
- γ is a discount rate, $0 \leq \gamma \leq 1$, which determines the rate at which expected future experiences are discounted in the Q-value calculation
- r is the reward an agent receives for transitioning to, or remaining in, the state s'.

Based on Q-values, an agent selects which action to execute in each given state. When selecting an action, RL-based agents face a trade-off between exploration and exploitation. An agent's goal is to maximize its long-term reward, and in order to do that it has to take actions known to yield high rewards (i.e., exploit current knowledge). However, in order to discover all such actions, it needs to sample new actions, that potentially might not yield high rewards (i.e., it needs to explore). A number of action-selection strategies exist that address this problem. Instead of acting in a purely greedy fashion, i.e., always exploiting to maximize immediate reward (by always selecting an action with the highest Q-value), agents can use ϵ-greedy action selection, that selects the action with the highest reward most of the time, but occasionally, with a small probability ϵ, selects an action at random. One drawback of ϵ-greedy action selection is that, when it explores, it chooses equally among all actions, therefore also selecting ones that can result in large negative payoffs. Softmax action selection strategies do not assign the same probability to all actions, but base the probability of an action being selected on its Q-value, so the actions with higher Q-values have higher probability of being selected. One of the most common softmax action selection strategies is Boltzmann. Boltzmann uses a parameter called the temperature (τ) to balance exploration and exploitation; high temperatures cause all of the actions to be nearly equiprobable, while at $\tau \rightarrow 0$ selection is mostly exploitative, i.e., mostly selecting actions with high payoffs. Generally, at the start of the learning process, temperatures are high to enable exploration in order to experience all states and actions, while in later stages, when suitable actions have been identified, the temperature is decreased.

From the elements of Q-learning described here we can observe that the outcome of the learning process depends on a number of RL elements, such as the state space design, the learning rate, the discount rate and the action selection strategy. Particular care should be taken when designing the reward function, as Q-learning aims to maximize the long-term reward received but receiving the maximum reward might not, in fact, mean meeting the original goal if the state space and reward model are not designed correctly.

2.3.2 Multi-policy RL

RL, as described in Section 2.3.1, is a single-agent, single-policy learning technique. However, an agent might be required to simultaneously optimize towards a number of policies. Therefore, multiple extensions to RL-based techniques have been developed to enable RL to solve single-agent multi-policy optimization problems. Some of the research described in this section refers to this type of problem as a multi-agent problem rather than a multi-policy problem, however, in such work multiple software agents are assumed to control a single set of actuators for which actions must be determined

(Humphrys, 1996b). In our work, each agent has exactly one set of actuators, and each set of actuators belongs to exactly one agent, and it is the multiple policies present on that agent, or the multiple goals that an agent has, that are competing for use of the actuators.

The techniques described in this section are not only motivated by the need for agents to optimize towards multiple goals, but also by the need to decompose large complex goals into multiple simpler goals, turning a complex single-goal problem into a multi-goal problem (Tham & Prager, 1994). These multiple goals can be required to be met in sequence, where the completion of one task is a prerequisite for initiating the completion of another task (Singh, 1992), or can be required to be completed in parallel. This thesis, and therefore this section, focuses on multiple goals that are to be implemented in parallel.

We identify two major approaches to dealing with multiple policies in RL: addressing multiple policies within a single learning process, or using an arbitrated approach where a dedicated learning process is created for each policy, and an arbitrating agent or an action selection method decides, based on some criteria, which of those policies should receive control over the actuators at each time step. We present these approaches in the following two sections.

2.3.2.1 Combined State Spaces

Multiple policies on an agent can be combined into a single learning process, where the state space of the joint learning process is a cross product of the state spaces of individual policies. Each resulting state has an associated payoff, which is a combination of payoffs of all the policies. In order to evaluate each state an agent can have a vector of weights representing the importance of each of its policies (Barrett & Narayanan, 2008). For example, assume that, in the two-policy case, a state A has an associated reward of 1 for one of the policies, and a reward of 5 for the other. If the agent's vector of weights assigns a weight of 0.6 to the first policy, and of 0.4 to the other, the total payoff an agent will receive for being in state A is $(0.6 \times 1) + (0.4 \times 5) = 2.6$.

Assigning the weight vectors is not trivial as there might not be a clear hierarchy of goals, and the weights, or the importance of a goal for an agent, can change over time. Gábor et al. (1998) addresses a problem with fixed goal weights that do not change over time, while Hiraoka et al. (2008) and Natarajan & Tadepalli (2005) address goals with changing relative weights. Hiraoka et al. (2008) consider calculating optimal actions for all combinations of policy weights, however this approach is prone to state explosion. Instead, they use an approximation approach with adaptive error margins, starting off with a large margin to learn in a smaller state set, and reducing the margin at the later stages to increase accuracy of the results. In (Natarajan & Tadepalli, 2005), goals' weights, i.e., their

relative importance, can change over time. Upon change of a goal's weight, agents learn the optimal actions for the current weights, by reusing the best policy learnt so far and adjusting it to current weights rather than starting from scratch. These approaches assume that the rewards received from the environment for different goals are comparable and reflect their relative priority. Shelton (2000) considers the case where rewards are received from multiple incomparable sources, and provides an algorithm for scaling the rewards so that policies of equal importance get equal weight.

The use of combined state space approaches is feasible when the number of objectives and the size of their state spaces are relatively small, however for situations with a large number of objectives with large state spaces this approach can be computationally expensive, policies can take a long time to converge to optimal results, and the addition of new policies leads to exponential growth of the state space (Cuayahuitl et al., 2006). As such, for larger and more complex problems, combining state spaces is not a scalable approach and is of limited use in multi-policy optimization. To address this problem, a combined state space approach can be utilized in combination with various algorithms used to reduce the state-space size or number of state-action combinations. For example, one way to improve the performance and scalability of the combined state space approach is presented in (Cuayahuitl et al., 2006), where an algorithm is proposed to reduce the state-action space (of 3.3 million state-action combinations in the authors' example) in order to achieve more feasible convergence times, memory requirements, and improved performance. The algorithm eliminates invalid actions per state and invalid states, as specified by domain-specific constraints. For example, in the dialogue system considered, all conversation slots needed to be confirmed, apart from slots filled with "yes" or "no", limiting the number of actions that can follow unconfirmed slots. However, this approach is limited to application domains where eliminating an invalid state-action pair will lead to a significant reduction in search space size. Paquet et al. (2004) also use a state-reduction algorithm, in which, instead of eliminating invalid state-action combinations, an agent starts off by perceiving only a single state and learns to split it into multiple states, distinguishing only between the states that need to be distinguished.

2.3.2.2 Arbitration-based Approaches

In arbitration-based approaches, multiple policies suggest actions to an arbitrator, which selects an action for execution based on some criteria. Arbitration-based approaches primarily differ in the way in which a winning action is chosen, and in whether a winning action is nominated by one of the policies explicitly, or is a compromise action. Some of the early work in this area includes the subsumption architecture by Brooks (1986, 1991). Multiple goals in a subsumption architecture are

met by separate layers arranged in a hierarchy, where higher layers in the hierarchy always win when they compete with lower-level ones. In (Gadanho & Hallam, 2001), the strength of a number of so-called "emotions" (perceptions of the state of the world) currently "felt" is used to decide on behaviour switching (i.e., to decide which RL process will gain control over the agent). In (Raicevic, 2006), a gating function is used to assign varying weights to action suggestions by individual policies. Instead of a single learning process being assigned to each goal/sub-goal, Mariano & Morales (2000) use a family of independent agents for the implementation of each goal, but similarly, results are reported to a negotiation mechanism that selects the action to be executed. It attempts to find an action that satisfies all of the objectives, and failing that, selects a random action from the proposed set.

Some approaches enable a compromise action to be selected, rather than give full control to only one policy. For example, in (Rosenblatt, 2000), policies calculate utilities for each of the outcome states; an arbitrator combines those utilities, and selects the action with the highest combined utility. Similarly, in (Russell & Zimdars, 2003), (Karlsson, 1997) and (Sprague & Ballard, 2003), policies report Q-values for each of the actions, and arbitrator selects the action with the highest cumulative Q-value.

The W-learning (Humphrys, 1996b) approach suggests making an action selection based on relative policy weights, where weights, instead of being predefined or assigned by an arbitrator, are learnt specific to the particular state in which the policy is currently. Our DWL algorithm is inspired by this approach, so we cover W-learning in more detail.

W-Learning W-learning (Humphrys, 1996b) is an action-selection technique in which each policy is implemented as a separate Q-Learning process with its own state space. An agent learns Q-values for state-action pairs for each policy and, at every time step, each policy nominates an action based on these Q-values. Using W-Learning, an agent also learns, for each of the states of each of its policies, what happens, in terms of the reward received, if the action nominated by that policy is not obeyed. Note that a policy does not need to know what action was actually executed, or which other policy suggested that action; it only needs to observe the effect of its nominated action not being executed. We consider this particularly suitable for multi-agent heterogeneous environments, where one agent might not know what actions are executed by other agents, or might not share the same action sets as other agents. The effect of one policy's nominated action being executed can be positive or negative for other policies. For example, an action nominated by one policy can also be a good action, or even the optimal action, for another policy, meaning that the policy receives a reward close to what it would have received if its own nominated action had been executed. However, one policy's nominated

action being executed can also have a negative impact on other policies, i.e., other policies might receive zero or negative rewards after that action's execution. W-learning captures this effect as a measure of the weight of the nominated action in that particular state s, called a W-value, W(s). W-values reflect the importance of the nominated action to the policy. For example, a policy might be in a state where the next action to be executed holds no importance, or it might be in a state where it is crucial for the action that it had nominated to be executed. It is important to note that this weight is not absolute, but is learnt relative to the other policies simultaneously deployed, i.e., to the actions those policies nominate, and to the particular policy state. For example, the weight can be low not just because a policy does not care what action is executed next, but also if the policy learns that the actions suggested by some other policy are as suitable as the action it nominates itself. Therefore, W(s) for a particular state, can be different depending on the other policy (or policies) that are deployed on an agent simultaneously and whether those policies are compatible, complimentary or conflicting. W(s) is updated after an action selection according to Formula 2.2, where r_i is the immediate reward received, s is the current state of the policy, s' is the next possible state, a_i is the current action, and a_i' is the possible next action for policy i.

$$W_i(s) = (1-\alpha)W_i(s) + \alpha(Q_i(s,a_i) - (r_i + \gamma \text{max}Q_i(s', a_i'))) \tag{2.2}$$

Action selection based on the W-values is performed as follows. At each learning step, multiple policies on a single agent nominate their actions, together with associated W-values for their current states. The winning policy can be selected in several ways based on the nominated W-values. These methods are divided into individual and group methods. Individual methods select a winning action so that it satisfies some criteria related to a single policy, for example, the one that can potentially be worse off than any other policy in the next step, while group methods take into account all of the policies, for example, select an action that satisfies the largest number of policies. We consider the "minimize the worst unhappiness" (Humphrys, 1996a) action selection particularly suitable to our multi-agent multi-policy domain. In this approach, an agent A selects the policy with the highest nominated individual W-value, or effectively, the policy which is going to suffer a higher loss if its nominated action is not executed, than any other policy will suffer if A's nominated action is executed. This method ensures that performance of all policies is addressed, while group methods satisfy a large number of policies, but the performance of a small number of policies, which are in conflict with the others, could be neglected. Neglecting a policy would be particularly detrimental to the agent's performance, if the neglected policy is of a higher priory than the policies addressed by the group method.

In Figure 2.7 we show an example of W-learning action selection. The figure represents an agent

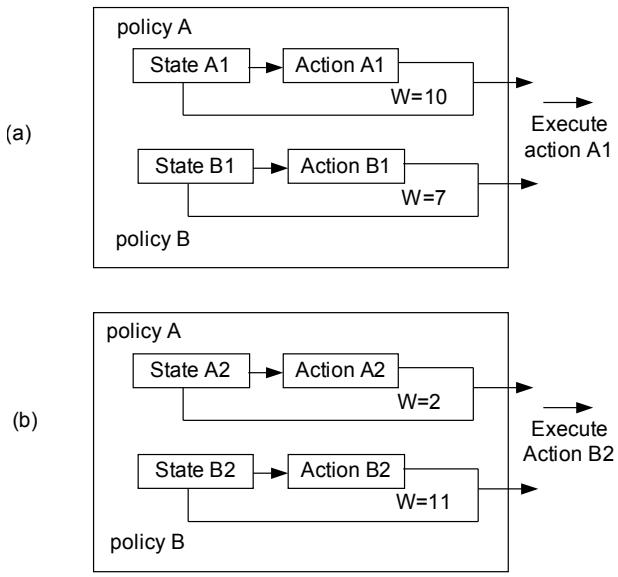


Fig. 2.7: W-Learning action selection (Humphrys, 1996a)

that has two policies, A and B. In state A1, the policy A has a higher W-value than policy B has in its state B1, so the action suggested by A is executed, i.e., action 1. When A is in state A2, and policy B in B2, B has a higher W-value, and action B2 is executed.

The relative priority of the policies should be built into the design of the rewards that policies can receive, i.e., an agent should receive a higher reward for a policy with a higher priority. As rewards received are used in calculation of a W-value, the size of the rewards is reflected in higher W-values for higher priority policies, which will therefore have a higher chance of winning the competition at action-selection time. Note, however, that this does not prevent a lower priority policy from winning at action-selection time when it is in a state of high importance or when a higher priority policy is in a low-importance state.

To ensure that all policies get a chance to execute suitable actions and that no policy gets neglected for extended periods of time, W-values are updated only for policies that did not win at the last action selection step. After action selection, the W-value of the winner stays the same, while the W-values of other policies are updated and therefore can potentially increase giving them a chance to catch up with the winner and win in the next action-selection step. Therefore, W-values are not learnt once and then remain unchanged, but change during the agent's operation. Actions nominated by agents will stay the same, as determined by converged Q-values and action-selection strategy, but at different times will be nominated with a different strength, based on their relative current importance.

W-learning has been shown to be a suitable approach for multi-policy optimization for a simple

ant world example, where an ant needs to balance a search for food with avoiding a predator, as well as in a more complex case of a house robot, which has 8 different policies for which to optimize, e.g., clean the house, put out fires, detect strangers.

2.3.2.3 Pareto-Based vs. Non-Pareto Approaches

Multi-objective optimization techniques can be categorized into Pareto-based and non-Pareto approaches, based on the characteristics of the solution(s) they generate.

In multi-objective optimization, a Pareto-optimal solution is a solution in which no improvements can be made to one objective without making other objective(s) worse off (Vamplew et al., 2008). For a multi-objective problem, generally a number of solutions exist (Jin & Sendhoff, 2008). Pareto-based optimization methods return a set of such solutions, representing different trade-off points between the solutions (Vamplew et al., 2008). A single solution is selected from the set based on user preferences or objective weights. Non-Pareto approaches incorporate the user preferences and/or objective weights into the optimization problem at the start, and return a single solution that is optimal for a given combination of weights (Jin & Sendhoff, 2008).

The RL approaches discussed in this section use non-Pareto techniques. In a multi-objective learning process using a single combined state space, the relative priority of objectives is specified through their rewards, the weight of user preferences can be given by a vector, while the RL solution aims to maximize the reward given these weights. In action selection problems, in which each separate learning process suggests an action for execution, an arbitrator selects only a single solution, i.e., an action for execution, based on some specified criteria, where that criteria can, for example, depend on relative policy priorities or user preferences.

2.3.3 Multi-Agent RL

In the previous section, we discussed RL-based systems for optimization towards multiple goals (policies) on a single agent. This section covers the optimization of the behaviours of multiple agents, that are cooperating in order to meet some global system goal. Note that we do not consider the case of competing agents as such systems are outside the scope of this thesis. We do not aim to provide a comprehensive overview of RL in multi-agent systems, but review a few representative examples that would allow us to discuss issues related to our research into multi-agent multi-policy RL. For a more comprehensive overview of multi-agent systems and multi-agent learning, please refer to (Busoniu et al., 2005) and (Vlassis, 2007).

In single-agent RL, an agent learns the actions with the highest payoffs for each of its states. In the

multi-agent case, a global state is a combination of all local states, and a global action is a combination of all local actions, leading to exponential growth of the state-action space (Guestrin et al., 2002). In order to determine jointly optimal actions without explicitly considering all possible combinations of local actions, a number of approaches have been proposed to break down the global problem into a group of local optimization problems. These approaches can be divided into two groups: those where each agent acts independently towards optimizing its local performance and ignores the presence of other agents, so called independent learners (IL), and approaches where agents cooperate with other agents, most commonly their immediate neighbours, in order to ensure that locally good actions are also globally good, so called joint action learners (JAL) (Vlassis, 2007).

IL approaches using RL are not justified theoretically, as the underlying assumption of a Q-learning environment being stationary does not hold, due to the influence of other agents in the system (Vlassis, 2007). However, experiments show that the IL approach still has practical use (Tan, 1993; Claus & Boutilier, 1998), although it achieves poorer performance when compared to communicating agents (Tan, 1993; Schneider et al., 1999), as communication is often required to ensure locally-good actions are also good for the system globally (Claus & Boutilier, 1998; Dowling & Haridi, 2008). As we also believe that cooperation is required to manage agent dependencies, as discussed in Chapter 1, in this review we discuss only JAL-based approaches.

Multi-agent RL techniques can also be categorized based on the observability of the environment in which they are applied, i.e., problems they aim to solve can be modelled as either MDPs or POMDPs. We primarily focus on techniques using MDPs (Sections 2.3.3.1, 2.3.3.2, and 2.3.3.3), as in our work presented in this thesis we assume full observability, however we also briefly introduce POMDP-based systems as an example of alternative approaches that can be used in partially observable environments (in Section 2.3.3.4).

2.3.3.1 Collaborative Reinforcement Learning

Collaborative Reinforcement Learning (CRL) (Dowling, 2005) enables global system optimization based on cooperation between RL agents. Each agent has its own state space, action space, value function, and only a local view of the environment, while an estimate of the global view of the system is achieved by agents periodically exchanging their value functions. Global optimization of system behaviour arises from optimizing the process of solving smaller tasks introduced locally, represented as Discrete Optimization Problems (DOPs). DOPs are defined as "the selection of minimal cost alternatives from among a finite set of feasible solutions, as defined by an objective function" (Dowling et al., 2006) and are modelled on agents as absorbing MDPs, i.e., MDPs that will enter a terminal

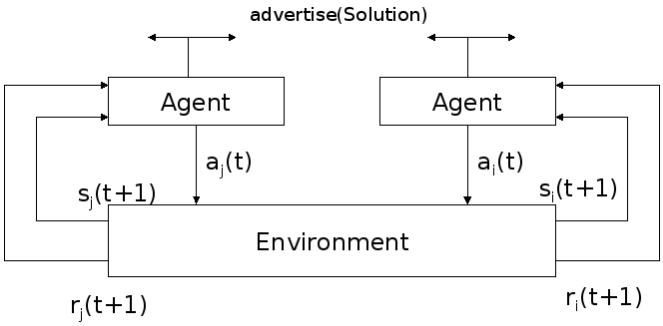


Fig. 2.8: Collaborative reinforcement learning (Dowling, 2005)

state after a finite amount of time. DOPs can be introduced in the system on any of the agents, and an agent needs to minimize the cost of solving them by either making an action towards solving it locally, or delegating it to one of its neighbours to solve or delegate further. Delegation incurs the cost of transferring the DOP as well as the cost of estimating whether any of the neighbours can solve it at a lower cost or can find another agent that can do so. Agents periodically advertise their estimated costs of solving DOPs to their neighbours, as this information can change dynamically during system operation (if agents are, for example, solving other DOPs). This process is depicted in Figure 2.8. Each agent executes its own RL process, i.e., performs actions in the environment and receives rewards for those actions, which it uses to update its value function, i.e., learn the solution to a problem. Periodically, that solution is advertised to its neighbours.

CRL is suited for multi-agent optimization problems where agents delegate actions to each other in order to solve a goal. For example, CRL has been successfully applied in simulations of load balancing (Dowling, 2005) and ad-hoc network routing (Dowling et al., 2006), where agents can delegate jobs for processing or packets for delivery to their neighbours. We discuss these application further in Sections 2.3.4.2 and 2.3.4.3, respectively, as examples of applications of RL in autonomic computing.

2.3.3.2 Coordination Graphs

Coordination graphs are used to coordinate the actions of multiple agents cooperating to control a single large MDP. The action space of the MDP is a joint action space of the entire set of agents, and grows exponentially with an increase in the number of agents in the system. To reduce the number of combinations that need to be taken into account when computing a joint action, Guestrin et al. (2001, 2002) propose factoring the problem space into smaller more manageable tasks using coordination graphs. They argue that any given agent does not need to coordinate its actions with

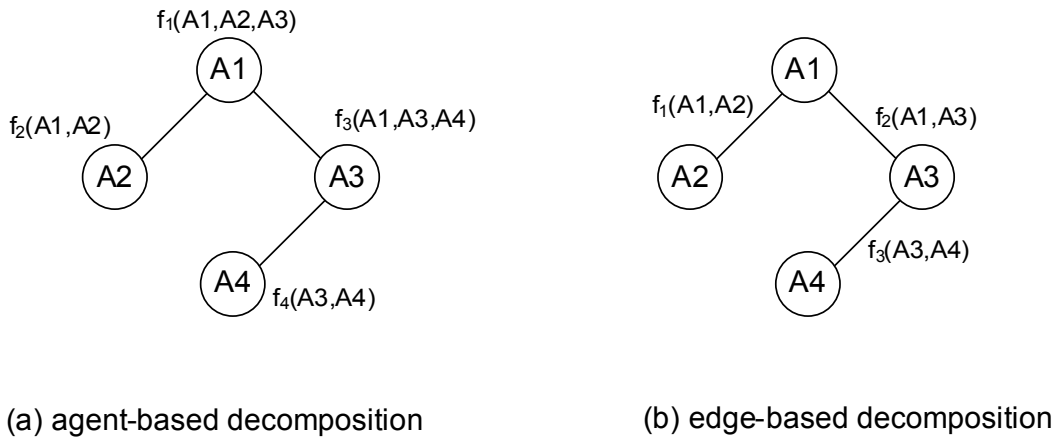


Fig. 2.9: Coordination graphs (Guestrin et al., 2001, 2002; Kok et al., 2005)

all of the other agents, but only with a small number of the agents in its proximity. To facilitate this, a global value function is approximated as a linear combination of local value functions, where each of the local value functions addresses only a small part of the system controlled by a small number of agents. Using these local value functions, an agent coordinates its action with its neighbours, which in turn coordinate their actions with their neighbours and so on, in order to optimize the behaviour towards a global function. A global MDP is represented as a coordination graph, where agents need to coordinate only with the neighbours with which they are connected in the graph. Local value functions can be modelled on a single agent and the set of its one-hop neighbours (so called agent-based decomposition (Guestrin et al., 2001, 2002)), or on two agents connected by an edge (edge-based decomposition (Kok et al., 2005)). The two approaches are depicted in Figure 2.9(a) and Figure 2.9(b), that show a graph consisting of four agents with three edges connecting them. In agent-based decomposition, the global function is broken down into four local functions, f_1, f_2, f_3, f_4, one per agent, while in edge-based decomposition, it consists of three functions, one per edge, f_1, f_2, and f_3. In agent-based decomposition, as presented in (Guestrin et al., 2001, 2002), it is assumed that dependencies between nodes (i.e., graph connections) are known up-front, while Kok et al. (2005) in their edge-based approach provide a means for dependencies to be learnt and reinforced during problem solving.

Coordination graphs with agent-based decomposition have been successfully applied in a simulation of a load-balancing application (Guestrin et al., 2002). The test system consists of a number of interconnected machines, whose status can be good, faulty or dead, and can change stochastically. The presence of a dead machine increases the probability that its neighbours will become faulty and

die. A machine in any state can perform a reboot action that results in the machine's status switching to good, however the current job executing on that machine is lost. The goal of the system is to maximize the number of jobs completed in the system. Therefore, each machine needs to coordinate with its neighbours, whose number of jobs completed could be influenced by the machine switching to a status of dead, but also needs to complete its local jobs.

Coordination graphs with edge-based decomposition have been successfully applied in a distributed sensor network problem (Kok & Vlassis, 2006). A number of sensors need to coordinate in order to "hit" the targets. The target is hit if at least three sensors are focused on it. Each focus action has an associated cost, however hitting a target has an associated high reward. If more than three sensors focus on a target, the target is still hit, but only three sensors receive a reward. Therefore, sensors need to coordinate with their neighbours in order to maximize the rewards received by hitting a target, but minimize the cost incurred by focus actions that do not result in hitting the target.

2.3.3.3 Distributed Value Functions

Schneider et al. (1999) propose an approach to implementing distributed RL on multiple agents by using shared value functions. Each agent learns its local value function based on its individual actions and rewards received, but it also incorporates the value functions of its neighbours into the local value function updates. In order to do this, each agent needs to have a weight function specifying how much the value functions of other agents contribute to its value function, with the weight of non-neighbours' value functions being zero. In such a way, each agent learns a value function that is an estimate of a weighted sum of its own expected rewards and those of its neighbours (whose value function is in turn a weighted sum of their own and that of their neighbours and so on), and can therefore select actions that are good not just locally but also for the other agents in the system. The distributed value function approach has been successfully applied in a simulation of power grid management, where it was used to coordinate the performance of a number of interconnected power distributors (that are also connected to a number of providers and a number of customers), in order to maximize the rewards received by providing the desired level of service to their customers.

2.3.3.4 POMDP-based Approaches

In the work covered in previous sections, agents' environments are represented as MDPs, however in the case where agents might only have partial, noisy, or probabilistic observations of their state space and limited communication with their neighbours, the environment can be modelled as a POMDP. Examples of such approaches are presented in (Goldman & Zilberstein, 2003, 2004), (Peshkin et al.,

2000), and (Yagan & Tham, 2007). Solving POMDP-based problems requires a different set of learning techniques than those used to solve MDPs. While MDPs are in general solved using value-search techniques (i.e., methods based on learning the value function, such as Q-learning), policy-search techniques, which learn optimal policies without learning a value function (Baird & Moore, 1999), are considered more suitable for solving POMDPs (Peshkin et al., 2000). In the work presented in this thesis we assume that agents have full observability of the environment, while the potential application of our proposed approach in partially observable environments is a subject for future work.

2.3.3.5 Learning to Cooperate

In the previous section we have presented a number of cooperative multi-agent systems, however, in most of those systems agents cooperate with a fixed set of neighbours (e.g., one-hop neighbours in a graph, as presented in 2.3.3.2), where those neighbours' inputs have a fixed weight (e.g., as determined by a weight function, as presented in 2.3.3.3). A much smaller body of work is concerned with agents in multi-agent systems learning with which other agents to cooperate, and in which situations. We present such work in this section.

Kok et al. (2005) extend the coordination-graph approach discussed in Section 2.3.3.2 to learn the dependencies between the agents, i.e., the strengths of the graph connections, rather than using predefined ones. The dependency graph (i.e., coordination graph) starts off with either no edges, or random edges, and based on the outcome of the cooperation (joint action decisions made) those edges either get weaker and disappear or grow stronger.

In (Melo & Veloso, 2009), agents do not learn whom to cooperate with, but when (i.e., in which states) to cooperate. This work considers only a two-agent case, but nevertheless proposes an interesting approach to learning when to cooperate with other agents. For each agent's state, a fictitious action "coordinate" is added. If this action is selected, an agent senses the other agent's state and bases its local action decision on the combination of the other agent's state and its own state, rather than only on its own local state. The Q-learning process proceeds as usual and learns Q-values for all of the state-action pairs, where one of the actions executed can be "coordinate" with the other agent. In this way, an agent learns in which states is it useful to coordinate with the other agent. For example, Melo & Veloso (2009) test this approach using two robots that need to coordinate their behaviour whilst attempting to pass through a narrow doorway. Agents learn that that coordination action is mostly appropriate when they are positioned in the vicinity of a doorway, as that is when they need to make sure not to collide with the other agent that is also trying to pass through the doorway.

Neither (Kok et al., 2005) or (Melo & Veloso, 2009) consider agents implementing multiple heterogeneous policies, and therefore do not investigate the potential impact of agent and policy heterogeneity on when and with whom to cooperate.

2.3.4 Reinforcement Learning in Autonomic Computing

RL is increasingly being used for the implementation of self-adaptive behaviour in autonomic systems. RL's ability to learn optimal behaviours without requiring domain knowledge (i.e., without requiring a model of the system or the environment) is removing the need to develop accurate models of autonomic systems, which is often a complex and time consuming task (Tesauro, 2007). Additionally, MDP-based RL is based on an underlying sequential decision theory that includes the possibility of a current decision having delayed consequences in the future, and is therefore able to account for the dynamic behaviour of autonomic environments (Tesauro et al., 2004). In this section we describe several applications of RL in autonomic computing systems, specifically in autonomic resource allocation, load balancing, ad-hoc network routing, and autonomic network repair.

2.3.4.1 Online Resource Allocation

RL algorithms have been applied for online resource allocation in a distributed prototype data centre in (Das et al., 2005; Tesauro, 2005; Tesauro et al., 2005). The system consists of a number of application environments where applications are deployed, and a resource arbiter, whose task is to dynamically assign resources to applications. The goal of a resource arbiter is to maximize the sum of resource utilities, i.e., maximize the long-term expected value of the allocation of a number of servers to given application(s). Each application has its own utility function that expresses the value that a particular application brings to the data center by delivering services to its customers at a particular service level, i.e., the value of being assigned a given number of servers. This work discusses a number of challenges faced when applying RL in complex distributed systems, e.g., state-space representation, which is potentially required to incorporate a large number of variables in order to accurately describe the system state, the duration of training time RL requires in live systems, and a lack of convergence guarantees in distributed RL. However, empirical results in a distributed prototype data centre show feasible training times, and the quality of the solution is comparable to solutions obtained using complex queue-theoretic system performance models that require detailed understanding of system design and user behaviour.

2.3.4.2 Load Balancing

CRL, a multi-agent collaborative RL technique discussed in 2.3.3.1, has been applied for load balancing in a simulation of a decentralized file storage system (Dowling, 2005; Dowling & Haridi, 2008). The system consists of ≈50 agents and several server agents, whose storage capacity is ten-fold that of other agents. The goal of the system is to store all inserted loads in as short a time as possible. Loads are entered into the system through the agent at position 0. Each agent has actions available to it that allows it to store the load itself, or forward it to one of the 10 neighbours to which each agent has a connection. Unsuccessful store actions result in an agent receiving a high negative reward, and successful ones a reward that is a function of the storage space available on an agent. The system was able to successfully store all loads 15 times faster than a random policy, was able to self-adapt to the addition of a new server in the system, and self-heal when connections between agents were broken.

2.3.4.3 Routing in Ad-Hoc Networks

CRL (as described in 2.3.3.1) has also been used to implement autonomic properties in ad-hoc network routing (Dowling et al., 2006). The network consists of a number of fixed and mobile agents whose goal is to optimize system routing performance. Each agent can deliver a packet to its destination (and receive a reward), deliver it to an existing neighbour (at an associated cost), or perform a discovery action to find a new neighbour (at an associated cost). The goal of each agent is to minimize the cost, i.e., either deliver a packet or forward it to the lowest cost neighbour. In order to do this, each agent learns and maintains a statistical model of its network links to estimate the cost of a given route, and exchanges the information on route costs with its neighbours. Using this approach agents learn to favour stable routes (consisting of fixed nodes) and re-route the traffic around congested areas of the network.

2.3.4.4 Autonomic Network Repair

In (Littman et al., 2004), RL has been used to learn how to efficiently restore network connectivity after a failure. A single agent, called a decision maker, is charged with repairing the network. An agent can perform a number of test actions, used to narrow down the source of a fault, and a number of repair actions, used to repair the fault. Each action is associated with a cost (time required for its execution), and the decision maker's goal is to minimize the cost of restoring the system to proper functioning. It is assumed that the decision maker does not have a complete set of information about faults, and therefore the learning problem is modelled as a POMDP. This approach was implemented in a live network, with a separate program injecting faults, and a decision maker successfully learnt

to attempt cheaper repair actions first.

2.3.4.5 Grid Scheduling

(Perez et al., 2008) use a combination of RL and ANNs to enable autonomic job scheduling on a resource grid. Users submit processing jobs to the grid, and a grid scheduler, implemented as an RL agent, is charged with selecting jobs for execution. The goal of the scheduler is to maximize user satisfaction (which decreases as a function of time that it takes to complete the job) and fairness (which is expressed as the difference between actual resources allocated and an externally-defined resource share that should be given to that user). A number of common issues with RL needed to be addressed in this work, i.e., the algorithm was initially trained offline, to overcome bad performance of RL during the training phase, and its convergence is only checked empirically, due to lack of theoretical guarantees. The approach is simulated using real traces of an existing grid workload with 100 processors and 5000 user jobs, over a period of 7 days. In the authors' simulation, after poor initial performance due to the exploration period has been overcome, this approach consistently outperformed the job scheduler currently used in the live system.

2.3.4.6 Summary of RL Applications in Autonomic Computing

In this section we have reviewed some of the existing applications of RL in autonomic computing. Applications range from single-agent single-policy problems (Section 2.3.4.4), cooperative multi-agent single explicit policy problems (Sections 2.3.4.2 and 2.3.4.3) to centralized or single-agent multi-policy problems (Sections 2.3.4.1 and 2.3.4.5), however, we are not aware of any existing decentralized cooperative multi-agent multi-policy applications. Applications presented encounter a number of common problems that arise in RL-based optimization, namely long training periods and poor performance during the same, no theoretical guarantees of convergence in multi-agent cases, and the problem of designing a feasible state-space representation in decentralized problems due to the large number of variables that influence the environment state. Nevertheless, all of the above applications show promising results in outperforming, or at least matching the performance of, existing model-based approaches that are time-consuming and require extensive knowledge of system structure to construct.

2.3.5 RL Summary and Conclusions

In this section we have introduced RL and the basic RL concepts required for an understanding of DWL. We have reviewed existing multi-policy RL optimization approaches, existing multi-agent RL optimization approaches, and applications of RL in autonomic systems.

RL-based techniques have been shown to be suitable for use in autonomic systems due to their ability to learn desired behaviours without requiring domain knowledge. However, for their wider application in autonomic systems, we believe RL techniques need to be able to be simultaneously implemented on a number of cooperative heterogeneous agents comprising an autonomic system, as well as be able to simultaneously address multiple, potentially conflicting, policies with different characteristics that autonomic systems might be required to implement.

Existing techniques, as summarized in Table 2.1, either address multi-policies on a single agent, or are multi-agent techniques but optimize only towards a single explicit system goal.

If we were to apply one of the multi-policy techniques in a multi-agent environment, there would be a number of issues that remain open. For example, we could address policies simultaneously on a single agent using W-learning or combined state spaces, however we do not have a means to take into account the performance of policies implemented on other agents, how other agents' policies affect the local agent, or how the local agent's actions affect other agents and their policies.

Similarly, if we were to use the reviewed multi-agent optimization techniques in multi-policy environments, e.g., CRL or coordination graphs, we would lack the means to combine multiple policies on those agents into a single optimization problem.

If we combined the existing multi-policy and multi-agent techniques, e.g., implemented one of the multi-agent algorithms globally and one of the multi-policy algorithms locally, we believe that heterogeneity of policies and heterogeneity of agents that implement them would give rise to a number of additional issues not present in either multi-agent or multi-policy cases individually. For example, we would need to decide if an agent should collaborate only with other agents that implement the same policies or with all agents in the system, should we, and how, make agents aware of each other's policies, and how will the priority of multiple policies be maintained if agents implementing lower priority policies are not aware of a higher priority policy being deployed on other agents.

Therefore, we believe that an algorithm that simultaneously addresses learning and optimization towards multiple-policies in multi-agent systems should draw on techniques from multi-policy optimization (e.g., it needs to balance the action preferences of multiple policies), techniques from multi-agent optimization (e.g., it needs to ensure that actions executed locally by an agent do not negatively affect another agent, or the system as a whole), as well as address additional issues that arise in multi-agent multi-policy environments.

To provide a basis for our research into such an algorithm we first evaluate several existing multi-policy RL-based optimization techniques on independent agents in Chapter 3, and based on the observations we design and present our proposed optimization algorithm, DWL, in Chapter 4. With

RL	single-policy	multi-policy
single-agent	e.g., Q-learning	e.g., combined state spaces, arbitration-based approaches
multi-agent non-collab	e.g., ILs in (Tan, 1993) and (Claus & Boutilier, 1998)	W-learning and combined state spaces case study (Chapter 3)
multi-agent collab	e.g., CRL, distributed value function, coordination graphs	DWL (Chapter 4)

Table 2.1: RL optimization techniques summary

this we complete the Table 2.1, providing a case study on use of ILs in multi-agent multi-policy scenario, and an algorithm for collaborative multi-agent multi-policy optimization, DWL.

2.4 Urban Traffic Control

In this section we present UTC, the application domain in which we have evaluated DWL. As we already argued in Chapter 1, we believe decentralized agent-based UTC is a suitable application area for DWL, as it consists of multiple heterogeneous agents (i.e., junctions with different layouts and traffic light settings) and needs to optimize towards multiple policies (i.e., address different vehicle types). Furthermore, there are potential dependencies present between agents and between policies, due to a shared operating environment (i.e., road network).

In the remainder of this section we first provide the UTC-related background necessary for a discussion of UTC techniques. We then review existing UTC approaches, and the latest research into agent-based optimization in UTC in order to position DWL's potential application in UTC optimization.

2.4.1 Glossary of UTC Terms

We first provide a glossary of terms that are frequently used in UTC and are necessary for an understanding of the UTC domain and UTC optimization techniques discussed in this section, as well as for an understanding of our simulation environment and evaluation scenarios presented later in this thesis.

Definitions in this section were taken from (Papageorgiou et al., 2003) unless otherwise stated.

- A *junction* (*intersection*) consists of a collection of approaches and a crossing area.
- An *approach* consists of one or more lanes with the same traffic direction.

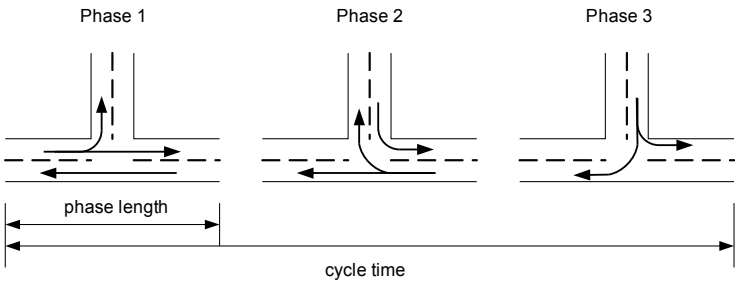


Fig. 2.10: Phases, phase length, signal cycle, and cycle time (Papageorgiou et al., 2003)

- A *signal cycle* is one repetition of the basic series of signal combinations at an intersection. The duration of a signal cycle is called *the cycle time*.
- A *stage* (*phase*) is a part of the signal cycle, during which one set of compatible streams of traffic have right of way. The duration of a phase is called the *phase length*. Figure 2.10 shows an example of a cycle that consists of three phases, and shows phase length as a part of the cycle time.
- A *phase scheme* (*staging*) refers to grouping of signal lights into phases and determining the order in which the corresponding phases will be deployed (Richter, 2006).
- A *split* is the relative duration of green time of a phase as a proportion of the cycle time.
- An *offset* is a phase difference between cycles for neighbouring intersections, that may give rise to a green wave.
- *Delay* is defined as the amount of time that is added to a journey time due to a vehicle being stopped at traffic lights (Pierre-Luc Gregoire & Chaib-draa, 2007).
- *Traffic density* is defined as the number of vehicles present on a road segment of a given length, and is expressed in vehicles per kilometer (veh/km). Density can also be expressed in percentage terms, as a ratio of vehicles present on a given road segment versus maximum number of vehicles that the road segment can accommodate.
- *Traffic volume* (*flow*, *throughput*) is defined as the number of vehicles crossing a given location during some time period, and is expressed in vehicles per hour (veh/hr).
- *Traffic demand* is defined as the number of cars that want to enter the road network at a given point in time or over a certain period of time (Richter, 2006).

- *Saturation flow rate* is the maximum number of vehicles from an approach that would pass through an intersection in one hour under current traffic conditions if that approach were given a continuous green signal for that hour (Kutz, 2003). A related measure is the *degree of saturation* (DS), defined as a ratio of the effectively used green time to the total green time given (Lowrie, 1982), and calculated as

$$DS = [available\,green - unused\,green]/available\,green \tag{2.3}$$

In the next section we review currently-deployed UTC systems and explain how they adapt traffic control parameters to optimize delay and throughput.

2.4.2 Commercial UTC Systems

In this section we review the currently most-widely deployed UTC systems, namely fixed-time plans, SCATS and SCOOT (Chowdhury & Sadek, 2003). We also review RHODES, a real-time traffic control system, which is not yet commercially available, but is currently undergoing field trials and is predicted to outperform current deployments of SCATS and SCOOT (Chowdhury & Sadek, 2003).

2.4.2.1 Fixed-Time Plans

The most basic UTC systems deploy fixed-time plans, where signal settings are fixed in that green periods and offsets do not vary from cycle to cycle. A set of phases, their order, and their length is preselected, and the traffic-light controller cycles through them in a round robin-like manner. Fixed time plans were originally designed manually by traffic engineers and in later deployments assisted by programs such as TRANSYT (Robertson & Bretherton, 1991), and are based on historical traffic data on a given intersection (Papageorgiou et al., 2003). Such UTC systems usually consist of several different fixed time plans designed for morning peak, midday, afternoon peak, and evening/nighttime conditions. In addition, special plans may be produced, for example for reoccurring major music or sporting events. The major disadvantages of fixed-time plans are that they are rarely kept up to date due to the complexity and duration of the design process for new plans and that they are not able to deal well with random fluctuations in traffic patterns (Robertson & Bretherton, 1991).

2.4.2.2 SCATS

SCATS (Sydney Coordinated Adaptive Traffic System) (Lowrie, 1982) was developed in the late 1970s by the Roads and Traffic Authority of New South Wales in Australia (Traffic Authority of New South Wales Australia, 2009) and is currently deployed in over 80 cities around the world, including

Dublin, Sydney and Hong Kong. SCATS obtains traffic counts and the distance between vehicles from traffic loops at lane stop lines and adjusts the cycle time and phase duration based on the degree of saturation at the approaches of a junction. SCATS aims to keep the junction saturation as close to a target percentage as possible, by shortening or lengthening the phase duration (Richter, 2006). Signals are grouped into subsystems, within which one critical junction decides on the sub-system's parameters. e.g., the cycle time and offset (Fellendort, 1997). Even though SCATS provides online adaptation in terms of the duration of a cycle and its phases, it still requires manual design of regional groups and selection of phases. Also, as SCATS is designed to equalize the saturation of conflicting approaches on a junction, it can fail to minimize delays on major roads (Chowdhury & Sadek, 2003).

2.4.2.3 SCOOT

SCOOT (Split Cycle Offset Optimization Technique) (Peek Traffic Limited, Siemens Traffic Controls, TRL Limited, 2009) has been developed in the early 1970s by the Transport and Road Research Laboratory in the United Kingdom, and is under continuous development with Siemens Traffic Control and Peek Traffic as industrial partners. SCOOT is currently deployed in over 200 cities worldwide including Cork, London, Madrid, Beijing and Toronto. SCOOT uses vehicle detectors positioned between 100 and 300 meters upstream from the stop line to count vehicles in the queue and anticipate traffic flows. Controllers communicate this information to a central computer that returns new signal timings by increasing or decreasing the offset or cycle time (McGuire & O'Keeffe, 2003). A disadvantage of SCOOT is that its traffic prediction models might not be accurate in traffic conditions where queues extend beyond the upstream detectors, i.e., oversaturated conditions (Abdulhai & Pringle, 2003).

2.4.2.4 RHODES

RHODES (Real Time Hierarchical Optimized Distributed Effective System) (Mirchandani & Head, 2001) was developed in the late 1990s by the University of Arizona. It uses a peer-to-peer communications approach to communicate traffic volumes from one intersection to another in real-time, based on counts obtained from both upstream and stop-line detectors. It recalculates phase timings every five seconds to take into account the most recent information based on three levels of estimation granularity - vehicles per hour, vehicle platoons, and individual vehicles (Mirchandani & Head, 2001). In the initial field trials, RHODES has been shown to outperform manually optimized plans, however, it is thought that currently its main applicability is limited to under-saturated traffic conditions and main traffic arteries (Curtis, 2003).

2.4.2.5 Selected Vehicle Priority

Existing UTC systems incorporate the possibility of giving priority at an intersections to certain vehicle types. In order of priority, from high to low, the vehicle types that need to be given priority are: emergency vehicles, trams/trolleys (i.e., vehicles constrained in their movement by tracks or overhead wires), and buses (S Jones & Fox, 1998).

Selected vehicle priority can be incorporated in UTC systems using so-called interventionist and non-interventionist strategies (S Jones & Fox, 1998). Non-interventionist strategies simply give more predefined green time to routes that, based on historical data, have a higher average flow of priority vehicles. Such strategies are generally applied within fixed-time plans, due to their inability to dynamically adapt when vehicles that should be prioritized are approaching. Interventionist strategies use green light extensions and recalls (i.e., either extend the green time to accommodate a priority vehicle or prematurely terminate a red signal), and stage skipping (i.e., violating the usual sequence of phases at the intersection by skipping to the phase that serves the vehicle that should be prioritized). SCATS and SCOOT use such interventionist strategies to incorporate public transport priority (S Jones & Fox, 1998), while RHODES provides an additional option of assigning public vehicles different levels of priority, where the priority increases with higher passenger numbers and when vehicles are behind schedule (Pitu et al., 2000).

Public-transport vehicle priority is generally not absolute, i.e., the impact of its prioritization on general congestion is evaluated, and vehicles are given priority only if the prediction of the resulting congestion is within set thresholds. On the other hand, emergency vehicles need to be given absolute priority, which is generally realized through use of green-waves, i.e., by setting the signal to green at all the intersections through which a vehicle is due to travel, at times that the vehicle is expected to reach them. The green wave can be triggered manually by a push button at the origin (e.g., a fire station) or by a vehicle detector at exit from the fire station (S Jones & Fox, 1998). This approach could be enhanced by the use of vehicle detectors along the route, in order to use the current vehicle position to readjust vehicle's estimated time of arrival at a particular location.

From the techniques currently used for selected vehicle priority as described above, we conclude that incorporating such vehicle priority into a UTC system is far from trivial. It requires a means of detecting vehicles that should be prioritized, deciding on the level of priority they should be given, evaluating the impact that prioritizing the vehicle could have on remaining traffic, evaluating the best approach to give priority to the vehicle (e.g., green wave, extension, recall, stage skipping), executing the signal change, and potentially compensating for the resulting congestion after the priority has been given (S Jones & Fox, 1998). Therefore, we believe selected vehicle priority needs to be integrated into

the design and development of a UTC system. However, as we will see in Section 2.4.3, this is often not the case, as the majority of RL-based UTC techniques currently address only personal vehicles and concentrate only on addressing overall traffic congestion.

2.4.2.6 Summary

Even though adaptive UTC systems (e.g., SCATS and SCOOT) show significant improvements when compared to fixed-time approaches, we believe that there is potential for further improvement in UTC through the application of decentralized agent-based learning techniques. Current adaptive systems require manual pre-configuration and phase selection (staging). Using learning techniques, UTC systems could learn the most suitable phases from a full set of possible phases, rather than only adapt the duration of phases in a pregiven set.

Both vehicle priority and general traffic flow could be additionally improved by learning the impact of one vehicle type on another by addressing all vehicle types simultaneously. For example, improvements could be obtained by clearing the bus or emergency vehicle routes prior to a vehicle arriving at a junction, and by learning to apply phases that serve the approach with the priority vehicle together with other non-conflicting approaches with the highest congestion. Additional improvements in the response time of controllers could result from the full decentralization of traffic light control.

In the next section we describe current research on decentralized agent-based UTC systems and the self-organizing learning algorithms used to implement them.

2.4.3 Agent-based Decentralized UTC Systems

Agent-based approaches to the management of UTC systems use a number of different self-organizing algorithms, e.g., RL, EAs, and ACO. We review examples of these implementations in this section with particular focus on RL-based techniques.

2.4.3.1 Reinforcement Learning

RL algorithms have been widely applied in the optimization of UTC systems (Abdulhai et al., 2003). Various RL-based approaches to UTC differ in whether the agents on which RL is deployed collaborate with each other or not, whether the reward that the agents obtain is calculated locally or globally, and on the scale at which the approaches were evaluated. Additionally, they can be categorized based on whether it is cars that implement learning agents (car-centric approach) or traffic lights (traffic-light (TL)-centric approach) or both.

Abdulhai et al. (2003) use a Q-learning agent to control a traffic light controller in order to minimize delay on a single, isolated junction. The state space encodes information about the queue length on all junction approaches, and the actions available to an agent enable it to either continue with the current phase or switch to the next one. Reward is inversely proportional to vehicle delay on all of the junction's approaches. Even though it is a very simple scenario and does not address any of the issues associated with the presence of multiple policies or multiple agents in the system, this work nevertheless contributes to assessing the suitability of RL to conditions in a UTC domain. Under uniformly distributed traffic conditions, this approach performs on a par with an existing pre-timed controller, while under variable traffic conditions it performs significantly better, due to its ability to learn behaviours suitable to varying traffic circumstances.

Camponogara & Kraus (2003) also use a TL-centric approach to UTC optimization, and implement Q-learning on two non-communicating junctions. The state space encodes information about queue lengths, actions are represented as TL phases, and the reward is inversely proportional to the number of vehicles waiting at an intersection. Simulations were performed with either just one or both of the junctions being controlled by an RL-agent. Not surprisingly, the largest improvements in vehicle waiting time were gained when both junctions implemented an RL process. This work shows that UTC can benefit from RL-based optimization even when junctions are modelled as independent agents and ignore the presence of other agents. However, this approach might not be suitable if policies on junctions are conflicting and we therefore believe that further improvements can be obtained by enabling cooperation between junctions.

In (Wiering et al., 2004a,b), both car-centric and TL-centric approaches were combined to minimize the average waiting time for vehicles. At each TL stop, cars estimate their waiting times for their total trip until they reach the destination (calculating estimates for both red and green lights at each intersection), and communicate this information to TLs. Based on this information, TLs, controlled by RL agents, learn to set the green signal for the TL configuration that minimizes the total estimated car waiting time. TL agents do not communicate with each other but base their decisions only on local information received from cars waiting on their approaches. This technique was evaluated on a relatively large scale, using 15 junctions, where it outperformed hand-designed algorithms used as baselines. However, we believe that, due to a lack of collaboration between agents, this approach might not be able to adequately address scenarios where dependencies between agents exist, particularly in the presence of multiple heterogeneous conflicting policies.

Even though it is primarily concerned with highway vehicle flow rather than UTC, we mention the work presented in (Pendrith, 2000) as an example of a car-centric approach to traffic flow optimization.

Each vehicle implements RL locally with the goal to optimize a global metric, i.e., optimize highway utilization. Each agent is assumed to be able to observe vehicles in 8 positions surrounding it (ahead left, ahead current, ahead right, clear left, clear right, behind left, behind current, behind right) and based on that information learn whether to stay in the current lane or change to the lane immediately left or immediately right of its current lane. We do not believe this approach is directly applicable to UTC, as in UTC scenarios the lane that the vehicle is in is often determined by its destination, i.e., desired direction at the next intersection.

Most closely related to our approach to UTC optimization using DWL is work by Richter (2006) and Salkham et al. (2008), due to their use of collaborative multi-agent RL optimization techniques.

Richter (2006) use the natural-actor critic RL algorithm (Peters et al., 2005) to optimize average vehicle travel time. Each intersection's environment is modelled as an MDP and each intersection implements an RL process to solve that MDP which receives a reward based on the number of vehicles it allows to proceed. TL agents communicate with their immediate neighbours, in order to use neighbours' traffic counts to anticipate their own traffic flows. The algorithm was evaluated for a number of different traffic patterns and on a large-scale involving up to 100 junctions, where it showed improvements over the simple saturation balancing algorithm SAT, which is based on SCATS (see Section 2.4.2). In a large-scale simulation, this approach required 3 days of real world experience to achieve performance equivalent to SAT.

Salkham et al. (2008) use a technique based on CRL (see Section 2.3.3.1) to implement a traffic-control technique called Adaptive Round Robin (ARR-CRL). On each agent, ARR cycles through phases available to a junction, and it either skips the phase or sets it with one out of a set of predefined durations, based on congestion levels on the approaches served by that phase. An agent is rewarded proportionally to the number of cars that pass through an intersection during the phase, and inversely proportional to the number of cars still waiting at an intersection. Agents periodically exchange information on their performance (their accumulated rewards) with their immediate neighbours and incorporate the information received from neighbours into their own reward. Agents, therefore, receive rewards for the good performance of their neighbours, ensuring cooperation between them. ARR-CRL was evaluated in large-scale (60+ agents) simulations and shows significant improvements over a simple saturation balancing algorithm based on SCATS (see Section 2.4.2) and fixed-time controllers.

Neither (Richter, 2006) nor (Salkham et al., 2008) address different vehicle types, but only aim to optimize general traffic flow. The collaboration approach in (Richter, 2006), where agents exchange their traffic flow information with their neighbours, could be utilized in multi-policy approaches to inform neighbouring junctions of a higher-priority vehicle approaching them. However, we do not

consider the overall approach suitable due to long training times required for large-scale applications. The collaboration approach in (Salkham et al., 2008) consists of exchanging rewards, and therefore we believe is not suitable to multi-policy optimization, at least not in its current form. In a multi-policy approach, exchanged rewards would need to be associated with information such as policies that neighbours' are optimizing for and their relative priorities.

2.4.3.2 Other Agent-Based Approaches

In the previous section we have reviewed RL-based approaches to UTC optimization. In this section we review examples of other self-organizing algorithms applied in UTC, specifically EAs and ACO.

EAs are commonly used in UTC in combination with other techniques, where they are utilized to find the optimal combinations of a set of parameters. For example, in (Mikami & Kakazu, 1994) and (Yang et al., 2005), RL is deployed locally on individual traffic light agents, but globally EAs are used to find optimal combinations of parameters for the local RL processes. In (Prothmann et al., 2008), EAs are used in combination with Learning Classifier Systems (LCS) (Bull, 2004). EAs evolve traffic light parameters for a specific traffic situation and evaluate them in an offline simulation. Determined parameters are, together with corresponding traffic conditions, stored in the LCS's rule-base. Based on observed traffic conditions, a traffic light selects one of the pre-determined sets of parameters from the rule-set. In (Bazzan, 2005), evolutionary game theory is used to model individual traffic controllers as agents that are capable of sensing their local environment and learning optimal parameters for continually changing traffic patterns. Agents receive both local reinforcement from their local detectors, and global reinforcement based on global traffic conditions. For example, if the majority of traffic travels westbound, agents receive higher payoffs for giving longer green signals to that direction. However, in this approach global matrices of payoffs need to be specified by the designer of the system for each set of traffic conditions (Bazzan, 2009), and as such require domain knowledge to construct.

ACO algorithms can be used in UTC when optimization is car-centric, i.e., to provide routing information to vehicles rather than to control traffic signals. For example, Hoar et al. (2002) combine car-centric ACO-based optimization with EA-based TL-centric optimization. Vehicles act independently to reach their destination and deposit information about their route in the form of a pheromone equivalent for potential use by other cars. In addition, each car that is stopped at a traffic light casts a vote on their status that is transmitted to that traffic light. Traffic lights then use EAs to, based on the votes received, adapt phase timing and sequence to improve the overall waiting time.

2.4.3.3 Vehicle Priority in Agent-Based UTC Systems

All of the approaches to multi-agent optimization in UTC reviewed above are concerned only with a single traffic policy of optimizing global traffic flow, by maximizing the throughput or minimizing travel/waiting time. There is significantly less research implementing multi-agent approaches to optimization towards other possible goals of a traffic management system, such as prioritizing emergency vehicles or public transport vehicles. We review such research in this section.

Oliveira & Duarte (2005) incorporate emergency vehicle priority into their traffic simulation. Each traffic light agent observes its local traffic conditions and, if an emergency vehicle is observed, sets the signal on the lane on which the vehicle is present to green. Additionally, an agent can communicate this event to downstream junctions, to inform them of the possibility of an emergency vehicle approaching. Even though this approach incorporates optimization for multiple vehicle types simultaneously, the prioritization of emergency vehicles is rule-based rather than based on learning. We believe this approach has the potential for further improvement by learning the most suitable phase setting for an approaching emergency vehicle, based on its previous experiences of routes that emergency vehicles take. Using this method, the relevance of the information sent to the downstream junctions can also be improved, as only the junction that is on the ambulance route need be informed, rather than all of the downstream junctions.

Febbraro et al. (2004) use Petri nets (Murata, 1989) to model each junction in a simulation of a UTC system, representing vehicle flows entering the junction, leaving the junction, and a traffic light controller. The TL control system consists of a local controller and a priority controller on each junction, and a global supervisor. Each local controller aims to minimize the traffic queues and equalize queue lengths across a junction's approaches. When an emergency vehicle enters the system, it notifies the global controller, which calculates the shortest path (in terms of waiting time) for an emergency vehicle to take. It then notifies all of the local junction controllers on the path of the time at which an emergency vehicle is estimated to reach them. Based on this information, the local priority controller can either extend the current green signal or shorten future red signals, to ensure an approach with an emergency vehicle receives a green light. Bus priority is implemented only by local priority controllers once they detect the bus on a junction's approach, and does not involve the global controller. The authors evaluate this approach in terms of emergency-vehicle and public-transport-vehicle waiting time, which show improvements over fixed-time controllers, however they do not evaluate the impact of such priority implementation on the remaining traffic, i.e., do not evaluate the impact of policy dependency. Traffic light controllers act independently and do not cooperate with their neighbours, therefore not accounting for potential agent dependencies either.

2.4.3.4 Summary

RL-based approaches to decentralized UTC management show significant improvements when compared to hand designed fixed-time controllers and simple adaptive techniques based on the systems currently in use. However, the existing approaches learn appropriate actions only for a single traffic policy (optimizing global flow) and either do not take into account other policies at all, or address them using rule-based priority systems, without addressing the impact that such priority has on the remaining traffic.

We believe that there is potential for further improvement in the performance of policies by addressing all vehicle types and policies simultaneously. Due to the shared road network, there are potential dependencies between different policies, i.e., between the performance of different vehicle types, and addressing only one at a time can have a negative impact on all policies deployed. The impact of agent-based optimization towards one policy on another needs to be investigated, as what are thought to be optimal actions for one policy in isolation, might not be optimal actions for that policy once other policies are simultaneously deployed.

Therefore, we believe that the implementation of UTC systems using DWL could improve their performance as it enables them to simultaneously optimize for multiple policies, and to learn and address dependencies between different vehicle types and between junctions, in order to improve the performance of all vehicle types and of the overall system.

2.5 Summary and Conclusion

This chapter introduced autonomic computing, focusing in particular on decentralized autonomic systems. We have argued that self-organizing multi-agent systems are a suitable technique for the engineering of such decentralized autonomic systems. We have focused on the engineering of self-optimizing behaviour in these systems, using self-organizing techniques such as ACO, EA and in particular RL. We argued that RL is particularly suitable for the implementation of decentralized autonomic systems, and have reviewed a number of RL techniques currently used in multi-agent and autonomic systems. We concluded that the techniques reviewed enable either self-optimization towards multiple policies on a single agent, or self-optimization of multi-agent systems towards only a single policy, identifying a gap for a multi-agent multi-policy RL-based optimization technique. Later in this thesis we propose such a self-optimization technique, DWL. In this chapter we have also introduced UTC, the domain in which we evaluate DWL. We have reviewed currently deployed UTC systems, as well as RL-based techniques for UTC optimization, concluding that UTC can benefit

from a DWL-style optimization approach, which addresses multiple traffic policies and vehicle types simultaneously.

Chapter 3

Non-Collaborative Multi-Policy Optimization in Autonomic System

"*Before software should be reusable, it should be usable.*"
– Ralph Johnson

In the previous chapter we presented a review of the existing multi-policy single-agent RL-based optimization techniques. In this chapter we present a case study in which we evaluate some of these techniques in a multi-agent scenario, in order to identify a suitable basis for designing a multi-policy multi-agent optimization technique. We implement multi-policy techniques simultaneously on multiple agents, to observe how they behave in multi-agent scenarios, and identify requirements for a collaborative multi-policy technique. Note, in particular, that the experiments presented in this chapter are implemented on independent non-communicating agents (i.e., independent learners), as existing techniques provide only for single-agent implementations. We first present our simulation environment, which is also used for the evaluation of DWL in later chapters. We then describe the policies and deployment scenarios we have evaluated in this case study, present the results, and draw the conclusions which motivate the design of DWL[1].

[1]Parts of the case study presented in this chapter have been published in (Dusparic & Cahill, 2009c) and (Dusparic & Cahill, 2009d)

3.1 Case Study Objectives

The case study presented in this chapter was designed to assess the suitability of existing multi-policy RL-based techniques for optimization in multi-agent autonomic systems and their potential to serve as the basis of a multi-policy multi-agent optimization technique. In order to do so, we have implemented and evaluated several single and multi-policy UTC scenarios. We use currently deployed UTC techniques as baselines for assessing the suitability of multi-policy RL-based approaches for optimization in UTC. We use single-policy scenarios to evaluate the impact that policies targeted at one vehicle type have on other vehicle types, as well as to serve as baselines for the evaluation of multi-policy scenarios.

As policy heterogeneity is a central issue, for the initial evaluation we selected two policies that differ in all three of our policy classification criteria: priority, temporal scope, and spatial scope. The single-policy scenarios we implemented are as follows:

1. Global Waiting Time Only (GWO) - a policy that aims to optimize waiting time for all the vehicles in the system.
    - Spatial scope: global (it addresses all of the vehicles in the system, and therefore is implemented by all of the junctions in the system).
    - Temporal scope: continuous (it is active for the entire duration of system operation, as there are vehicles always present in the system).
    - Priority: standard (it treats all vehicles with the same standard priority).
2. Emergency Vehicles Only (EVO) - a policy that aims to prioritize emergency vehicles only.
    - Spatial scope: regional (implemented only by the junctions through which an emergency vehicle travels).
    - Temporal scope: sporadic (active only when an emergency vehicle is present in the system).
    - Priority: high (emergency vehicles need to be given a priority over other vehicles in the system).

We combined the policies above using two single-agent multi-policy techniques to implement the following multi-policy scenarios:

1. Combined state space (GWEV-c), where GWO and EVO are combined into a single learning process over a single state space.

2. W-Learning (GWEV-w), where GWO and EVO learn the best actions separately as two separate Q-learning processes, but W-learning is used to determine which action is to be executed.

We have selected the multi-policy scenarios above to represent the two main types of approaches in which RL-based methods address multiple policies: combined learning and arbitration-based approaches (see Chapter 2). We implemented a combined state space approach as it enables the precise description of the state space in terms of both vehicle types that both policies are addressing, and aims to learn the most suitable action for all possible combinations of individual policy states. We implemented an arbitration-based approach, as, even though it is less precise in terms of state space description for a combination of policies, it is expected to be less computationally expensive and more scalable. In particular, we selected W-learning as, unlike most other arbitration-based approaches discussed in Chapter 2, it does not use the absolute priority of policies in order to select an action for execution, but enables policies to learn their own importance specific to the particular state and to the other policies deployed simultaneously. Also, W-learning implementations have an option of either selecting a compromise action to be executed, or giving control to a single policy. Giving control over action selection to a single policy is of crucial importance when one of the policies has a high priority, as executing a compromise action might lead to meeting the requirements of several low-priority policies instead of the policy with the highest priority.

3.2 UTC Simulation Platform

For the evaluation of the scenarios described in the previous section, and for the evaluation of DWL as described later in this thesis, we use an urban traffic simulator developed in Trinity College Dublin (Reynolds, Cahill, & Senart, 2006). The simulator uses a microscopic traffic simulation approach (i.e., it simulates vehicles on an individual level), and can simulate traffic over any road network defined by a map described in a specific XML format. The simulator can distinguish between multiple vehicle types, such as cars, public transport vehicles, and emergency vehicles. Vehicles implement different behaviors based on their type. Emergency vehicles are capable of driving above the allowed speed limit, as well as driving through red lights if it is safe to do so. Public transport vehicles (buses) are larger in size and stop at designated bus stops.

The map we used for the initial experiments presented in this chapter is shown in Figure 3.1. The map is based on road layout details provided by Dublin City Council and corresponds to one of the highest profile areas of Dublin's road network, O'Connell Street (Dublin's main street) and several side roads that feed traffic onto this road. Using a map based on real road network provides a more realistic

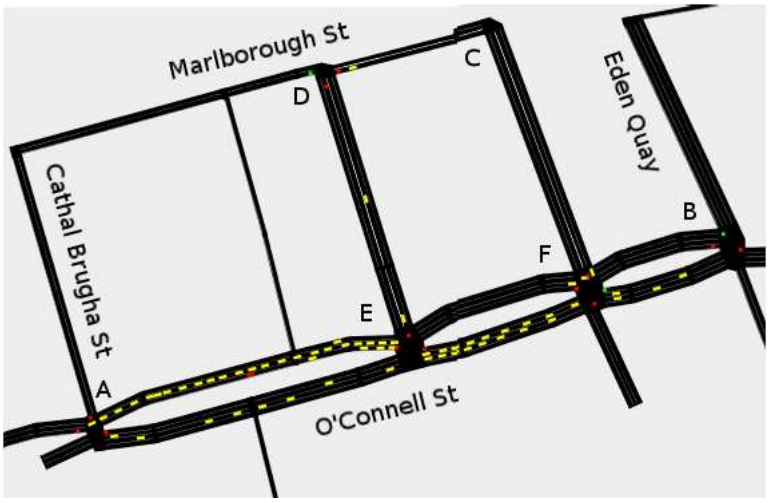


Fig. 3.1: Experiment map in Dublin UTC simulator

simulation; many of the simulations used for the evaluation of multi-agent systems in UTC covered in Chapter 2 use either a single junction, or multiple junctions and road links that have similar layouts, while the map that we use includes junctions of various layouts (e.g., junctions with two, three, and four approaches and exits), roads of differing width (e.g., roads with two, three, and four lanes), as well as one-way and two-way roads. The map covers 8 junctions, 5 of which are signalized junctions (junctions A, B, D, E and F in Figure 3.1) and are controlled by the agents described in the following section. Each agent has a set of available phases, generated based on intersection layout and allowed traffic directions. Each phase is mapped to an action that the agent is able to execute. This leads to RL-based agents having different action sets. For example, due to differences in road layout, the size of the action set for junction D (which has three approaches and two exits) is 3, while the size of the action set for junction F (which has four approaches and three exits) is 9.

3.3 Policies and Agent Implementation

In this section we describe the design of agents that implement our policies as well as the design of agents that simulate the performance of existing UTC optimization techniques, which we have used as a basis for comparison.

3.3.1 Baselines

As baselines for the evaluation of performance of RL agents we have used a round robin controller and a controller implementing a simple adaptive technique.

Round robin Round Robin (RR) junction controller, at each junction, continuously cycles through all of the available phases at that junction, setting them for a fixed duration in a fixed order.

SAT Simple adaptive technique (SAT) is a simple SCATS-like traffic-responsive UTC technique, as defined by Richter (Richter, 2006), that adjusts phase duration based on the degree of saturation at a junction. The degree of saturation is defined as a ratio of the effectively used green time to the total available green time. At each junction, SAT, in a similar manner to SCATS, aims to keep the junction saturation as close to 90% as possible, by shortening or lengthening the phase duration. A SAT implementation depends on three parameters: the *minimum duration* of each phase, the *phase increment* (the length by which a phase duration can increase or decrease in a single step), and a *maximum cycle length factor* (the maximum duration of a cycle is determined by multiplying the number of phases at a particular junction by the minimum phase duration and a cycle length factor).

3.3.2 Single-Policy deployments

The single-policy deployments we have implemented are global, continuous, standard priority policy GWO, and regional, sporadic, high priority policy EVO. We present their implementation details below.

GWO - Optimizing global waiting time The first policy we implemented, optimizing Global Waiting Time Only (GWO), optimizes waiting time for all vehicles in the whole system. Since global waiting time is a sum of waiting times for all cars at all junctions in the system, and we assume no collaboration between agents, we aim to minimize the waiting times at each individual junction.

Each agent is capable of sensing the number of vehicles at each of its approaches, and maps that to a state space that orders approaches according to their level of congestion. For example, on a junction with two approaches, a_1 and a_2, a state can be "Congestion order: a_1, a_2", meaning that approach a_1 has more traffic waiting than a_2, or "Congestion order: a_2, a_1", meaning that approach a_2 has more traffic waiting than a_1. The state space does not encode how many vehicles are waiting at which approach as the numbers are relative to overall congestion in the system (e.g., knowing that there are 9 vehicles waiting at an approach would not tell us whether that is a high or low number

relative to the current traffic conditions). It also contains information about whether the total number of vehicles waiting at the junction is more or less than at the previous phase change (e.g., a state can be "Congestion order: a_1, a_2, less vehicles than before" or "Congestion order: a_1, a_2, more vehicles than before"). Note that arrival rates are uniform in this set of experiments, so a change in the number of vehicles waiting is caused by an agent's action, rather than a drop in demand, and as such is a reflection on an agent's performance. We designed the state space in this manner to facilitate rewarding an agent (100 points in this set of experiments) for being in a state with less traffic waiting than at the previous decision point, i.e., to motivate it to execute actions that clear more traffic than arrives at the junction during the action execution. Agents receive no reward for being in a state where there are more vehicles waiting at an approach than at the previous time step. This enables agents to learn how to reduce the number of vehicles waiting at the junction's approaches, thus reducing global waiting time for the system. Note that, since junctions have different layouts, the size of the state space designed in the way described depends on the number of approaches, and is calculated as $NumberOfApproaches! \times 2$ (The factorial of the number of approaches allows for all combinations of congestion order that approaches can be in, and is multiplied by two to allow for each state to be split into a "less vehicles than before" and "more vehicles than before" states).

EVO - Prioritizing emergency vehicles The other single policy that we implemented minimizes waiting times for Emergency Vehicles Only (EVO). An agent's state space encodes information about which approach(es), if any, have emergency vehicles present on them (e.g. "Ambulance present on a_1"). The size of the state space depends on the layout of junction, i.e., on the number of approaches on the junction. Emergency vehicles can be present on all of the approaches, on none of the approaches, on only one approach at a time, or on various combinations of two or three approaches at a time. Agents are rewarded (200 points in this set of experiments) for being in a state where there is no emergency vehicle present at any of the approaches. This motivates the agents to, as soon as possible, return to the state with no emergency vehicle present, by enabling emergency vehicles to travel through the junction. This policy does not address any other vehicle types and only takes emergency vehicles into account when making action decisions.

3.3.3 Non-Collaborative Multi-Policy Deployments

The multi-policy deployments we have implemented are GWEV-c and GWEV-w, which combine GWO and EVO using combined state spaces and W-learning, respectively. We present the details of these deployments below.

GWEV-c: Merging RL processes One way to combine multiple policies on a single agent is to encode all the information relevant for all the policies into a single state space and a single learning process. We use this approach to combine GWO and EVO into a single policy, GWEV-combined (GWEV-c). The state space of GWEV-c consists of the cross product of the state spaces for GWO and EVO. Therefore, the size of the state space is a product of the sizes of the state spaces for individual policies leading to a quite large state space (e.g., for four-approach junction, the size of the GWO state space is 32, and for EVO it is 15, so the size of GWEV-c state space is 32 $\times$15 =465). An agent receives a reward of 100 points for being in any of the states with less traffic than in the previous phases (i.e., states for which GWO receives a reward), a 200-points reward for any of the states with no emergency vehicles present (i.e., states for which EVO receives a reward), and the sum of both rewards for being in a state that satisfies both criteria. We acknowledge that the size of the state space in GWEV-c is not only large for the policies we are evaluating in this scenario, but also, as the number of policies to be combined increases, will not be scalable due to state-space explosion, however, our purpose was to compare its performance to other techniques in order to provide insight into how to deal with multiple policies.

GWEV-w: W-learning In the other multi-policy deployment that we implemented, GWEV-w, both policies are addressed using W-learning action selection (see Chapter 2). GWO and EVO are first deployed separately, to learn Q-values, and then deployed simultaneously to learn W-values. At every time step, on each agent, both policies nominate an action, based on their Q-values, together with an associated W-value for the states in which they are currently. The action proposed by the policy with the higher W-value is executed. In our experiments, since EVO is a sporadic policy, we deem it inactive when there are no emergency vehicles present, and set the weight of the action that the EVO policy nominates in such a state to zero. As W-learning is deployed on top of existing single-policy approaches, there is no increase in state-space size for the policies in GWEV-w, but the duration of the learning process is increased, as after Q-values are learnt, W-learning requires learning of W-values as well.

3.3.4 Implementation Remarks

A major distinction between the performances of RR, SAT, and our RL-based approaches is the way in which they adapt to the traffic demand. RR does not provide any adaptivity and cycles through the same set of phases of the same duration regardless of the demand. SAT also cycles through the same set of phases in all traffic conditions, but can shorten or lengthen the duration of each phase to

respond to the demand. In our RL-based approach, agents start the exploration process with a full set of phases that are possible based on the intersection layout and allowed traffic maneuvers. During the learning phase, an agent learns the most suitable phases to set for each state, based on traffic demand and direction, so only a smaller subset of those phases is used in the exploitation phase. This approach ensures that the best phases are selected from the full available set and removes the need for manually selecting the phase set at design time by traffic engineers, as required by currently deployed UTC systems.

The design of state spaces and reward models for both single- and multi-policy scenarios have been influenced by the decentralization of the system, lack of global view or control, and the decision to implement non-communicating agents in this set of experiments. Therefore, agents can measure only their own local performance, based on their local environment conditions, and are not aware of the performance or rewards received by other agents in the system or system as a whole.

3.4 Evaluation

This section presents the details of the experimental parameters used in the evaluation of the existing RL-based techniques for multi-policy optimization. It presents the vehicle paths and vehicle numbers used in the evaluation, the RL parameters, and the metrics used for comparison of the performance of our evaluation scenarios.

3.4.1 Simulation Setup

In our simulation, cars enter the road network at four different points (marked A, B, C, D in Figure 3.1) and exit the system at two different points (A, B), following 1 of 4 paths: A to B, B to A, C to A, and D to B. Emergency vehicles tend to use major routes wherever possible, so in our simulation they only travel on paths A to B, and B to A. Therefore, the EVO policy is only deployed on agents A, B, E, and F. All vehicles follow the shortest path from source to destination. Vehicle routes are the same for all of the experiments we ran.

Agent performance is tested under three different traffic loads to simulate different traffic conditions. The loads are as follows:

- low load - a total of 28,140 vehicles are inserted over 2000 minutes (7,000 cars on each of the car routes and 70 ambulances on each of the emergency vehicle routes) corresponding to a flow of ∼850 vehicles per hour.

- medium load - a total of 56,280 vehicles are inserted over 2000 minutes (14,000 cars on each of the car routes and 140 ambulances on each of the emergency vehicle routes) corresponding to a flow of ∼1700 vehicles per hour.

- high load - a total of 100,500 vehicles are inserted over 2000 minutes (25,000 cars on each of the car routes and 250 ambulances on each of the emergency vehicle routes) corresponding to a flow of ∼3000 vehicles per hour.

According to traffic counts from February 2009 (Ghosh, 2009), O'Connell street (the area which we have simulated in this experiment) has an average hourly flow during peak time (7am-9am) of ∼1700 vehicles. We have chosen to use that figure to represent our medium load, and use low load to simulate off-peak traffic, and high load to simulate extremely congested conditions, for example, during major sporting events.

Each signalized junction in the simulation has a different set of available phases, generated at simulation start-up based on junction layout. Junctions can cycle through their available phases using RR, or can be controlled by SAT or one of the RL agents described in the previous section. For this set of experiments, the duration of each phase in RR and RL-based approaches is set to 20 seconds, while in SAT 20 seconds is used as a minimum phase duration (as SAT has the ability to adapt phase duration).

3.4.2 Experiment Parameters

The single-policy RL experiments presented here ran in two parts: 2010 simulation minutes of exploration, and 2010 minutes of exploitation. GWEV-w, the W-learning based experiment, was run for an additional 2010 minutes between the exploration and exploitation phase, to enable W-value exploration after Q-values have been learnt. The duration of 2000 minutes enables Q-learning and W-Learning to execute 6000 learning steps (as our actions are of 20 seconds duration each) which, we found sufficient for agents to learn the Q-values for their state-action pairs. An additional 10 minutes were added to allow an opportunity for the last inserted vehicles to leave the system. GWEV-c has a much larger state space than the other policies and therefore was given a longer exploration phase of 20,000 minutes to enable a larger portion of the state space to be visited a sufficient number of times. In all scenarios actions are selected for execution (or per policy nomination in the case of W-learning) using Boltzmann action-selection (see Section 2.3.1), with the temperature starting at 10000 and cooling down uniformly for the duration of the exploration period. The exploitation period is initialized using Q-values and W-values learnt during the exploration phase, and the Boltzmann

temperature is set to 1 for the duration of exploitation phase.

Prior to the experiments presented here, each RL process has been run multiple times to determine the best combination of α and γ (refer to Section 2.3.1 for the meaning of these parameters). The final combinations used are, for GWO: $\alpha = 0.1$ and $\gamma = 0.3$, for EVO: $\alpha = 0.9$ and $\gamma = 0.1$, for GWEV-c: $\alpha = 0.1$ and $\gamma = 0.1$ and for GWEV-w: $\alpha = 0.1$ and $\gamma = 0.7$. The best parameters determined for SAT performance with a minimum action duration of 20 seconds are 10 for the phase increment, and 1.2 for the maximum duration of the cycle factor. Each experiment is repeated three times, and the average results from the exploitation phase only are presented. We performed two-tailed t-tests on sample data to compare the performance of various algorithm and policy combinations, and define a statistically-significant difference in performance to be one where a test resulted in a p value of $p > 0.05$.

3.4.3 Metrics

We compared the performance of the RL agents based on the following metrics:

- Traffic density - Traffic demand is the same for all of the experiments described in this chapter. Traffic density can, therefore, be used as a measure of an agent's performance, as the differences in the number of vehicles present on the road will be determined by how quickly vehicles arrive at their destination and are cleared out of the system. Higher density therefore means poorer agent performance, since traffic that is not successfully cleared and is still in the system is causing an increase in density.

- Vehicle waiting time - Average waiting time per vehicle for the duration of the experiment. We separate waiting times per vehicle type, so we can measure performance of each of the individual policies that address different vehicle types.

- Number of vehicles served - The number of vehicles attempting to join the system and travel to their destination is the same for all experiments performed for a certain traffic load. However, not all vehicles are served, as for them to join the system, there needs to be available road space. If traffic light agents are not successfully clearing the traffic, the density increases, there is less available road space, and more vehicles get turned away. Lower number of vehicles served during the whole experiment, therefore, means poorer agent performance.

Even though the goal of single-agent single-policy RL-based learning is to optimize the long term reward received, we have not used total reward obtained in the system as a metric for this set of

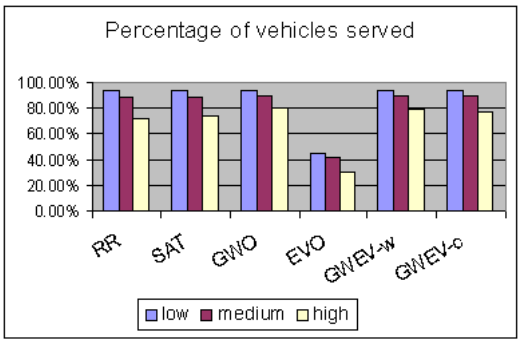


Fig. 3.2: Number of vehicles served per agent type per load

experiments, as maximizing the total reward is not the goal of all of our agents. GWEV-w, which is W-learning based, does not aim to maximize the reward obtained, but to minimize the worst unhappiness (as discussed in Section 2.3.2.2), i.e., to prioritize the policy that has more to lose, which is generally a higher priority policy. Also, GWO and EVO optimize only for a single policy and can receive rewards only in relation to a single policy, while GWEV-w and GWEV-c address two policies so can receive rewards from both policies, making their total reward incomparable to total reward received by single-policy approaches. Also, RR and SAT baselines do not use RL (i.e., they do not receive rewards for their performance), so total reward of RL-based approaches would not provide insight into how these approaches compare to our baselines. Consequently, we have decided to use UTC-related performance metrics, namely density, vehicle waiting time, and number of vehicles served, as described above.

3.5 Results

Figure 3.2 shows the number of vehicles served during each experiment, Figure 3.3 presents average waiting times for cars and emergency vehicles for the experiments performed, and Table 3.1 shows traffic density, all separated by traffic load. We have analyzed the results with respect to the case study objectives outlined in Section 3.1. We assess the suitability of multi-policy RL-based methods to UTC by comparing them to RR and SAT, compare single-policy deployments with multi-policy deployments, and compare the two multi-policy deployments to one another.

3.5.1 Multi-Policy RL vs. Baselines

In this section we compare the performance of RL-based agents with the performance of the baselines, RR and SAT.

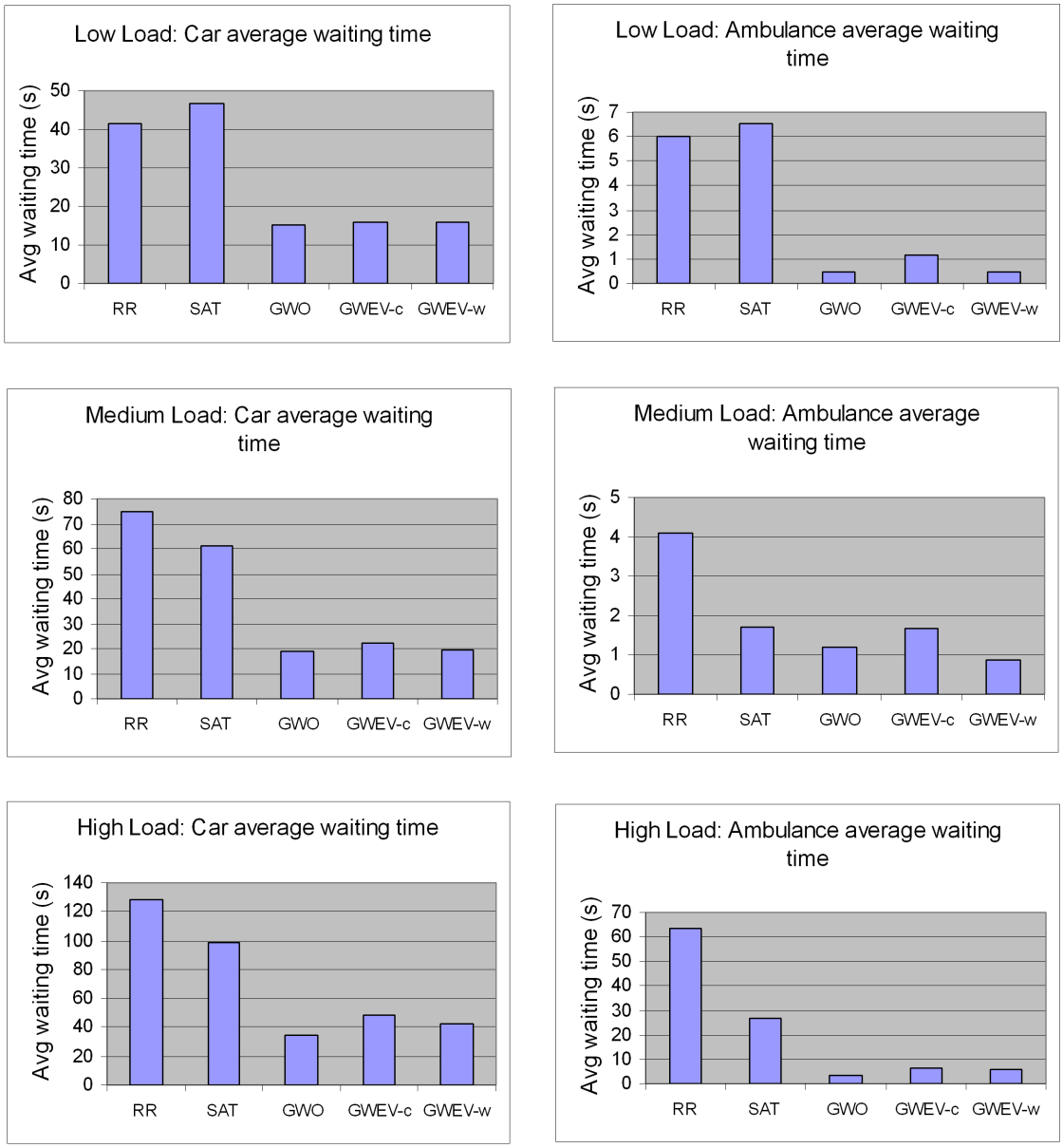


Fig. 3.3: Non-collaborative multi-policy optimization waiting time results

	RR	SAT	GWO	EVO	GWEV-c	GWEV-w
Low	2.96	2.76	1.66	12.30	1.60	1.49
Medium	5.60	5.20	3.31	11.37	3.47	3.09
High	11.04	9.50	5.84	15.85	6.03	5.06

Table 3.1: Average density per traffic load (percentage)

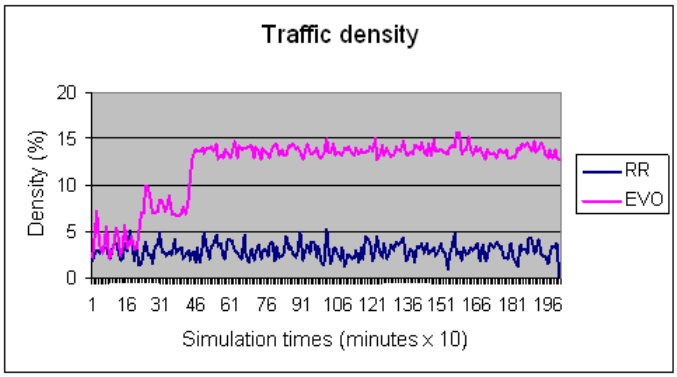


Fig. 3.4: EVO density during low load

Before we present a detailed analysis of the results, we need to address the performance of EVO in our experiments. As shown in Figure 3.2 (which shows number of vehicles served during the operation of each algorithm), we observe that the number of vehicles served by EVO is less than half of the vehicles served by any other algorithm, for all traffic loads. Figure 3.4 and Table 3.1 showing traffic density for EVO shed more light on this; density during the performance of EVO increases to the point of maximum saturation about 45 minutes into the simulation, and is kept at the maximum from that point on, due to EVO's inefficiency in clearing traffic.

We believe that this inefficiency is due to the fact that EVO addresses only emergency vehicles, which make up only 0.5% of the traffic in our simulation. Cars, which make up a remaining 99.5% of the traffic, are not addressed, and create a backlog in the system. The system fills up with the traffic that is not adequately addressed by the policy, creating high density and preventing new vehicles from joining the system. This results in EVO being able to serve only 30,000 vehicles at high load, while other algorithms serve between 72,000 and 80,000. For this reason, EVO waiting time results are not comparable to other results and we exclude them from the graphs showing waiting time results for other algorithms, as presented in Figure 3.3. We believe that the poor performance of EVO in our experiments highlights policy dependency in multi-policy systems where policies exist in a shared environment. If only one policy is addressed, performance of other policies can degrade to the point where it negatively affects the performance of the policy being addressed as well. In this case, cars that were not addressed by EVO, saturate the system and negatively affect the performance of ambulances, which then cannot be served due to the lack of available road space.

Apart from EVO, all of the other RL-based approaches that we have implemented perfom statistically significantly better than the RR and SAT techniques, with t-tests returning values of p ranging between 1×10^{-6} and 5×10^{-6}, depending on the scenario and algorithm. RL-based techniques reduce

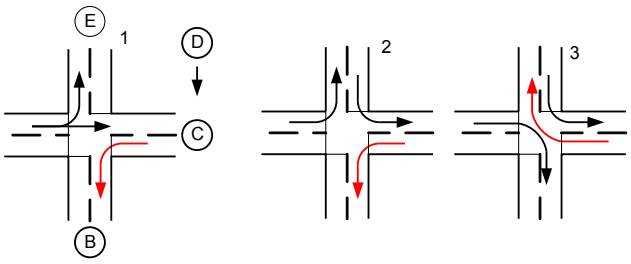


Fig. 3.5: Traffic light phases available to agent F

vehicle waiting time by 42-92% compared to their waiting time in RR, and by 3-93% compared to their waiting time in SAT, depending on the vehicle type, traffic load, and specific RL approach. The density and number of vehicles served results also confirm these findings. RL-based approaches have a lower density at all three loads than both SAT and RR, and serve between 3000-9000 more vehicles at high load than RR/SAT. These results indicate that RL-based approaches are suitable for optimization in UTC systems, as they outperform currently deployed UTC control techniques in these experiments.

Superior performance of RL-based approaches over our baselines is a result of RL agents learning which phases to use in order to release the most congested approach, rather than cycling through all phases regardless of the traffic conditions. Additionally, as several different phases can serve a single approach (enabling different travel directions, i.e., left, straight, right), an agent also learns which of those phases results in receiving a reward, therefore learning to use the phase that corresponds to the desired travel direction of majority of traffic on that approach. For example, consider Figure 3.5, representing a subset of phases of agent F. All three phases shown serve an approach C, however, if the highest amount of traffic that is waiting on the approach C is travelling towards B, junction F learns to prefer phases 1 and 2, rather than phase 3, as if it learns that traffic from that direction tends to turn left (following D to B route in our experiments).

It is also interesting to observe the performance of our baselines, SAT and RR, in relation to each other. At low load, in terms of waiting time, RR performs better than SAT, shortening vehicle waiting times by 12% for cars and 8% for ambulances. However, at medium and high load, SAT performs better than RR, where SAT vehicle waiting times are 19-24% shorter for cars and 59% shorter for ambulances than those in RR. This indicates that when the loads in the system are very low, running an adaptive algorithm such as SAT, might have adverse effects on traffic performance, potentially due to extending phase times to longer than is required and creating larger backlogs. However, these results also emphasize the importance of adaptation at higher loads, which increases as the load in

the system increases (indicated by the difference between the performance of RR and SAT being more significant during high load than during medium load).

3.5.2 Single-Policy vs. Multi-policy RL

In this section we compare the performance of single-policy GWO to multi-policy GWEV-c and GWEV-w. The performance of other single-policy approach, EVO, has already been discussed in the previous section.

Addressing only cars using a single-policy GWO is shown to result in the lowest average car waiting time in this set of experiments. GWEV-w, however, results in lower traffic density than GWO, indicating that the introduction of the second policy that multi-policy GWEV-w is addressing does not have a significant negative impact on car performance (as cars make up 99.5% of the traffic and density, therefore, reflects mostly the performance of cars). The difference in resulting average waiting time of cars under GWO and under GWEV-w is statistically significant under low load, while it is not statistically significant under medium and high load. GWO also results in lowest average ambulance waiting time at low and high load (even though it does not explicitly prioritize them but treats them in the same manner as other vehicles), and follows GWEV-w closely as the second best approach for ambulances at medium load. The difference in resulting average waiting time of ambulances under GWO and under GWEV-w is statistically significant under high load, while it is not statistically significant under low and medium load. This set of results therefore suggests very similar peformance of GWO and GWEV-w, where their relative suitability depends on the load and vehicle type. The similar performance of single-policy GWO and multi-policy GWEV-w suggests a high dependency between the performance of different vehicle types, i.e., the performance of different policies. It emphasizes the importance of clearing general traffic, as GWO does, to free up the road space for emergency vehicles so they can freely proceed once they enter the system, and suggests a high dependency between a policy that addresses emergency vehicles and one that addresses private vehicles.

The other multi-policy approach GWEV-c is outperformed by single-policy GWO in terms of average vehicle waiting time for all vehicle types and loads. The difference in average vehicle waiting time is statistically significant under low load for both vehicle types ($p = 0.0001$ for cars and $p = 0.01$ for ambulances), and is not statistically significant under medium and high load (p ranging between 0.08 and 0.6 depending on the load and vehicle type). We discuss GWEV-c performance in more detail in the next section and compare it to the other multi-policy approach that we implemented, GWEV-w.

3.5.3 Combined State Space vs. W-Learning

GWEV-w outperforms GWEV-c in terms of emergency vehicle average waiting time at all loads, reducing it by 20-60% when compared to their waiting time in GWEV-c. This difference in average waiting time is statistically significant under low traffic load (p=0.009) and is not statistically significant under medium and high load (p = 0.4 and p = 0.7, respectively). In terms of car waiting time, at low load GWEV-c outperforms GWEV-w by 2%, however, GWEV-w reduces car waiting time by 10-13% when compared to GWEV-c at medium and high load. The difference is not statistically significant under any of the traffic loads tested.

GWEV-c and GWEV-w serve a similar number of vehicles at low and medium loads, however during high load, GWEV-w serves 3000 vehicles more than GWEV-c. Also, GWEV-w has a lower average density at all loads than GWEV-c.

From these results we conclude that GWEV-w performs as well as, or better, than GWEV-c, under all traffic loads tested with respect to both vehicle types. Due to its scalability and smaller state space requirements, without any adverse effects on the performance when compared to GWEV-c, we conclude that GWEV-w is a more suitable technique for multi-policy optimization than GWEV-c in our experiments. GWEV-c had to be given a training period ten times longer than GWEV-w, to enable it to learn suitable actions for all states (training period of 20,000 minutes of traffic or $\sim$14 days). We do not rule out the possibility that GWEV-c might have performed better if given an even longer training period, however extending it even further would render it unfeasible for applications in real systems. These results suggest that GWEV-c, even though it outperforms SAT and RR, is not effective even for combinations of only two policies and would not be scalable to the addition of any further policies.

3.6 Conclusions

From the case study on non-collaborative multi-policy optimization described in this chapter we have made the following main observations.

Both RL-based techniques, GWEV-w and GWEV-c, outperform our baselines, both in terms of emergency vehicle and car waiting times, showing that RL-based techniques are promising approaches to multi-policy optimization in autonomic systems.

GWEV-w performs better than GWEV-c in terms of density and number of vehicles, and although the improvement in average vehicle waiting time under certain traffic conditions was not statistically significant, GWEV-w can additionally benefit from scalability, as well as shorter training times due

to smaller state spaces. The results, therefore, indicate that the W-learning-based approach is a more suitable approach for multi-policy optimization than combining learning processes into a single learning process.

We also observe a high dependency between the policies reflected in their performance, which emphasizes the importance of simultaneous optimization towards all policies present in the system. The dependency is reflected in the policy that addresses only emergency vehicles (EVO) generating a backlog of other vehicles, and as a result performing very badly both in terms of car and emergency-vehicle waiting times. Also, GWO, which addresses only cars, performs well in terms of emergency vehicle waiting times, which it does not explicitly address, as clearing cars clears the congestion on the roads and enables emergency vehicles to proceed. Our results also show that the importance of the optimization increases with the traffic load, where the gap between the performance of adaptive techniques (e.g., SAT) and non-adaptive techniques (e.g., RR) grows larger. In the next chapter we discuss how these observations have motivated the design of our proposed multi-policy multi-agent collaborative optimization technique, DWL.

Chapter 4

Distributed W-Learning

"*There is no selfish good deed, sorry*!"
- Joey, TV show Friends

In Chapter 3 we analyzed the performance of non-collaborative multi-policy RL-based optimization techniques and presented our observations. We first use those observations and the analysis of decentralized autonomic environments presented in Chapter 1, to derive a set of requirements for a collaborative multi-agent multi-policy optimization technique for such environments. We then present the design of Distributed W-learning (DWL), an algorithm for multi-policy optimization in large-scale heterogeneous agent-based autonomic systems, as motivated by these requirements.

4.1 Requirements for a Collaborative Multi-Policy Optimization Technique

In the case study on non-collaborative multi-policy optimization presented in the previous chapter we observe a high dependency between the multiple policies for which the system is optimizing, principally due to the shared deployment environment. Policy dependency can also cause agent dependency, as different policies might be deployed on different agents, due to different regional and temporal policy scopes. This dependency between agents could also potentially be increased due to the shared environment in which agents are situated, as discussed in Chapter 1. If this is the case, agents, and the system as a whole, could benefit from agents cooperating to select actions that are not only suitable for their own policies, but for the other agents' policies as well. However, designing an algorithm for

cooperative problem solving raises numerous issues related to cooperation, such as with whom, when, how much, and how to cooperate. Collaboration needs to be enabled between heterogeneous agents, as agents in the system can have different state-space representations and different action sets, and between heterogeneous policies, as policies can vary in their temporal and regional scope and priority. Additionally, as dictated by the large-scale of the autonomic systems considered, an algorithm needs to be decentralized and not rely on a global view of the system. Suitable optimization actions should be learnt rather than predefined at design time, as predefining all behaviours for all combinations of conditions is not possible due to the large and dynamic operating environment.

Based on these observations we derive a list of requirements for a multi-agent multi-policy optimization technique in large-scale autonomic systems:

1. Decentralized control
2. Learning-based optimization
3. Support for simultaneous deployment of multiple policies
4. Support for cooperation between agents (even when cooperation means sacrificing own performance)
5. Support for cooperation between policies (even when cooperation means sacrificing own performance)
6. Respect of policy priorities
7. Support for optimization and collaboration in heterogeneous environments, where the source of heterogeneity can be:
   (a) agents - in terms of their state spaces or action spaces
   (b) policies - in terms of their different spatial scope, temporal scope, or priority
8. Enable agents to determine cooperation criteria such as:
   (a) the other agents with which to cooperate
   (b) the situations in which to cooperate, i.e., in which states (or combinations of states)
   (c) how much to cooperate, i.e., how to weight other agents' action preferences vs. their own preferences

In the remainder of this chapter we present DWL and analyze how its design addresses the above specified requirements.

4.2 DWL Design

In this section we use the requirements specified above to derive the design of DWL. We consider a number of alternatives, evaluate them against the requirements, and assess their suitability to the multi-agent multi-policy decentralized environments.

The design of a multi-agent multi-policy optimization technique can be broken down into two main components: a means of addressing multiple heterogeneous policies on a single agent, and a means of addressing collaboration between multiple heterogeneous agents. We consider these two components below.

4.2.1 DWL as an Extension of W-Learning

As a result of the evaluation of existing multi-policy RL-based techniques in non-collaborative multi-agent scenarios presented in the previous chapter, we have found that W-learning is a promising technique for multi-policy optimization in autonomic systems. As already discussed in Chapter 2, the advantage of W-learning over other multi-policy arbitration-based RL optimization techniques is the unique way in which it learns the relative importance of policies specific to the current state in which they are (using W-values) rather than imposing a strict hierarchy of policies. By learning how one policy's actions affect the rewards received by another policy, W-learning learns the dependencies between policies, and exploits them to avoid executing actions that are particularly harmful for a policy in a given state, or to execute actions that are particularly suitable for a policy in a given state. W-learning respects policy priorities, as higher-priority policies can receive higher rewards, which are reflected in higher W-values for important states of the higher-priority policy, and W-learning selects the actions associated with higher W-values for execution. W-learning also enables optimization towards multiple policies regardless of their characteristics, i.e., it is suitable for heterogeneous policies, as the action selection process relies only on W-values, and does not depend on the policy's state-action space. Based on these characteristics of W-learning, we have made the decision to use W-learning as a basis for DWL. In DWL, each policy on each individual agent in the system is implemented as a Q-learning process, which learns the values associated with executing particular actions in each of its states, as well as a W-learning process, which learns the importance of its preferred action being executed when the policy is in a given state. Through use of W-learning as the basis for DWL, we address requirements 2 and 3 as specified in Section 4.1, i.e., we use RL to learn optimal actions rather than use predefined rules and we use W-learning to enable optimization towards heterogeneous policies. We also partially satisfy requirements 5 and 6, i.e., we ensure cooperation between policies

and ensure that policy priority is respected on individual agents, but do not address these requirements for a multi-agent scenario. W-learning only enables optimization towards multiple policies on a single agent, however, we also need to enable optimization towards multiple policies deployed on multiple agents, while also enabling collaboration between those agents. We discuss collaboration-related issues in the next section.

4.2.2 Collaboration in DWL

We have argued in Chapter 1 and Chapter 2 that collaborative agents in a MAS can achieve globally superior performance when compared to a group of independent agents. However, collaboration is useful only if its implementation is suitable to the agents' circumstances. Tan (1993) shows that "collaboration done intelligently" improves the performance of all agents, however, he also shows an example where collaborative agents are outperformed by a group of independent agents due to collaborative agents exchanging insufficient or irrelevant information. Therefore, designing a collaborative mechanism, deciding on the type of information to be exchanged, and deciding how that information will be used by a receiving agent, are crucial to achieving performance improvements by agents in a collaborative MAS. We address these issues in the following sections.

4.2.2.1 Purpose of Collaboration

The first major step in the implementation of a collaboration mechanism is deciding on the purpose of collaboration. Agents can collaborate in order to improve the quality or speed of their local learning, or to improve global performance of the system by ensuring that the actions they consider locally good are also beneficial for overall system performance. In DWL, both types of cooperation could be useful. It would be beneficial to the system if agents could learn from each other in order to shorten the duration of the learning phase, however, more crucially, due to agent dependency, agents need to cooperate to make sure the actions they execute locally are not detrimental to their neighbouring agents and to the system as a whole. Therefore, we focus on implementing collaboration with the purpose of ensuring that one agent's actions do not have negative effects (and can potentially give rise to positive effects) on the performance of other agents.

4.2.2.2 Collaboration in Heterogeneous Environments

The second major step when designing a collaboration mechanism is to identify the type of information that would be beneficial for agents to exchange and to identify how that information can be used by a receiving agent. When considering potential information from which DWL agents can benefit, we

analyze the collaboration mechanisms of RL-based MAS reviewed in Chapter 2, as well as analyze other elements of an RL process that could potentially be exchanged, and evaluate their suitability to DWL. The collaboration mechanisms we have identified are as follows:

1. Action coordination, i.e., make joint action decisions, as done in, e.g., (Guestrin et al., 2002; Kok et al., 2005) and discussed in 2.3.3.2

2. Exchange accumulated rewards, i.e., exchange the sum of a number of the latest rewards received, as done in, e.g., (Salkham et al., 2008) and discussed in 2.4.3

3. Exchange value functions or Q-values, as done in, e.g., (Dowling et al., 2006; Schneider et al., 1999) and discussed in 2.3.3.1 and 2.3.3.3.

4. Exchange immediate sensations, e.g., state, reward, as done in (Tan, 1993).

All of these collaboration implementations result in performance improvements in the domains in which they were applied (see Chapter 2 for more details), however we need to assess their suitability for our domain, particularly as determined by the presence of multiple heterogeneous policies and agents.

Action coordination In an action coordination approach (e.g., coordination graphs, see Section 2.3.3.2), agents in a neighbourhood, instead of making local action decisions and maximizing their own reward, make joint action decisions that maximize the value function for the neighbourhood. Effectively, the learning processes on the agents and in the neighbourhood are combined into a single optimization function. In DWL case, this would require merging all of the policies on all of the agents in a neighbourhood into a single learning process. We have discarded this option for even the single agent case, as it is prone to state-space explosion in the presence of numerous policies and its performance was inferior to the performance of W-learning in our case study presented in Chapter 3.

Exchanging accumulated rewards RL agents can also exchange their accumulated rewards, to inform each other of how good has their overall performance been in a number of previous time steps. A receiving agent can incorporate this information into its own local reward, receiving a higher reward if its neighbours have been performing well. Therefore, an agent is motivated to help its neighbours in their performance, as it is aiming to maximize its overall reward received which includes neighbours' rewards. This collaboration approach is suitable for a single-policy scenario, however, a number of issues arise in considering its applications in multi-policy scenarios.

Firstly, in the presence of multiple heterogeneous policies, we need to decided should an agent's primary task still be to maximize its overall reward, or should it be to respect the priority of policies and aim to satisfy higher priority policies first. Policy priority can be incorporated into rewards, however, in the presence of multiple policies and multiple agents, meeting several lower priority policies could result in a higher reward than meeting a single high-priority policy. We argue that an agent's task is to respect policy priorities, and aim to maximize its reward only when policy priorities are maintained.

Therefore, simply exchanging accumulated rewards might result in agents satisfying a number of lower-priority policies (or a single lower-priority policy on several neighbouring agents) instead of a higher-priority policy. To maintain relative policy priorities, accumulated rewards would need to be grouped by the policy by which they are received, in order to give an accurate reflection of an agent's performance towards its individual policies. Analogously, on the receiving agent, neighbours' rewards should be incorporated into the local rewards of the matching policies only. However, due to agent and policy heterogeneity, neighbours might not be implementing the same sets of policies. Arguably, agents should then have different sets of neighbours for different policies, collaborating only with neighbours that implement the same policies (if there are any), however, this again could result in agents not respecting overall policy priorities. If a single agent is implementing a policy that has a high priority for the overall system, surrounding agents need to ensure not to interfere, and to possibly help its performance, regardless of the fact that they might not be implementing the policy themselves.

Due to the numerous issues that arise when applying this approach to multiple policies, as discussed in this section, it is not clear if, and how, this approach could be applied in multi-policy heterogeneous environments.

Exchanging value functions Instead of exchanging accumulated rewards, agents can also exchange estimates of their expected long-term rewards per state-action pair, i.e., Q-values. Receiving agents can then scale the Q-values according to some weight function and incorporate them into their own Q-values. In this way agents learn to take actions which are beneficial in the long-term not just for themselves but also for their neighbours. Exchanging Q-values in multi-policy environments raises the same issues related to policy heterogeneity as exchanging accumulated rewards, i.e., how to use the Q-values at a receiving agent if the sending agents do not implement the same policies as the local agent. Q-values are learnt specific to state-action pairs, and if a receiving agent does not have matching state-action pairs, received Q-values cannot be incorporated locally. In fact, this approach is not directly applicable even in single-policy environments where agents are heterogeneous in terms of their state-action spaces due to different layouts of the environments in which they are situated

(as is the case in our UTC application). Since agents do not share a state-space representation and action sets, Q-values for a given neighbour's state-action pairs can not be integrated into the receiving agent's Q-values, as it does not have the corresponding state-action pairs.

Exchanging immediate sensations Agents can, as part of collaboration, exchange information about immediate sensations, e.g., latest state, latest reward received, or latest action executed. This form of information offers more flexibility in the manner in which it can be used by a receiving agent, as it is not tied to fixed state-action pairs (unlike exchanging Q-values), and it can still maintain the information about the policy to which it corresponds (unlike exchanging accumulated rewards). However, the onus is on the receiving agent to find a meaningful way to use this information in order to enable useful collaboration. In the next section we discuss several ways in which the immediate sensations exchanged between agents can be interpreted and used in the design of a collaboration mechanism for a multi-agent multi-policy optimization technique.

4.2.2.3 Collaboration in DWL

The central reasons for not being able to apply the collaboration mechanisms discussed in the previous section to collaboration in DWL were related to the heterogeneity of agents and policies. Q-values are tied to a specific state-action pair and as such cannot be exchanged between heterogeneous agents and policies, while exchanging accumulated rewards may be difficult to incorporate while maintaining relative policy priorities. Therefore, any form of collaboration to be used in DWL cannot be state-space or action-set specific, and must support the retention of relative policy priorities.

The underlying assumption in all of the collaboration mechanisms discussed is that an agent's performance is influenced by a combination of its local actions and the actions of its neighbours; an agent is, therefore, learning to execute actions that are suitable for itself and for its neighbours. Potentially, an agent's performance is influenced by the actions of all other agents in the system, however, as considering the influence of all agents on each other is not feasible, most approaches consider only immediate neighbours (e.g., (Guestrin et al., 2002), (Salkham et al., 2008)), whose actions are more likely to have the strongest influence. In DWL, we also adopt this approach, and have made a design decision that the collaboration mechanism will be implemented only between neighbouring agents. However, we aim to design the collaboration mechanism such that collaboration can be implemented between any two agents, so long as a communication link between them exists.

From the analysis of collaboration mechanisms in the previous section we have concluded that there are no obstacles (at least not as imposed by the heterogeneity of agents and policies) to agents

exchanging immediate sensations. In order to design a collaboration mechanism in DWL, we need to consider which immediate sensations agents should exchange, and how they can use them to learn about the performance of other agents, and to ensure their own actions do not negatively influence other agents. We considered two approaches: using immediate sensations to learn how other agents' actions affect a local agent's performance, and using immediate sensations to learn how a local agent's actions affect other agents' performance.

Learning the influence of other agents on local states If each agent knew how other agents' actions affected it, it could make action suggestions to those agents, especially if any of the actions that the other agents execute have a particularly positive or a particularly negative effect on an agent. In order for an agent to learn the influence of other agents' actions on its local states, each agent could implement a Q-learning process using its own state space and a neighbouring agent's action set. Since actions can have different impacts on different local policies, an agent would need to implement such a Q-learning process for each of its policies. In order to do this, an agent needs to get information from its neighbour about its action set. The neighbours' actions do need to be interpretable on the local agent or match the local agent's actions, but an agent only needs a representation of those actions so that it can generate state-action pairs to be updated. After each learning step, an agent could request information about the last action executed from its neighbour, and update the Q-value for the local state - neighbours' action pair using its local rewards received. At action selection time, each of the policies on an agent would have an action it can suggest to the neighbour, which it has learnt to be the most suitable for the particular state in which it is currently. However, after receiving those action suggestions, a neighbouring agent also needs the means to decide which of the action suggestions to execute; each agent will have action suggestions from its local policies, but also one or more suggestions from each of its neighbours. More crucially, an agent does not have a motivation to execute any of the neighbours' suggestions, as, in terms of its local reward, it could be better off executing an action that will result in the highest reward locally. Note that collaboration approaches where agents exchange accumulated rewards or Q-values do not suffer from this drawback, as a local agent incorporates neighbours' rewards/Q-values into its local rewards/Q-values, and therefore is motivated by receiving a reward for a neighbours' good performance. In the next section we consider a collaboration approach that addresses the agent motivation issue.

Learning the influence of local actions on other agents' states RL agents learn which actions to execute so as to maximize rewards received in the long-term, i.e., they are motivated by receiving rewards. In order to motivate an agent to help its neighbours' performance, we need to reward an

agent for the good performance of its neighbours. Instead of each agent learning how other agents' actions influence its local performance, as suggested in the previous section, we can enable each agent to learn how its actions influence its neighbours' policies. Each agent needs to implement a Q-learning process for each of the policies that each of its neighbours implement. In order to do this, each agent needs to obtain information from each of its neighbours on the full state space of each of its policies. An agent then, at each time step, updates Q-values for each of the neighbours' policy's states and its local action, using the reward neighbours' policies received at that time step, and receiving the same reward locally as the neigbours' policies have received. Using this approach, an agent both learns how its actions affect the policies on its neighbours and receives rewards for those policies performing well. Note that an agent does not need to implement the same policies as its neighbours, or even know which policies the neighbours implement, and it does not need to be able to interpret the meaning of the neighbours' policies' states; it only needs to have a representation of those state spaces in order to create the neighbours' state - local action pairs. This approach, therefore, enables collaboration between heterogeneous agents implementing heterogeneous policies. At action selection time, an agent receives, from each of its neighbours, information on the current state of each of their policies, and identifies a suitable action for each of those policies based on neighbours' state - local action Q-values. It also needs to know the reward that each of the policies on each of its neighbours has received for the previous action executed, in order to update those Q-values.

Using this collaboration mechanism we both motivate an agent to help its neighbours and enable it to do so even in the presence of policy and agent heterogeneity. However, the issue relating to how an agent should choose between the actions nominated by its local policies and its mappings of the neighbours' policies remains open. We focus on this issue next.

Action Selection At each time step, a local agent has a set of action suggestions that are suitable for its local policies and a set of action suggestions that are suitable for each of its neighbours' policies. An agent needs a mechanism to decide which of these to execute, so as to maximize its own reward, help neighbours maximize their rewards, and respect the relative priority of policies. However, the agent does not have information about relative policy priorities; the only information it has available when making an action selection are action nominations and their associated Q-values. If we were to enable agents to know the relative priorities of their local policies and neighbours' policies, we would require a global hierarchy of policy priorities in the system, i.e., we would require a global view of the system. Also, an agent needs a way of distinguishing how important the actions are to the policies that are suggesting them; only one action can be executed so it should be one that is really crucial

for good performance of a policy that is suggesting it at that particular time-step.

When implemented as outlined above, the problem of action selection within a one-hop neighbourhood resembles the problem of action-selection on a single agent implementing multiple policies, as discussed in Section 2.3.2.2. The agent has a number of action nominations to select from and needs to pick one of them based on some criteria. As in a single-agent case, the action selected for execution can be based on a group criterion, so that it satisfies the largest number of policies, or can be an action nominated by the policy to which the next action is the most important. We believe it should be the latter, since if we were to use group selection methods, the priority of policies might not be respected - we could end up addressing several lower-priority policies instead of one high-priority one. However, an agent still needs a way to learn relative policy weights, both in terms of their absolute priority and their current importance. This is exactly the type of problem for which we found the use of W-learning suitable on an agent locally, when selecting an action amongst the action suggestions of its local policies. We propose extending the use of W-learning from action selection on a single agent to action selection between agent's local policies and action suggestions from its one-hop neighbours. Use of such an action selection mechanism, distributed within the neighbourhood, was the rationale for naming our algorithm Distributed W-Learning.

Action selection in a neighbourhood using W-Learning In DWL's approach to collaboration, as described above, each agent has a set of value functions for its local policies and for policies of each of its one-hop neighbours. At each time step, each of the local and neighbours' policies has an action suggestion based on learnt Q-values. In order to learn the importance of these action suggestions relative to the state the policies are in currently, each agent also implements a W-learning process for each of its neighbours' policies, i.e., learns W-values relative to neighbours' policies' states. An agent receives all the information required for this process during the Q-learning process; in order to learn W-values an agent only needs to know neighbours' policies' current states and the rewards received, which are also required by the Q-learning process. By using a combination of Q-learning and W-learning, an agent not only learns the suitability of its actions for neighbours' policies, but also learns how not executing those actions can affect the neighbours' policies. If the neighbour's policy receives a high reward regardless of the local agent executing its preferred action or not, the weight associated with that particular state of the neighbour's policy will be low, as the dependency between the local action and neighbour's policy performance is either low (as an action executed locally does not affect the neighbour's reward) or the policies are complementary and one policy's action is also good for the other policy. Conversely, if the neighbour's policy receives a high reward when its action suggestion is

executed, but it does not otherwise, the weight associated with that particular state of the neighbour's policy will be high, as the dependency between the local action and neighbour's policy performance is high (as an action executed locally significantly affects the neighbour's reward) and policies are conflicting (i.e., actions suitable for one policy are not suitable for the other). We argue that, in this manner, W-learning enables an agent to learn the dependencies, in terms of reward, between its local actions and neighbours' policies' performance, i.e., it learns the dependencies between agents.

In Section 4.4.1 we formalize this mechanism of enabling an agent to learn how and to what extent its actions influence its neighbours by using a concept of "remote policy".

4.2.2.4 Degree of Cooperation

By using Q-learning and W-Learning, each agent, at each time step, has a set of action suggestions with associated weights from both its local policies and all of its neighbours' policies. We have already argued that action selection should be informed by policy priority and current policy importance. However, another factor that should be considered at action selection time is whether an agent should take into account all action suggestions as nominated by local and neighbours' policies, or should weight its local suggestions more, as, potentially, its local actions can have a larger influence on its immediate operating environment than its neighbours' actions. We expect a suitable weight ratio to be dependent on an application area and a set of environment conditions, so the design of DWL should remain flexible in regards to the weight ratio. We should enable an agent to give equal weight to neighbours' policies as to its own, i.e., be fully cooperative, as well as to completely ignore neighbours' policies, and be fully non-cooperative. We also need to enable a range of cooperation levels in between the two extremes; for example, an agent might want to rate its own policies higher, but still be willing to defer to its neighbours if the next action is really important to them, i.e., their W-values exceed some threshold. In order to specify this threshold, we introduce a cooperation coefficient C, which a local agent uses to scale the weight of neighbours' action preferences at action selection time. C can have a value $0 \leq C \leq 1$, where C=0 denotes a selfish non-collaborative agent, and C=1 a fully cooperative agent that weights its neighbours' policies' suggestions as much as its own. C can be externally defined by the designer, similar to (Dowling et al., 2006) and (Schneider et al., 1999), where neighbours' value functions are incorporated locally using predefined coefficients. However, we also investigate the possibility of an agent learning a suitable value for C itself.

4.2.2.5 Learning the Degree of Cooperation

The cooperation coefficient, as described in the previous section, can be set as a predefined DWL parameter to be the same across all agents during both learning and exploitation. There are a number of limitations to this approach. The best performing C for a given set of agents in a given set of circumstances can only be determined by trial-and-error by running simulations with varying values of C, requiring an extensive training period. Additionally, a single value of C might not be appropriate for all agents in the system. For example, a very important agent in the system whose performance has a major impact on overall system performance, might need to have a low value of C, in order to act more selfishly, as its performance is more important to the system than the performance of its neighbours. Conversely, an agent whose local contribution to overall system performance is smaller than that of its neighbours might require a high value of C, i.e., it should have a lower threshold for deferring to its more important neighbours. Therefore, DWL needs to allow for the possibility of agents having different values of C rather than having the same value across the whole system. In order to make this feasible, we extend DWL with the capability of each agent to learn its local optimal C, rather than use the same predefined value of C across all agents. Each agent has a range of C values available, and implements a Q-learning process to learn the most suitable local C with which to scale all neighbours' suggestions, such as to maximize the reward received in the neighbourhood. Note, however, that agent is not learning which action, out of all available actions, is maximizing the reward in the neighbourhood, but only learns the value of C that maximizes the reward by executing one of the actions nominated, and as such, we argue, is still respecting policy priorities. We show this is the case later in this chapter (in Section 4.4.3), after we have introduced all of the necessary DWL concepts.

4.2.2.6 Collaboration Summary

In this section we have given the rationale for the design of a collaboration mechanism for a multi-agent multi-policy optimization technique. The main issues addressed are the purpose of collaboration, how to motivate agents to collaborate, how to enable collaboration between heterogeneous agents implementing heterogeneous policies, and how to ensure that the relative priorities of policies are maintained. We have adopted a "remote policy" approach, where each agent learns how its actions affect all policies that all of its one-hop neighbours are implementing, and at each time step enables each of those policies to make an action suggestion. Agents are motivated to collaborate by receiving rewards for their remote policies when their neighbours receive rewards. Using remote policies as a collaboration mechanism does not depend on state space or action sets of the policies. Using W-

learning, DWL also learns the weights (W-values) of these action suggestions relative to the current policy state and policy priority. DWL scales the W-values of neighbours' policies using either a predefined cooperation coefficient C, or a learnt value of C that maximizes the overall reward in the neighbourhood, and executes the action with the highest weighted W-value. Our collaboration mechanism satisfies requirements 3, 4, 5, 6, and 7 through use of remote policies, and requirement 8 through use of cooperation coefficient C and an ability of each agent to learn it locally.

4.2.3 Decentralized Control

DWL has been designed so that each agent optimizes towards its policies locally, and collaborates with its one-hop neighbours to help their performance when feasible. The only communication required in DWL is between neighbouring agents, and no global information, central or external control is required. Therefore, DWL is designed to be decentralized and self-organizing, meeting requirement 1.

As the goal of DWL is to optimize global system behaviour, we are foreseeing that globally good behaviours will emerge from agents' local optimization and optimization within neighbourhoods. We do not foresee radically new behaviours emerging, but we do expect a better quality of global behaviours than we obtain when each agent optimizes for its own local policies only. We expect the performance of all policies to be improved through cooperation, with policy priority being respected and higher priority policies being given preference in the system, even on agents that do not directly implement them. We will return to this point in Chapter 6, when evaluating and analyzing the performance of DWL.

4.3 Definition of DWL

In the previous section we have motivated the design of the elements of DWL and discussed how they address the requirements. In this section we formalize the definition of DWL elements.

A DWL-based system consists of the following:

- A set of agents $A = \{A_1, ..., A_n\}$, where each agent controls a set of actuators.
- Each agent A_i has a set of neighbours $N_i = \{N_{i1}, ..., N_{im}\}$ consisting of all agents $A_j \in A$ that are one-hop neighbours of the agent A_i.
- A set of policies $LP_i = \{P_{i1}, ..., P_{ip}\}$ are deployed at each A_i. We refer to these policies as *local policies* of A_i. Local policies can be active or inactive at each time step, based on their temporal scope.

- Each agent A_i has a set of policies $RP_i = \{RP_{ij1}, ..., RP_{ijr}\}$ whose goal is to contribute to the implementation of each local policy LP_{jk} deployed at each $A_j \in N_i$. We refer to these policies as *remote policies* of A_i. A remote policy can be active or inactive at each time step, based on the temporal scope of the corresponding local policy of neighbour A_j, i.e., whether that policy is currently active or not.

- Each policy, both local and remote, on each agent is implemented as a combination of a Q-learning and a W-learning process. It has Q-values associated with each of its state-action pairs and W-values associated with each of its states. An agent's current action is denoted as a_i and its previous action a_{i-1}. A policy's current state is denoted as s_i and its previous state s_{i-1}.

- Cooperation coefficient C, which is used to scale the importance of action suggestions by remote policies. C can be predefined to the same value for all agents in A, or can be learnt individually by each $A_i \in A$.

We further focus on the details of these elements in the next section.

4.4 DWL Elements

In the previous section we defined the elements that a DWL-based optimization system consists of. In this section we focus on three main concepts in DWL that enable it to meet the requirements for multi-agent multi-policy optimization technique. These are: remote policies, cooperation coefficient C, and agents' ability to learn values of C.

4.4.1 Remote Policies

To enable collaboration between agents, as well as between policies deployed on different agents, DWL agents not only learn to select actions that are suitable for their local policies, but also learn how their local actions affect their immediate neighbours. To motivate an agent to take into account its neighbours' action preferences (i.e., to collaborate), each agent, as well as its own policies, implements a "*remote*" policy "help neighbour N_i to implement its policy P_{ik} " for each of the policies deployed on each of its immediate neighbours. This policy receives a reward each time policy P_{ik} receives a reward on neighbour N_i, therefore motivating an agent to select actions suitable for this policy. By using remote policies, DWL enables cooperation between heterogeneous agents, i.e., agents that implement different policies, and have different state space and action sets. Each remote policy on each agent is implemented using a Q-learning and a W-learning process. The Q-learning process uses the remote

agent's policy's states and the local agent's actions to learn how local actions affect a remote policy. The W-learning process learns how important it is for a remote policy, in each of its particular states, that its preferred local agent's action is executed.

4.4.2 Cooperation Coefficient C

At each time step, both local and remote agent's policies, suggest their preferred action based on their current state, together with an associated weight (W-value) for that state. An agent uses these W-values to select the next action for execution.

By introducing a cooperation coefficient C, local policy action nominations are taken into account with their full W-values, while remote policy nominations are multiplied by C, where $0 \leq C \leq 1$, to enable an agent to give varying weight to its neighbours' action preferences. C=0 means that the local agent is non-cooperative, i.e., it does not consider its neighbours' performance when selecting an action, while C=1 means that a local agent is fully cooperative, i.e., it cares about its neighbours performance as much as it cares about its own. A value of C between 0 and 1 enables a local agent to specify the relative importance of its local preferences and its neighbours' preferences. The local agent is still selfish to a degree, i.e., its local policies still have a higher relative importance, but it does defer to its neighbours' action suggestions when their importance exceeds the relative threshold as set by C. For example, if C=0.2, the strength of a neighbour's suggestion, i.e., its W-value, needs to be more than five times greater in order for an agent to defer to one of its neighbours.

4.4.3 Learning Values of C

In DWL, agents also have an ability to learn their own suitable values of C, rather than use a predefined one. Each agent can learn a value of C such as to maximize the overall reward received in the neighbourhood, i.e., by all of its local and remote policies.

In order to explain choosing an overall neighbourhood reward as criteria for learning C, we need to analyze the DWL action selection process. Consider Figure 4.1. Each local and remote policy suggests a single action for execution. Therefore, DWL's action selection is not from the full action set of actions available on an agent, but from the set of actions suggested by the policies. For example, in Figure 4.1 only actions belonging to the set $\{a_{11}, a_{12}, a_{121}, a_{122}, a_{131}\}$ are considered for execution. Action selection also depends on the W-values associated with each nominated action. Action a_{11} has an associated W-value W_{11}, action a_{12} has an associated W-value W_{12} and so on. The action for execution is picked in three ways depending on the value of C, as follows:

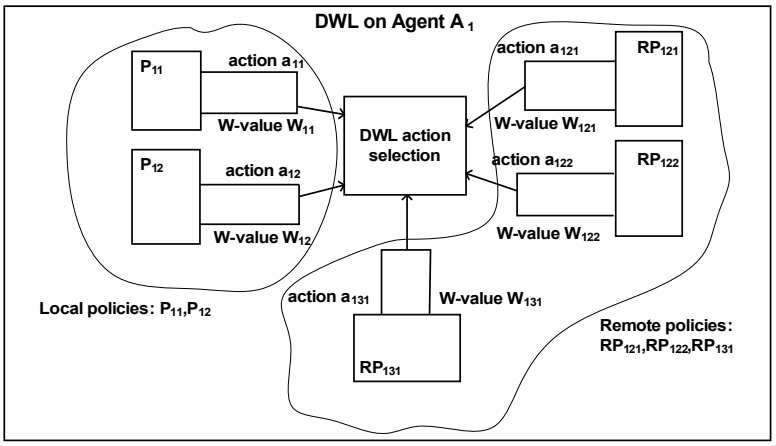


Fig. 4.1: DWL action selection: local vs remote policies

- for C=0, the action selected for execution, denoted as a_{max}, is an action associated with the W-value W_{max}= max (W_{i1}, ..., W_{ip}) where W_{i1}, ..., W_{ip} are the W-values for the current states of agent's A_i's *local* policies P_{i1},..., P_{ip} (i.e., a_{max} is the action associated with the maximum local W-value).

- for C=1, a_{max} is an action associated with the W-value W_{max}= max (W_{i1}, ..., W_{ip}, W_{i11}, ..., W_{ijk}) = max (max (W_{i1}, ..., W_{ip}), max (W_{i11}, ..., W_{ijk})) where W_{i1}, ..., W_{ip} are the W-values for the current states of agent's A_i's *local* policies P_{i1},..., P_{ip} and W_{i11}, ..., W_{ijk} are the W-values for the current states of agent's A_i's *remote* policies RP_{i11},..., RP_{ijk} (i.e., a_{max} is the action associated with either maximum local W-value or maximum remote W-value).

- for $0 < C < 1$, a_{max} is an action associated with W-value W_{max}= max (W_{i1}, ..., W_{ip}, C $\times W_{i11}$, ..., C $\times W_{ijk}$) = max (max (W_{i1}, ..., W_{ip}), C $\times$ max (W_{i11}, ..., W_{ijk})) where W_{i1}, ..., W_{ip} are the W-values for the current states of agent's A_i's *local* policies P_{i1},..., P_{ip} and W_{i11}, ..., W_{ijk} are the W-values for the current states of agent's A_i's *remote* policies RP_{i11},..., RP_{ijk} (i.e., a_{max} is the action associated with the maximum local W-value or an action associated with the maximum remote W-value).

Therefore, an action selected for execution, for all values of C, can either be an action associated with the maximum W-value on the agent's local policies, or an action associated with the maximum W-value on the agents remote policies, i.e., it has to belong to a set $\{a_{l-max}, a_{r-max}\}$, where a_{l-max} is associated with W_{l-max} = max (W_{i1}, ..., W_{ip}) , and a_{r-max} is associated with W_{r-max} = max (W_{i11}, ..., W_{ijk}). The value of C, together with the W-values themselves, will determine which of the two actions will get selected for execution. We argue that neither always selecting local maximum

(C=0) nor always selecting overall maximum (C=1) are optimal approaches as the local agent needs to prioritize its own performance, but still needs to defer to its neighbours when the importance of their action nominations exceeds some threshold. However, we need a criteria on which to select that threshold rather than set an arbitrary C or determine it experimentally. We believe that a suitable criteria for choosing between local and remote maxima is to assess how those actions affect other agents and policies. Therefore, an agent should learn how both a_{l-max} and a_{r-max} affect other agents and policies in the immediate neighbourhood, apart from those by which they are nominated, and select the one that has the most desired effect. Therefore, an optimal C is the one that enables an action that receives a higher sum of rewards on all local and remote policies to be selected., i.e., an action that has a higher value of (r_{i1}+ ... + r_{ip}+ r_{i11}+ ... + r_{ijk}) where r_{i1} ... r_{ip} are the rewards received by local policies after execution of the selected action a_{max} and r_{i11} ... r_{ijk} are the rewards received by remote policies after execution of the selected action a_{max} .

Note that, in this case, the agent is not actually learning a specific action to execute, as the a_{l-max} and a_{r-max} that are nominated differ based on the states in which their local and remote policies are; an agent is learning whether, in general, selecting its nominated local or remote action results in higher reward, regardless of the specific actions nominated. Basically, an agent is learning how selfish or how cooperative it should be overall, by learning the degree of cooperation (cooperation coefficient) such that it maximizes the reward received locally and by its immediate neighbours, regardless of the specific actions nominated at that time step.

Also note that, the agent is not learning to select an action that maximizes the sum of rewards on all of the agent's local and remote policies (i.e., maximizes the sum of rewards on an agent and its immediate neighbourhood) out of the full action set. The agent is only learning which C enables it to select an action, out of the nominated two actions (ones with local maximum W-value and remote maximum W-value), that generates a higher reward. The difference between the two approaches is that if an agent was to learn an action that maximizes the sum of rewards on local and remote actions, it might not necessarily respect priorities, e.g., meeting a lower priority policy on several agents would generate a higher total reward than meeting a high priority policy on a single agent. By using DWL to first select a subset of actions nominated based on the current state importance, we ensure that policy priorities are respected, and that only actions with a high current importance to the local and remote policies are nominated (where the higher priority is expressed using a higher reward, therefore resulting in higher W-values).

Learning C, as it is reward-based, is also implemented as a Q-learning process, one on each agent. An action set consists of various values of C, where $0 \leq C \leq 1$. In our implementation we use A

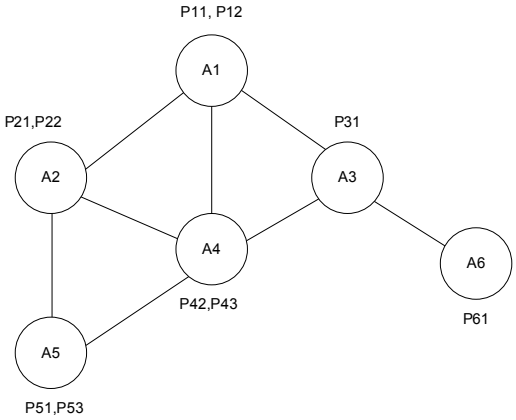


Fig. 4.2: Example of DWL agent network

= {C=0, C=0.1, C=0.2, C=0.3, C=0.4, C=0.5, C=0.6, C=0.7, C=0.8, C=0.9, C=1}, however the granularity of actions can be finer or coarser to provide for more precise learning, or for faster learning times. The state space of the Q-learning process used to learn C consists of only a single state, i.e., is stateless, as C is learnt regardless of the current state of an agent's local/remote policies and regardless of specific actions nominated for selection. Arguably, learning C specific to the particular state each policy is in might provide more precise results, however, the system would have to learn C for each combination of local and remote states, rendering the learning intractable and prone to state explosion. Learning C only requires the sum of rewards on local and remote policies for updating its Q-values, and as receiving reward information from neighbours is already required for learning Q-values and W-values for remote policies, learning C does not introduce any additional communication overhead.

Now that we have introduced all of the DWL elements, we move on to explain the processes of DWL initialization, learning and action selection.

4.5 DWL Initialization and Learning Process

In this section we explain the process required to initialize DWL-based optimization systems and explain the DWL learning and the action selection processes which are executed by each agent at each time step.

4.5.1 DWL Initialization

Consider a network of agents depicted in Figure 4.2. The network consists of a set of agents A = $\{A_1,..., A_6\}$. Each agent has a set of neighbours it is connected to by the edges of the graph. For

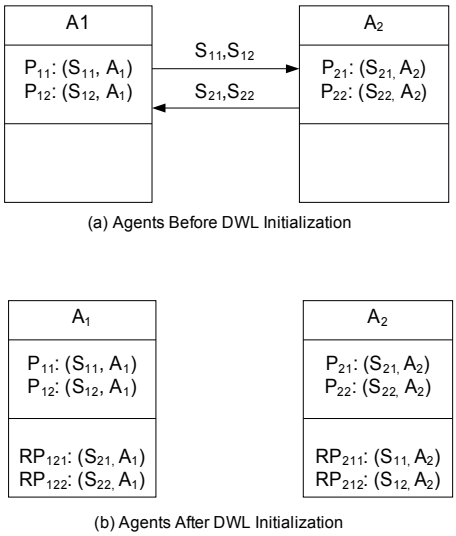


Fig. 4.3: Agents A_1 and A_2 before and after DWL initialization

example, Agent A_1 has a set of neighbours $N_1 = \{A_2, A_3, A_4\}$. Each agent also has a set of local policies it implements, e.g., agent A_1 has a set of local policies $LP_1 = \{P_{11}, P_{12}\}$.

During the initialization of DWL optimization process, each DWL agent initializes a Q-learning and a W-learning process for each of its local policies. Q-learning processes for local policies are initialized using a local policy state space and local actions (as they are used to learn how an agent's actions affect the state of the local agent's policies and to learn optimal actions for each of the local policies' states). W-learning processes for local policies are initialized using a local policy state space (as they are used to learn how important it is, for each of the local policy states that an action nominated by that policy is executed on the local agent).

After local initialization, each DWL agent exchanges state-space representations for each of its policies with each of its immediate neighbours. For example, consider Figure 4.3(a). Agent A_1 has two policies: P_{11} which provides mapping from its set of states S_{11} to set of actions of A_1, and P_{12} which provides mapping from its set of states S_{12} to set of actions of A_1. Agent A_2 also has two policies: P_{21} which provides mapping from its set of states S_{21} to set of actions of A_2, and P_{22} which provides mapping from its set of states S_{22} to set of actions of A_2. During the initialization, agent A_1 sends state representation for both of its policies, S_{11} and S_{12} to A_2 , and A_2 sends state representation for both of its policies, S_{21} and S_{22} to A_1.

This state-space information is used to initialize Q-learning and W-learning for remote policies. Q-learning processes for remote policies are initialized using a remote policy state space and local actions (as they are used to learn how an agent's local actions affect the state of the remote agent's

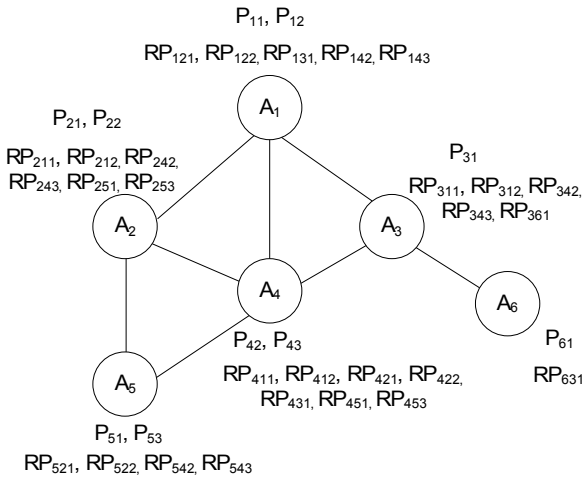


Fig. 4.4: Example of DWL agent network after DWL initialization

policies and to learn optimal local actions for each of the remote policies' states). W-learning processes for remote policies are initialized using a remote policy state space (as they are used to learn how important it is, for each of the remote policy states that an action nominated by that policy is executed on the local agent).

For example, consider Figure 4.3 (b). Agent A_1 now also has two remote policies: RP_{121} which provides a mapping from a set of states S_{21} (which is a set of states received from A_2's policy P_{21}) to a set of actions of A_1, and RP_{122} which provides a mapping from a set of states S_{22} (which is a set of states received from A_2's policy P_{22}) to a set of actions of A_1. Agent A_2 also has two new remote policies: RP_{211} which provides a mapping from a set of states S_{11} (which is a set of states received from A_1's policy P_{11}) to a set of actions of A_2, and RP_{212} which provides a mapping from a set of states S_{12} (which is a set of states received from A_1's policy P_{12}) to a set of actions of A_2.

Note that both local and remote policies have the same set of actions, which represent the set of capabilities of the local agent. Policies therefore learn which of the available local actions are most optimal for the local agent's performance as well as for the performance of the agent's immediate neighbours.

Figure 4.4 shows the updated sets of policies after DWL has been deployed and initialized on a network of agents represented in 4.2. Agent A_1 now, as well as a set of local policies LP_1, also has a set of remote policies RP_1= {RP_{121}, RP_{122}, RP_{131}, RP_{142}, RP_{143},}. Remote policies RP_{121} and RP_{122} correspond to local policies P_{21} and P_{22} on agent A_2, remote policy RP_{131} corresponds to a local policy P_{31} on agent A_3, and remote policies RP_{142} and RP_{143} correspond to local policies P_{42} and P_{43} on agent A_4.

The DWL initialization process is summarized in Algorithm 1.

Initialization steps outlined here need to be performed on each DWL agent at system start up, or on an agent and its neighbours when it joins the system. Some initialization steps (i.e., relating to a single policy only) need to be performed when a new policy is deployed, i.e., a new policy needs to be initialized locally, its state space information exchanged with an agent's neighbours, and remote policies on all neighbours initialized. If an agent leaves the system, initialization steps relating to remote policies need to be performed on agents that become neighbours as a result of an agent leaving.

Algorithm 1: DWL Initialization on an Agent A_i

```
/* Initialize local policies */
foreach P_il in LP_i do
    /* Initialize Q-learning */
    InitQLearning(P_il states, A_i actions);
    /* Initialize W-learning */
    InitWLearning(LP_il states);
end
/* Create remote policies */
foreach A_j in N_i do
    foreach LP_jk in LP_j do
        Add corresponding RP_ijk to RP_i;
    end
end
/* Initialize remote policies */
foreach RP_ijk in RP_i do
    /* Initialize Q-learning */
    InitQLearning(RP_ijk states, A_i actions);
    /* Initialize W-learning */
    InitWLearning(RP_ijk states);
end
```

4.5.2 DWL Learning Process

After initialization, the DWL learning process is repeated on each agent, at each time step, for the duration of the DWL-based system's operation, and consists of the steps described in Algorithm 2.

Algorithm 2: DWL one learning step on an Agent A_i

```
/* Get action nominations by local policies */
foreach P_il in LP_i do
    determine P_il's state s_il;
    get reward r_il from A_i's environment;
    /* Update Q-values */
    update Q(s_il-1, a_i-1) with r_il;
    /* Update W-values */
    update W(s_il-1) with r_il;
    nominate action a_il based on Q-values for s_il;
    get W(s_il);
end
/* Get action nominations by remote policies */
foreach RP_ijk in RP_i do
    get RP_ijk's state s_ijk from A_j;
    get reward r_ijk from for s_ijk from A_j;
    /* Update Q-values */
    update Q(s_ijk-1, a_i-1) with r_ijk;
    /* Update W-values */
    update W(s_ijk-1) with r_ijk;
    nominate action a_ijk based on Q-values for s_ijk;
    get W(s_ijk);
end
/* Select and execute action */
pick winning action a_i according to Formula 4.1
```

Each agent, using Q-learning and W-learning, learns Q-values for its local-state/local-action pairs and W-values for its local states. To do this, at each time step, each policy needs to sense the conditions in the environment, map them to its local state representation, and obtain a reward for being in that state. Based on the reward, each policy updates its local Q-values and W-values.

Additionally, each agent learns Q-values for its remote-policy-state/local-action pairs and W-values for its remote policies' states. To do this, at each time step, an agent needs to receive information about its neighbours' current states for each of its policies and the rewards that they have received. Based on the rewards received, an agent updates Q-values and W-values for its remote policies. For

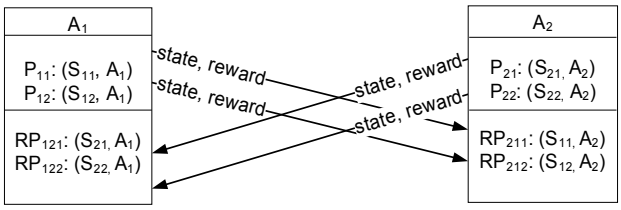


Fig. 4.5: Exchange between agents during each DWL learning step

example, consider Figure 4.5. In order for agent A_1 to update its remote policy RP_{121}, it needs to receive from agent A_2 current state of A_2's policy P_{21}, as well as the value of the reward that P_{21} has received in this time step. Note from Algorithm 2 that the action used to update Q-values for all local and all remote policies is the one that is last executed on the agent locally, i.e. a_{i-1}.

4.5.3 DWL Action Selection

At each time step, after Q-values and W-values for local and remote policies have been updated, an action needs to be selected for an execution by each agent based on action nominations by local and remote policies. Each policy, both local and remote, nominates an action, based on learnt Q values for state-action pairs, together with an associated learnt W-value. For example, in Figure 4.6, agent A_1 has four policies, two local and two remote, and at each time step receives four action nominations. Note that, as described in Chapter 2, the W-value is specific to the agent's current state, rather than specific to a nominated action. W-values of inactive policies are set to 0, as nominations of sporadic policies are not taken into account during their inactive periods. An action with the highest associated W-value is selected, to prioritize the policy that currently has the most to lose (or "to minimize greatest unhappiness" as discussed in Chapter 2). Using this method of action selection, actions nominated by local policies have the same importance as actions nominated by remote policies, i.e., by remote agents. To allow agents to give a higher priority to their own local preferences (i.e., to their own locally nominated actions), we can use a cooperation coefficient C.

The action that is executed on an agent, i.e., the one that wins the competition between policies at a given time step, is the one with the highest W-value (W_{win}), after remote W-values have been scaled by the cooperation coefficient C:

$$W_{win} = \max(W_{il}, C \times W_{ijk}) \tag{4.1}$$

where W_{il} are W-values associated with actions nominated by local policies of A_i and W_{ijk} are W-values associated with actions nominated by remote policies of A_i. In the event of a tie, where W_{il}

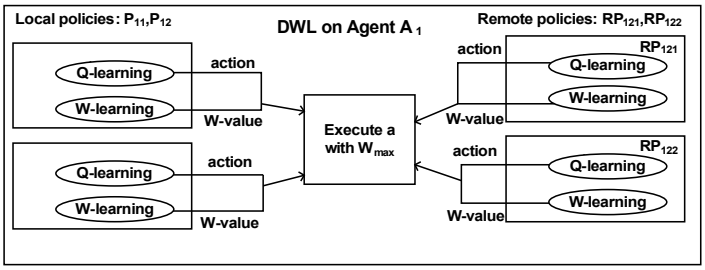


Fig. 4.6: DWL action nomination

and $C \times W_{ijk}$ have the same value, preference is given to a local action suggestion.

Algorithm 3: DWL action selection: learning values of C

```
/* Sum up total reward received on local and remote policies */
```
int totalReward=0;
```
/* Get rewards obtained on local policies and add to total reward */
```
foreach P_{il} **in** LP_i **do**
| totalReward+=r_{il};
end
```
/* Get rewards obtained on remote policies and add to total reward */
```
foreach RP_{ijk} **in** RP_i **do**
| totalReward+=r_{ijk};
end
```
/* Use total reward to update Q-value for previous C used */
```
Update Q(s_i, C_{i-1});
```
/* Select next value of C based on Q-values */
```
select C;
```
/* Multiply W-values of all remote policies by C */
```
foreach RP_{ijk} **in** RP_i **do**
| W(s_{ijk})x= C;
end
```
/* Execute an action with max W-value across all local and remote policies */
```
W_{max} = max (W(s_{il}), W(s_{ijk}));
execute a_{max};

In the case of agents learning their individual values of C rather than using a predefined one, Q-values for the process of learning C need to be updated at this point as well. In that case, instead

of simply selecting the highest W-value according to Formula 4.1, an agent needs to execute the steps presented in Algorithm 3. An agent needs to calculate the sume of the rewards received on all of its local and remote policies, i.e., the rewards received in the neighbourhood, and use the sum to update the Q-value for previously used C. In this way, an agent learns which values of C result in the highest long-term rewards received in the neighbourhood.

The learning process and action selection steps are repeated on each agent until the agent leaves the system or until the operation of the system is terminated externally, as individual learning processes on agents have no terminating states.

4.6 DWL and the Requirements for a Multi-Policy Collaborative Optimization Technique

In DWL, each agent implements a Q-learning process for each of its local and remote policies, a W-learning process for each of its local and remote policies, and a single Q-learning process for learning the cooperation coefficient. At each time step, on each agent, each policy, both local and remote, suggests an action based on the outcome of the Q-learning process; that action is associated with a W-value for the current policy state, based on the outcome of the W-learning process. In this way, action selection is narrowed down to two actions, the action nominated with the highest W-value amongst local policies, and the action nominated with the highest W-value amongst remote policies. Selection between the two is done by selecting the maximum W-value, after the remote W-value is multiplied by C. C can be predefined or learnt individually by each agent such as to maximizes the sum of rewards received on all of the agent's local and remote policies.

Through a combination of local policies (that use Q-learning and W-learning to learn optimal local behaviours as well as to learn dependencies and enable collaboration between policies on a single agent), remote policies (that use Q-learning and W-learning to learn optimal behaviours for neighbouring agents and to learn dependencies and enable collaboration between policies on neighbouring agents), and cooperation coefficient and ability to learn its value, DWL addresses all of the requirements for multi-agent multi-policy optimization technique outlined in Section 4.1.

Table 4.1 summarizes which features address which specific requirement. DWL is a self-organizing decentralized algorithm (Req. #1) and hence suitable for optimization in environments with no central control or environments where using a centralized optimization approach is intractable. DWL does not require a global system view; and optimization of the overall system behaviour is designed to emerge from local optimization and agent cooperation. In DWL, action selection on each agent is

Req #	Requirements	DWL feature
1.	Decentralized	Local learning processes for each policy
2.	Learning-based	Q-Learning and W-Learning
3.	Simultaneous multiple policies	W-Learning
4.	Collaboration between agents	Remote Policies
5.	Collaboration between policies	Remote Policies
6.	Respect policy priorities	Remote Policies, W-Learning
7.	Suitable for heterogeneous policies/agents	Remote policies
8.	Know with who/when/how much to cooperate	W-Learning, Cooperation Coefficient

Table 4.1: Requirements vs. DWL features

based on learning (Req. #2) the optimal action for all of its own policies, as well as for all of the policies on its immediate neighbours (Req. #3, 4, 5). Collaboration is enabled between heterogeneous agents and heterogeneous policies, regardless of whether the source of heterogeneity are different state space representations, different action sets (i.e., different agent capabilities) or different policies (or policy characteristics) (Req. #7). DWL agents exchange their state space representations during initialization, and rewards received during the optimization process. Aside from this, agents do not need to know anything about other agent's action sets, policies, policy priority, or if they match their own policies, state spaces, or action sets.

Agents are motivated to cooperate by introducing remote policies for which they receive rewards. Agents learn with which other agents they share the highest dependencies by learning the Q-values and W-values for the pairs of remote agent's states and their local actions. The relative priority of policies is specified by different rewards received for different policies, which is reflected in the Q-values and W-values. A higher reward results in higher W-values, and higher W-values gain control over action selection, respecting policy priority (Req. #6). Therefore, agents learn to cooperate with the neighbours who implement policies with higher relative W-values, where those values are scaled using cooperation coefficient C, selected to maximize a payoff in the immediate neighbourhood. Hence, through a combination of W-learning and learning C, DWL can determine if and to which policy on which neighbour to defer to and which deferral threshold to use (Req. #8).

DWL is primarily designed for multi-policy optimization, however, due to its approach to cooperation and action selection it can also be used to improve single-policy optimization by exploiting collaboration capabilities. Even though all agents implement the same policy, and there is no higher priority policy in the system to be deferred to, agents can still learn how their actions affect their neighbours, and defer to neighbours' action suggestions in the case of high dependencies between them.

4.7 DWL Assumptions and Scope

DWL is designed to optimize global behaviours of multi-agent autonomic systems. Due to the scale and complexity of autonomic systems, rather than adopting a top-down approach, which models the whole system as an MDP that individual agents are contributing to solve, we adopt a bottom-up approach, where we model each agent's immediate environment as an MDP. Agents optimize their local behaviours and the behaviours of agents in their immediate neighbourhood, while global behaviour is expected to arise from local and neighbourhood behaviours. In order to achieve globally optimal behaviours, agents need to be cooperative, as selfish local behaviours might negatively affect the global system behaviour rather than optimize it (Dowling & Haridi, 2008). Therefore, DWL does not aim to optimize the individual behaviours of self-interested non-cooperative agents, but optimizes global system behaviour by enabling local agent cooperation.

By using Q-learning and modelling each agent's local environment as an MDP, DWL makes a number of assumptions that might not strictly hold in dynamic multi-agent systems (Tesauro, 2007). Namely, environments might not be fully observable, might be non-stationary, and might be history dependent.

In multi-agent systems, the environment of an agent might not be affected only by its own actions, but also the actions of its neighbours, making an agent's local learning environment no longer Markovian (Busoniu et al., 2005). When the Markovian property is violated, the theoretical guarantee of convergence of Q-learning on a single-agent no longer holds (Busoniu et al., 2005). However, even though there are no theoretical guarantees, empirical results show that RL based on MDPs is still widely applicable in multi-agent systems (Busoniu et al., 2005; Dowling & Haridi, 2008).

Additionally, as DWL relies on sensors to obtain information about the environment, this information may be noisy or incomplete, violating the full observability assumption of an MDP. However, we assume these violations will not be severe enough to affect the learning agent to the point where it would require modelling as a POMDP, and can be safely ignored (Busoniu et al., 2005).

4.8 Summary

In this chapter we presented the requirements for a multi-agent multi-policy cooperative optimization algorithm for large-scale autonomic systems. Based on these requirements, we have developed such an algorithm, DWL, presented its design, and discussed how it addresses the requirements specified. In the next chapter we present the implementation of a DWL-based optimization system, and in Chapter 6 we evaluate the suitability of DWL in a collaborative multi-agent multi-policy simulation of UTC.

Chapter 5

DWL Implementation and Simulation Environment

"When one has finished building one's house, one suddenly realizes that in the process one has learned something that one really needed to know in the worst way –before one began."

- Friedrich Nietzsche

This chapter describes the implementation of DWL agents and the UTC simulation environment. We introduce the Collaborative Reinforcement Learning (CRL) framework (Salkham et al., 2008) that we used to implement RL agents. We detail the extensions to the framework that we implemented in order to facilitate development of RL agents with W-learning capabilities (referred to as RL-W agents in this chapter). We then describe the implementation of DWL agents, relationships between RL agents, RL-W agents and DWL agents, and communication between agents and the simulation environment. We identify the methods that RL-W agents need to implement in order to be compatible with DWL, as DWL is implemented as an additional hierarchical layer on top of W-learning agents. We also describe the interface between agents and the simulation environment, which acts as a collection of sensors and actuators, to collect data from the environment that agents require and to affect the environment through execution of agents' actions.

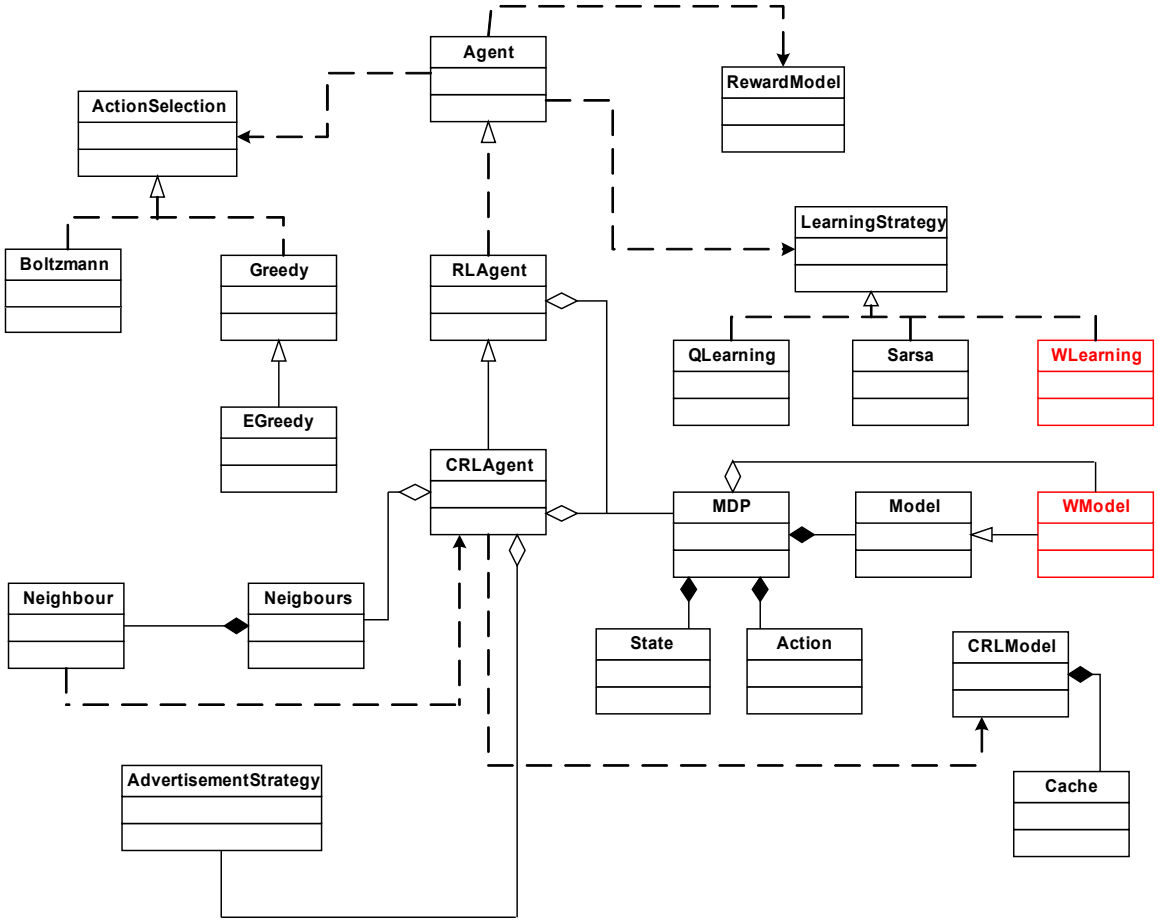


Fig. 5.1: CRL Framework (Salkham et al., 2008)

5.1 CRL Framework

For the development of our agents we use Salkham's CRL Framework (Salkham et al., 2008), a C++ library that provides components required for building single-policy collaborative RL agents (where collaboration is implemented as described in Section 2.4.3.1).

In order to enable multi-policy implementations, we have extended the CRL framework to support multi-policy RL-W, by including `WModel` and `Wlearning` classes. Classes that are a part of our extension are denoted in red in the Figure 5.1.

WModel

`WModel` extends the `Model` class provided by the CRL framework. An instance of `Model` holds information about learnt Q-values for <state, action> pairs for a given agent, and an instance of `WModel`

holds information about learnt W-values for the states of a given agent.

The methods provided by the `WModel` class are as follows:

- `void writeWModel (string location, int agentID)`, which is used to save W-values that were learnt during exploration.
- `void readWModel (string location, int agentID)`, which is used only at the start of the exploitation, to read W-values learnt during exploration.
- `void setWValue (StateValuePair svp)`, where `StateValuePair` is defined by

  ```
  typedef pair <State*,double> StateValuePair
  ```

  and is used to update the W-value for a given state.
- `double getWValue (State* )`, which is used to retrieve the current W-value for a given state.

A `Model` instance is contained with an instance of `MDP` associated with an RL agent. Instances of `MDP` associated with RL-W agents contain instances of both `Model` and `WModel` (see Figure 5.1).

WLearning

The `WLearning` class implements the W-learning process, by enabling update of W-values associated with agents' states. The update is performed using the W-learning formula presented in Chapter 2, and requires the following values:

- learning rate α
- discount rate γ
- reward received at the last time step r
- current value of W for the state being updated, $W(s)$, where the state being updated is the agent's state at the previous time step (as update is performed just before the next action is take at the next time step, after the reward has been received for the effects of the previously selected action).
- Q-value for the previous state, and action executed in the previous state $Q(s, a)$
- maximum Q-value available to an agent in the next state, max $Q(s', a')$.

α and γ are set as W-learning parameters at the start of the experiment, a reward is received from the environment at each time step, and Q-values and W-values are contained within instances of `Model` and `WModel,` which in turn are contained within an instance of `MDP` associated with the agent for which we are performing an update. The `MDP` object also keeps track of the current and previous state and the current and previous action that an agent was/is in, as required for Q-value and W-value updates.

The `W-learning` class implements and exposes the following methods:

- `void setAlpha (double alpha)`, which is used to set the value of α
- `void setGamma (double gamma)`, which is used to set the value of γ
- `void update (MDP* mdp, int rwd)`, which is used to update a W-value, by providing the reward received, as well as a pointer to an `MDP` object associated with the agent fow which the update is being performed. Access to the `MDP` is required to provide access to the Q-values and W-values required for the update.

The CRL framework, as provided by (Salkham et al., 2008), enables implementation of single-policy RL agents. After the addition of WModel and W-Learning, we can use the CRL framework to implement multi-policy RL-W agents. In the next section we describe specific RL-W agents that we use to implement UTC policies, as well as the agent generator, which enables us to generate multiple agents that implement the same system policy, but with different state spaces and action sets, according to the environment layout.

5.2 UTC RL Agent Implementation

As we have evaluated DWL in a simulation of UTC, the operating environment of the agents is represented as a road network, where each group of traffic lights at a single intersection is controlled by a DWL agent. RL actions represent traffic-light phases available to an agent, and the state-space representation is specific to each policy that an agent implements. Policies are first implemented as non-collaborative RL-W agents, and then a DWL layer is added for the creation of remote policies and the implementation of the DWL action-selection mechanism. We first describe the implementation of non-collaborative RL-W agents.

In order to implement specific UTC policies, we need to instantiate the CRL framework with policy-specific implementations of `RLAgent`, `MDP`, `Action`, and `Reward`. In order for agents to communicate with the environment, i.e., sense the traffic conditions and execute actions, we provide an

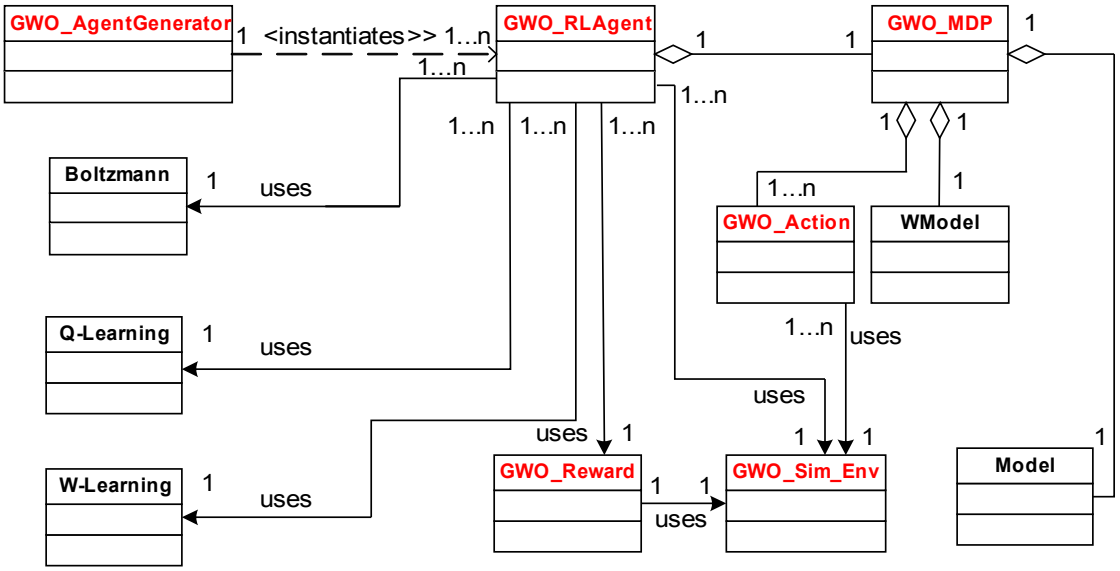


Fig. 5.2: Generation of multiple GWO RL agents

interface for communication between the UTC simulator and agents, `Sim_Env`. We also implement `AgentsGenerator`, a class that instantiates agents for all junctions in the road network, with states and actions specific to the layout of that particular junction.

To provide these agents with W-Learning capabilities, i.e., to extend them from single-policy RL agents to multi-policy RL-W agents, we need to instantiate a `WModel` object that will store W-values for specific policy states, and associate it with the instance of `MDP` associated with each instance of `RLAgent`. We also need to set `WLearning` as action nomination strategy of each instance of `RLAgent`.

An example of this process for the implementation of the GWO policy which we discussed in Chapter 3 is presented in Figure 5.2. `GWO_RLAgent`, `GWO_MDP`, `GWO_Reward`, and `GWO_Action` extend `RLAgent`, `MDP`, `Reward`, and `Action`, respectively, overloading the methods that use generic state, action, and reward representations to use states that describe congestion levels as described in Chapter 3, actions that represent traffic-light phases available at the specific junction, and a 100-point reward for being in the states specified for receiving rewards, i.e., the states where congestion is lower than at the previous time-step. `GWO_AgentGenerator` instantiates a `GWO_RLAgent` for each traffic light junction together with a corresponding `GWO_MDP`, `GWO_Reward`, and `GWO_Action`. In order for a `GWO_RLAgent` to determine the environment state and execute actions (i.e., set phases on the group of traffic lights that it controls) it needs to communicate with the environment using `GWO_Sim_Env`. `GWO_Sim_Env` provides the following methods to enable that interaction:

- `void (int junctionID, int& vehicleCount)` simulates a sensor at the junction with identifier `junctionID,` returning the total number of vehicles on all of the junction's incoming approaches.
- `Map* getMap()` provides access to the full environment layout, i.e., all of the junctions with associated incoming and outgoing junctions (neighbours), access to links and lanes that constitute the approaches, and all of the vehicles present on all of the lanes.
- `bool switchPhase(int Junction_ID, int Phase_ID)` simulates an actuator at the junction with identifier `junctionID,` setting the phase at that junction to the phase with identifier `Phase_ID.`

As it contains details of the full junction layout and can provide access to vehicle objects through junction objects, `GWO_Sim_Env` is also used to gather various statistics on simulation and agent performance, e.g., waiting times for all vehicles in the system, the number of stops for all vehicles in the system, the total number of vehicles per vehicle type and/or per junction.

In order to have a W-learning capability a `GWO_RLAgent` needs to have W-learning assigned as its action selection strategy using the following method:

```
void setNominationStrategy(LearningStrategy*);
```

where `LearningStrategy` is an instance of `WLearning:`

```
WLearning* nominationStrategy = new WLearning();
```

Agents implementing other UTC policies are implemented in a similar manner. EVO, a policy that optimizes the travel time of emergency vehicles, which we discussed in Chapter 3, is implemented using `EVO_RLAgent`, `EVO_MDP`, `EVO_Reward`, and `EVO_Action,` while `EVO_Sim_Env` provides an interface for communication with the simulation environment. Instead of a method that counts all vehicles on the junction approach that `GWO_Sim_Env` implements, `EVO_Sim_Env` enables an agent to sense if there are any (and if so, how many) emergency vehicles present in its local environment, using a method:

- `void getAmbsCount(int junctionID, int& ambCount).`

An additional policy that we implemented for DWL evaluation, that we have not used in non-collaborative experiments, is a regional, sporadic policy that optimizes Public Transport vehicles Only (PTO). We implemented this policy using `PTO_RLAgent`, `PTO_MDP`, `PTO_Reward`, and `PTO_Action`. We implemented `PTO_Sim_Env` for communication between the simulator and PTO agents, which

PTO_Reward uses to determine the current policy state, by sensing if there are any (and if so, how many) public transport vehicles present on any of the agent's approaches, using a method:

- void getBusCount(int junctionID, int& busCount).

EVO_RLAgent, PTO_RLAgent and GWO_RLAgent have full agent capabilities, i.e., can be deployed on their own for non-collaborative optimization for single policies they are implementing. When the DWL layer is added, EVO_RLAgent, PTO_RLAgent and GWO_RLAgent retain all of their agent capabilities (e.g., sensing the environment and learning) apart from executing actions in the environment directly, as all of them are controlling the same set of actuators. Control over action selection is transferred to a DWL_RLAgent, to avoid interference between action executions by different RL agents implementing different policies. Once a DWL_RLAgent has selected an optimal action, it initiates action execution by one of the agents. We outline the implementation of this action selection process in the next section.

5.3 UTC DWL Agent Implementation

In this section we present the implementation of DWL agents, the steps required for generation and initialization of DWL agents at the start of the simulation, as well as the sequence of steps required for action selection using DWL.

5.3.1 DWL Agent Generation

In order to implement multi-policy collaborative UTC agents using DWL agents, we have developed the following classes: DWL_AgentGenerator, DWL_RLAgent, DWL_MDP, and DWLCoop_MDP. We discuss the use of each below.

5.3.1.1 DWL_AgentGenerator

DWL agents are generated using an instance of DWL_AgentGenerator that instantiates DWL agents for all of the junctions in the road network. DWL_AgentGenerator also instantiates agent generators for all policies that are to be deployed in the system, which in turn generate an agent for each policy on each agent at which the policy is to be deployed (see Figure 5.3). Note that due to the potentially different spatial scope of policies, not all policies will be implemented on all of the agents. DWL agents are generated for each agent on the network, while DWL_AgentGenerator reads the list of other policies that are to be deployed, together with the corresponding agents at which they are to be deployed, from the AgentTypes configuration file. Each policy in the system is assigned a unique

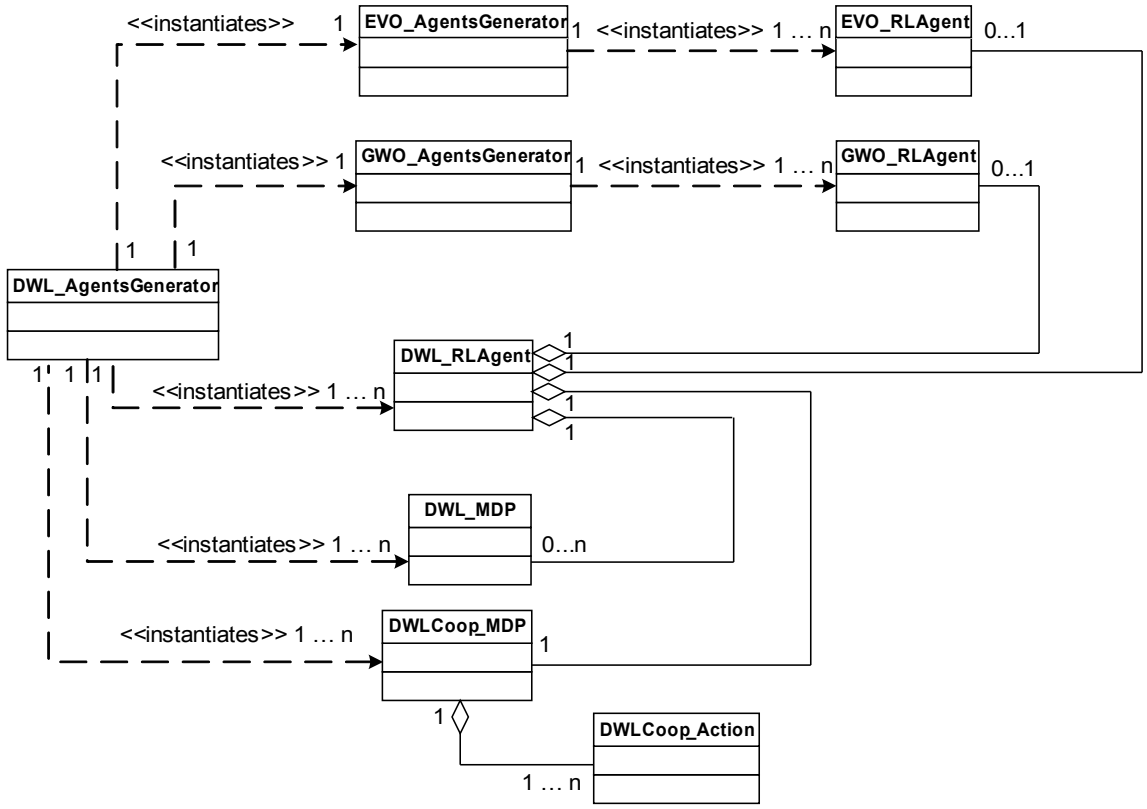


Fig. 5.3: DWL agent generation

identifier, which is used in `AgentTypes` to identify it and used by `DWL_AgentGenerator` to instantiate the correct agent generators. Single-policy collaborative DWL optimization is implemented by only listing a single policy identifier in `AgentTypes.`

5.3.1.2 DWL_RLAgent

`DWL_RLAgent`, even though it also extends `RLAgent`, differs in implementation from other RL agents that implement single policies, as they are charged with different tasks. `DWL_RLAgent` does not implement a specific policy itself, so it does not have an `MDP` or actions associated with it. `DWL_RLAgent` is in charge of action selection on a single junction, where the actions considered for selection are received from all local policies (implemented as RL-W agents, e.g., `EVO_RLAgent`, `GWO_RLAgent`, `PTO_RLAgent`) and all remote policies, implemented as instances of the `DWL_MDP` class. Therefore, a `DWL_RLAgent` contains the following members:

- a map of pointers to all local agents, `map <int, RLAgent*> allagents`, where `int` is the identifier of the policy type that `RLAgent*` implements.
- a map of pointers to MDPs for the remote policies grouped by the neighbour with which they are associated

  `map <int, NeighborMDPs> remotePolicies` where `int` is the junction identifier of a neighbour, and `NeighborMDPs` is defined as

  `map <int, DWL_MDP*> NeighborMDPs` where `int` is the identifier of the policy type `DWL_MDP`* is implementing.

During the initialization process, all local policies are instantiated and associated with their corresponding `DWL_RLAgent` instances. After this is done, each `DWL_RLAgent` obtains the list of its one-hop neighbours from the environment (i.e., vectors of all the first upstream and downstream traffic-light junctions), obtains from the `DWL_RLAgent` on each of its neighbouring junctions the list of all policies (i.e., the list of local RL agents stored in `allagents`) that the neighbour implements, and instantiates a `DWL_MDP` object for each of those policies.

In order to be able to engage in different levels of cooperation, each DWL_RLAgent needs to either have a variable which stores a predefined cooperation coefficient, or an instance of `DWLCoop_MDP` to store the Q-values associated with each available cooperation coefficient. In order to update these Q-values, a `DWL_RLAgent` needs to be associated with an instance of `Q-learning` as a learning strategy for learning values of C, as follows:

```
QLearning* coopLearningStratetgy = new QLearning();
DWLagent→setCoopLearningStrategy(coopLearningStrategy);
```

where `DWLagent` is an instance of `DWL_RLAgent`.

In order to select a C based on these Q-values for use at each action selection step a `DWL_RLAgent` also needs to have an associated ActionSelection instance, as follows:

```
Boltzmann* coopActionSelectionStrategy = new Boltzmann();
DWLagent→setCoopActionSelectionStrategy(coopActionSelectionStrategy);
```

where `DWLagent` is an instance of `DWL_RLAgent`.

5.3.1.3 DWL_MDP

`DWL_MDP` is extended from the `MDP` class, and therefore contains `Model` and `WModel` instances, which store Q-values and W-values for the remote policies. `Model` is built using local actions and neighbour's policy's states, and `WModel` is built using neighbours' policy states. Therefore, in order for `DWL_AgentGenerator` to be able to get access to the actions and states required to build `Model` and `WModel` for `DWL_MDP`, RL-W agents need to provide access to the list of their actions and states through providing access to their `MDP`. E.g.:

```
std::map<string,GWO_Action*>localActions=GRLLocal→getMDP()→getActions();
```

where `GRLLocal` is an instance of `GWO_RLAgent` and

```
std::map<string,State*>remoteStates=PRLRemote→getMDP()→getStates();
```

where `PRLRemote` is an instance of `PTO_RLAgent`.

The above two methods are the only methods that RL-W agent and its corresponding MDP need to implement to facilitate the initialization of DWL remote policies. From the obtained map of remote policy states and local agent actions, a new `DWL_MDP` is created for each remote policy as follows:

```
DWL_MDP* dwlmdp = new DWL_MDP (ID, remoteStates, localActions);
```

As already noted, local policies are implemented as fully functioning RL agents, however, remote policies do not have full agent capabilities, but are only implemented as an `MDP`, without the capability to sense the environment or execute actions. The reason for this is that remote policies, in fact, are never required to sense the environment, as they receive state information from the corresponding local policy on a corresponding neighbour, and never need to execute actions in the environment, as they only provide action suggestions to the `DWL_RLAgent` which then instructs one of the local agents which action to execute.

5.3.1.4 DWLCoop_MDP

Each `DWL_RLAgent` also contains an MDP associated with a process for learning a cooperation coefficient C, `DWLCoop_MDP`. `DWLCoop_MDP` extends `MDP` and contains a `Model` that stores Q-values for cooperation actions. Cooperation actions are implemented as instances of `DWLCoop_Action`. Note that all Q-values are associated with a single default state, as we do not distinguish between policy states when learning C.

5.3.1.5 Initial Agent Wake-up

After all of the agents have been instantiated, the simulator schedules their first wake up, to execute the first round of actions. Further wake ups are scheduled based on the duration of actions that agents execute. In our simulation, action duration is set to 20 seconds, so all of the agents are woken up every 20 seconds. However, if actions (i.e., in our simulation, traffic light phases) were to have varying duration, the wake up time for each agent can be scheduled separately in order to enable asynchronous wake-ups. At each wake up, on each traffic light-controlled junction in the network, one step of DWL action selection is performed. We describe the implementation of DWL action selection in the next section.

5.3.2 DWL Action Selection Implementation

The process of action selection performed at each learning step at each of the `DWL_RLAgent` objects is depicted in Figure 5.4. The simulator wakes up `DWL_RLAgent` on each of the traffic-light junctions. `DWL_RLAgent`, in turn, wakes up all RL-W agents implementing local policies. Local policies sense the environment and map the conditions to their local state representation. Based on the reward received for being in that state, each local policy updates its Q-value for the previous state and previous action executed, and the W-value for the previous state. Based on the current state, each local policy, upon request, sends an action suggestion to `DWL_RLAgent`. Action suggestions are stored in the vector of suggestions, together with the W-values for the state that the policy is currently in, and the identifier of the agent and policy suggesting it, as shown below.

```
vector <suggestion> actions;
```

where class `suggestion` is specified as:

```
class suggestion{
   string action_id;
```

```
    double w_value;
    string agentID_policyID;
};
```

After suggestions from all of the local policies have been received, `DWL_RLAgent` wakes up all of its associated remote policies, i.e., the `DWL_MDP` objects implementing them. Remote policies request from the corresponding local policies on the corresponding neighbour the current policy state and the reward received for being in that state. These are the only two messages that need to be exchanged between an agent and each of its neighbours' policies at each action selection step. Based on the reward received, each remote policy updates its Q-value for the previous state and previous action executed and W-value for the previous state. Based on the current state, each remote policy, upon request, sends an action suggestion to `DWL_RLAgent`, together with an associated W-value. W-values are multipled by a cooperation coefficient C. Either a predefined value of C is used, or in the case of a DWL deployment that learns C, a value is selected using an instance of `ActionSelection` set as a `coopActionSelectionStrategy` in `DWL_RLAgent`. In the case of a DWL deployment that learns C, prior to selection of C for the current action-selection step, a `DWL_RLAgent` updates the Q-value for C used in the previous step using the sum of the rewards received on all local and remote policies for their current states.

Finally, after W-values of remote policies have been multipled by C, a `DWL_RLAgent` identifies the maximum W-value, and selects the associated action for execution. It notifies all local and remote policies of the action selected, so they can use it in the next action selection step to update their Q-values, and instructs one of the local RL-W agents to execute the action. An RL-W agent executes an action, i.e., it changes the phase of the traffic lights controlled by an agent to a phase selected in the action selection process.

In order for `DWL_RLAgent` to execute an action selection as described above, underlying RL-W agents do not need to implement any additional methods, but only need to ensure they provide `DWL_RLAgent` with access to some of the methods they already implement for single-policy or multi-policy non-collaborative implementations. Each RL-W agent needs to provide access to its associated `Reward` and `MDP`, and `MDP`, in turn, needs to provide access to the associated `Model` and `WModel`, as well as the current/previous state and current/previous action. `DWL_RLAgent` also needs access to an instance of `ActionSelection`, in order to select the next action for a given `MDP`, and an instance of `W-Learning`, in order to update W-values in a given `MDP` using the latest reward.

Each RL-W agent has a `wakeup()` method, which is used to wake the agent up at set intervals during the simulation and executes one algorithm learning step. One learning step consists of sensing

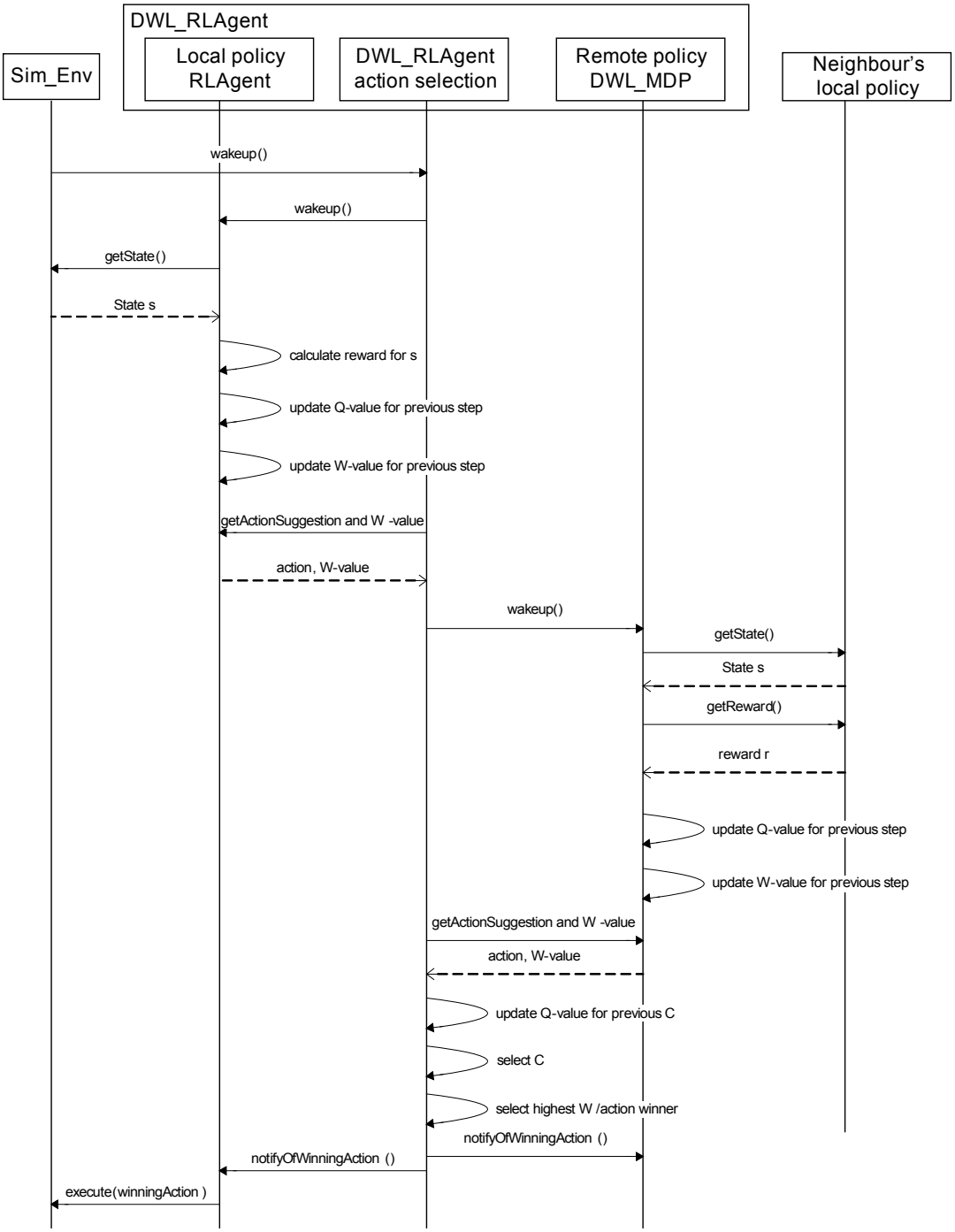


Fig. 5.4: DWL action selection sequence diagram

the environment conditions, mapping the conditions to one of the policy states, getting a reward for being in that state, updating Q-values for the previous state-action pair and W-values for the previous state, selecting an action to execute in the current state, and finally executing that action. In order for an RL-W agent to be compatible with DWL, action execution needs to be called from outside the `wakeup()` method, so an agent can still perform the full learning process, but enable a `DWL_RLAgent` to take control over action selection and intercede just before action execution. `DWL_RLAgent`, after it has selected an action for execution based on all local and remote policy suggestions, changes the current action of all RL-W agents to the one that it has selected.

5.3.3 DWL Agent Summary

The implementation of DWL presented in this section follows the DWL requirements outlined in the previous chapter. DWL is decentralized and uses only local actions and interactions, i.e., agents use only local information (vehicle types and vehicle counts only on their approaches) and communicate only with their one-hop neighbours (first upstream and downstream traffic light junctions). DWL enables collaboration between heterogeneous agents and policies, as action suggestions received from local policies and remote policies have the same format and are received in the same manner regardless of policy type, scope, or agent layout.

5.4 Summary

In this chapter we have presented the implementation of DWL agents as defined by the algorithm design in Chapter 4. We have presented the CRL framework that we used to implement single-policy RL agents. We have presented extensions to the framework we have implemented in order to enable the development of multi-policy W-learning agents. We have then presented the implementation of DWL agents, which are added as an action-selection layer on top of W-learning agents, to implement multi-policy collaborative agents. This layered implementation approach enables an easy comparison between the performance of single-policy, non-collaborative multi-policy, and collaborative multi-agent multi-policy implementations, while also enabling the easy addition of further policies.

Chapter 6

DWL Evaluation

"*An optimist is a person who sees a green light everywhere,*
while a pessimist sees only the red stoplight
. . . The truly wise person is colorblind."
Albert Schweitzer

In this section we present an evaluation of DWL as a technique for multi-agent multi-policy optimization in decentralized autonomic systems. We present the objectives of the evaluation, along with the metrics that we used to measure the performance of DWL. We describe the experiments we used for the evaluation, and present and analyze their outcomes.

6.1 Evaluation Objectives

The goal of the evaluation of DWL presented in this chapter is to establish how well DWL addresses the requirements specified in Chapter 4. The main objective of the design of DWL was to provide a decentralized, self-organizing, multi-policy, multi-agent, optimization technique that improves system performance by enabling simultaneous deployment of heterogeneous policies and collaboration between heterogeneous agents. In Chapter 4 we outlined how the design of DWL addresses these requirements, however, the success of DWL as a technique for multi-agent multi-policy optimization depends on its performance in a variety of evaluation scenarios.

DWL can be said to have succeeded in addressing our requirements for a multi-agent multi-policy optimization technique if it satisfies the following performance requirements:

1. DWL outperforms existing UTC optimization techniques. As we evaluate DWL in a simulation of UTC, to be deemed a success, DWL would need to outperform existing techniques used for optimization in that domain. Note that we only consider stationary traffic conditions, as, once a suitable behaviour for that set of conditions has been learnt, DWL does not have the ability to adapt to a change in traffic pattern. Pattern change detection and adaptation to new patterns is a subject for future work (see Chapter 7).

2. DWL outperforms non-collaborative multi-policy deployments using existing RL-based techniques. As a representative of a non-collaborative multi-policy deployment we use independent agents implementing W-learning. W-learning has been shown to be a suitable technique for non-collaborative multi-policy optimization in UTC as presented in Chapter 3.

3. The ability to learn the optimal levels of collaboration improves the performance of DWL over using predefined collaboration coefficients.

4. DWL is suitable for single-policy collaborative deployments, i.e., it outperforms single-policy non-collaborative deployments. If a system needs to optimize its behaviour towards only a single policy, DWL can improve the performance of the system towards that policy by utilizing agent collaboration.

5. DWL respects policy priorities. When multiple system policies are addressed using DWL, the priority of those policies is respected; the performance of a lower priority policy might be sacrificed to some extent for an increase in the performance of the higher priority policy but not vice versa.

6. DWL multi-policy deployments improve the performance of at least one vehicle type when compared to single-policy DWL deployments, if multiple vehicle types are present in the system. For example, if both cars and buses are present in the system, we expect a single-policy deployment addressing only cars to achieve good performance with respect to cars, while buses might be neglected resulting in their poor performance. If a policy that addresses buses is added to the system, we expect a multi-policy DWL deployment to improve the performance of buses, while potentially having small negative effects on cars, as cars would not be the only vehicle type whose performance is being addressed. While single-policy deployments, in the presence of multiple vehicle types, neglect the performance of the vehicle type that they are not addressing, multi-policy deployments ensure that both vehicle types are addressed simultaneously and no vehicle type is neglected. Therefore, we consider this to be a performance improvement, as nei-

ther of the vehicle types are neglected, even though one vehicle type might suffer small adverse effects in order for performance of the other vehicle type to be improved.

7. DWL improves the performance of system policies under a variety of environmental conditions (e.g., traffic load, traffic patterns) and policy characteristics (e.g., scope, priority).

8. DWL does not require prohibitively long training times to learn performance-improving behaviours.

As part of this evaluation we also investigate the impact of policy relationships on the W-values of local and remote policies deployed on DWL agents, as DWL is designed to learn W-values that reflect the dependencies between agents and between policies, and to take advantage of those dependencies to improve performance.

In the next section we present the metrics we use to assess the performance of DWL.

6.2 Evaluation Metrics

To evaluate the performance of DWL we use a number of metrics relevant to our application area, UTC. These metrics are as follows:

1. Average vehicle waiting time, per vehicle type.

2. Number of vehicles served.

3. Traffic density.

4. Average number of stops per vehicle, per vehicle type.

The metrics 1-3 used in the evaluation scenarios presented here are the same ones used for the evaluation of single-agent, multi-policy techniques presented in Chapter 3. Please refer to that chapter for an explanation of the metrics, and the rationale for their use and suitability.

In the evaluation scenarios in this chapter we have also measured the average number of vehicle stops, per vehicle type for the duration of the experiment. We introduce this as an additional metric to help us assess the suitability of DWL for optimization in UTC, as it reflects traffic congestion and traffic flow (Klein, 2001). As we believe this metric is related to vehicle waiting time (i.e., the less stops the vehicle makes, the lower its total waiting time will be), we do not analyze it in as much detail as other metrics, but only analyze its consistency with other metrics.

6.3 Evaluation Scenarios

In this section we describe the scenarios that we used to evaluate the suitability of DWL for multi-policy optimization in large-scale decentralized autonomic systems. We first introduce the UTC policies we used for evaluation, present the simulation environment and parameters, and then describe the specific details of each scenario used.

6.3.1 UTC Policies

As part of the evaluation, we have implemented UTC baselines, single-policy deployments of DWL, and multi-policy deployments of DWL, as described below.

6.3.1.1 Baselines

As baselines with which to compare DWL's performance, we use the same baselines that we used in our case study on non-collaborative multi-policy optimization, namely RR and SAT. For more details on these please refer to Chapter 3.

6.3.1.2 Single-Policy Deployments

As DWL has been designed to support multi-policy optimization in heterogeneous environments, the policies we have selected for evaluation differ in their regional and temporal scope, and in their priority. Different scopes and priority, i.e., heterogeneity of policies, also causes heterogeneity of agents with respect to the policies that they are implementing. The policies we implemented are presented below.

GWO - Optimize global vehicle waiting time GWO has already been introduced in Chapter 3, as it was used in our preliminary case study. For details of the state space and reward function for this policy please refer to that chapter. However, in Chapter 3, GWO had only been implemented in a non-collaborative single-policy deployment using Q-learning, and in combination with EVO in a non-collaborative multi-policy deployment using W-learning. In the evaluation scenarios that we describe in this chapter, we have implemented GWO using DWL with a range of predefined collaboration coefficients and in a DWL deployment where C is learnt rather than predefined.

PTO - Prioritize public transport vehicles PTO is a regional, sporadic policy that prioritizes public transport vehicles over other traffic. An agent's state space encodes information about which approach(es), if any, have public transport vehicles (in our simulation buses) waiting (e.g., "Bus present on a_1"). The size of the state space depends on the layout of the junction, i.e., on the number

of approaches on the junction. Bus(es) can be present on all of the approaches, on none of the approaches, on only one approach at a time, or on various combinations of two or three approaches at a time. Agents are rewarded (120 points in this set of experiments) for being in a state where there is no bus present at any of the approaches. This motivates the agents to, as soon as possible, return to a state with no buses present, by enabling buses to pass. This policy does not address any other vehicle type and only takes buses into account when making action selections. PTO has been implemented using DWL with a range of predefined collaboration coefficients and in a DWL deployment where C is learnt rather than predefined.

The implementation of PTO is similar to the implementation of EVO introduced in Chapter 3 and used in our case study on existing single-agent multi-policy approaches. As EVO addressed only emergency vehicles that accounted for only 0.5% of the overall traffic, in this set of experiments we use PTO, which addresses public transport vehicles that make up a larger proportion of overall traffic. As public transport vehicles have a higher priority than cars, but a lower priority that emergency vehicles, we use 120 points as a reward in PTO, whereas EVO used a 200-point reward.

6.3.1.3 Multi-Policy Deployments

GW-PT Both GWO and PTO are addressed simultaneously using a multi-policy DWL deployment, GW-PT. GW-PT has been implemented using a range of predefined collaboration coefficients, as well as in a DWL deployment where C is learnt rather than predefined.

When comparing GW-PT to other non-DWL approaches in this section, we refer to it as "DWL".

6.3.2 Simulation Setup

Evaluation of DWL was performed in the same traffic simulator as our case study presented in Chapter 3. The map we used for evaluation is shown in Figure 6.1.

The map corresponds to Dublin's city center, and consists of 270 junctions, 62 of which are controlled by traffic lights. The traffic-light junctions are marked on the map by a circle, and correspond to junctions which are currently controlled by Dublin's traffic control system. As the policies that we use for evaluation address cars and public transport vehicles, we use both of these vehicle types in our simulation.

Personal vehicles (i.e., cars) enter the system through 14 junctions on the edge of the map (representing the junctions where traffic enters the city centre area) and 3 junctions in the center of the map (representing the city centre parking lots), as shown in Figure 6.2 using black dots, and travel along 260 different routes, to exit the system through one of the 17 entry junctions that also serve as vehicle

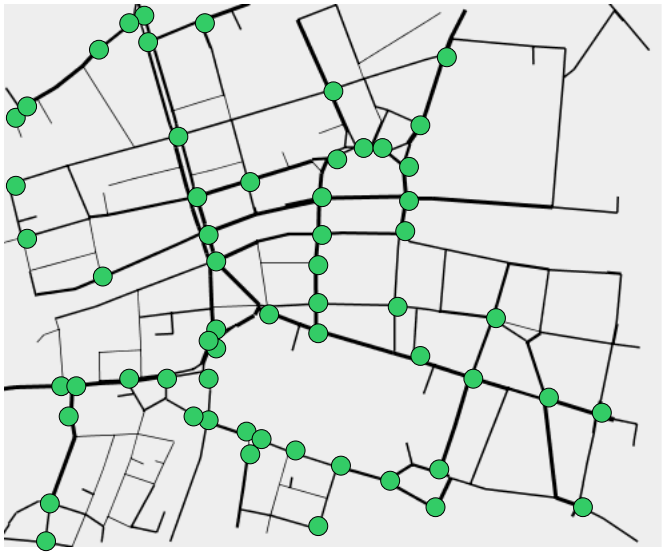


Fig. 6.1: DWL evaluation map

finishing points. Vehicles follow the shortest route between their start and end junctions. Such a large variety of routes ensures that each traffic-light junction in the system has traffic travelling through it, but also that this traffic flow is not spread evenly across the map, as some junctions will be visited by a larger number of vehicles on their routes.

Buses join and exit the system only through a subset of the entry/exit junctions, and travel only along 20 designated bus routes. Each bus route has at least one bus stop where each bus makes a mandatory 20-second stop. Due to an overlap of routes, some buses stop at 2 or more bus stops during the simulation. 47 out of the 62 traffic-light junctions are positioned on bus routes. PTO is deployed only at these junctions, and is therefore of regional spatial scope, implemented by ∼75% of agents in the system. Buses make up 5% of the overall traffic in the simulation, reflecting PTO's sporadic temporal scope.

6.3.3 Simulation Parameters

The simulation of baselines, RR and SAT, was run for a duration of 750 minutes of simulation time. Traffic was joining the simulation for the first 720 minutes (12 hours), while the final 30 minutes were added to allow all of the vehicles to arrive to their destinations and exit the system. The action duration for RR was fixed to 20 seconds, and SAT parameters used were those determined to yield the best performance; the minimum action duration was set to 20 seconds, the phase increment to 10 seconds, and the maximum duration of the cycle factor was set to 1.2. Please refer to Chapter 3 for

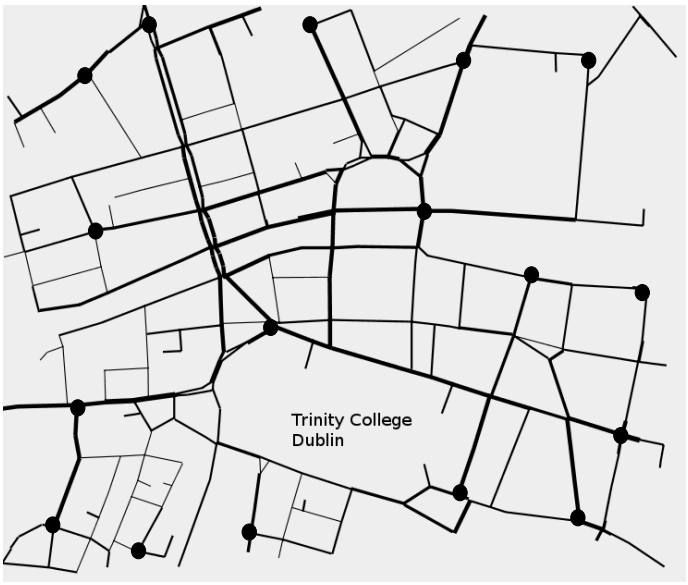


Fig. 6.2: Vehicle route start/end junctions

more details on these parameters.

Simulations for both single- and multi-policy DWL deployments were run in five separate stages[1], each lasting 750 minutes of simulation time (unless otherwise stated when describing individual scenarios), to allow the exploration of actions and the learning of Q-values and W-values, as well as learning the optimal cooperation coefficients. Deployments of DWL with predefined cooperation coefficient need to include only stages 1-3, while deployments of DWL that learn C need to include stages 1-5. The five stages are as follows:

1. Learning Q-values for all local and remote policies. At the start of this stage all Q-values for all local and remote policies are initialized to 0. The action selection strategy used is Boltzmann with temperature parameter set to 10000 at the start of the stage. Temperature cools down uniformly to reach 1 at the end of this stage. The task of this stage is to explore the outcomes of all the actions that local and remote policies can take. The cooperation coefficient is set to a predefined value $0 < C < 1$ during this stage.

2. Learning W-values for all local and remote policies. This stage is initialized using the Q-values learnt in stage 1, which are also updated during this stage. W-values are initialized to 0 at the start and are learnt during this stage. The action selection strategy used is Boltzmann with

[1]Stages 1-5 were executed separately in order to clear leftover traffic from the previous stage before starting a new one, i.e., to make sure that congestion that may have occurred during the exploration stages does not have negative effects on the performance in exploitation stages.

temperature parameter set to 1, i.e., Q-values are mostly exploited, allowing local and remote policies to learn W-values relative to the actions that are most likely to be selected during the exploitation stage. The cooperation coefficient is fixed, meaning that Q-values and W-values are learnt given a predefined value of C.

3. Exploiting Q-values and W-values. This stage is initialized using the Q-values and W-values as learnt during stages 1 and 2. The action selection strategy used is Boltzmann with temperature parameter set to 1, so both Q-values and W-values are exploited, but also updated during this stage. The cooperation coefficient is fixed to a predefined value during this stage. Therefore, the performance during this stage is the best that DWL can achieve given a certain fixed value of C.

4. Learning optimal values of C per agent while exploiting Q-values and W-values. This stage is initialized using Q-values and W-values saved after execution of stage 3. The action selection strategy used is Boltzmann with temperature parameter set to 1, so both Q-values and W-values are exploited, but also updated during this stage. However, unlike stage 3 where C is predefined, in this stage Q-values associated with values of C are learnt. The action selection strategy used for this process is also Boltzmann but the temperature parameter is set to 10000, and cools down uniformly during this stage to reach 1 at the end. This enables exploration of various values of C.

5. Exploiting learnt Q-values, W-values and values of C. This stage is initialized using Q-values and W-values returned as output of stage 4, and Q-values for various values of C, also learnt in stage 4. This stage is mostly exploitative and the temperature for all action selection processes is set to 1. Therefore, performance during this stage is the optimum that DWL can achieve and it takes full advantage of all of DWL's features, i.e., remote policies, the cooperation coefficient, and the ability to learn cooperation coefficient.

α and γ for all Q-learning and W-learning processes in the evaluation scenarios presented in this chapter are in all stages set to 0.1 and 0.3, respectively, which are values experimentally determined to result in best performance in our scenarios. The duration of each action selected in any of the stages is fixed to 20 seconds. There are no restrictions on repeating actions so an approach can be given a green signal for longer than 20 seconds if traffic demand requires it.

The results presented in this chapter represent performance measured during exploitation stages, namely, stage 3 for deployments of DWL with predefined values of C, and stage 5 for deployments of DWL where C is learnt by agents. Each stage for each deployment is executed five times, and the

average values across the five executions are presented. Two-tailed t-tests were performed to investigate the statistical significance of the differences in performance of different algorithms deployed. Where t-tests return values of $p < 0.05$, results are considered statistically significant.

We evaluated the performance of DWL in five different traffic scenarios. Within a single scenario, all five executions ran with the same vehicle input files, i.e., vehicles' join times, their origins, routes and final destinations were constant. Vehicle join times were generated randomly, by specifying only the total number of vehicles to join at a given location during the simulation (e.g., during 12 hours, 3500 vehicles should enter the simulation at junction X). Varying the input files, especially at higher loads, would not create significant changes in traffic patterns, due to the high frequency of vehicle join times, and the high density of vehicles present in the system. Instead, differences in traffic conditions were created due to the RL-based scenarios selecting actions stochastically at each time step. Different traffic lights' actions result in different traffic conditions, i.e., even if vehicles join the simulation across all five executions at the same time, they arrive at different junctions on their route and their destinations at different times and their waiting time at those junctions varies, as determined by traffic-light signal settings. More significant differences in traffic conditions were simulated by generating five different vehicle input files for five different evaluation scenarios, reflecting different traffic patterns that might occur.

In the next section we describe the specific evaluation scenarios that we implemented, outline their objectives and present additional scenario-specific details relating to traffic load and vehicle routes.

6.3.4 List of Evaluation Scenarios

In this section we present details of the scenarios that we implemented, in terms of the number of junctions, traffic load, traffic patterns, vehicle types, and vehicle routes, as well as the rationale behind evaluation of DWL performance using a given scenario.

6.3.4.1 Scenario 1: Multi-Policy Optimization of Uniformly Distributed Traffic Under Different Traffic Loads

This scenario was implemented to evaluate the performance of DWL in uniform traffic conditions under different traffic loads. We consider a uniform traffic pattern to be a set of traffic conditions where an approximately equal number of vehicles travel through the map area in opposing directions, i.e., the number of vehicles travelling North to South is similar to the number of vehicles travelling South to North, and the number of vehicles travelling East to West is similar to the number of vehicles travelling West to East.

The scenario was implemented using the Dublin city center map, as shown in Figure 6.1. Cars travel along 260 routes, with an equal number of vehicles on each route, and buses travel along 20 routes, where those routes are a subset of the 260 car routes, and each route has an equal number of buses. However, note that these routes overlap, so even though the demand at the start of the routes is the same, demand on various roads and junctions in the inner parts of the map might vary. For example, major four-way junctions are situated on more routes than smaller two-approach junctions and will therefore have a higher number of vehicles travelling through them. The length of the vehicle routes also differs; the shortest vehicle route spans across 4 junctions, the longest route spans 37 junctions, and the average route length is 19 junctions. Buses represent 5% of total traffic.

This scenario has been implemented for two traffic loads:

- low load: ∼35,000 vehicles are inserted in the system over 12 hours, or the equivalent of ∼3000 vehicles per hour.
- high load: ∼60,000 vehicles are inserted in the system over 12 hours, or the equivalent of ∼5000 vehicles per hour.

The loads are based on traffic-flow data reported by the Dublin Transportation Office (DTO, 2006). During the morning rush hour, between 7:30 and 9:30, on average 12,000 vehicles per hour enter the inner-city area, consisting of approximately 250 junctions. Our map corresponds to approximately one quarter of that area, or ∼60 junctions, indicating an estimated figure of one quarter of the traffic flow, or 3000 vehicles per hour. We take that figure to represent the lower end of our estimate, i.e., low load. For the high load, we use a figure of 5000 vehicles per hour, taking into account that our map represents the heart of the city including Dublin's main street, O'Connell street, so one quarter of the area might attract more than one quarter of the traffic, in this case an estimated 40% of the total traffic entering the inner city.

As part of this scenario we ran simulations with the following policy deployments:

- RR and SAT
- GWO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GWO using DWL with learnt values of C
- PTO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- PTO using DWL with learnt values of C
- GW-PT using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}

- GW-PT using DWL with learnt values of C.

This set of deployments enables us to compare the performance of DWL with the performance of the baselines, to compare the performance of single-policy deployments to multi-policy deployments in the presence of multiple vehicle types, to compare non-collaborative (C=0) deployments with collaborative DWL deployments, and to compare DWL deployments with predefined values of C to DWL deployments with learnt values of C, all under uniform traffic conditions for both low and high traffic loads.

6.3.4.2 Scenario 2: Multi-Policy Optimization of Non-Uniformly Distributed Traffic (3:1 pattern)

This scenario was implemented to evaluate the performance of DWL for optimization of non-uniformly distributed traffic. We consider a non-uniform traffic pattern to be a set of traffic conditions where the numbers of vehicles travelling in opposing directions on the map are not equal. In this scenario the number of vehicles travelling from the North side of the map to the South side of the map is three times larger than the number of vehicles travelling from the South side to the North side. Therefore we call this traffic pattern the 3:1 pattern. This set of conditions is designed to simulate morning or evening rush hour traffic, when most of the traffic tends to move towards the city centre or is leaving the city center. The rest of the simulation conditions, in terms of the simulation map, number of junctions, routes and vehicle type ratios are identical to the conditions in Scenario 1 described in the previous section.

As part of this scenario we ran simulations with the following policy deployments:

- RR and SAT
- GWO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GWO using DWL with learnt values of C
- PTO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- PTO using DWL with learnt values of C
- GW-PT using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GW-PT using DWL with learnt values of C.

This set of deployments enables us to compare the performance of DWL with the performance of the baselines, to compare the performance of single-policy deployments to multi-policy deployments in

the presence of multiple vehicle types, to compare non-collaborative (C=0) deployments with collaborative DWL deployments, and to compare DWL deployments with predefined values of C to DWL deployments with learnt values of C, under non-uniform traffic conditions.

6.3.4.3 Scenario 3: Multi-Policy Optimization with Conflicting Traffic

This scenario was implemented to evaluate the performance of DWL for optimization of conflicting traffic, i.e., conflicting policies. We consider a conflicting traffic pattern to be a set of traffic conditions where vehicles of one type, addressed by one policy, travel in different directions from the vehicles of the other type, addressed by the other policy.

In terms of the map used, number of junctions, and vehicle routes, the traffic conditions in this scenario are the same as in Scenarios 1 and 2. However, in previous scenarios, the 20 bus routes were a subset of the overall 260 routes, while in this scenario car traffic was removed from the 20 routes used by buses, and is present only on the remaining 240 routes. Buses, therefore, use different routes from cars, and are more likely to approach a junction from a different approach than cars, and proceed in a different direction to cars. We believe this scenario is not a realistic scenario in UTC, as most major routes are used by both private and public vehicles, however, we implemented the scenario to evaluate DWL's performance in conflicting situations (i.e., where buses and cars are travelling on different routes), should they occur in UTC, or in any other potential application areas for DWL. This experiment was performed for a single traffic load of ∼3000 vehicles per hour.

As part of this scenario we ran simulations with the following policy deployments:

- RR and SAT
- GWO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GWO using DWL with learnt values of C
- PTO using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- PTO using DWL with learnt values of C
- GW-PT using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GW-PT using DWL with learnt values of C.

This set of deployments enables us to compare the performance of DWL with the performance of the baselines, to compare the performance of single-policy deployments to multi-policy deployments, to compare non-collaborative (C=0) deployments with collaborative DWL deployments, and to compare

DWL deployments with predefined values of C to DWL deployments with learnt values of C, under conflicting traffic conditions.

6.3.4.4 Scenario 4: Optimization of Multiple Policies with Different Spatial Scope and Priority Relationships

This scenario has been designed to investigate the relationship between policy spatial scope and policy priority, and verify that performance improvements in DWL arise from a higher policy priority rather than from a policy's spatial scope, i.e., that DWL respects policy priorities regardless of other policy characteristics.

The set of traffic conditions in this experiment is the same as in Scenario 1, apart from the value of rewards obtained by individual policies. Instead of using the default rewards for PTO and GWO in the DWL deployment, we varied their rewards and the relationship between the rewards, to simulate varying relationships between policy priorities and their spatial scopes. In all of the previous scenarios the regional policy (PTO) had higher priority than the global policy (GWO). In this scenario, we evaluate DWL in a set of traffic conditions where a regional policy has the same priority as a global policy, and in a set of traffic conditions where a global policy has a higher priority than a regional policy. This experiment was run for only one traffic load of ∼3000 vehicles per hour.

The combinations of PTO and GWO rewards for which this set of experiments was performed were as follows:

1. PTO r = 120 and GWO r = 100, i.e., a scenario where the regional policy PTO has a higher priority than the global policy GWO. This is a default priority relationship used in our other scenarios, so this set of experiments was not repeated, instead, results from Scenario 1 were reused.

2. PTO r = 100 and GWO r = 100, i.e., a scenario where the global and regional policy are of the same priority.

3. PTO r = 100 and GWO r = 120, i.e., a scenario where the global policy GWO has a higher priority than the regional policy PTO.

As part of this scenario we ran simulations with the following policy deployments:

- GW-PT using DWL with a predefined value of C = {0, 0.2, 0.4, 0.5, 0.6, 0.8, 1}
- GW-PT using DWL with learnt values of C

for all three combinations of PTO and GWO rewards listed above. This set of deployments enables us to evaluate if DWL respects relative policy priorities under variety of priority and regional scope relationships, both when deployed with predefined values of C and with learnt values of C.

6.3.4.5 Scenario 5: Multi-Policy Optimization in Time-Constrained Scenarios

This scenario was designed to evaluate the feasibility of the training time required by DWL. The set of traffic conditions in this scenario is identical to the conditions in Scenario 1. The experiment was run for only a single traffic load, of ∼3000 vehicles per hour. However, instead of running each of the stages for 750 minutes, the duration of the training stages in this scenario was gradually reduced in order to identify the minimum training time required for DWL to start outperforming the baselines. For simplicity, this set of experiments was only run with GW-PT with predefined C=0.2, which was experimentally determined to be the best-performing predefined C during the low traffic load as presented in Scenario 1. We ran only phases 1-3, i.e., we did not run DWL for the further two stages in which agents learn C, as we were not aiming to achieve the best overall performance of DWL, but identify the point at which DWL starts to outperform the baselines. The duration of training phases 1 and 2 was set to 50 minutes, 100 minutes, 200 minutes, 300 minutes, 400 minutes, 500 minutes, 600 minutes, 700 minutes and 750 minutes. The exploitation phase 3 was of 750 minutes duration regardless of the duration of the training phase, so that vehicle waiting times remain comparable.

6.3.4.6 Summary of Evaluation Scenarios

In this section we presented the details of the scenarios that we used to evaluate the performance of DWL. For ease of reference we summarize these scenarios in Table 6.1. In the next section we analyze the experimental results obtained from these scenarios to address the evaluation objectives outlined in Section 6.1.

6.4 Results and Analysis

In this section we analyze the performance of DWL with respect to the evaluation objectives outlined in Section 6.1. We evaluate the performance of DWL when compared to the performance of baselines, we compare the performance of single-policy deployments to multi-policy deployments in the presence of multiple vehicle types, compare non-collaborative (C=0) deployments to collaborative DWL deployments, DWL deployments with predefined values of C to DWL deployments with learnt values of C, we evaluate DWL performance for conflicting policies, assess DWL's ability to respect

Scenario number	Name	Description
1	Uniform traffic	Uniform vehicle insertion rates at all points on the map Two loads: low (35k vehicles), high (60k vehicles)
2	3:1 traffic pattern	Ratio of traffic travelling N → S to S → N is 3:1 Single traffic load: 60k vehicles
3	Conflicting traffic	Buses do not travel on same routes as cars Single traffic load: 35k vehicles
4	Reversed priority	3 deployments: buses have a higher priority than cars, both vehicle types have same priority, and cars have a higher priority than buses
5	Measure learning time	9 deployments: learning (exploration) duration of 50, 100, 200, 300, 400, 500, 600, 700, 750 minutes

Table 6.1: Summary of DWL evaluation scenarios

	Uniform Low Load		Uniform High Load		3:1 Traffic	
	Car	Bus	Car	Bus	Car	Bus
RR	77.38%	80.22%	86.36%	84.91%	76.42%	81.88%
SAT	83.11%	85.00%	87.20%	87.89%	73.00%	79.34%

Table 6.2: DWL vs. baselines: Maximum waiting time improvement

policy priorities, evaluate the feasibility of the training duration required by DWL, and analyze the ability of DWL to learn policy and agent dependencies. As our analysis draws on multiple scenarios, in each section we state which scenarios contribute to the results presented in that section, describe the results, and discuss how they relate to the evaluation objectives.

6.4.1 DWL vs. Baselines

In this section we compare the performance of DWL against the UTC baselines, RR and SAT. DWL results presented in this section are the results obtained from DWL deployments with learnt C, as they were the best performing deployments of DWL. We present results for three sets of traffic conditions: the uniform traffic pattern at low and high loads, and the 3:1 traffic pattern, i.e., Scenarios 1 and 2.

Figure 6.3 shows average waiting time per vehicle type per experiment, for DWL and both baselines. We see from both graphs that DWL clearly outperforms both baselines in terms of average vehicle waiting time under all tested traffic conditions.

To emphasize the extent of improvement, in Table 6.2 we show how much shorter, in percentage

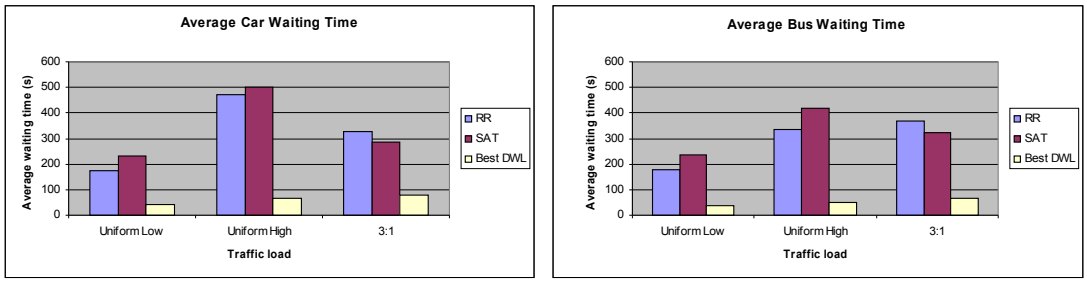


Fig. 6.3: DWL vs. baselines: Vehicle waiting time

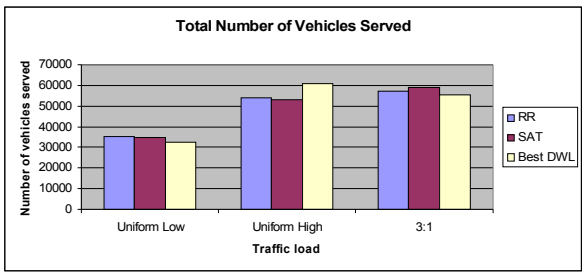


Fig. 6.4: DWL vs. baselines: Number of vehicles served

terms, are average vehicle waiting times in DWL when compared to RR and SAT. The improvements range from ∼73% (which is how much average car waiting time is shortened compared to SAT under 3:1 traffic conditions), to ∼88% (which is the improvement of average bus waiting time compared to SAT during high uniform traffic load). These improvements are statisticallty significant with all p-values being lower than 9×10^{-14}.

We also compare DWL to baselines in terms of the total number of vehicles served and the traffic density. Note that these two metrics are not measured per vehicle type, but for the system as a whole, and as such mostly reflect the performance of cars, as they make up 95% of the traffic in our simulation setup.

In terms of the total number of vehicles served (see Figure 6.4), under uniform high traffic load, DWL serves ∼14% more vehicles that SAT and ∼13% more vehicles than RR, however, under uniform low load it serves ∼7% and ∼8% less vehicles than SAT and RR, respectively, and under the 3:1 traffic pattern ∼7% and ∼6% less vehicles than SAT and RR, respectively. We believe that the drop in number of vehicles served by DWL under certain traffic conditions is caused by DWL prioritizing buses resulting in a small negative effect on other vehicles joining the system. Note that vehicles attempting to join the system are not placed in a queue for joining at a later time, but are discarded if they cannot join the system at the time and at the junction at which they are scheduled. If an

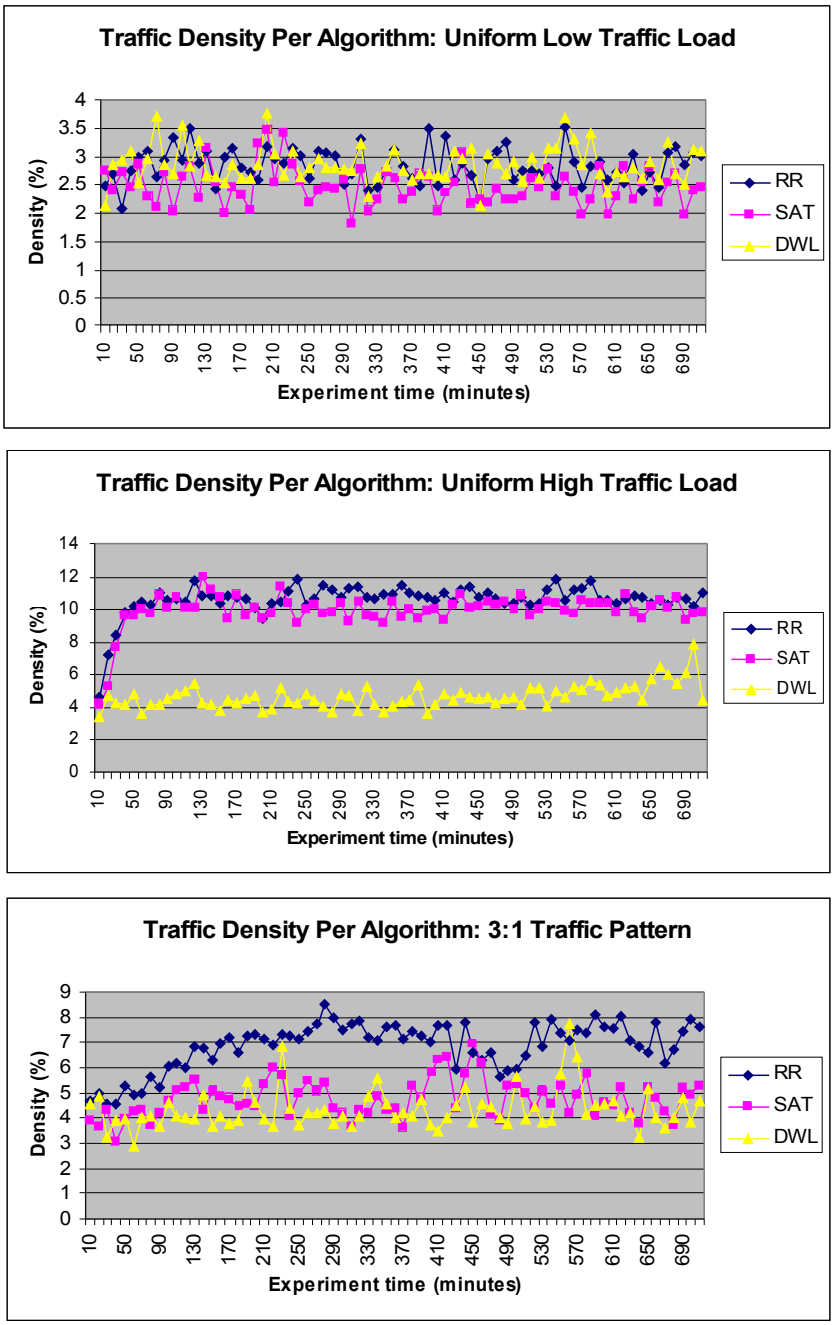


Fig. 6.5: DWL vs. baselines: Traffic density

approach is congested at that particular time, due to DWL serving other approaches with higher priority vehicles, i.e., buses, the vehicles cannot join the system at that approach, resulting in a lower number of vehicles being served. We confirm this when analyzing the performance of DWL in systems with a single policy only (in Section 6.4.3), where a single-policy DWL deployment that addresses only cars serves 3% - 20% more vehicles than baselines under all traffic scenarios, as there is no negative effect of bus prioritization.

In terms of overall traffic density (as shown in Figure 6.5), under uniform low load SAT results in traffic density that is ∼13% lower than that of DWL, however, under all other traffic conditions DWL has a lower density than both of the baselines, with the largest improvement observed under uniform high load, where DWL density is ∼53% lower than in SAT, and ∼56% lower than in RR.

Although it was not our primary goal when performing this set of experiments, it is also interesting to observe the relative performance of the baselines to each other. In terms of vehicle waiting time, SAT performs worse than RR during uniform low load and uniform high load for both vehicle types, but outperforms RR in terms of waiting time for both vehicle types under the 3:1 traffic pattern. SAT results in lower density during all three sets of traffic conditions and serves more vehicles than RR during the 3:1 traffic pattern, while RR serves more vehicles during uniform traffic loads. We therefore observe that SAT performs better than RR in terms of all three metrics used under the 3:1 traffic pattern, emphasizing the need for adaptive control strategies under non-uniform traffic conditions.

Overall, we conclude that DWL is a suitable approach to optimization in UTC, with improvements over the baselines in terms of vehicle waiting time reaching nearly 90%. DWL shortens both bus and car waiting times, under different traffic loads, and under different traffic patterns.

6.4.2 Collaborative vs. Non-Collaborative Deployments: Impact of Collaboration

In this section we evaluate the impact of collaboration, as implemented in DWL, on multi-policy optimization. We draw on the results from evaluation Scenarios 1 and 2, addressing DWL's performance under uniform low load, uniform high load, and the 3:1 traffic pattern.

We evaluate the impact of DWL's capability to enable collaboration, i.e., the impact of remote policies, by comparing collaborative scenarios to non-collaborative (C=0) scenarios, which use the equivalent of basic W-learning as presented in Chapter 2. We also evaluate the impact of DWL's ability to allow agents to engage in different levels of collaboration ($0 < C < 1$) rather than necessarily being fully collaborative (C=1), i.e., the impact of the presence of collaboration coefficient C, by comparing the results obtained with the best performing C (where $0 < C < 1$) with those obtained

when C=1. Finally, we evaluate the impact of DWL's capability to learn C per agent, by comparing the performance of DWL with learnt C, with the performance of DWL with the best performing predefined value of C.

The best performing predefined value of C was determined experimentally by running the simulation of DWL with C = {0, 0.2, 0.4, 0.6, 0.8, 1} for each set of traffic conditions. Under uniform low load the best performing predefined C was determined to be C=0.2, for uniform high load C=0.2, and for 3:1 traffic pattern C=0.4.

6.4.2.1 Collaborative vs. Non-Collaborative Deployments

Figure 6.6 shows average vehicle waiting time per vehicle type for DWL with C=0, C=1, and with the best performing value of C (in graphs: "Best DWL"). It is clear that DWL deployments result in by far the best performance under all tested traffic conditions, for both cars and buses. Therefore, we observe that agents can benefit from engaging in cooperation to a degree (i.e., that C should not be 0), but that collaboration should not result in agents taking into account their neighbours' action suggestions with as much weight as their own (i.e., that C should not be 1) as that can have negative effects on performance. In Table 6.3 we summarize the improvements in average vehicle waiting time gained by the addition of collaboration over the performance of non-collaborative deployments, and observe that collaboration can shorten the average vehicle waiting time by up to ∼90%, depending on the vehicle type and the traffic scenario. The improvements are statistically significant, with all p-values smaller than 2×10^{-11}.

Results in terms of the number of vehicles served (as shown in Figure 6.7) and traffic density (as shown in Figure 6.8) confirm this observation. DWL deployed with the best performing value of C serves more vehicles than DWL deployed with either C=0 and C=1, and has lower average density for all three sets of traffic conditions.

When considering the density results, it is also interesting to compare traffic density when C=0 with collaborative DWL. We see that in all three graphs shown in Figure 6.8, the density of the non-collaborative deployment has multiple peaks, i.e., periods of high density, during system operation. We believe that, as when C=0 all agents act independently and do not help each other, poor performance of individual junctions, which is not corrected with help from neighbours, has negative effects on the performance of the whole system, temporarily increasing congestion in the neighbouring areas until it clears its local congestion. We see that both fully collaborative deployments and those using the best performing value of C result in a much smoother and more uniform traffic density during the whole simulation period.

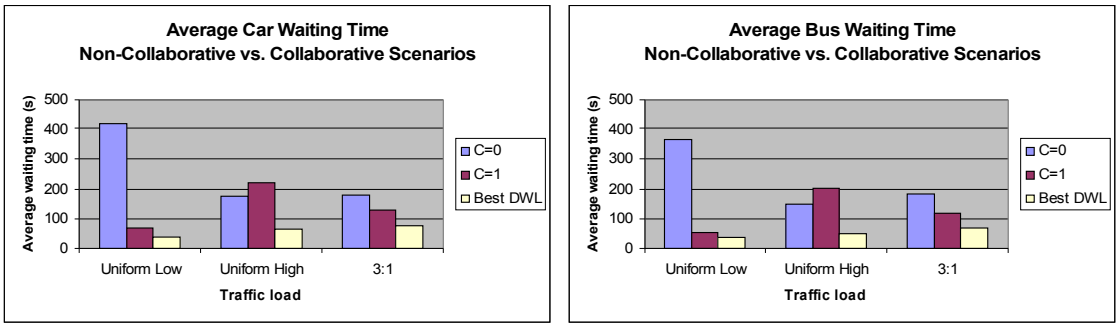


Fig. 6.6: Collaborative vs. non-collaborative scenarios: Vehicle waiting time

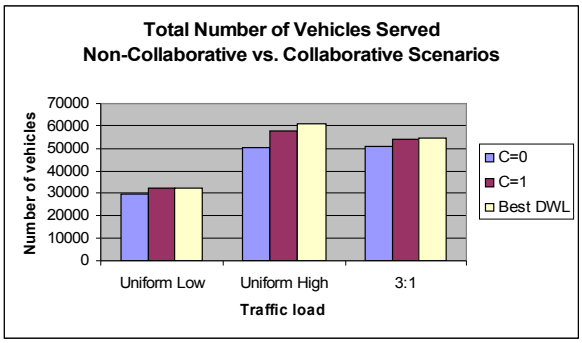


Fig. 6.7: Collaborative vs. non-collaborative scenarios: Number of vehicles served

From the results above we conclude that DWL's ability to enable collaboration between agents using remote policies improves system performance, however that collaboration should not be full, i.e., C=1, but scaled to a degree using a cooperation coefficient C.

6.4.2.2 Collaboration Using Predefined vs. Learnt Values of C

We now consider how DWL's ability to learn C per agent, rather than using a predefined value of C for the whole system, affects system performance. In terms of average vehicle waiting time (shown in Figure 6.9), DWL with learnt value of C can outperform the DWL deployment with the best predefined value of C under all traffic conditions for both vehicle types, apart from car waiting time under the 3:1 traffic pattern. A summary of the differences is shown in Table 6.4 where we see that learning C achieves as high as ∼60% improvement over the best predefined value of C. These improvements are statistically significant for both vehicle types under high load ($p = 0.02$) and for buses under low load ($p = 0.01$), but not statistically significant for cars under low load ($p = 0.1$). In the 3:1 traffic pattern

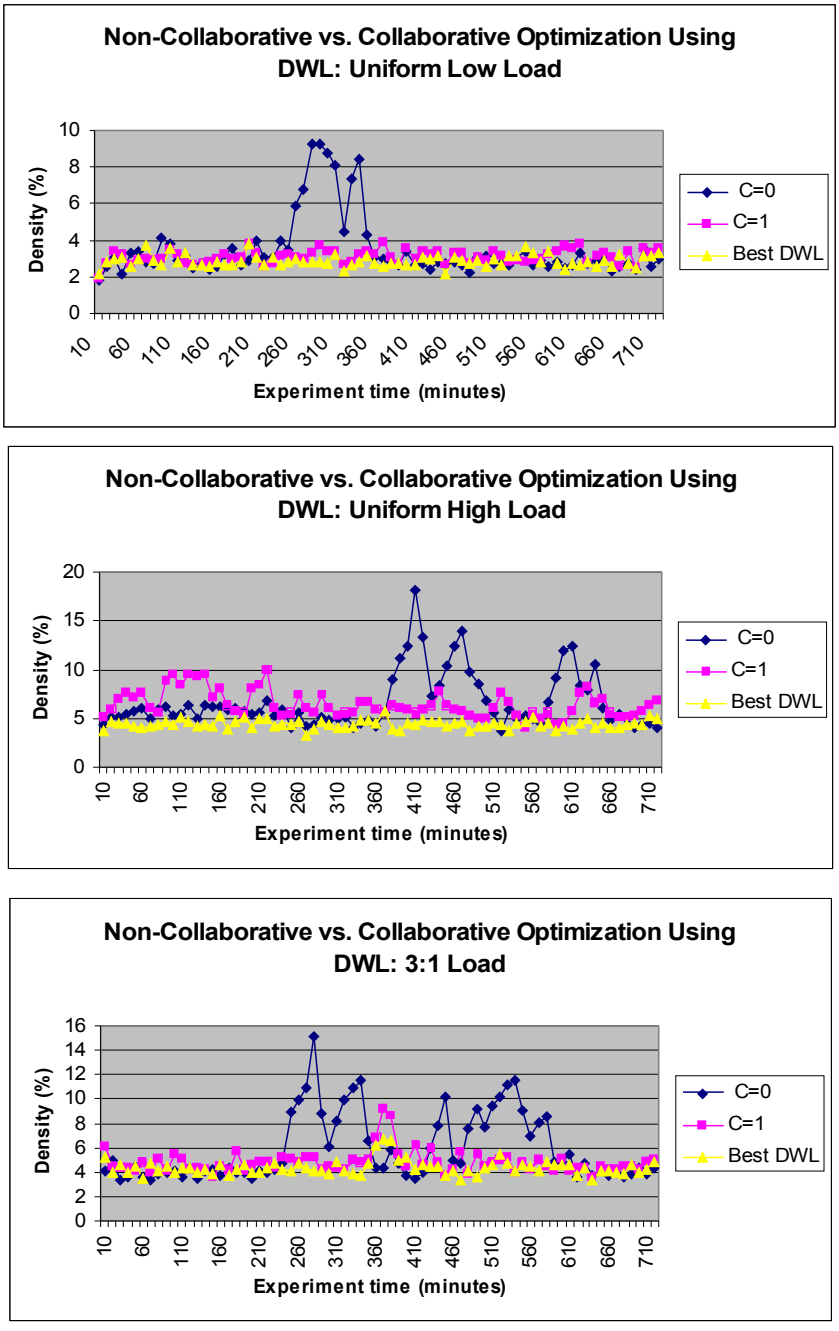


Fig. 6.8: Collaborative vs. non-collaborative scenarios: Density

	Car	Bus
Uniform Low	90.62%	90.38%
Uniform High	63.73%	66.21%
3:1 Load	56.45%	62.69%

Table 6.3: Waiting time improvement in collaborative scenarios over non-collaborative scenarios

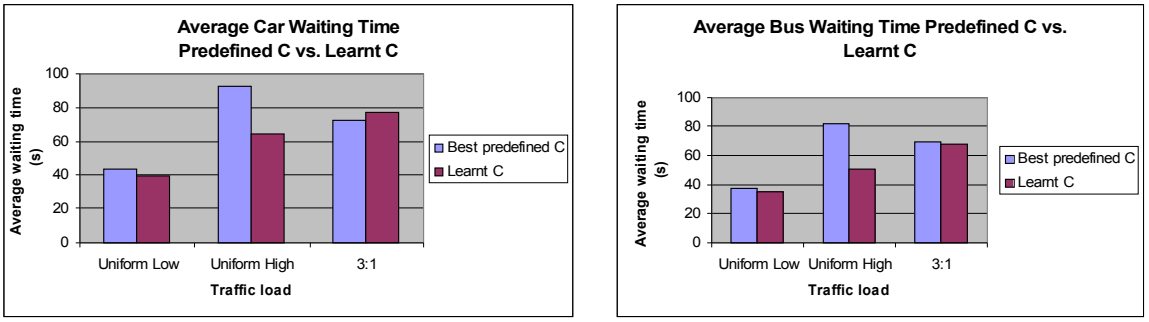


Fig. 6.9: Predefined C vs. learnt C: Vehicle waiting time

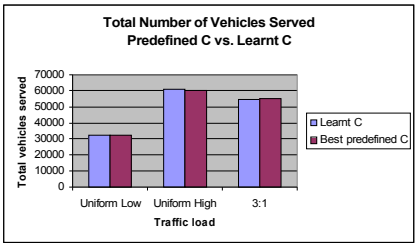


Fig. 6.10: Predefined C vs. learnt C: Number of vehicles served

	Car	Bus
Uniform Low	10.77%	6.82%
Uniform High	44.25%	61.49%
3:1 Load	−6.64%	2.77%

Table 6.4: Maximum waiting time improvement of DWL with predefined C over learnt C

average car waiting time is ∼7% worse when C is learnt rather than when C is predefined, while bus waiting time is simultaneously improved by ∼3%. However, neither of these differences is statistically significant (p = 0.6 for cars and p = 0.8 for buses).

In terms of the total number of vehicles served, both DWL with the best performing predefined C and learnt C have very similar performance (as shown in Figure 6.10), showing no major advantage or disadvantage of learning C over using a predefined C.

We also compare the performance of deployments using the best predefined C to those using learnt C in terms of density. Figure 6.11 shows density with the best predefined C under uniform low load (C=0.2) when compared to learnt C, density with the best predefined C under uniform high load (C=0.2) when compared to learnt C, and density for the best predefined C for 3:1 pattern (C=0.4) when compared to learnt C. Densities are similar in terms of their average values, but we observe that the performance of the DWL deployment with a predefined C shows peaks in traffic density, similar to the behaviour we observed in the C=0 deployment. These peaks are much less extreme than the peaks observed when C=0; peaks in C=0 deployment correspond to up to a 10-12% increase in density, while those observed with predefined C result in only up to a 4% increase. Nevertheless, we observe that learnt C results in the steadiest traffic density throughout the experiments, with density fluctuations from minimum to maximum being only ∼2%.

From the vehicle waiting time and density results presented above, we conclude that DWL's ability to learn values of C per agent results in as good as or significantly improved performance comparing to using the best predefined value of C. An additional advantage of learning C over using a predefined C is that the best performing predefined C can only be determined experimentally, by performing simulations with various values of C, while in DWL with learnt C agents can learn suitable individual values themselves during a single run. This can significantly shorten DWL training time, by removing the need to evaluate DWL performance with numerous values of C in order to identify the best performing value.

Overall, based on the results presented in this section, we see a significant improvement in global behaviour emerging from agent collaboration, as performance of the policies is improved by up to ∼90% when compared to each agent only optimizing its local performance towards its local policies. The improvements are a result of DWL's use of remote policies as a means of collaboration and DWL's ability to engage in variable levels of collaboration, where the appropriate levels of collaboration can be learnt by agents themselves.

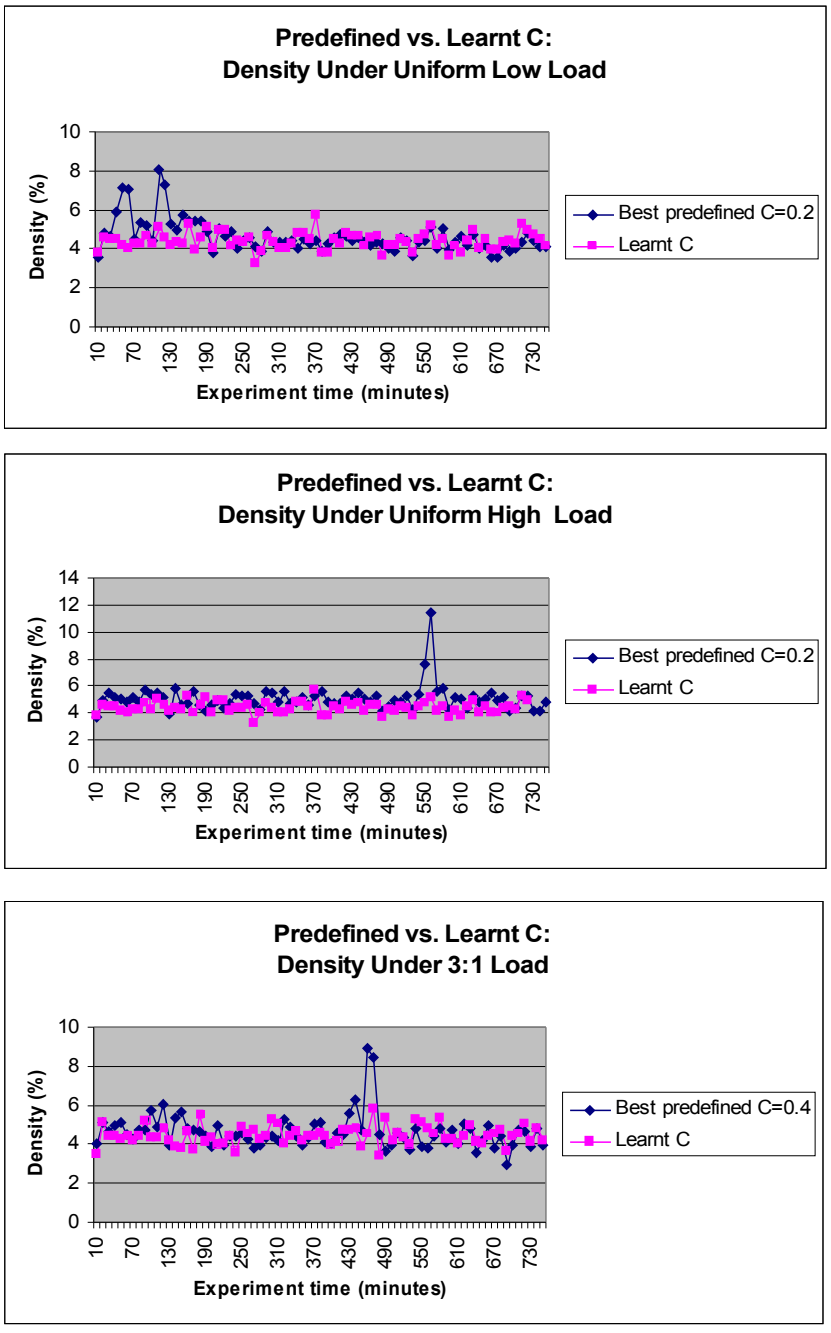


Fig. 6.11: Predefined C vs. learnt C: Traffic density

6.4.3 DWL for Single-Policy Optimization

In this section we evaluate the performance of DWL for single-policy optimization. DWL is primarily designed to improve system performance in the presence of multiple policies, however, we verify that DWL can be used even in circumstances where only a single policy is present in the system, by taking advantage of DWL's agent collaboration capabilities. We compare the performance of single-policy DWL with baselines and compare collaborative scenarios with non-collaborative scenarios. Results presented in this section are obtained from single-policy GWO deployments as part of Scenarios 1 and 2, i.e., under uniform low traffic load, uniform high load, and 3:1 traffic pattern.

Figure 6.12 shows the average vehicle waiting time and total number of vehicles served for RR, SAT, single-policy DWL with C=0 (equivalent to basic W-learning), fully collaborative single-policy DWL with C=1, and collaborative single-policy DWL with a learnt value of C, for all three sets of traffic conditions. In terms of waiting time we see that single-policy DWL with a learnt value of C outperforms both baselines, and both fully collaborative and non-collaborative deployments, for all sets of traffic conditions evaluated. These improvements are statistically significant, with all values of $p < 0.02$, apart from the difference between the performance of learnt C and C=0 under low traffic load, which is not statistically significant, with $p = 0.2$. In terms of the number of vehicles served, single-policy DWL outperforms the baselines and non-collaborative deployment, as well as the C=0 and C=1 deployment under high uniform load and 3:1 traffic pattern, while under uniform low load the C=1 deployment serves slightly more vehicles (<0.5%) than DWL with a learnt value of C.

When we have compared the total number of vehicles served by multi-policy DWL and the baselines (in Section 6.4.1), we observed that under certain traffic conditions multi-policy DWL served less vehicles than the baselines, and concluded that this is caused by bus priority having negative effects on throughput of cars. Our observations on single-policy DWL presented here confirm this, as GWO outperforms both baselines in terms of total number of vehicles served, as it addresses only cars and there is no negative effect of bus priority observed.

In terms of density, as shown in Figure 6.13, the best performing DWL, which is DWL with a learnt value of C, has a lower density than both baselines, under all three sets of traffic conditions. Deployments with C=0 and C=1 result in a similar average density to the best performing DWL, however, as also observed in the previous section, the C=0 deployment results in periods of very high density during the system operation, as represented by peaks in the graph. Introducing collaboration eliminates those peaks, and results in more even density throughout.

From the results above we conclude that DWL's collaboration capabilities, as implemented by the use of remote policies, and learning of the cooperation coefficient, can be used to improve performance

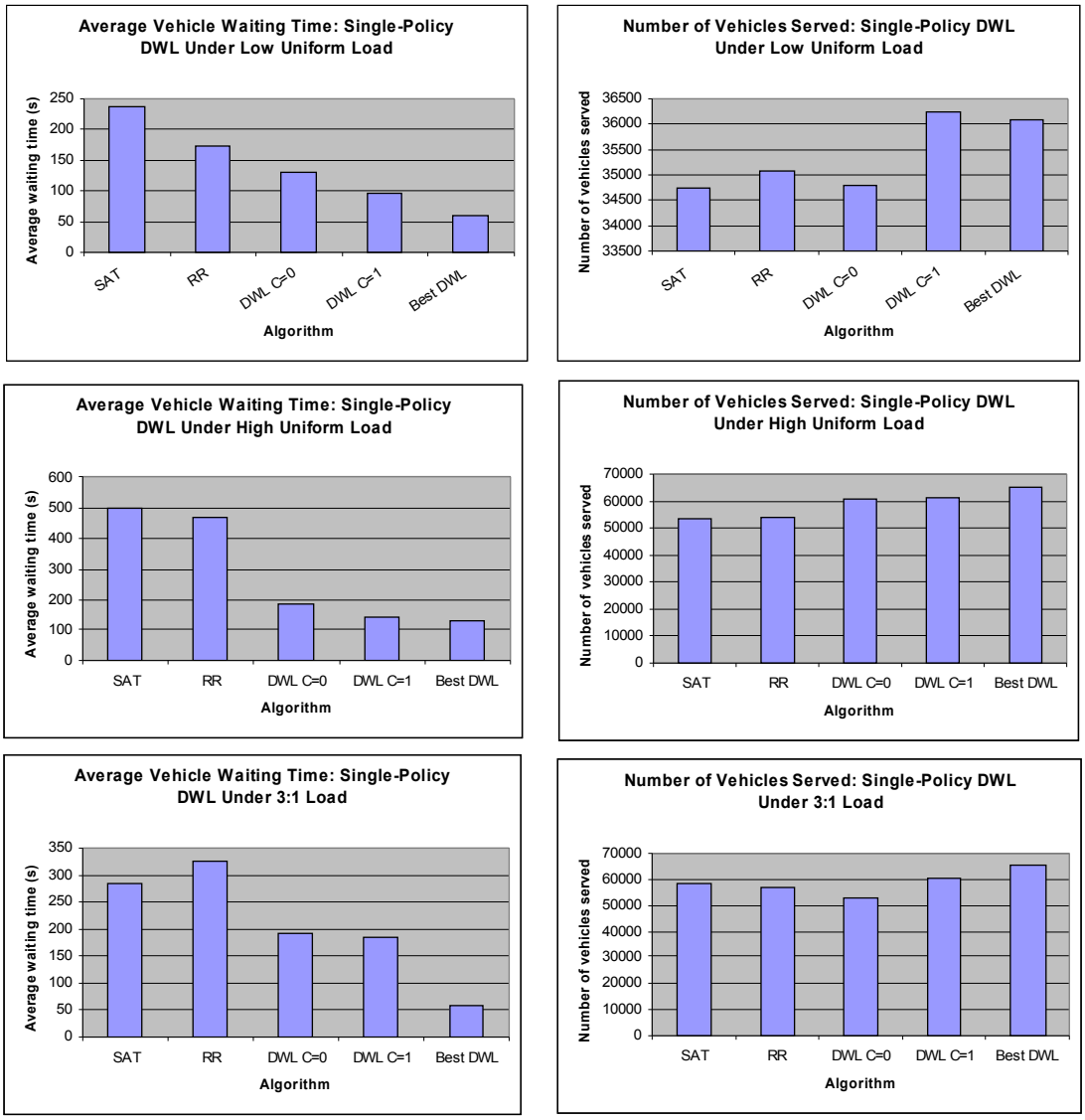
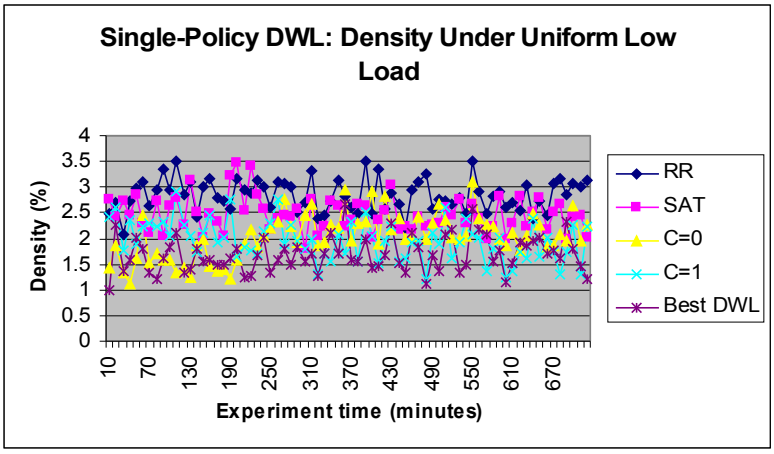
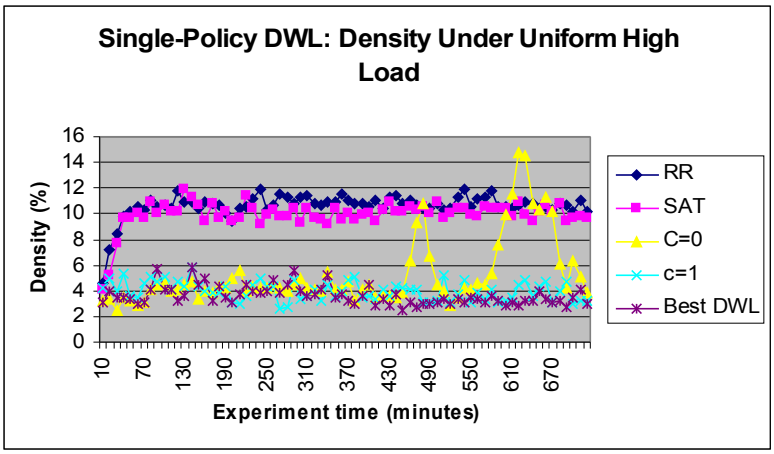


Fig. 6.12: Single policy DWL: Vehicle waiting time and number of vehicles served





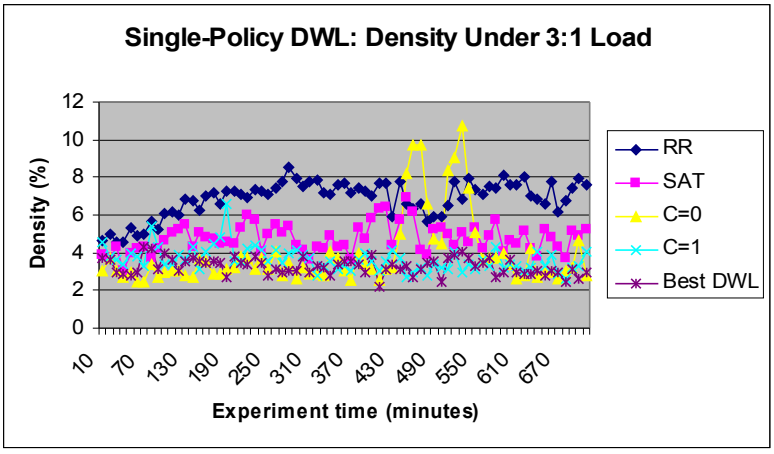


Fig. 6.13: Single policy DWL: Traffic density

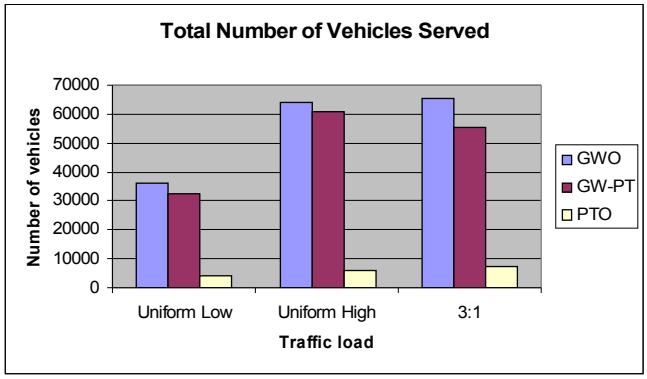


Fig. 6.14: Single-policy vs. multi-policy DWL: Number of vehicles served

of single-policy systems, and not only multi-policy systems as already shown in Section 6.4.2.

6.4.4 Single-Policy vs. Multi-Policy DWL Deployments

In this section we evaluate how the performance of a single policy deployed in the system is affected when another policy, addressing a different vehicle type, is added. For this analysis we draw on Scenarios 1 and 2, using single-policy GWO and PTO deployments, as well as a multi-policy deployment that addresses both of these policies simultaneously, GW-PT.

6.4.4.1 PTO vs. GW-PT

We first analyze the performance of PTO, the policy that addresses only buses, and compare it to the performance of GW-PT, i.e., analyze how buses are affected when an additional policy that addresses cars is added.

Figure 6.14 shows the total number of vehicles served by PTO, GWO and GW-PT deployments using DWL. We can observe that PTO, under all three sets of traffic conditions serves only a small fraction of the vehicles served by GW-PT.

Density results for GWO, PTO, and GW-PT (as shown in Figure 6.15), are consistent with the number of vehicles served. PTO, under all three sets of traffic conditions, has much larger density than GW-PT, which starts to increase early in the experiment, eventually over-saturating the system. Over-saturation prevents new vehicles joining the system, resulting in very small numbers of vehicles served.

This behaviour of PTO is similar to the behaviour of single-policy EVO as discussed in our preliminary case study in Chapter 3. PTO prioritizes public transport vehicles, which represent only 5%

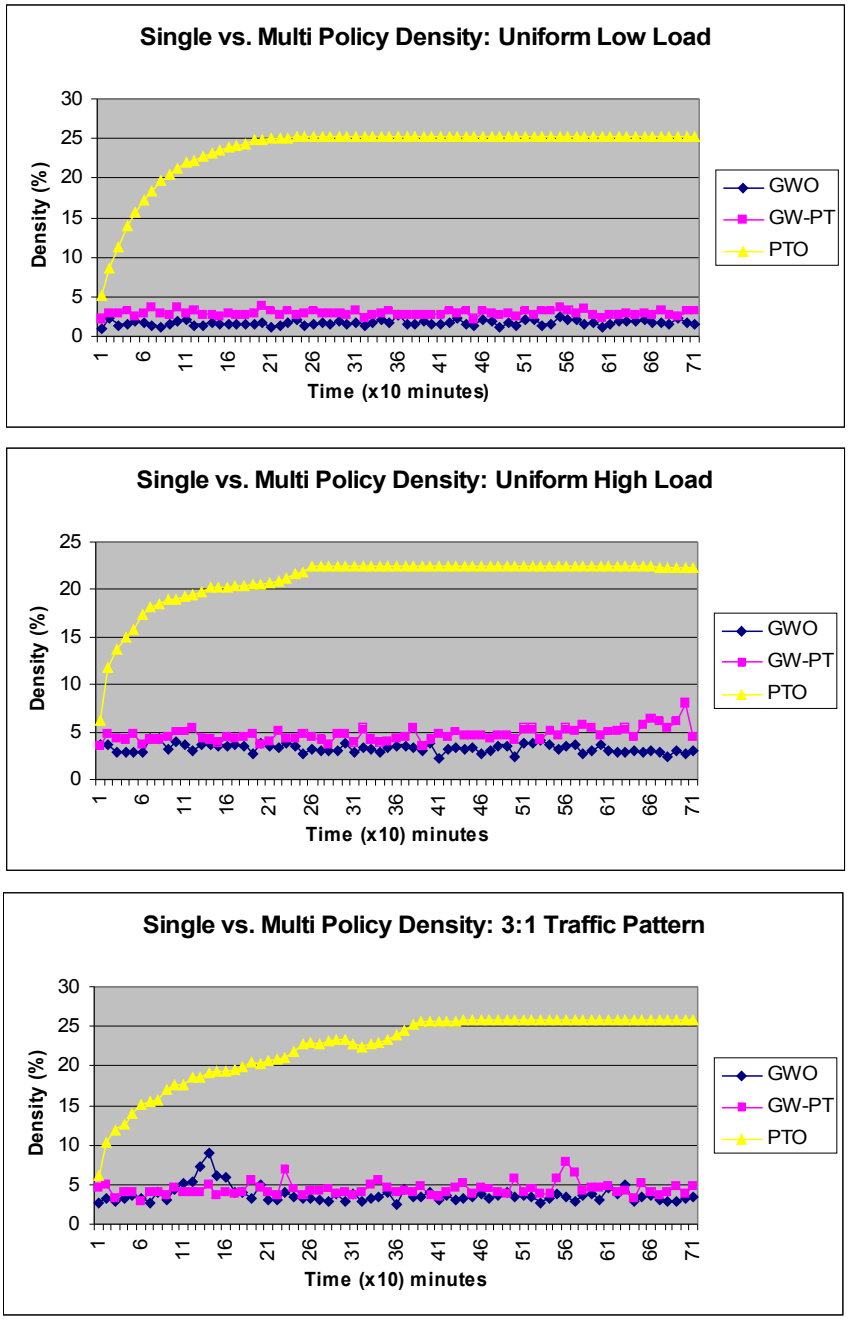


Fig. 6.15: Single-policy vs. multi-policy DWL: Traffic density

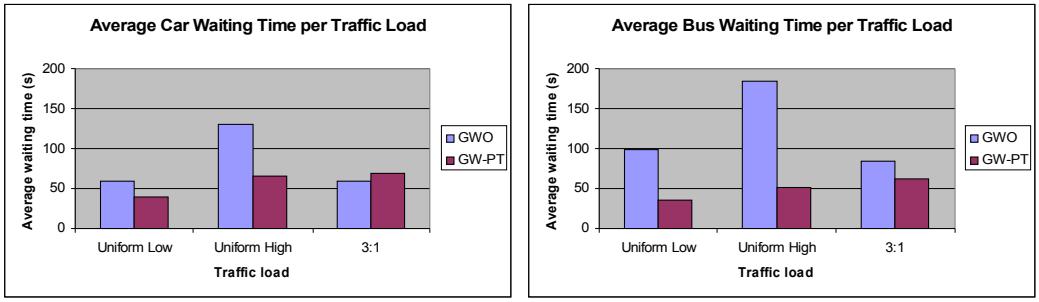


Fig. 6.16: Single-policy vs. multi-policy DWL: Vehicle waiting time

	Car	Bus
Uniform Low	−33.06%	−64.12%
Uniform High	−50.57%	−72.36%
3:1 Load	+16.03%	−28.55

Table 6.5: Average vehicle waiting time: Difference between GW-PT and GWO

of total traffic, so attempting to optimize only their performance negatively affects the performance of the other 95% of vehicles. Other vehicles that are not adequately addressed then create a backlog in the system, in turn affecting the performance of public transport vehicles themselves, as, given the shared infrastructure, there is no available road space for them to proceed, or road space for new vehicles to join. Due to the very low numbers of vehicles served by PTO, its waiting time results are not comparable to waiting time results of GWO and GW-PT, and we therefore do not address them in this section.

We conclude that optimizing for only a single policy with a very small temporal scope and regional scope that addresses a very small percentage of vehicles, such as PTO, is not feasible due to the shared infrastructure and operating environment creating policy dependencies. In such situations, adding an additional policy that addresses the remaining vehicles is beneficial for both policies, i.e, for the system overall, as confirmed by GW-PT serving significantly more vehicles than PTO and resulting in much lower traffic density.

6.4.4.2 GWO vs. GW-PT

We now compare the performance of GWO, the policy that addresses only cars, to the performance of GW-PT, i.e., analyze how are cars affected when an additional policy that addresses buses is added.

Figure 6.16 compares average vehicle waiting times, per vehicle type, per set of traffic conditions, for GWO and GW-PT. The differences, in percentage terms, between the average vehicle waiting times achieved in GWO and in GW-PT are listed in Table 6.5. We observe that GW-PT outperforms GWO under all sets of traffic conditions in terms of bus waiting time. This result is expected as GW-PT prioritizes public transport vehicles, while GWO treats all vehicles the same. However, under two sets of traffic conditions (uniform low and uniform high load), GW-PT also outperforms GWO in terms of car waiting time. When a new policy is added to the system (PTO), we expected the performance of the original policy (GWO) to slightly deteriorate, as the new policy prioritizes a different vehicle type from the one addressed by GWO. However, there was no significant deterioration of performance of either of the vehicle types under any of the traffic conditions (all p-values were greater than 0.8), while the average waiting time of buses has been significantly improved under low traffic load ($p = 0.02$). We believe this is a result of policy dependency, where the two policies are complementary and contribute towards the performance of each other. Therefore, optimizing for both policies simultaneously results in as good as (or under certain traffic conditions) improved average vehicle waiting time for both vehicle types, when compared to optimizing for a single policy only.

In terms of total number of vehicles served (as shown in Figure 6.14) and traffic density (as shown in Figure 6.15) however, we observe slight negative effects for all three sets of traffic conditions of addition of a second policy. GW-PT serves less vehicles than GWO, and results in larger traffic density than GWO, under both low and high uniform load, and under 3:1 traffic pattern.

6.4.4.3 Single-Policy vs. Multi-Policy Summary

Overall, from the results presented in this section, we conclude that the addition of a second policy to the system, in the presence of multiple vehicle types, can result in either a positive impact on the performance of both the newly added and the original policy, or result in no deterioration to the performance of the original policy with the improvement to the performance of the vehicle type that newly added policy is addressing.

We expected that addition of a second policy would have a small negative impact on the performance of the original policy, due to the system having to address the new policy as well. However, under certain circumstances, the addition of the second policy, due to policy dependency, resulted not just in improvements to the performance of the vehicle type addressed by the new policy, but also in improvements in the performance of the original policy. This improvement was most evident when comparing the performance of PTO and GW-PT, where PTO performed very badly as it addressed only 5% of vehicles, and the introduction of a second policy that addresses the remaining 95% of

vehicles significantly improved the performance of both vehicle types. Additionally, we have observed small improvements in the car waiting time in GW-PT when compared to single-policy GWO under certain sets of traffic conditions, when prioritization of buses also helped improve the waiting time of cars.

In terms of the performance of the system as a whole, if there are two vehicle types present in the system, addressing both simultaneously using DWL results in the best overall system performance. Single-policy deployments either perform well for one vehicle type, but neglect the other vehicle type that they are not addressing (e.g., GWO), or perform poorly for both vehicle types, as neglecting one vehicle type causes negative effects on the performance of the other vehicle type (e.g., PTO). Multi-policy deployments either improve the performance of both vehicle types over corresponding single-policy deployments (e.g., GWO vs. GW-PT under uniform patterns), or improve the performance of one vehicle type with smaller negative effects on the other (e.g., GWO vs. GW-PT under 3:1 pattern).

6.4.5 DWL in the Presence of Conflicting Policies

In this section we assess the ability of DWL to address conflicting policies by analyzing the results from the evaluation Scenario 3. We assess the performance of DWL in this conflicting-policy scenario when compared to baselines, evaluate the impact of the addition of a second policy (i.e., compare single-policy to multi-policy DWL deployments), and compare non-collaborative with collaborative DWL deployments.

From Figure 6.17, showing average waiting time for both vehicle types, for RR, SAT, GWO, and DWL, we see that DWL outperforms both baselines, with decreases in waiting time in the range of ∼75-83%. The differences in performance are statistically significant with all p-values under 8×10^{-9}. GW-PT outperforms single-policy GWO in terms of bus waiting time by ∼126% (a statistically-significant improvement with $p = 0.008$), with ∼0.2% increase in car waiting time (a not statistically-significant increase with $p = 0.4$). We have observed this behaviour in the 3:1 traffic pattern as well; when the second policy was added to the system, car waiting time was slightly increased in order to allow a significant improvement in bus waiting time.

In terms of total number of vehicles served, as shown in Figure 6.18, single-policy GWO serves the highest number of vehicles, followed by the baselines, and followed by multi-policy GW-PT. PTO serves only a small fraction of vehicles served by the other deployments. This is consistent with observations on total number of vehicles served in Scenarios 1 and 2, and observations on the poor performance of PTO when deployed as a single-policy approach, due to creating a backlog of cars.

Density graphs for a conflicting scenario are consistent with the observations on total number of

vehicles served. Figure 6.19 shows traffic density, first for all deployments (RR, SAT, GWO, PTO, and DWL), and then the graph with PTO removed to allow for closer inspection of density for other deployments. GWO has the lowest density, being the most efficient in clearing the general traffic as it does not prioritize buses, followed by GW-PT, and the baselines. PTO, again, has by far the worst performance, due to the backlog of cars created in the system.

Therefore, multi-policy DWL (GW-PT) outperforms baselines under the conflicting-traffic scenario as well, and outperforms corresponding single-policy deployments in the presence of both vehicle types. When only GWO is present in the system and PTO is added, cars suffer a very small decrease in performance, but the performance of buses is significantly improved. When only PTO is present in the system and GWO is added, the performance of both vehicle types is significantly improved, as all vehicles in the system are addressed and there is no congestion created by the neglected vehicles.

We now compare the performance of non-collaborative multi-policy deployments to collaborative DWL deployments under conflicting traffic conditions. Figure 6.20 graphs vehicle waiting times for non-collaborative deployments, for fully collaborative deployments, for DWL deployed with the best predefined value of C, and DWL deployment with learnt value of C. We observe that C with a value between 0 and 1 results in shorter average waiting times than both fully collaborative and fully non-collaborative scenarios, as was also observed for other traffic patterns. Learning the value of C provides a small improvement in bus waiting time ($\sim$2%), while increasing car waiting time by $\sim$4%, however, neither of the differences in performance are statistically significant ($p = 0.6$ and $p = 0.2$, respectively). Nevertheless, we conclude that learning values of C is a preferred approach, as it performs on a par with the best predefined value of C, without requiring extensive training periods to establish the most suitable values of C. A similar relationship between the predefined and learnt values of C was also observed under the 3:1 traffic pattern, where learning C slightly improved the performance of buses with slight negative effects on the performance of cars, as buses have a higher priority in the system.

From the similar performance of DWL in a conflicting scenario and in the 3:1 traffic scenario, we observe that both scenarios create similar levels of conflict within the system. In the conflicting-policy Scenario 3, buses use different routes than cars, so they approach junctions from different directions than cars, while in the 3:1 traffic pattern (Scenario 2), 75% of the overall traffic approaches junctions from different directions than the remaining 25% of traffic. As observed, in such conflicting scenarios, DWL can make trade-offs in the performance of different vehicle types that respect policy priorities.

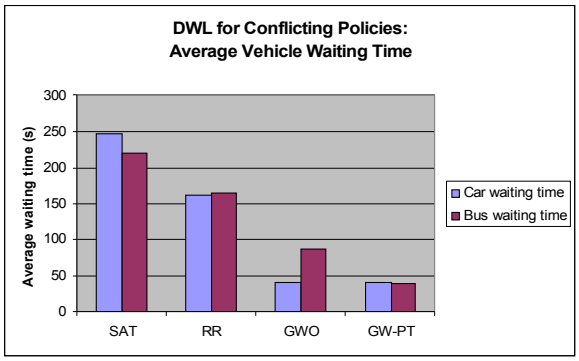


Fig. 6.17: DWL for conflicting policies: Vehicle waiting time

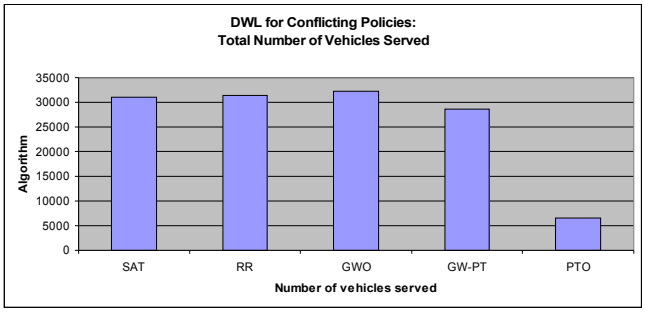


Fig. 6.18: DWL for conflicting policies: Number of vehicles served

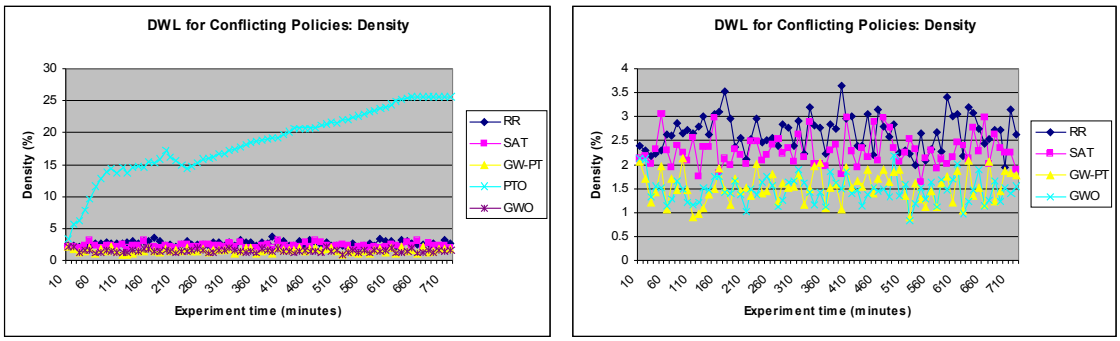


Fig. 6.19: DWL for conflicting policies: Traffic density

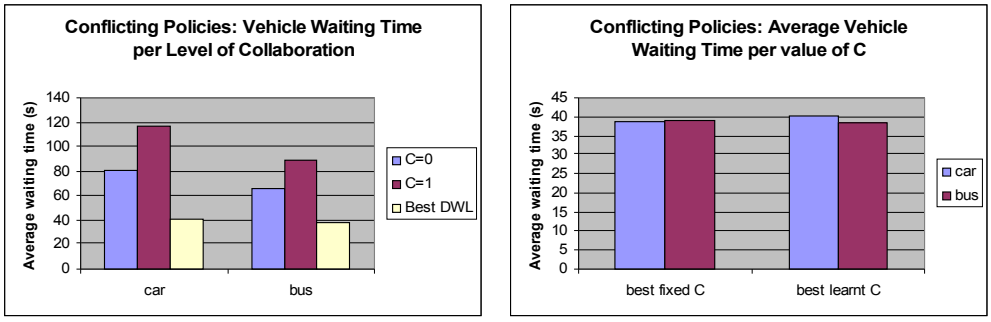


Fig. 6.20: DWL for conflicting policies: Impact of collaboration

6.4.6 Policy Priority in DWL

In this section we focus on the ability of DWL to respect policy priorities. From the analysis of DWL performance when compared to baselines (Section 6.4.1) and to non-collaborative multi-policy deployments (Section 6.4.2), as well as the comparison of DWL deployments with predefined values of C and learnt values of C (Section 6.4.2.2), we observe that in most of these cases DWL improves the performance of both policies, i.e., even of the policy with a lower priority. We observe two situations in which sacrifices in the performance of one policy had to be made in order to improve the performance of the other policy: when comparing deployments with a predefined value of C against deployments with a learnt value of C under the 3:1 traffic load and in presence of conflicting policies. In these two scenarios, the waiting time of cars, which are addressed by a lower priority policy, was slightly increased in order to shorten the waiting time of buses, which are addressed by a higher priority policy. Under no traffic conditions and under no DWL deployment in this set of experiments were sacrifices made in the performance of a higher priority policy.

For the analysis of policy priority in DWL we also use experimental results obtained from Scenario 4, where we vary the relative priority of policies and observe their performance. As described in Section 6.3.4.4, experiments were performed with three combinations of rewards: GWO 100/PTO 120 (i.e., buses are given a higher priority), GWO 100/PTO 100 (i.e., cars and buses have equal priority), and GWO120/PTO 100 (i.e., cars are given a higher priority). The results in terms of waiting time for DWL with the best performing predefined C and DWL with learnt C are shown in Figure 6.21.

From the graphs we can observe that, as the relationships between car and bus priority changes from buses having a higher priority, to both vehicles having the same priority, to cars having a higher priority, the waiting time of buses increases and that of cars decreases. We note that buses still have smaller waiting times even when cars have a higher priority. However, as the average lengths of routes for different vehicle types differ, we cannot compare the absolute values of vehicle waiting times, but

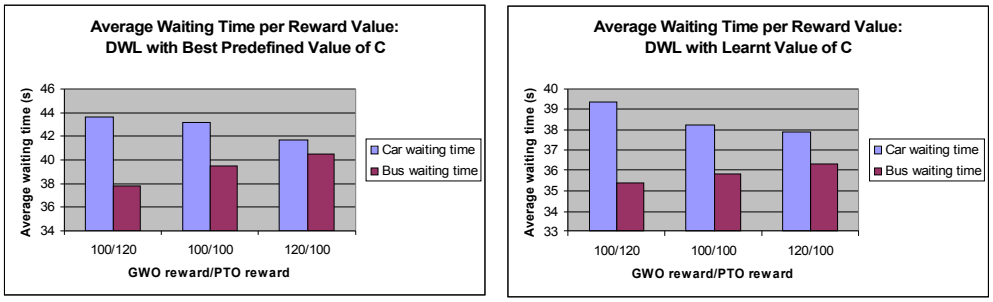


Fig. 6.21: DWL with varying reward values: Car and bus waiting time

	Best predefined C			Learnt C		
	Car	Bus	Difference	Car	Bus	Difference
GWO 100/PTO 120	43.60s	37.82s	−15.26%	39.36s	35.41s	−11.15%
GWO 100/PTO 100	43.14s	39.50s	−9.24%	38.25s	35.86s	−6.69%
GWO 120/PTO 100	41.73s	40.44s	−3.18%	37.91s	36.31s	−4.40%

Table 6.6: DWL with varying reward values: Difference between car and bus average waiting time

we compare the relative difference between their values at different relative priorities. We can observe that the gap between the waiting times of cars and buses is largest when cars have a lower priority, decreases when cars and buses are given the same priority, and decreases further when cars are given a higher priority than buses. The ratio of these differences is shown in Table 6.6. When both vehicle types have the same priority, bus waiting time is shorter by ∼7-9%. This increases to ∼11-15%, when buses are given higher priority, and decreases to ∼3-4%, when cars are given higher priority. Therefore, we observe that in DWL the relative improvements in policy performance are directly related to their relative priorities.

From this set of experiments we conclude that DWL respects relative policy priorities and improves vehicle waiting times accordingly, both when deployed with predefined values of C and when DWL agents learn their own individual values of C.

6.4.7 DWL Learning Times

In this section we present the results of Scenario 5, as described in Section 6.3.4.5. In all of the other experiments, we ran the experiment in 5 stages for 750 minutes, allowing what we considered sufficient time for DWL agents to learn Q-values and W-values for their local and remote policies and Q-values

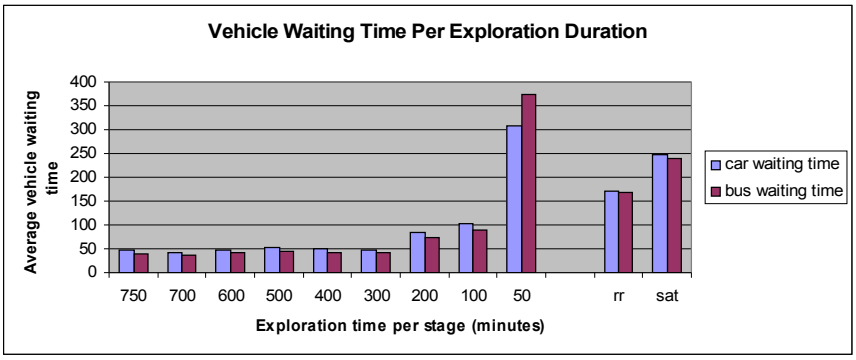


Fig. 6.22: DWL performance vs. duration of exploration period

for the process of learning C. This set of experiments was designed to determine the minimum duration of training time that DWL requires to show benefits over our baselines. The results are graphed in Figure 6.22, which shows average waiting times per vehicle type for RR, SAT, and DWL with a variety of lengths of training period. From the graph we observe that the minimum training time required for DWL to start outperforming the baselines is 100 minutes per experiment stage, i.e, 100 minutes to learn Q-values (stage 1) and 100 minutes to learn W-values (stage 2). As we already noted when describing this Scenario, we do not run stages 4 and 5 for this set of experiments, as we only aim to identify the minimum training time for DWL to start showing benefits over the baselines, rather than to achieve the best possible performance of DWL.

This shows that DWL has very feasible training times, of 200 minutes, or just under 3.5 hours per set of conditions in this set of experiments. For example, in order to train DWL for morning rush hour conditions, this would only require data to be gathered over two days, taking into account that morning rush hour generally has similar traffic conditions for 2 hours, 7:30-9:30am.

6.4.8 Policy and Agent Dependencies in DWL

In this section we examine DWL's ability to learn the dependencies between policies and between agents, as captured by the W-values of local and remote policies of an agent. We believe that DWL's ability to improve the performance of multiple policies is based on its ability to learn these dependencies and exploit them, by learning the situations (states) in which, some policy (or some agent, through the use of remote policies) should gain control over the selection of the next action. We first show that W-learning on a single agent with two policies can learn dependencies between policies, where policy compatibility is reflected in low or negative W-values (as policies do not need to compete for control over actuators, since the actions of one policy also suit the other policy), and policy conflict

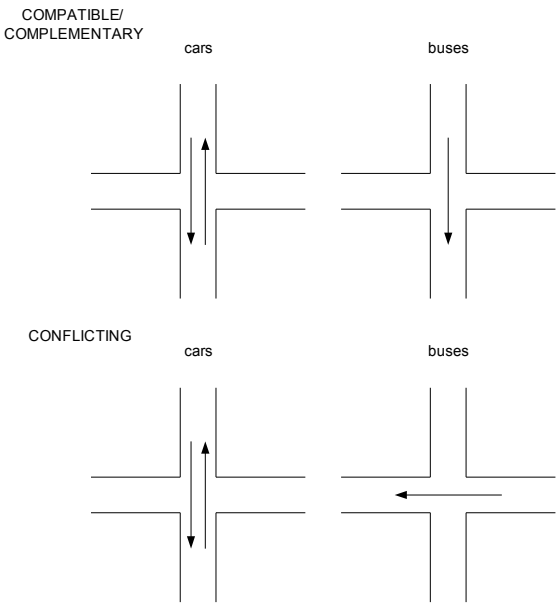


Fig. 6.23: Compatible vs. conflicting scenarios: Policy dependency

is reflected in high W-values, higher for the policy of higher priority, as it is more important for that policy to take over control over actuators. Analogously, we then show that DWL can learn these dependencies between agents, by examining the W-values of remote policies, that reflect the level of compatibility between a local agent's actions and a remote agent's performance.

6.4.8.1 Policy Dependency

To examine policy dependency on a single agent, we performed experiments on a single four-approach junction using single-policy GWO and single-policy PTO combined using W-learning. We implement two traffic scenarios: where two policies are at least compatible (and possibly even complementary), and a conflicting traffic scenario. These scenarios are shown in Figure 6.23. In the compatible scenario, cars approach the junction from two directions, that can both be served using the same phase, and buses approach the junction from one of the directions from which cars also approach. Therefore, the policies are compatible or complementary as both cars and buses can be served by the same phase. In the conflicting scenario, buses approach the junction from a direction orthogonal to approaching cars, and cannot be served using the same phase as cars.

We show the W-values for GWO and PTO for both the compatible and conflicting traffic scenarios in Figure 6.24. We observe that for the compatible scenario, the W-values for both GWO and PTO all have negative or a very low positive values, averaging -5 for GWO and -14 for PTO. For the

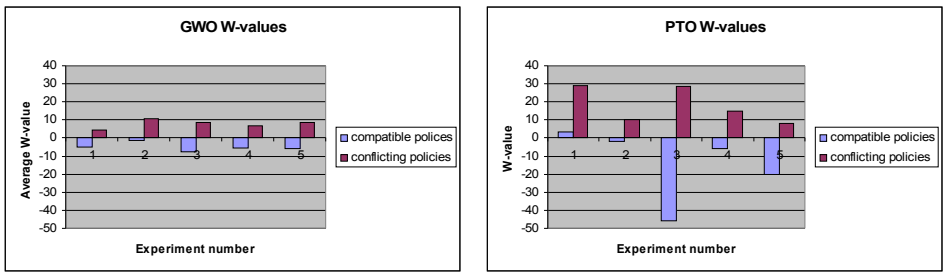


Fig. 6.24: Compatible/complementary vs. conflicting scenarios: DWL W-values for multiple policies on a single agent

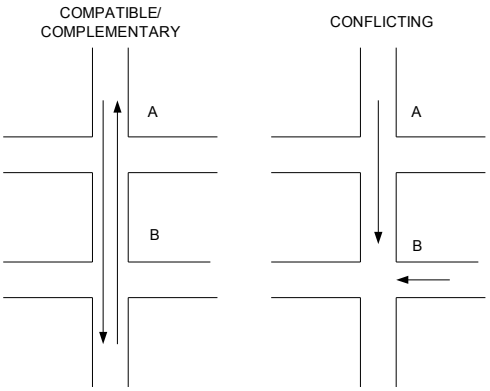


Fig. 6.25: Compatible/complementary vs. conflicting scenarios: Agent dependency

conflicting scenario, however, all W-values for both GWO and PTO have positive values, averaging to 8 for GWO and 18 for PTO. Therefore, in a conflicting scenario policies need to suggest their actions with higher W-values to increase the chance of having the highest W-value and gaining control over actuators. PTO has a higher priority, which is reflected in a higher average of its W-values.

6.4.8.2 Agent Dependency

To analyze agent dependency, and how it is reflected in the W-values of remote policies, we examined W-values for PTO from Scenario 1, which we consider a compatible/complementary scenario, and Scenario 3, which we consider a conflicting scenario. We selected two junctions in the centre of the simulation map, on O'Connell Street, and examined the traffic behaviour on its approaches. Observed traffic behaviours are shown in Figure 6.25.

In a compatible/complementary scenario, both junctions A and B have buses approach from the same directions, where both directions can be served using the same phase. In a conflicting scenario, junction B has buses approaching from the junction A, and from another direction orthogonal to it,

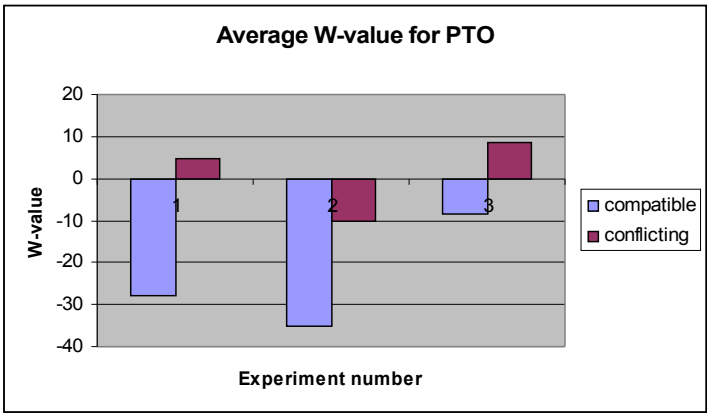


Fig. 6.26: Compatible vs. conflicting scenarios: W-values for remote policies

i.e., from two directions that cannot be served using the same phase. We examine the W-values for a remote policy on A, which learns how A's actions reflect on B's performance towards its PTO policy, i.e., learns the dependency between A and B towards PTO. Observed W-values are shown in Figure 6.26. For the compatible/complementary scenario, all W-values are large-negative values, averaging -23, while for the conflicting scenario they are either positive or low negative value, averaging 1.

The above results show that DWL successfully extends W-learning's capability to learn the dependencies between policies on a single agent, by implementing the capability to learn dependency between agents by using remote policies.

6.4.9 Additional Observations: Number of Vehicle Stops

In our simulations we have also measured the average number of vehicle stops per vehicle type for all experiments performed. We did not analyze these results in detail when analyzing other metrics as they are mostly consistent with the average vehicle waiting time and as such would not provide any additional insight. In this section we present these results, and focus only on the instances where average number of stops is not consistent with other metrics presented.

Figure 6.27 shows the average number of stops per vehicle type for the traffic conditions from Scenario 1 and Scenario 2, i.e., uniform low load, uniform high load, and the 3:1 traffic pattern. We show the results for baselines RR and SAT, for non-collaborative multi-policy deployments (C=0), for fully collaborative DWL deployments (C=1), for DWL deployments with an experimentally determined to be the best-performing predefined value of C for the respective traffic conditions, and for DWL deployments with learnt values of C. We see that all DWL deployments outperform (i.e., result in a lower average number of vehicle stops) the baselines, collaborative deployments outperform the

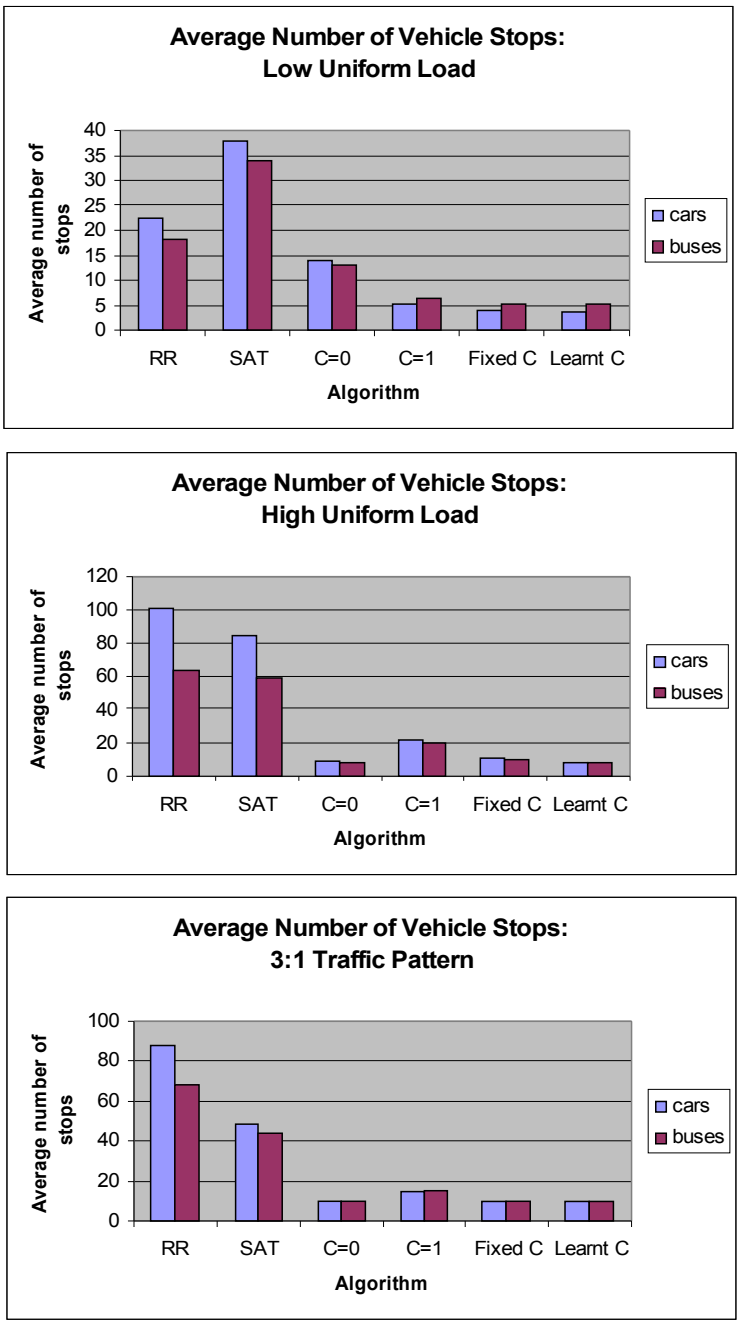


Fig. 6.27: Average number of vehicle stops per vehicle type: DWL and baselines

non-collaborative ones, and deployments that learn values of C outperform or perform as well as those that use predefined values.

This is the same behaviour we observed when analyzing average vehicle waiting time. However, there are two instances where the relationship between the performance of deployments in terms of average number of stops presented here is not consistent with the relationship between their performances in terms of average vehicle waiting time. Under the 3:1 traffic pattern, the C=0 deployment has a lower number of stops than the C=1 deployment, although when we compared the average vehicle waiting times between these two deployments in Section 6.4.2, C=0 resulted in a higher average vehicle waiting time. Also, under uniform high load, SAT outperforms RR in terms of number of stops, as shown on the graph, however it has worse vehicle waiting times, as discussed in Section 6.4.1. Therefore, in these two instances, vehicles make fewer stops, but the stops they do make are longer. In the SAT vs. RR case, we can attribute this to the ability of SAT to extend the phase duration, so that when vehicles are stopped at a traffic light they might need to wait longer than they do in RR which uses a fixed 20s phase duration. In the C=0 vs. C=1 case, the vehicle stops might be longer when C=0 due to a lack of collaboration, e.g., vehicles might need to wait at a junction even when their signal is set to green, because the backlog of vehicles created by a downstream junction might be preventing them from proceeding.

Overall, the results in terms of average number of vehicle stops confirm suitability of DWL to collaborative multi-policy optimization, as collaborative DWL deployments, and deployments where DWL learns suitable values of C, outperform the baselines by lowering the average number of vehicle stops by ∼62-92%.

6.5 Evaluation Summary

In this chapter we presented details of the evaluation of DWL as a multi-agent multi-policy optimization technique for decentralized autonomic environments. We have presented the evaluation objectives, described the evaluation scenarios, and presented and analyzed the results.

From the analysis of the results we conclude that DWL is a suitable algorithm for multi-agent multi-policy optimization in our application area, UTC, and as such could be a promising approach to optimization in other large-scale decentralized autonomic systems with similar characteristics.

DWL outperforms both existing UTC algorithms that we have used as baselines, with improvements in average vehicle waiting time ranging from ∼73% to ∼88%, depending on the specific baseline used, the evaluation scenario, and the vehicle type (Objective #1, as listed in Section 6.1). Collab-

orative DWL deployments outperform non-collaborative multi-policy techniques, with improvements in average vehicle waiting time ranging from ∼56% to ∼90%, depending on the evaluation scenario and the vehicle type (Objective #2). Learning C rather than using a predefined value of C can also improve the performance of the system, by, depending on the evaluation scenario, either improving the performance of both policies (by up to ∼61% depending on the evaluation scenario and the vehicle type), or performing as well as the best predefined value of C while removing the need for extensive testing to determine the most suitable values of predefined C (Objective #3). DWL, even though it has primarily been designed to enable simultaneous optimization towards multiple policies, can also be deployed in single-policy systems to improve their performance by enabling collaboration between agents. Single-policy collaborative deployments of DWL that learn values of C outperform the baselines, non-collaborative scenarios, and either outperform or perform as well as deployments with a predefined value of C (Objective #4). DWL respects policy priorities (Objective #5), as it either improves the performance of both policies, or, when trade-offs are required, it improves the performance of the higher priority policy. Additionally, we observe that in DWL the relative improvements in policy performance are directly related to their relative priorities. We have also shown that addressing both policies simultaneously using DWL, in the presence of multiple vehicle types, improves the performance of the system when compared to single-policy deployments (Objective #6). If there are multiple vehicle types in the system (e.g., buses and cars), single-policy deployments either perform well for the vehicle type they are addressing and neglect other vehicles, or perform poorly for both vehicle types due to policy dependency. Multi-policy DWL deployments can improve the overall system performance by either improving the performance of both vehicle types or improving the performance of one vehicle type with smaller negative effects on the other. DWL showed performance improvements under all four sets of traffic conditions evaluated (uniform low load, uniform high load, non-uniform 3:1 traffic pattern, and a conflicting scenario) and under different policy priority and scope relationships, showing its suitability for a variety of environmental conditions and policy characteristics (Objective #7). Finally, we have shown that DWL requires only 3.5 hours of traffic data, under the evaluated set of traffic conditions, to start outperforming the UTC baselines used (Objective #8).

Chapter 7

Conclusions and Future Work

"*The duty of helping one's self in the highest sense involves the helping of one's neighbors.*"

- Samuel Smiles

In this chapter we summarize the thesis and review its most significant achievements. We then conclude with a discussion of the remaining open research issues related to this work.

7.1 Thesis Contribution

The work in this thesis addresses multi-policy optimization in decentralized autonomic systems.

Chapter 1 motivated the work by outlining issues in multi-policy optimization in decentralized autonomic systems implemented using multi-agent technologies. We argued that the main challenges in multi-policy optimization in such systems arise from the heterogeneity of agents and the heterogeneity of the policies that they implement, and dependencies between agents and policies caused by shared operating environments. Due to these dependencies, we concluded that cooperation between agents could be beneficial, however, we have identified that collaboration introduces further issues, for example, how to motivate agents to cooperate, with what other agents should they cooperate, to what degree should they cooperate and when.

We then, in Chapter 2, analyzed the autonomic computing domain in more detail, focusing on multi-agent systems and existing self-organizing techniques used for optimization in large-scale decentralized systems. We reviewed existing work in multi-policy RL and multi-agent RL, identifying a gap for an RL-based multi-agent collaborative optimization technique capable of addressing multiple

policies simulatenously. We also introduced UTC, as an example of a decentralized autonomic system, in which we have evaluated DWL.

In Chapter 3 we presented a case study in which we evaluated two existing single-agent multi-policy RL techniques in a non-cooperative multi-agent simulation of UTC in order to establish a baseline for our research into collaboratieve multi-policy optimization techniques. We found W-learning to be a suitable technique for non-collaborative multi-agent multi-policy optimization. In our simulations W-learning outperformed the baselines that we used for comparison, RR and SAT, which are based on techniques currently used for optimization in UTC. W-learning also either outperformed or performed as well as the other single-agent multi-policy RL-based optimization technique that we evaluated, which combined multiple learning processes into a single learning process with a single state space. The results of the experiments in this preliminary case study also highlighted policy and agent dependency, confirming our belief that cooperation could be beneficial for system performance.

In Chapter 4 we derived a set of requirements for a collaborative multi-agent multi-policy optimization technique in decentralized autonomic systems. We argued that such a technique needs to be self-organizing, i.e., not require external control; needs to be decentralized, i.e., not require a global system view; needs to be capable of learning suitable behaviours rather than using predefined rules; needs to be capable of addressing heterogeneous system policies simultaneously while respecting their relative priorities; and needs to enable collaboration between heterogeneous agents, as well as enable agents to learn the levels of cooperation in which to engage.

We then presented DWL, the main contribution of this thesis, and outlined how it meets the requirements specified. DWL is an RL-based multi-agent multi-policy optimization technique which enables simultaneous optimization towards multiple policies, regardless of their scopes or relative priorities, and enables collaboration between heterogenous agents, i.e., agents whose state-space representations and action-sets differ. In DWL, each agent uses Q-learning to learn suitable actions for its local policies, and W-learning to learn the relative weights of the actions nominated by local policies in given states. In DWL, each agent also has a set of so-called "remote policies", that use Q-learning to learn how the agent's local actions affect its one-hop neighbours' policies (i.e., it learns the dependencies between agents), and uses W-learning to learn the weights of the actions nominated by remote policies in given states. At each time step, each agent executes the action with the highest associated W-values across all local and remote policies, after the W-values of remote policies have been scaled using a cooperation coefficient C. C determines the level of cooperativness of an agent, and ranges from C=0, where agents are fully non-collaborative, to C=1, where agents are fully collaborative, i.e., take their neighbours' action suggestions into account with the same relative weight

as their own. C can be predefined to the same value on all agents, or can be learnt by each agent individually so as to maximize the rewards received by its local and remote policies, while respecting relative policy priorities. Designed in such a way DWL can be used to implement fully self-organizing systems, i.e., it does not require external control, central control or a global system view. All actions and interactions are performed locally within the one-hop neighbourhood of an agent, while global optimization towards system policies emerges from those local actions and interactions.

In Chapter 5 we presented details of the implementation of DWL and how it is applied to generate traffic-light agents implementing multiple traffic optimization policies in a simulation of UTC.

In Chapter 6 we evaluated DWL as a multi-policy optimization technique for collaborative multi-agent autonomic systems in a simulation of UTC. We evaluated DWL's performance in a variety of traffic conditions, addressing compatible as well as conflicting policies, different policy spatial scopes and their relative priorities, and different lengths of training time.

Our experiments show that DWL is suitable for optimization in UTC, as it outperforms our baselines, RR and SAT, under all tested scenarios, with improvements in average vehicle waiting time ranging from 73% to 88%, depending on the evaluation scenario and vehicle type. DWL also improves the performance of both policies over non-collaborative deployments, through the use of remote policies and the cooperation coefficient C. The best performing DWL deployments with a predefined value of C outperform non-collaborative multi-policy deployments under all tested scenarios, with improvements in average vehicle waiting time ranging from 56% to 90%, depending on the evaluation scenario and vehicle type. We also show that the performance of both policies can be further improved under some traffic conditions by enabling agents to learn suitable values of C rather than using predefined values. We have also investigated the impact of the addition of a second policy to the system on the original policy, in the presence of multiple vehicle types. Addressing both policies simultaneously using DWL improves the performance of both policies over their corresponding single-policy deployments (if policies are compatible or complementary), or the performance of one vehicle type is slightly degraded for improvements in the performance of the other vehicle type (if policies are conflicting) but no vehicle type is neglected, as it might be the case in single-policy deployments. Additionally, even though it is primarily aimed at multi-policy environments, we show that DWL's collaboration approach can also improve the performance of the system in the presence of a single policy only. Finally, we show that DWL has feasible training times, as it requires only 3.5 hours worth of exploration to start outperforming the baselines in the traffic scenario tested. All of the above results hold for a variety of traffic conditions, showing DWL is a suitable technique for collaborative multi-policy optimization in a simulation of UTC, and suggesting its wider applicability in other heterogenous

large-scale decentralized autonomic systems.

7.2 Open Research Issues

When designing and evaluating DWL, we have identified several areas where DWL's performance and applicability could be extended and identified a number of areas for potential future research. We outline these areas and discuss the remaining open research issues below.

As DWL is based on RL, several issues inherent in RL should be addressed before DWL is applied in live systems. First, DWL could be extended to the optimization of multiple policies in non-stationary environments. Learning processes can be extended with mechanisms that detect significant changes in the environment and either reinitiate the learning process, or switch to another previously learnt set of behaviours for a given set of environment conditions. Due to the presence of multiple policies in the system, the main challenge with this approach would be detecting whether a change in environmental conditions affect one or more policies, and whether it changes the dependencies between policies and dependencies between agents, and adapting accordingly. Secondly, as Q-learning and W-learning require exploration periods while learning optimal actions and weights, the use of DWL with batch reinforcement learning algorithms (Kalyanakrishnan & Stone, 2007) could be investigated and applied in environments where online learning is not feasible. Additionally, applicability of DWL for multi-policy optimization in partially observable environments could be addressed.

In terms of the features of DWL, an implementation of learning the most suitable cooperation coefficient per pair of agents rather than for the whole agent's neighbourhood could be considered. This would enable agents to give a higher priority to action suggestions from the neighbours whose behaviours a more significant effect on global system behaviour than other neighbours. Additionally, mechanisms to detect the levels of compatibility between policies (based on W-values and performance) could be implemented. In the case of conflicting policies, where trade-offs between performance of the policies is needed, a desired (e.g., a minimum or maximum) value of performance metrics for policies could be specified, and recommendations on relative values of policy rewards made.

One of the main open research issues related to DWL is providing formal guarantees on its performance. Convergence and optimality guarantees associated with single-agent Q-learning do not extend to our multi-agent multi-policy case. The problem of exactly solving a multi-agent problem has been shown to be intractable as it has a non-deterministic explonential time complexity (Bernstein et al., 2000), and therefore, most multi-agent learning algorithms (Dowling & Haridi, 2008), including DWL, use approximation techniques, and cannot provide convergence and optimality guarantees.

Even though empirical results show DWL is a promising approach to multi-policy optimization in decentralized autonomic systems, some application areas might require stronger behavioural guarantees, for example, providing at least performance boundaries for system behaviour.

Bibliography

Abdelwahed, S., & Kandasamy, N. (2007). *Autonomic Computing: Concepts, Infrastructure, and Applications*, chap. A Control-Based Approach to Autonomic Performance Management in Computing Systems. CRC Press.

Abdulhai, B., & Pringle, R. (2003). Autonomous reinforcement learning - 5 GC urban traffic control. Tech. rep., Transportation Research Board Annual Meeting.

Abdulhai, B., Pringle, R., & Karakoulas, G. (2003). Reinforcement learning for the true adaptive traffic signal control. *Journal of Transportation Engineering*, *129*(3), 278–285.

Angus, D., & Woodward, C. (2009). Multiple objective ant colony optimisation. *Swarm Intelligence*, *3*(1), 69–85.

Anthony, R., Butler, A., & Ibrahim, M. (2007). *Autonomic Computing: Concepts, Infrastructure, and Applications*, chap. Exploiting Emergence in Autonomic Systems. CRC Press.

Babaoglu, O., Jelasity, M., & Montresor, A. (2005). *Unconventional programming paradigms*, chap. Grassroots Approach to Self-management in Large-Scale Distributed Systems. Springer Berlin / Heidelberg.

Babaoglu, O., Meling, H., & Montresor, A. (2002). Anthill: A framework for the development of agent-based peer-to-peer systems. *International Conference on Distributed Computing Systems*.

Baird, L., & Moore, A. (1999). Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems 11*, (pp. 968–974). MIT Press.

Baran, B., & Schaerer, M. (2003). A multiobjective ant colony system for vehicle routing problem with time windows. In *Proceedings of IASTED International Conference on Applied Informatics*.

Barrett, L., & Narayanan, S. (2008). Learning all optimal policies with multiple criteria. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, (pp. 41–47).

Bazzan, A. L. (2005). A distributed approach for coordination of traffic signal agents. *Autonomous Agents and Multi-Agent Systems*, *10*(1), 131–164.

Bazzan, A. L. C. (2009). Opportunities for multiagent systems and multiagent reinforcement learning in traffic control. *Autonomous Agents and Multi-Agent Systems*, *18*(3), 342–375.

Bernstein, D. S., Zilberstein, S., & Immerman, N. (2000). The complexity of decentralized control of markov decision processes. In *Mathematics of Operations Research*.

Blum, C., & Merkle, D. (Eds.) (2008). *Swarm Intelligence: Introduction and Applications*. Natural Computing Series. Springer.

Brooks, R. (1986). Achieving artificial intelligence through building robots. Tech. rep., Cambridge, MA, USA.

Brooks, R. A. (1991). How to build complete creatures rather than isolated cognitive simulators. In *Architectures for Intelligence*, (pp. 225–239). Erlbaum.

Bull, L. (2004). Learning classifier systems: A brief introduction. In *Applications of Learning Classifier Systems*. Springer.

Busoniu, L., Schutter, B. D., & Babuska, R. (2005). Learning and coordination in dynamic multiagent systems. Tech. Rep. 05-019, Delft Center for Systems and Control, Delft University of Technology, Delft, The Netherlands.

Bustard, D. W., & Sterritt, R. (2007). *Autonomic Computing: Concepts, Infrastructure, and Applications*, chap. A Requirements Engineering Perspective on Autonomic Systems Development. CRC Press.

Camponogara, E., & Kraus, W. (2003). Distributed learning agents in urban traffic control. In *Portuguese Conference on Artificial Intelligence (EPIA)*, (pp. 324–335).

Choi, S. P., Yeung, D. Y., & Zhang, N. L. (2002). Multi-model approach to non-stationary reinforcement learning. In *Proceedings of Artificial Intelligence and Soft Computing*.

Chowdhury, M. A., & Sadek, A. W. (2003). *Fundamentals of Intelligent Transportation Systems Planning*. Artech House Publishers.

Claus, C., & Boutilier, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, (pp. 746–752). AAAI Press.

Coello, C. A. C. (1999). A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, *1*, 269–308.

Cuayahuitl, H., Renals, S., Lemon, O., & Shimodaira, H. (2006). Learning multi-goal dialogue strategies using reinforcement learning with reduced state-action spaces. In *International Journal of Game Theory*, (pp. 547–565).

Cui, X., Potok, T., & Palathingal, P. (2005). Document clustering using particle swarm optimization. In *Swarm Intelligence Symposium*.

Curtis, D. (2003). Adaptive control software. Tech. rep., U.S. Department of Transportation, Federal Highway Administration.

Das, R., Tesauro, G. J., & Walsh, W. E. (2005). Model-based and model-free approaches to autonomic resource allocation. Tech. rep., IBM.

Di Caro, G., & Dorigo, M. (1998). AntNet: Distributed Stigmergetic Control for Communication Networks. *Journal of Artificial Intelligence Research*, *9*, 317–365.

Di Caro, G., Ducatelle, F., & Gambardella, L. M. (2005). AntHocNet: An adaptive nature-inspired algorithm for routing in mobile ad hoc networks. *European Transactions on Telecommunications, Special Issue on Self-organization in Mobile Networking*, *16*, 443–455.

Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., & Phung, D. (2005). Self-managing systems: A control theory foundation. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, (pp. 441–448). Washington, DC, USA: IEEE Computer Society.

Doerner, K., Hartl, R., & Reimann, M. (2003). Are COMPETants more competent for problem solving? - the case of full truckload transportation. *Central European Journal of Operations Research*, *11*(2), 115–141.

Dorigo, M., & Di Caro, G. D. (1999). *The Ant Colony Optimization Meta-Heuristic*, (pp. 11–32). London: McGraw-Hill.

Dowling, J. (2005). *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. Ph.D. thesis, Trinity College Dublin.

Dowling, J., Cunningham, R., Curran, E., & Cahill, V. (2006). Building autonomic systems using collaborative reinforcement learning. *Knowledge Engineering Review*, *21*(3), 231–238.

Dowling, J., & Haridi, S. (2008). *Reinforcement Learning*, chap. Decentralized Reinforcement Learning for the Online Optimization of Distributed Systems. I-Tech Education and Publishing.

DTO (2006). Road user monitoring report. Tech. rep., Dublin Transportation Office.

Dusparic, I., & Cahill, V. (2009a). Distributed W-Learning: An algorithm for multi-policy optimization in decentralized autonomic systems (poster). In *Proceedings of the 6th International Conference on Autonomic Computing and Communications*.

Dusparic, I., & Cahill, V. (2009b). Distributed W-Learning: Multi-policy optimization in self-organizing systems. In *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*.

Dusparic, I., & Cahill, V. (2009c). Multi-policy optimization in decentralized autonomic systems (extended abstract). In J. S. S. Carles Sierra, Keith S. Decker, & C. Castelfranchi (Eds.) *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, (pp. 1203–1204).

Dusparic, I., & Cahill, V. (2009d). Using reinforcement learning for multi-policy optimization in decentralized autonomic systems - an experimental evaluation. In W. Reif, G. Wang, & J. Indulska (Eds.) *Proceedings of the 6th International Conference on Autonomic and Trusted Computing*, vol. 5586 of *Lecture Notes in Computer Science*, (pp. 105–119).

Ehrgott, M., & Gandibleux, X. (2002). *Multiple Criteria Optimization. State of the art annotated bibliographic surveys*. Kluwer Academic, Dordrecht.

Eiben, A., & Smith, J. (2003). *Introduction to Evolutionary Computing*. Springer, Natural Computing Series.

Eiben, A. E. (2005). Evolutionary computing and autonomic computing: Shared problems, shared solutions?. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. P. A. van Moorsel, & M. van Steen (Eds.) *Self-star Properties in Complex Information Systems*, vol. 3460 of *Lecture Notes in Computer Science*, (pp. 36–48). Springer.

Febbraro, A. D., Giglio, D., & Sacco, N. (2004). Urban traffic control structure based on hybrid petri nets. *IEEE Transactions on Intelligent Transportation Systems*, *5*(4), 224–237.

Fellendort, M. (1997). Public transport priority within SCATS - a simulation case study in dublin. Tech. rep., PTV.

Flake, G. W. (2000). *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. The MIT Press.

Gábor, Z., Kalmár, Z., & Szepesvári, C. (1998). Multi-criteria reinforcement learning. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, (pp. 197–205). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Gadanho, S. C., & Hallam, J. (2001). Robot learning driven by emotions. *Adaptive Behaviour*, *9*(1), 42–64.

Gambardella, L. M., Taillard, E., & Agazzi, G. (1999). MACS-VRPTW: a multiple ant colony system for vehicle routing problems with time windows. (pp. 63–76).

Ganek, A. G., & Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, *42*(1), 5–18.

Gershenson, C., & Heylighen, F. (2003). When can we call a system self-organizing? In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, & J. Ziegler (Eds.) *Advances in Artificial Life, 7th European Conference, ECAL*, (pp. 606–614).

Ghosh, B. (2009). Applied transportation analysis - short-term traffic forecasting. Presentation.

Goldman, C. V., & Zilberstein, S. (2003). Optimizing information exchange in cooperative multi-agent systems. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, (pp. 137–144). New York, NY, USA: ACM.

Goldman, C. V., & Zilberstein, S. (2004). Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research (JAIR)*, *22*, 143–174.

Goldsby, H. J., Cheng, B. H. C., McKinley, P. K., Knoester, D. B., & Ofria, C. A. (2008). Digital evolution of behavioral models for autonomic systems. In *International Conference on Autonomic Computing*, (pp. 87–96). IEEE Computer Society.

Guestrin, C., Koller, D., & Parr, R. (2001). Multiagent planning with factored MDPs. In *14th Neural Information Processing Systems (NIPS-14)*, (pp. 1523–1530). Vancouver, Canada.

Guestrin, C., Lagoudakis, M., & Parr, R. (2002). Coordinated reinforcement learning. In *Proceedings of the ICML-2002 The Nineteenth International Conference on Machine Learning*, (pp. 227–234).

Hiraoka, K., Yoshida, M., & Mishima, T. (2008). Parallel reinforcement learning for weighted multi-criteria model with adaptive margin. (pp. 487–496). Berlin, Heidelberg: Springer-Verlag.

Hoar, R., Penner, J., & Jacob, C. (2002). Evolutionary swarm traffic: if ant roads had traffic lights. In *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*, (pp. 1910–1915). Washington, DC, USA: IEEE Computer Society.

Humphrys, M. (1996a). Action selection methods using reinforcement learning. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, (pp. 135–144). MIT Press.

Humphrys, M. (1996b). *Action Selection methods using Reinforcement Learning*. Ph.D. thesis, University of Cambridge.

IBM (2005). An architectural blueprint for autonomic computing. Tech. rep., IBM.

Jin, Y., & Sendhoff, B. (2008). Pareto-based multiobjective machine learning: An overview and case studies. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, *38*(3), 397–415.

Kadrovach, B. A., & Lamont, G. B. (2002). A particle swarm model for swarm-based networked sensor systems. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, (pp. 918–924). New York, NY, USA: ACM.

Kaelbling, L. P., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kalyanakrishnan, S., & Stone, P. (2007). Batch reinforcement learning in a complex domain. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, (pp. 650–657). New York, NY, USA: ACM.

Karlsson, J. (1997). *Learning to solve multiple goals*. Ph.D. thesis, Rochester, NY, USA.

Kennedy, J., & Russell, E. C. (2001). *Swarm Intelligence (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann.

Kephart, J. O. (2005). Research challenges of autonomic computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, (pp. 15–22). New York, NY, USA: ACM Press.

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41–50.

Kephart, J. O., & Walsh, W. E. (2004). An artificial intelligence perspective on autonomic computing policies. *Policies for Distributed Systems and Networks, IEEE International Workshop on*.

Kirk, D. E. (2004). *Optimal Control Theory: An Introduction*. Dover Publications.

Klein, L. A. (2001). *Sensor technologies and data requirements for ITS*. Artech House.

Kok, J. R., 't Hoen, P. J., Bakker, B., & Vlassis, N. (2005). Utile coordination: learning interdependencies among cooperative agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, (pp. 29–36). Colchester, United Kingdom.

Kok, J. R., & Vlassis, N. (2006). Collaborative multiagent reinforcement learning by payoff propagation. *Journal of Machine Learning Research*, *7*, 1789–1828.

Kutz, M. (2003). *Handbook of Transportation Engineering*. McGraw-Hill Professional.

Lekavy, M. (2005). Optimising Multi-agent Cooperation using Evolutionary Algorithm. In M. Bielikova (Ed.) *Proceedings of IIT.SRC 2005: Student Research Conference in Informatics and Information Technologies, Bratislava*, (pp. 49–56). Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava.

Littman, M. L., Ravi, N., Fenson, E., & Howard, R. (2004). Reinforcement learning for autonomic network repair. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, (pp. 284–285). Washington, DC, USA: IEEE Computer Society.

Lowrie, P. (1982). SCATS: The sydney co-ordinated adaptive traffic system - principles, methodology, algorithms. In *Proceedings of the IEE International Conference on Road Traffic Signalling*.

Maniezzo, V., Gambardella, L. M., & Luigi, F. D. (2004). *New Optimization Techniques in Engineering*, chap. Ant Colony Optimization. Springer-Verlag Berlin Heidelberg.

Mariano, C., & Morales, E. F. (2000). A new distributed reinforcement learning algorithm for multiple objective optimization problems. In *IBERAMIA-SBIA '00: Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI*, (pp. 290–299). London, UK: Springer-Verlag.

McCann, J. A., & Huebscher, M. C. (2004). Evaluation issues in autonomic computing. In *Grid and Cooperative Computing Workshops*, (pp. 597–608).

McGuire, J., & O'Keeffe, D. (2003). The limerick adaptive urban traffic control system project. Tech. rep., Limerick City Council and Arup Consulting Engineers.

Melo, F., & Veloso, M. (2009). Learning of coordination: Exploiting sparse interactions in multiagent systems. In *Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems*.

Mikami, S., & Kakazu, Y. (1994). Genetic reinforcement learning for cooperative traffic signal control. In *International Conference on Evolutionary Computation*, (pp. 223–228).

Mirchandani, P., & Head, L. (2001). RHODES: a real-time traffic signal control system: Architecture, algorithms, and analysis. Tech. rep., The University of Arizona and Gardner Transportation Systems.

Montresor, A., Meling, H., & Babaoglu, O. (2002). Messor : Load-balancing through a swarm of autonomous agents. Tech. Rep. UBLCS-02-08, Departement of Computer Science, University of Bologna, Bologna, Italy.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.

Natarajan, S., & Tadepalli, P. (2005). Dynamic preferences in multi-criteria reinforcement learning. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, (pp. 601–608). New York, NY, USA: ACM.

OLeary, D. (2008). Supporting decisions in real-time enterprises: autonomic supply chain systems. *Information Systems and E-Business Management*, *6*(3), 239–255.

Oliveira, E., & Duarte, N. (2005). Making way for emergency vehicles. In *Proceedings of the 2005 European Simulation and Modelling Conference*, (pp. 128–135).

Oxford (2000). *The Oxford English Dictionary*. Oxford University Press.

Papageorgiou, M., Diakaki, C., DInopoulou, V., Kotsialos, A., & Wang, Y. (2003). Review of road traffic strategies. In *Proceedings of the IEEE*, vol. 91.

Paquet, S., Bernier, N., & Chaib-draa, B. (2004). Multi-attribute decision making in a complex multiagent environment using reinforcement learning with selective perception. In *Canadian Conference on AI*, (pp. 416–421).

Parsons, S., & Wooldridge, M. (2002). Game theory and decision theory in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, *5*(3), 243–254.

Parsopoulos, K. E., & Vrahatis, M. N. (2002). Particle swarm optimization method in multiobjective problems. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, (pp. 603–607). New York, NY, USA: ACM.

Peek Traffic Limited, Siemens Traffic Controls, TRL Limited (2009). SCOOT. http://www.scoot-utc.com/.

Pendrith, M. D. (2000). Distributed reinforcement learning for a traffic engineering application. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, (pp. 404–411). New York, NY, USA: ACM Press.

Perez, J., Germain-Renaud, C., Kegl, B., & Loomis, C. (2008). Grid differentiated services: A reinforcement learning approach. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, (pp. 287–294). Washington, DC, USA: IEEE Computer Society.

Peshkin, L., eung Kim, K., Meuleau, N., & Kaelbling, L. P. (2000). Learning to cooperate via policy search. In *Proceedings of the 16th Annual Conference on Uncertainty in Artificial Intelligence (UAI-00)*, (pp. 489–496). Morgan Kaufmann.

Peters, J., Vijayakumar, S., & Schaal, S. (2005). Natural actor-critic. In *European Conference on Machine Learning*, (pp. 280–291).

Pierre-Luc Gregoire, J. L., Charles Desjardins, & Chaib-draa, B. (2007). Urban traffic control based on learning agents. In *Proceedings of the 10th International Conference on Intelligent Transportation Systems (ITSC'07)*.

Pitu, M., Anna, K., Larry, H., & Wu, W. (2000). An approach towards the integration of bus priority, traffic adaptive signal control, and bus information/scheduling systems. In S. Vos, & J. Daduna (Eds.) *International conference on computer-aided scheduling of public transport (CAPST)*, (pp. 319–334). Springer-Verlag.

Prothmann, H., Rochner, F., Tomforde, S., Branke, J., Müller-Schloer, C., & Schmeck, H. (2008). Organic control of traffic lights. In *ATC '08: Proceedings of the 5th international conference on Autonomic and Trusted Computing*, (pp. 219–233). Berlin, Heidelberg: Springer-Verlag.

Pugh, J., Zhang, Y., & Martinoli, A. (2005). Particle swarm optimization for unsupervised robotic learning. In *Swarm Intelligence Symposium*, (pp. 92–99).

Raicevic, P. (2006). Parallel reinforcement learning using multiple reward signals. *Neurocomputing*, *69*(16-18), 2171–2179.

Ramdane-Cherif, A. (2007). Toward autonomic computing: Adaptive neural network for trajectory planning. *International Journal of Cognitive Informatics and Natural Intelligence*, *1*(2), 16–33.

Reyes-Sierra, M., & Coello, C. A. C. (2006). Multi-objective particle swarm optimizers: A survey of the state-of-the-art. *International Journal of Computational Intelligence Research*, *2*(3), 287–308.

Reynolds, V., Cahill, V., & Senart, A. (2006). Requirements for an ubiquitous computing simulation and emulation environment. In *InterSense '06*. NY, USA: ACM.

Richter, S. (2006). Learning traffic control - towards practical traffic control using policy gradients. Tech. rep., Albert-Ludwigs-Universitat Freiburg.

Robertson, D. I., & Bretherton, R. D. (1991). Optimizing networks of traffic signals in real time - the SCOOT method. *IEEE Transactions on Vehicular Technology*.

Rosenblatt, J. K. (2000). Optimal selection of uncertain actions by maximizing expected utility. *Autonomous Robots*, *9*(1), 17–25.

Rosenschein, J. S., & Zlotkin, G. (1994). *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. Cambridge, Massachusetts: MIT Press.

Russell, S., & Norvig, P. (2003). *Aritifical Intelligence - A Modern Approach*. Prentice Hall.

Russell, S. J., & Zimdars, A. (2003). Q-decomposition for reinforcement learning agents. In T. Fawcett, & N. Mishra (Eds.) *International Conference on Machine Learning*, (pp. 656–663). AAAI Press.

S Jones, M. H., & Fox, K. (1998). State of the art and user needs for selected vehicle priority. Tech. rep., UK Department of Transport.

Salkham, A., Cunningham, R., Garg, A., & Cahill, V. (2008). A collaborative reinforcement learning approach to urban traffic control optimization. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, vol. 2, (pp. 560–566).

Schneider, J., Wong, W.-K., Moore, A., & Riedmiller, M. (1999). Distributed value functions. In *Proceedings of the Sixteenth International Conference on Machine Learning*, (pp. 371–378). Morgan Kaufmann.

Sen, S., & Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chap. Learning in Multiagent Systems, (pp. 259–298). Cambridge, MA, USA: MIT Press.

Serugendo, G. D. M., Foukia, N., Hassas, S., Karageorgos, A., Mostefaoui, S., Ulieru, O. R. M., Valckenaers, P., & Aart, C. V. (2003). Self-organization: Paradigms and applications. In *Proceedings of The International Workshop on Engineering Self-Organizing Applications*.

Shelton, C. R. (2000). Balancing multiple sources of reward in reinforcement learning. In *Neural Information Processing Systems*, (pp. 1082–1088).

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. In *Machine Learning*, (pp. 323–339).

Sprague, N., & Ballard, D. (2003). Multiple-goal reinforcement learning with modular Sarsa(0). In *International Joint Conference on Artificial Intelligence*.

Srinivasan, D., Choy, M. C., & Cheu, R. L. (2006). Neural networks for real-time traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, *7*(3), 261–272.

Sterritt, R. (2005). Autonomic computing. *Innovations in Systems and Software Engineering*, (pp. 79–88).

Sterritt, R., Parashar, M., Tianfield, H., & Unland, R. (2005). Autonomic computing. *Advanced Engineering Informatics*, *19*(3), 181–187.

Stone, P., & Veloso, M. M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, *8*(3), 345–383.

Suton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: A Bradford Book. The MIT Press.

Sycara, K. (1998). Multiagent systems. *AI Magazine*, *19*(2).

Tan, K. C., Khor, E. F., Lee, & Heng, T. (2005). *Multiobjective Evolutionary Algorithms and Applications (Advanced Information and Knowledge Processing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 330–337). Morgan Kaufmann.

Tesauro, G. (1999). Pricing in agent economies using neural networks and multi-agent Q-learning. In *Proceedings of Workshop ABS-3: Learning About, From and With other Agents*.

Tesauro, G. (2005). Online resource allocation using decompositional reinforcement learning. Tech. rep., IBM.

Tesauro, G. (2007). Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, *11*(1), 22–30.

Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., & White, S. R. (2004). A multi-agent systems approach to autonomic computing. *International Joint Conference on Autonomous Agents and Multiagent Systems*, (pp. 464–471).

Tesauro, G., Das, R., Walsh, W. E., & Kephart, J. O. (2005). Utility-function-driven resource allocation in autonomic systems. *International Conference on Autonomic Computing*, (pp. 342–343).

Tesauro, G., Jong, N. K., Das, R., & Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, (pp. 65–73). Washington, DC, USA: IEEE Computer Society.

Tham, C. K., & Prager, R. W. (1994). A modular Q-learning architecture for manipulator task decomposition. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann.

Traffic Authority of New South Wales Australia, R. (2009). www.rta.nsw.gov.au/.

Vamplew, P., Yearwood, J., Dazeley, R., & Berry, A. (2008). On the limitations of scalarisation for multi-objective reinforcement learning of pareto fronts. In *AI '08: Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence*, (pp. 372–378). Berlin, Heidelberg: Springer-Verlag.

Van Veldhuizen, D. A., & Lamont, G. B. (2000). Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary Computation*, *8*(2), 125–147.

Vlassis, N. (2007). *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Morgan and Claypool publishers.

Watkins, C. J. C. H., & Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, *8*(3), 279–292.

Weijters, A. J. M. M., & Hoppenbrouwers, G. A. J. (1995). Backpropagation networks for grapheme-phoneme conversion: a non-technical introduction. In *Artificial Neural Networks: An Introduction to ANN Theory and Practice*, (pp. 11–36). London, UK: Springer-Verlag.

Weiss, G. (Ed.) (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press.

Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J., & Waldbusser, S. (2001). Rfc 3198: Terminology for policy-based management. Tech. rep., The Internet Society.

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., & Kephart, J. O. (2004). An architectural approach to autonomic computing. *Autonomic Computing, International Conference on*, (pp. 2–9).

Wiering, M., van Veenen, J., Vreeken, J., & Koopman, A. (2004a). Intelligent traffic light control. Tech. rep., Institute of Information and Computing Sciences, Utrecht University.

Wiering, M., Vreeken, J., Van Veenen, J., & Koopman, A. (2004b). Simulation and optimization of traffic in a city. In *IEEE Intelligent Vehicles Symposium (IV'04)*. IEEE.

Wolf, T. D., & Holvoet, T. (2004). Emergence and self-organisation: a statement of similarities and differences. In *Lecture Notes in Artificial Intelligence*, (pp. 96–110). Springer Verlag.

Wolf, T. D., & Holvoet, T. (2007). *Autonomic Computing: Concepts, Infrastructure, and Applications*, chap. A Taxonomy for Self-* Properties in Decentralized Autonomic Computing. CRC Press.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. Wiley.

Yagan, D., & Tham, C.-K. (2007). Coordinated reinforcement learning for decentralized optimal control. In *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*.

Yang, Z., Chen, X., Tang, Y., & Sun, J. (2005). Intelligent cooperation control of urban traffic networks. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, (pp. 1482 – 1486).