8

Decentralized Reinforcement Learning for the Online Optimization of Distributed Systems

Jim Dowling and Seif Haridi
Swedish Institute of Computer Science
Sweden

1. Introduction

Distributed reinforcement learning is concerned with what action an agent should take, given its current state and the state of other agents, so as to minimize a system cost function (or maximize a global objective function). In this chapter, we give an overview of distributed reinforcement learning and describe how it can be used to build distributed systems that can adapt and optimize their operation to a dynamic environment. In particular, we focus on decentralized systems, where an agent has only a partial view of the system and does not have access to the system cost (or reward) function, that is, an agent does not have full observability of the state of all other agents in the system and system utility (performance) is not directly measurable in real-time.

Theoretical results that establish convergence and optimality guarantees for single-agent reinforcement learning algorithms do not hold for distributed (multi-agent) systems. This is because distributed environments are inherently non-stationary: agents can independently learn, adapt, and initiate new tasks. In fact, Bernstein et al. have shown that the problem of optimizing agent behavior in such a decentralized multi-agent system has non-deterministic exponential time-complexity (Bernstein et al., 2002). Thus, most existing approaches use approximate algorithms for distributed reinforcement learning.

In order for agents to learn globally good policies, we assume the need for agents to cooperate. This is because greedy policies at agents, based only on local state at the agent, do not necessarily improve global utility; in fact, they can even decrease global utility, as demonstrated in the "tragedy of the commons" problem. Another key property of distributed reinforcement learning is scalability; learning algorithms will cause message passing between agents and the algorithms need to make efficient use of the network.

Examples of distributed systems that perform online optimization using distributed reinforcement learning include packet routing in MANETs (Dowling, 2005), information-directed routing in Sensor Networks (Ghasemaghaei et al, 2007; Zhang et al, 2006), and optimization of application configurations in pervasive computing environments (Rigole, 2006). Although these systems all employ different variants of distributed reinforcement learning, they are all cooperative; agents selflessly contribute towards a common goal.

We see distributed reinforcement learning having similar potential to function approximation with reinforcement learning (RL). In both cases, the theoretical foundation that RL adopts from dynamic programming no longer applies, but that does not prevent the

Source: Reinforcement Learning: Theory and Applications, Book edited by Cornelius Weber, Mark Elshaw and Norbert Michael Mayer
ISBN 978-3-902613-14-1, pp.424, January 2008, I-Tech Education and Publishing, Vienna, Austria

development of useful systems. From a RL perspective, the potential advantages of adding distribution to reinforcement learning algorithms include the ability to handle larger state spaces by partitioning the state space over agents, an increased rate of learning through parallel learning over more computational hardware, and the ability to build more robust systems using redundant agents. For distributed systems, distributed reinforcement learning is an approach that can be used to build robust, self-managing and self-optimizing systems.

In the next sections, we address these questions, describe existing and related approaches to distributed reinforcement learning and decentralized control, and, finally, we present our work on collaborative reinforcement learning, and show how it can be used to build an adaptive load-balancing system.

2. Challenges of physical aspects of distributed systems

Physical aspects of distributed systems should be accounted for in any distributed reinforcement learning algorithm. For example, agents communicate by message passing over a network, but messages may be lost and message delivery is typically not guaranteed within a fixed time bound. Message passing is required in distributed reinforcement learning algorithms to enable agents to collaborate to solve distributed problems and to enable agents to collectively learn improved policies by sharing their local information with one another. Care must be taken when designing distributed reinforcement learning algorithms, to ensure that collective learning strategies do not generate excessive amounts of network traffic, flooding the system. Techniques for improving network utilization should also be considered, such as the caching of recent data received from neighbors, asynchronous message passing, and sending batches of messages.

Given that accounting for the network is crucial to distributed reinforcement learning, there are a number of related issues that must be taken into account when designing distributed reinforcement learning algorithms. These include:

- *degree of centralization*: centralization of system state or cost (reward) signals introduces both a bottleneck and a single point of failure in a system. However, distribution of system state and cost functions requires adapting the Markov Decision Process (MDP) framework to a distributed (multi-agent) system.
- *non-stationary environments*: distributed systems cannot be modeled as strict MDPs, as they may have non-Markovian aspects such as multiple, concurrent decision-making agents and history dependence (Tesauro, 2007). However, RL is only guaranteed to work in stationary (or almost stationary) environments. Network connections, however, are non-stationary, but in practice may be stable for long enough periods to enable learning or at least be amenable to modeling. Learning algorithms should not just guarantee eventual convergence on a near-optimal policy, but also guarantee *timely convergence* to ensure that real-time distributed system constraints are met. Realistic experimentation plays an important role in validating the timeliness of convergence of algorithms.
- *agent and network dynamism*: in the system of interest, will the number of agents be fixed or limited, and will there be any form of control on how and when agents join the system? Can agents adapt their connections to change their neighbors at runtime, thus

changing the system topology? We must assume there is no control over agents leaving the system, as hosts (and, therefore, agents) can fail arbitrarily.

- *message passing costs*: what events in the learning algorithms cause message passing? We need to modify learning algorithms so that message passing costs are included when calculating the costs of actions that induce message passing. In general, acquiring experience in distributed systems is costly, as it affects system performance.
- *agent views*: do agents collaborate to solve system problems? How is an agent's local view of the system represented? What update strategies are used to update an agent's view? Are update strategies synchronous or asynchronous to the execution of the agent's learning algorithm?
- *model-based learning*: model-free learning is generally not useful where acquiring real-world experience is more expensive than computation. How do we integrate model-based learning into distributed reinforcement learning algorithms?
- *approximating the system cost signal*: how can agents approximate the system cost signal, given that the global utility of a distributed system is not directly measurable at runtime? Solutions need to address the *spatial credit assignment problem*: how agents determine which other agents and states were responsible for taking good actions. Solutions must quickly and efficiently propagate changes to relevant agents. Solutions may also address non-linear system cost functions, where a small action, relative to the size of the system, by one agent can produce large changes in system utility.

3. Distributed reinforcement learning problem definition

Goldman and Zilberstein characterize multi-agent reinforcement learning as a decentralized control problem for stochastic systems (Goldman and Zilberstein, 2004). In decentralized control systems, agents take decisions without complete state information with the goal of optimizing some system performance measure. The general distributed learning problem can be characterized as how an agent can learn a policy, using partially-observable state information that minimizes a partially-observable system cost function in the presence of other independent agents, who are also learning a policy under the same conditions.

We now formulate the problem in terms of discrete-time stochastic control problems, based on (Cogill et al, 2006). The system has a finite state space *S*, and a finite set *A* of actions available at each time step. A cost $g(s,a) \rightarrow \Re$ is incurred when an action $a \in A$ is taken while in state $s \in S$. At the time step after an action *a* is taken in state *S*, the system state transitions with probability $p(s'|\,s,a)$ to $s' \in S$. The goal of the system is to learn a policy for choosing actions that minimizes the overall cost incurred over a given time period, where costs may be geometrically discounted over time. (The alternative, equivalent formulation is to maximize accumulated rewards; we minimize costs to explicitly model the cost of message passing over a network). A policy $\pi : S \rightarrow A$ describes which action is chosen by the system, based on the current system state. Here we consider the problem of choosing a policy to minimize the *expected total discounted cost*

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t g(s_t, \pi_t(s_t)) \middle| s_0 = s\right] \tag{1}$$

where 0 ≤ γ < 1. For a policy π, we can compute the total cost V^{π} from state *s* by solving:

$$V^{\pi}(s) = g(s, \pi(s)) + \gamma \sum_{s' \in S} p(s' \mid s, \pi(s)) V^{\pi}(s') \tag{2}$$

For the expected total discounted cost, there is a unique optimal value function, $V^*(s)$, which minimizes the total cost for all initial states:

$$V^*(s) = \min_{a \in A}\left(g(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V^*(s') \right) \tag{3}$$

A closely related function is the optimal action-value function, Q*(*s*, *a*), given by

$$Q^*(s,a) = g(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V^*(s') \tag{4}$$

An optimal policy π^*, not necessarily unique, is obtained from Q^* by taking $\pi^* = \arg\min_a Q^*(s,a)$.

The above equations describe an optimal policy for a system with globally observable state and system actions, independent of whether a single agent controls the policy or many agents collaboratively control the policy. In some versions of distributed reinforcement learning system, many agents use local actions and message passing to control the evolution of the single system MDP.

Decentralized reinforcement learning is a different decentralized control model, where many independent RL agents learn their local policy using both local state, and a model of neighboring agents built using message passing (Schneider et al, 1999; Dowling et al, 2005). In *decentralized reinforcement learning,* an agent has a model for its neighbors, called its *view* of the system. There is no global state identifying the current state of the system and no global actions. The desired system behavior must be realized by providing agents with actions to affect their local environment, as well as the ability to both collaborate and communicate with other agents. At this point, we can no longer model the system as a single MDP, and formally reason about the system's optimal policy. The system still has a cost function that should be minimized, although its representation is now distributed.

For problems that can be factored, agents can often *approximate* the system cost function, by knowing that the agent and its neighbors make a linear contribution to the system cost. Here, an agent learns a policy that minimizes the cost of its actions based both on its local state and its view, where a view is the agent's set of neighboring agents. This way, local cost functions at agents can, over time, converge on a good approximation of the system cost. The policy will only be approximate, because other agents are simultaneously learning local policies (and may execute conflicting actions that reduce system utility), and the agent's view is typically not always fully consistent with the actual state of neighbors (as it would be overly expensive to maintain synchronized views of neighbors). Also, agents may need to learn about costs at remote parts of the system, not represented in their local view. In this case, it is important that the view model enables estimated costs to propagate over multiple views to reach the relevant agents in the system.

We now present our simplified model for decentralized reinforcement learning. A decentralized reinforcement learning agent n_i is described by the tuple

$$n_i = (S_i, A_i, v_i) \tag{5}$$

, where $n_i \in N$ is an agent from the set of all agents in the system N , S_i is the set of local states at n_i , A_i is the set of local actions at n_i , $v_i \subset N$ is the set of neighboring agents of n_i . The global state the system is a function of all local states at all agents in the system. A neighbor relation is defined from one agent to another agent, if one agent can send messages directly to the other agent. An agent may have a local representation of its set of neighbors, $v_i = (v_0, \ldots, v_{N-1})$, defined as the agent's *view* of the system. The set of all agents and their neighbors defines the *topology* of the system as a graph; where agents are the vertices and neighbor relations are the directed edges. Typically, an agent has much fewer neighbors than there are agents in the system. Finally, the behavior of the system containing *K* agents is defined as a set of policies, one for each of the agents, $\pi_i : S_i \rightarrow A_i$ for i = 1, . . . ,K-1. The decentralized reinforcement learning problem is defined as how the set of policies minimizes a system cost function.

4. Related work on distributed reinforcement learning

In distributed reinforcement learning, actions by individual agents can potentially influence any other agent in the system. However, a naïve approach to action selection where agents are required to reach full consensus on the best action for the system does not scale, due to the excessive message passing required to reach consensus. In this section we cover some of the existing distributed reinforcement learning models that reduce the amount of system knowledge that an agent requires to select an action that is approximately optimal for the system. We also discuss Ant Colony Optimization (ACO), a multi-agent learning model not directly related to RL, but which addresses the same problem of decentralized control.

4.1 Coordination graphs for the collaborative multi-agent control of a system MDP

A multi-agent RL system can be modeled as the collaborative multi-agent control of a single MDP (Guestrin et. al, 2003). In this model, the distributed control problem involves the online or offline computation of a coordinated action for a group of *n* agents. Each agent *i* selections a local action $\mathrm{a_i}$ and the joint action of all *n* agents is $\mathrm{a} = (\mathrm{a_0}, .., \mathrm{a_{n-1}})$. The joint action generates a cost $\mathrm{g(a)}$ for the group of agents, where the optimal joint action is $a^* = \arg\min_a g(a)$. The goal of the agents is to select actions that minimize the received costs over a sequence of actions. However, if agents have full observability of the system, the size of the joint action space increases exponentially in the number of agents in the system.

One way to reduce the size of the joint action space is to enable agents to exclude those states that do not need to be estimated when computing joint actions. This approach is viable for problems that can be *factored*. Factored problems can be sub-divided, solved separately by agents and the overall result can be calculated as a linear combination of the results of the sub-problems. Guestrin introduced a coordination graph (CG) as a model for representing factored problems (Guestrin et. al, 2003), where the global coordination problem is approximated as a set of local coordination problems involving a smaller number of agents. In a CG, the global cost function, **g**(a), is decomposed into a sum of local cost functions, f_i, calculated independently at each agent using every possible action combination within the agent's neighborhood:

$$g(a) = \sum_{1}^{n} f_i(a_i) \tag{6}$$

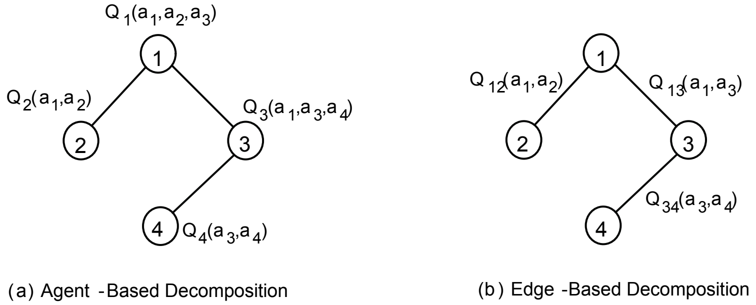


Fig. 1. The global Q-function for a 4-agent problem can be modeled in a coordination graph (Kok and Vlassis, 2006).

Although Guestrin's CG model was designed for off-line approximation, Kok and Vlassis adapted the model for online learning. In both models, the global Q-function is factored in a CG. In *Fig. 1*, we can see how Guestrin decomposes the global Q-function using an agent's set of neighbors, called agent-based decomposition (Guestrin et. al, 2002), while Kok and Vlassis decompose the global Q-function using connections between pairs of agents, called edge-based decomposition (Kok and Vlassis, 2006). In edge-based decomposition, an edge from agent *i* to agent *j* is represented as a Q-function, $Q_{i,j}$, where the sum of all edges (Q-functions) defines the global Q-function. Local Q-functions are updated based on the local Q-functions of the pair of agents that form the edge. This compares to agent-based decomposition where local Q-functions are updated based on the local Q-functions of all neighbors. In order to calculate the best joint action, agents in Kok and Vlassis' model use an approximate algorithm called max-plus, while agents in Guestrin's model use an exact algorithm called variable-elimination. The edge-based decomposition approach scales linearly to the width of the CG, while the agent-based decomposition approach scales exponentially (Kok and Vlassis, 2006).

The main problem with applying coordination graphs to online optimization of distributed systems is that it is often communication constraints that determine the topology of distributed systems, not problem constraints as in coordination graphs. The approach also assumes that problems can be factored, and does not explicitly account for the possibility of agent failure.

4.2 Independent learners

An alternative to selecting joint actions is to allow agents take individual actions, with the goal that the collective behavior of the agents will minimize the system cost function (Kok and Vlassis, 2007). Experiments by Claus and Boutilier with groups of independent Q-learning agents showed the need for agent cooperation to ensure that local agent actions are globally good (Claus and Boutilier, 1998). Agent's that are unaware of other agents can choose actions which are suboptimal for the system, as they use local Q-values that are incorrectly assumed to be independent of the actions selected and rewards received by the other agents. Another approach to building an independent learner model, where agents are unaware of one another, is Wolpert's Collective Intelligence (COIN) model (Lawson and Wolpert, 2002). In the COIN model, problems are structured such that independent agents' local cost models are adapted to approximate the system cost model, ensuring that actions that are locally good are always globally good. This approach, however, has limited applicability.

4.3 Distributed value functions

Schneider et al., 2002, have designed a distributed reinforcement learning algorithm where independent agents coordinate learning by sharing value functions between one another. Agents define a weight function *f(i,j)* that defines an agent's fixed set of neighbors (through weights being zero to non-neighbors, and non-zero for neighbors), and the weight of the Q-values from neighbors that should be contributed to updates to Q-values. The weight function is quite a general, as it defines both the static topology of the system and how value information is transferred over the network to a state-action pair from a successor state. As it is an approximate algorithm, they do not provide any convergence guarantees. The update function is defined as:

$$Q_i(s_i, a_i) = (1-\alpha).Q_i(s_i, a_i) + \alpha\left[R_i(s,a) + \gamma \sum_{neighbours} f(i,j) \max_{a_j} \; Q_j(s'_j, a'_j)\right] \quad (7)$$

However, the weight function assumes that agents can exchange information about their local values at no cost and that the environment is stationary. Also, the model does not provide support for reducing network utilization, such as caching data and asynchronous or batched message passing.

4.4 DEC-POMDP-Com

Goldman and Zilberstein address an offline version of the decentralized reinforcement learning problem for independent agents. They firstly assume that agents have probabilistic observations of the state space (noisy observations), that is, agents are described locally by a partially observable Markov Decision Process (POMDP). A group of agents is defined as a decentralized POMDP (DEC-POMDP). They also provide agents with an explicit language of communication, consisting of an alphabet of messages, to develop a communication policy. Communication policies model agent communications and its associated costs and the complete model of decentralized POMDPs with communication support is called a DEC-POMDP-Com. With this model, a joint policy can be defined over agents as a set of local policies, where each policy is composed of the communication and action policies for each agent. The result is a complex model, which eschews a potentially simpler approach of integrating messaging costs into reward functions. As their model is an off-line approach, it is unsuitable for online learning in distributed systems.

4.5 Ant colony optimization

The distributed control problem is also addressed by Ant-Colony Optimization (ACO), a non-reinforcement learning based approach, best described as a meta-heuristic for producing approximate solutions to combinatorial optimization problems (Dorigo and Stuetzle, 2004). Combinatorial optimization problems involve finding the minimum cost solution from a set of possible solutions, and problems are defined as

$$\Pi = \left(S, f, \Omega\right) \tag{8}$$

, where *S* is the set of candidate solutions, *f* is the objective function that assigns a value $f(s)$ to each candidate solution, $s \in S$, and Ω is a set of constraints (Dorigo and Stuetzle, 2004). ACO can be used to solve both static and dynamic combinatorial problems, where dynamic problems have non-stationary stochastic dynamics. Dynamic problems are defined as a function of some quantities whose value is set by the dynamics of an underlying system.

In order solve dynamic combinatorial optimization problems, ACO algorithms construct a problem with the following structure (Curran, 2003):

- a set of *components* $C = \{c_1, \ldots, c_M\}$, which correspond to agents with a single state in RL;
- a set of *L connections* among *C*, which correspond to neighbor relationships between agents;
- a *connection cost* function, $J : L \times R \rightarrow \Re$ defined over the connections, and parameterized by time. *J*(*l*, *t*) corresponds loosely to the expected cost g(*s*,*a*);
- ACO defines a *solution* to a combinatorial optimization problem as the lowest cost feasible path through the topology, that is, the graph of components and connections, which satisfies the set of problem requirements;
- In ACO a *solution cost* function is typically a summation of the connection cost over the connections that the solution contains. This is similar to factoring the system cost function in factored MDPs, as it is also a linear combination of the cost of the sub-problems.

ACO works by a population of agents, called ants, finding minimum cost paths (solutions) in the component graph by exploring, measuring the cost of edges as it traverses them, and storing estimated path costs at components as a *pheromone trail*. Components represent the environment of the agents, and pheromone trails store path costs from the current component c_i to another (often the terminal) component c_M . The pheromone trail at c_i can be observed by other agents as they traverse component c_i, in a form of indirect communication, known as stigmergy. Other agents can use the pheromone trail as a partial solution to their (potentially different) optimization problem.

Similar to agent action selection in RL, ants choose a connection in the graph using a probabilistic decision rule, and must balance exploration and exploitation to ensure that good quality solutions are found in reasonable time. However, an ant also has a memory of the path it has traversed that can be used in decision making, for example, to prevent loops in paths. When an ant has reached its termination conditions, it generally uses its memory to retrace its path and update the pheromone trails to reflect the cost of the solution found. This process of sending backward ants is very similar to multi-step backups in RL. Also, the backup approach in ACO naturally factors optimization problems by only attempting to update some subset of the states in the system, those states traversed by the ant from the start state. This is similar to approaches in RL used to reduce the amount of states updated after an action is taken, such as coordination graphs and prioritized sweeping (Sutton and Barto, 1998).

An important difference with RL is ACO's *pheromone trail decay*, in which the value of the pheromone trails decreases automatically over time. Decaying discovered solutions over time prevents too early convergence on sub-optimal solutions and increases exploration. Decay also helps adaptation from old solutions to new solutions in response to changes in the environment. Later, in collaborative reinforcement learning, we show how the similar mechanisms of decaying of estimated costs in RL can be used by agents learning in non-stationary environments.

ACO problems can be viewed as a subset of RL problems (Curran, 2003). In particular, ACO is applicable to problems where:

- agents are independent learners;
- states are discrete and all paths eventually terminate, that is, absorbing Markov Decision Processes in RL;
- there are *start states*, which are those where optimization is initiated, and whose value must be optimized.

A significant difference between ACO and RL are connection costs in ACO which may be time-varying, enabling adaptation to a non-stationary environment. RL has no time-based model for decaying discovered solutions.

5. Collaborative reinforcement learning

The rest of this chapter concerns our work on collaborative reinforcement learning (CRL), (Dowling et al, 2005; Dowling, 2004). CRL's system model defines a decentralized reinforcement learning system as a set of independent, collaborative learner agents. Similar to ACO, collaborative reinforcement learning models system optimization problems as a set of discrete optimization problems (DOP), that can be initiated at any agent and solved at any agent in the system. A DOP is defined as the combinatorial optimization problem, see Equation 8, of finding the agent from a discrete set of agents (the system) that can solve a particular problem with lowest cost. The ACO-like notion of a DOP as the problem to be solved by an agent can also be viewed as a task from traditional RL literature.

In CRL, a system is modeled as a graph, $G(V,E)$, where agents are vertices and views of neighbors, defined later in Equation 10, define the edges in the graph. From the agent-perspective the goal of CRL is to solve a DOP at the lowest cost agent in the system. From a system perspective, the goal of CRL is to minimize the total cost of solving all DOPs in the system. Each agent is an independent learner with its own local MDP; there is no global MDP or any joint actions defined over agents. Agents have only local actions and local states, but may collaborate with neighbors to solve DOPs and share estimated costs of solving DOPs with one another.

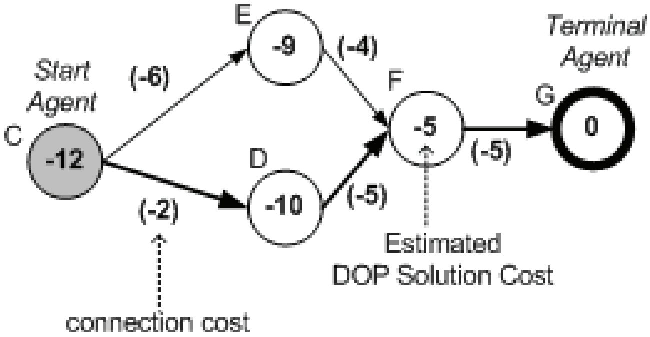


Fig. 2. A Discrete Optimization Problem (DOP) in CRL involves agents finding the lowest cost path from a Start Agent to a Terminal Agent that can solve the DOP.

In CRL, a DOP is solved as a sequential decision making problem, where the objective is to solve the DOP at lowest cost at some agent in the system (see

Fig. 2). This system problem can be viewed as an absorbing MDP (although no system MDP is explicitly represented) where the agents are states, that is, the DOP is guaranteed to enter a terminal state after a finite amount of time. Each agent has at least one state, an *initial state*, where the solution to the DOP is started, and at least one (and possibly all) agent(s) have a *terminal state*, where the DOP is solved.

There are three types of action that are supported in CRL for an agent to solve a DOP: a *local action* that contributes to solving the DOP at the current agent, a *delegation action* that forwards the DOP to a neighbor, and a *discovery action* that attempts to find a new neighbor (that may be able to solve the DOP at lower cost). Discovery actions are necessary for system bootstrap, when an agent does not have any neighbors, and for discovering new neighbors online. The three action types are illustrated in

Fig. 3, where it is shown how a MDP can be started either by an application at the agent's host or by a neighbor delegating the DOP to it. In distributed systems, delegation actions map to message passing over a network and discovery actions map to some form of underlying service or host discovery protocol. A discovery action that finds a new neighbor adds a new delegation action for the neighbor and a new Q-Value entry in its lookup table for the relevant state(s). For delegation actions, CRL provides a *connection cost* as an explicit model for the cost of using a network link. Agents attempt to learn a policy that solves DOPs with minimal cost, treating all three types of action equally, and attempting to select the action with the minimal estimated cost, given the agent's current state.

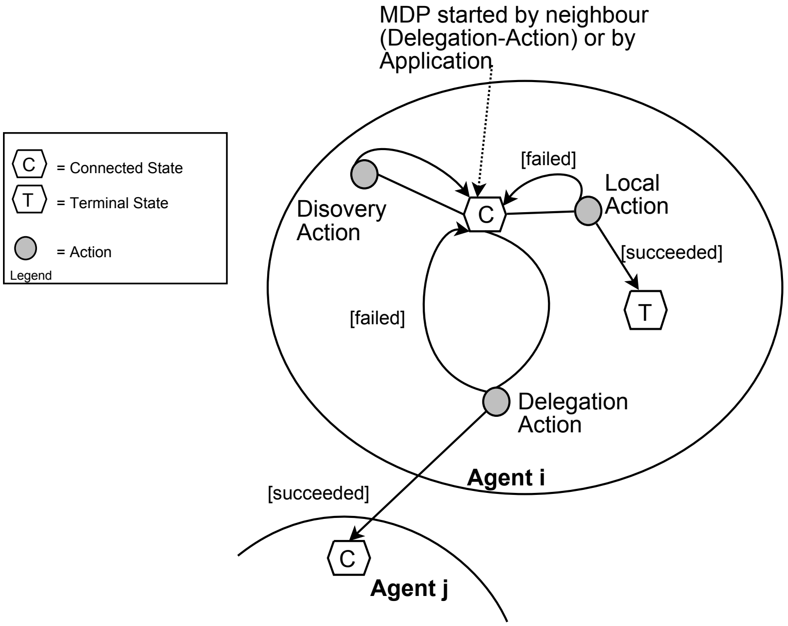


Fig. 3. Agent i has two states and three actions (a delegation, local and discovery action). The connected state is an initial state for the MDP at agent i and when the terminal state is reached at some agent the DOP has been solved. These are the three main types of action in CRL: local actions attempt to solve the DOP at the agent, delegation actions attempt to forward the DOP to a neighbor, and discovery actions try to find new neighbors. DOPs can be started by either an application or a neighbor; delegation and local actions may fail.

5.1 CRL system model

We now present the system model for Collaborative Reinforcement Learning (CRL). An agent n_i is described by the tuple

$$n_i = (S_i, A_i, v_i, C_i) \tag{9}$$

, where S_i is its set of local states, A_i is the set of local actions, $v_i \subset N$ is the set of neighboring agents of n_i , and C_i is the *cache*, a set of all *cached views* of neighbors of n_i . A cached view is defined as a pair containing a Q-function $Q_i(s_i, a_j)$ at n_i , where a_j is a delegation-action, and a V-value $V_j(s_v)$ for a state s_v at a neighboring agent n_j , for which a state transition is possible from s_i at agent n_i to s_j at agent n_j ,

$$(Q_i(s_i, a_j), V_j(s_v)) \in C_i, \; if \; P(s_v | s_i, a_j) > 0 \tag{10}$$

, where $s_i \in S_i$, $a_j \in A_i$, $s_v \in \bigcup_{j=0}^{|v_i|} S_j, S_j \in v_i$, and $V_j(s_v) \in \Re$. A state transition from a state s_i at n_i to a state s_j at n_j is realized by the termination of the local MDP at n_i and the initiation of a new MDP at n_j . The state s_j is called a connected state as it represents the connection of the MDPs at the two different agents. On a successful state transition state s_i to state s_j , a backup of $V_j(s_v)$ is made to its entry in the cache at agent n_i . A cached view of a neighbor, thus, represents a directed connection from agent n_i to agent n_j for the delegation of DOPs and the backup of V-value information. Model-based Q-learning algorithms can use the cached V-values to reason about the cost of delegation actions to neighbors, without having to send messages to those neighbors; thus helping improve network utilization. As we will see, the cached V-values can also be updated asynchronously by neighbors, using advertisements, and over time, by decay of the cache.

5.2 Advertisement of estimated DOP costs between agents

When an agent executes a delegation action to successfully forward a DOP to a neighbor, it receives an instantaneous cost (backup) from its neighbor that is used to update the agent's cached V-value. However, agents can also asynchronously advertise to their neighbors updates to the V-values of their connected states. This asynchronous advertisement enables agents to learn about remote parts of the system without executing actions.

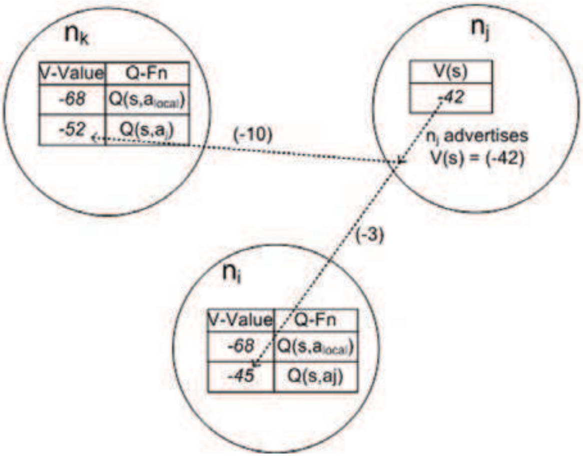


Fig. 4. Agent j advertises its V-value for state s to neighbors k and i. Advertisement enables agents to collectively learn about the state space by agents sharing updated V-values; agents do not have to take actions to learn about changes in the state space.

In *Fig. 4*, we can see how agent n_j sends its updated V-value to neighboring agents n_i and n_k . In this example, at agent *k* the V-value for a local action is -68, whereas its estimated cost of its neighbor agent *j* solving it is -52 (of which -10 is the estimated connection cost to *j*). Agent *k* will, therefore, have a higher probability of delegating a DOP to agent *j* than attempting to solve it locally. In this example, the advertisement may be caused by a local action at n_j or possibly by an advertisement received by n_j. Advertisements can cause cascading changes in MDPs at many agents caused by a change in a MDP at a single agent.

Implementation strategies for advertisement of V-values include broadcasting (useful in wireless networks where the network medium is shared), gossiping values periodically, conditional notification, and return values from delegation actions. Advertisements should be sent regularly to neighbors to indicate the agent's availability and refresh any cached V-values.

5.3 Decaying of cached Q-values for connected states

In the absence of advertisements, agents *decay* the V-values in their cache over time. The decay model allows agents to remove non-contactable agents from their set of neighbors when a cost threshold is reached. Decay enables agents to adapt their policies to a dynamic set of neighbors, as neighbors that were lower cost in the past are gradually forgotten over time in the absence of advertisements from them, for example, because they left the system. The rate of decay of costs in the cache is configurable, with higher rates more appropriate for more dynamic networks. Cached V-values for connected states are decayed using the following equation:

$$Decay(V(s))=V(s).\rho^{td} \tag{11}$$

where *td* is the amount of time elapsed since the last received advertisement for $V(s)$ from a neighbor, $V(s) \in \Re$, and ρ is a scaling factor that sets the rate of decay.

5.4 CRL learning algorithm

The CRL learning algorithm is a *distributed model-based reinforcement learning algorithm* with a cost model for network connections. In CRL, agents maintain a model of the state transition probabilities, $P_i(s'|s,a)$, which is particularly useful for state transitions to connected states, as they represent network links between agents. Even a simple statistical model based on recent observations of the network link can be a powerful predictor of whether it will function or not (Dowling et al., 2005).

Executing actions in CRL returns a cost signal, $g_i(s,a)$ from the agent's local environment. However, delegation actions receive an additional connection cost, $D_i(s'|s,a)$, that provides the estimated network cost of delegating the DOP from the local agent to a neighboring agent. Our distributed model-based reinforcement learning algorithm *for delegation actions* is

$$Q_i(s,a)=g_i(s,a)+\sum_{s'\in S_i} P_i(s'|s,a).(D_i(s'|s,a)+V_j(s')) \tag{12}$$

, where *s* is the current state, *a* the action to be executed from the set of possible actions, and *s′* is the next state from the set of possible next states. $V_j(s')$ is the estimated cost of the neighbor *j* solving the DOP, that is, it is the value function for agent *j*'s connected state *s'*. The value, $V_j(s')$, is retrieved from the local cache at agent *i*, and no message needs to be sent to agent *j* to calculate it. $P_i(s'|s,a)$ is calculated from the local state transition model, as the probability of the next state being *s'* after action *a* was executed in state *s*.

If, however, the action is not a delegation action, the connection cost, $D_i(s'|s,a)$, becomes zero. Thus, the distributed model-based reinforcement learning algorithm *for local actions and discovery actions* is:

$$Q_i(s,a) = g_i(s,a) + \sum_{s' \in S_i} P_i(s'|s,a).(V_j(s')) \tag{13}$$

The pseudo-code for the algorithm that a CRL agent executes at each time-step is outlined in Fig. 5. In this version, advertisement and decay are implemented as synchronous activities in the single thread; an alternative implementation would be to execute them asynchronously in a second thread.

```
Thread 1
repeat forever
listen for a DOP/advertisement from a neighbor/application;
if (no DOP) execute asynchronous advertisements; update/decay cache;

repeat
  observe the local environment's current state s,
  select and execute some legal action (local, delegation or discovery action) in state s
      using a probabilistic action selection strategy;
  receive a cost g followed by an observed transition to a state s';
  update the state transition model;
  execute the model-based distributed learning algorithm;
  execute asynchronous advertisements;
  decay cached delegation costs;
until DOP solved locally or DOP delegated
```

Fig. 5. Pseudo-Code for the CRL algorithm.

5.5 System optimization in CRL

The system optimization problem in CRL involves minimizing the cost of solving all DOPs in the system, that is, terminating all MDPs at all CRL agents. The system optimization problem is defined as minimizing the total cost of solving all DOPs

$$\sum_{i=0}^{K-1} \sum_{j=0}^{Mi-1} g(n_i, DOP_j) \tag{14}$$

, where K is the number of agents in the system and Mi is the number of DOPs to be solved at agent i. When DOPs are not competing for finite resources in the system and the network environment is stable, the DOPs can be optimized using local information and cached views of neighbors. However, when there are several agents attempting to solve DOPs concurrently at neighboring agents or the network environment is dynamic, the local estimated DOP solution cost at agent i quickly becomes stale, and agents need feedback from their neighbors to improve the quality of their local models.

5.6 Non-Markovian aspects in CRL

There are several heuristics in CRL that seek to overcome non-Markovian properties of distributed systems. For example, the DOP can have a local memory with information such as the list of currently visited agents, and a timeout value for when the DOP becomes invalid due to real-time constraints. The DOP memory can ensure that DOPs do not traverse loops, something MDP cannot prevent. The timeout value for a DOP could be added to an agent's MDP state space, although it could reduce the potential for collective learning of advertised V-Values, as the advertised values would now include a timeout value.

6. Load balancing experiments using CRL

We now define the load balancing problem for CRL. These experiments extend some previous work on load balancing using CRL (Dowling, 2004). We define the load balancing problem as a set *J* of *n* loads, and a set *M* of *m* hosts, where the goals are to store all load in the system (maximize resource utilization) in as short a time as possible (minimize the number of messages sent when storing the load), as well as to balance load equally among agents in the system. These last two goals can conflict, as storing loads as quickly as possible may not equalize load in the system. In our problem, we assume that any load can be potentially stored at any agent, subject to available local storage at the agent. We simplify the network cost problem by assuming that all network connections have equal latency. The loads have an entry point at some host (or hosts) in system; a load is routed from an entry point to a host that handles the load, constrained by the network topology of the system, which may contain cycles.

In our experiments, we define two different cost models for CRL that only differ by their cost models: in the *discrete cost model*, a successful store action for a load has no cost and an unsuccessful store action has some fixed high cost; in the *gradient cost model,* a successful store action has a cost that is a function of the spare capacity at the agent and an unsuccessful store action has some fixed high cost. In both models, we assigned a fixed connection cost for forwarding a load to a neighboring agent (in a real network, the connection cost could be measured from the underlying network message sending cost). The purpose of these two different CRL models is to show how the algorithm can be tuned exploit the first available space at agents (discrete model will prefer the first agent with spare capacity), or to exploit space at higher capacity agents (gradient cost model will prefer agents with higher spare capacity).

We ran our experiments in a discrete event simulator written in Java. The experiments define a topology of agents with connections between them, and a set of DOPs, where each DOP is a load storage problem. The unit of time in the experiments corresponds to a step in the simulator, where a step involves the execution of an action (storage, delegation or discovery) by an agent.

6.1 Experiment 1: Grid topology: Balancing of load over 48 homogenous agents and 2 server agents

The goal of this experiment is to evaluate whether agents can exploit their total load capacity (maximize resource usage), and minimize the amount of messages sent between agents (minimize time required to store all load). Sub-goals include evaluating how the different CRL policies exploit the higher load capacity at two servers in topology, and how well load is equalized among agents in the system. The topology in this experiment is a Grid; there are 48 agents with a capacity of 20 units, and 2 server agents with a capacity of 200 units. Each agent has 10 neighbors, where each agent *i* is connected to neighbors ((i+1) mod 50, … , (i+9) mod 50). The servers are placed at positions 47 and 48 in the grid, with the starting position at 0. Load units are sent into the system via the agent at position 0. We compare the two different CRL policies (CRL-discrete and CRL-gradient) with both a random policy (that selects a random action), and a dynamic programming (DP) solution that performs a breadth-first balancing of the load from the entry point. The CRL policies use a Boltzmann action selection strategy with a temperature parameter used to control the ratio of explorative to exploitative actions (Sutton and Barto, 1998). The results for the CRL and random policies were averaged over 5 experimental runs. The configuration parameters for the CRL policies are given in *Table 1*, below.

Parameter	**Value**
initial Q-Values for storage/discovery actions	-3
initial Q-Values for delegation actions	(-3*numNeighbours) + connection cost
connection cost	-10
MinQValue	-200
MaxQValue	0
delegation success reward	-2
cache decay rate	0.01
action store success reward for CRL-gradient	(MinQValue / (2 * (Capacity – CurrentLoad))
action store success reward for CRL-discrete	0
action store failure CRL-gradient/CRL-discrete	MinQValue
Temperature	0.9

Table 1. CRL-gradient and CRL-discrete configuration parameters.

As illustrated in

Fig. 6, all the policies successfully exploit the available capacity in the system. The random policy eventually finished after ~120,000 time steps. The dynamic programming (DP) policy is near optimal with respect to the number of time steps (i.e., actions executed) required to store the load. The CRL-discrete policy is also near optimal, as it is an exploitative policy that favors storing load over delegating load until an agent has reached its maximum capacity. The CRL-gradient policy is also near optimal until there is roughly 5% capacity left in the system, after which it performs similar to the random policy. This is because, at 5% spare capacity, the advertised V-Values from neighbors with a spare capacity reach the MinQValue, and DOPs effectively have to do a random walk to find the agents with spare capacity. This effect can be seen in Fig. 9.

In *Figures 7-10*, we can see how long the different policies take to discover and exploit the servers. Fig. 11 shows how well the loads are equalized among agents. The CRL and DP policies are more exploitative and tend to fill agents capacities sequentially from agent 0, leading to suboptimal load equalization, but they still compare favorably with the Random policy. The measure we used to determine load equalization is the standard deviation of agent loads from the mean agent load (in terms of percentage of capacity used) at agents.

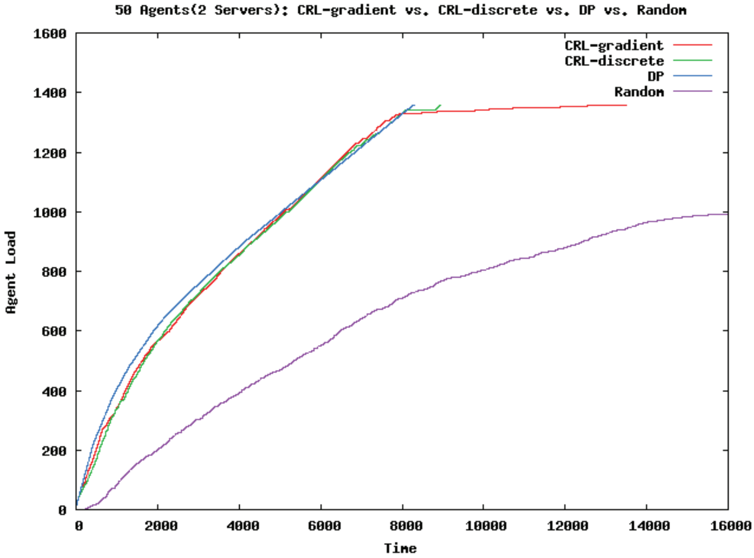


Fig. 6. Grid Topology: Load Balancing in a static topology with 48 agents and 2 servers (at positions 48, 49 in the grid) with CRL-gradient, CRL-discrete, Dynamic Programming and a random policy. Resource utilization is eventually maximized by all policies, while message passing is near-optimally minimized in both DP and CRL-Discrete.

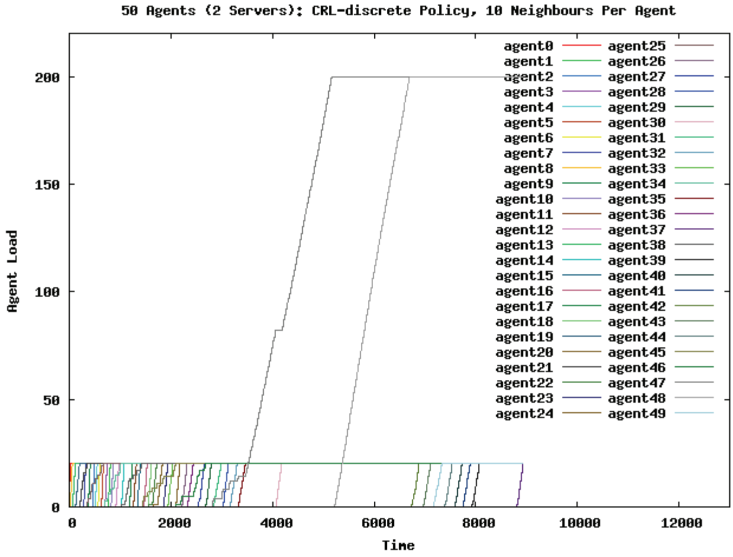


Fig. 7. Grid Topology: CRL-Discrete Policy. This exploitative policy favors storage actions at agents that are not fully loaded. Notice how agents are filled almost sequentially from the source of the load, agent 0.

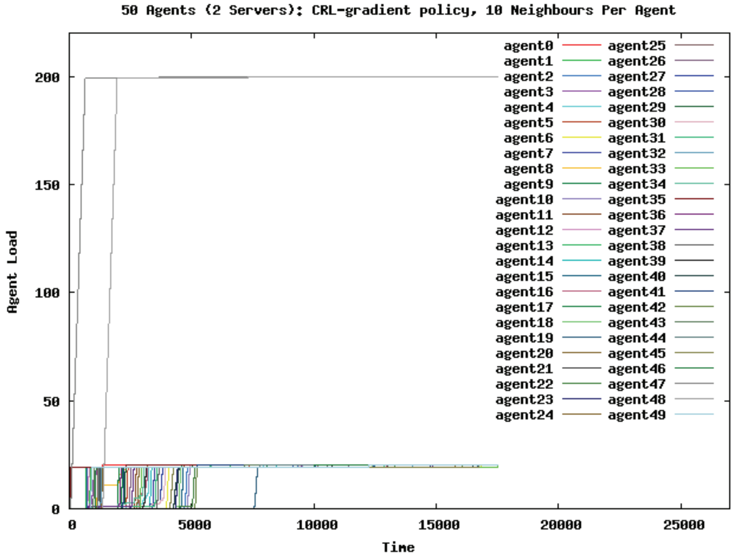


Fig. 8. Grid Topology: CRL-Gradient Policy. This policy favors storage actions on servers, but becomes random when agents are almost fully loaded. Notice how quickly the server agents are almost discovered and exploited, even though they are 3 hops from agent 0.

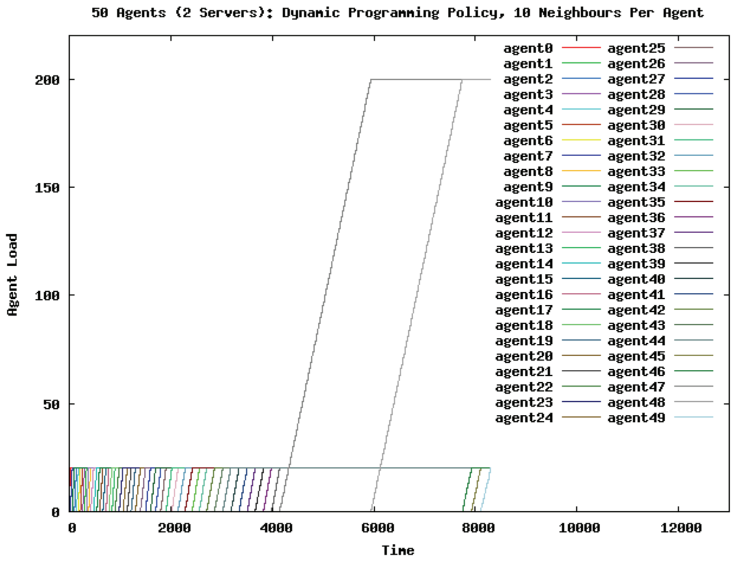


Fig. 9. Grid Topology: Dynamic Programming policy. Notice the similarity of this exploitative policy to the CRL-Discrete Policy.

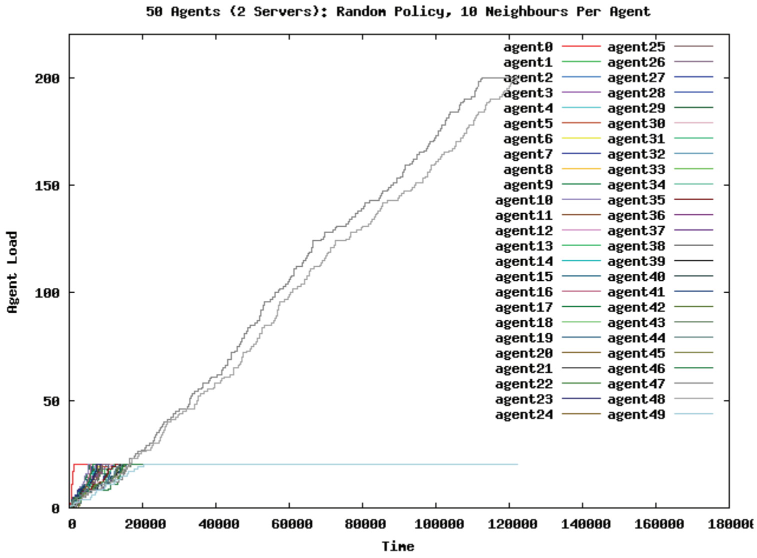


Fig. 10. Grid Topology: The random policy took over 15 times longer than the DP and CRL-Discrete policies to store all the load.

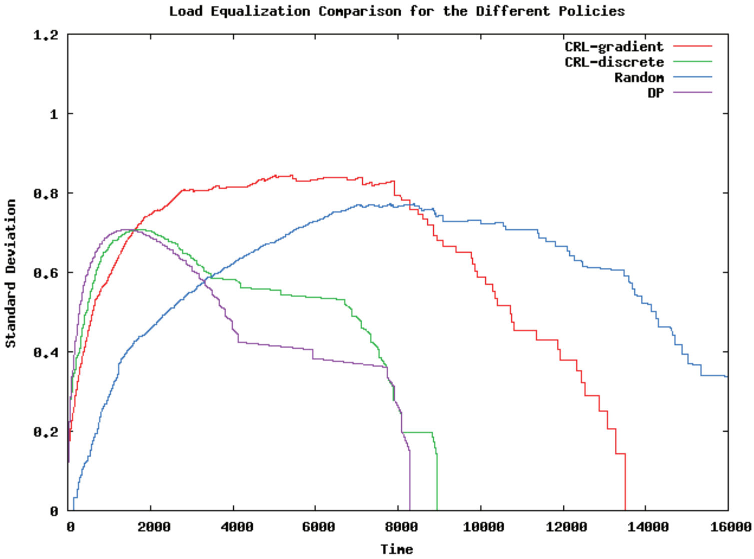


Fig. 11. Grid Topology: Standard Deviation of the Percentage of Load Capacity used at Agents for DP, CRL-Discrete, CRL-Gradient and Random Policies.

6.2 Experiment 2: Random topology: Balancing of load over 48 homogenous agents and 2 server agents

The goal of this experiment is, again, to maximize resource usage, and minimize message overhead, but this time in a random graph topology. In this topology, there are 48 agents with a capacity of 20 units, and 2 server agents with a capacity of 200 units. Each agent has 10 different neighbors, randomly distributed from the set of available agents, but where an agent cannot be a neighbor to itself. The servers are placed at positions more than 1-hop away from the starting position at 0. Load units are sent sequentially into the system via the agent at position 0. The same random topology was used to evaluate all policies, and the results were averaged over 5 experimental runs.

Again, we compare the two different CRL policies (CRL-discrete and CRL-gradient) with a random policy (that randomly selects an action from the local store action and the set of delegation actions), but the DP policy was not included as it does not finish, due to the presence of cycles in the topology. The CRL configuration parameters were the same as in experiment 1, with the exception of the Temperature, which was set to 0.85 for slightly increased exploration.

In *Fig. 12*, we can see that the CRL-Discrete policy is very effective at exploiting all available load in the system, while the CRL-Gradient policy again becomes a roughly random policy for the last ~5% spare capacity in the system. The Random Policy took an increased amount of time, over the grid topology, and terminated at time step ~150,000.

In another experiment, we added some dynamism to the environment by adding an extra server to the system when the system's load capacity was full (see *Fig. 13*). At the same time step, we added a discovery action to the 1-hop neighbors from agent 0 to enable them to discover the server (located at a 2-hop distance from the source agent 0). As can be seen, the

CRL-discrete policy quickly discovers and exploits the new server, while the CRL-gradient policy takes a longer time to do so. There was also quite high variation in how long the CRL-gradient policy took to discover the new server, indicating a high level of randomness in its exploration. The upward spike near the end of CRL-gradient policy was a point where collaborative feedback (that is, advertisements) from the new server eventually influenced reached the origin agent 0.

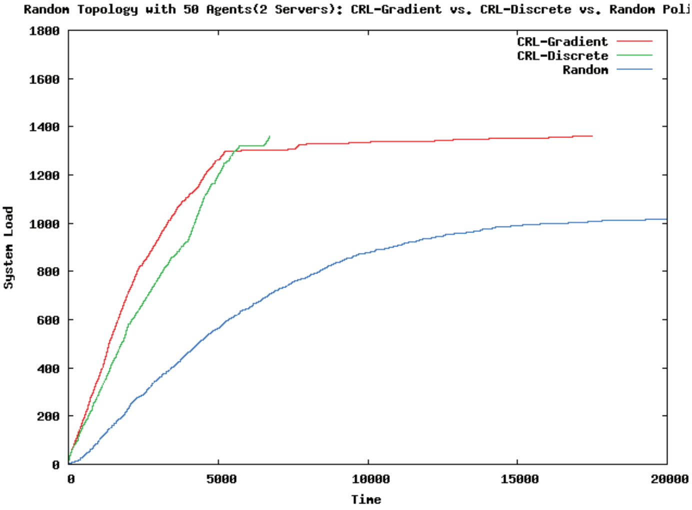


Fig. 12. Random Graph: Load Balancing for CRL-gradient, CRL-discrete and Random Policies.

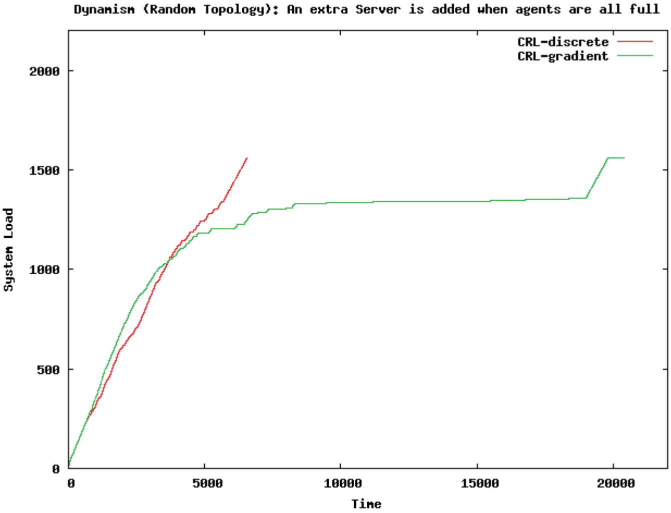


Fig. 13. Dynamism in the Random Topology: a new server was added, when the 50 agents were full, and was quickly exploited by the CRL-Discrete policy, less so by CRL-Gradient.

6.3 Discussion of experiments and CRL

The experiments show how CRL can be used to build a system that adapts to a dynamic environment. Agents interact with their local environment by storing loads and receiving feedback on available local storage capacity. Agents interact with their neighbors by delegating load storage to them, receiving estimated costs for neighbors' storing loads. Through delegating load and locally storing load, agents collaboratively provide a load balancing service that is robust, adaptive, and can learn about and exploit new agents introduced into the system. The experiments could easily be extended to improve performance by adding asynchronous advertisements and adding heuristics, for example, adding memory to the DOP of the list of agents already visited prevent DOPs entering network loops.

CRL, itself, is an approximate approach to online, decentralized reinforcement learning. It has similarities with population-based techniques such as ACO, particle swarm intelligence (Kennedy and Eberhart, 2001) and evolutionary computing: the system takes a diverse set of DOPs as input, and it reinforces the selection of agents that were successful at solving the DOPs given the state of the system environment; this process improves system utility for a stable environment and can also adapt a system to better match its changing environment. Rather than having agents die and be replaced by fitter agents, CRL agents decay their solutions to purge the system of stale information and use collaborative feedback to cooperatively learn new solutions.

7. Future of distributed reinforcement learning

Distributed reinforcement learning is an emerging field that offers the promise of enabling the construction of distributed systems that can adapt and optimize their operation online.

Existing approaches to distributed reinforcement learning include multi-agent control of a single MDP that describes system behavior, and decentralized approaches where agents are independent learners that collaborate to provide system services and collectively learn from one another to build local policies that improve system utility.

Designers of distributed reinforcement learning algorithms should give careful consideration to real-world properties of distributed systems, such as the high cost of message passing, and the possibility of failure for both agents and network connections. As a proof of concept, in this chapter, we showed how collaborative reinforcement learning can be used to build a load balancing system that can adapt to a dynamic environment.

In the future, we anticipate that distributed reinforcement learning algorithms will be increasingly applied in a variety of domains, from large-scale grid computing systems, to optimize resource usage, to small-scale wireless and sensor networks, where power usage and radio transmission usage should be minimized. In both cases, the goal of distributed reinforcement learning will be to replace existing parametric models with online learning models that can demonstrate improved adaptation to dynamic environments.

8. Acknowledgements

This research was supported by a Marie Curie Intra-European Fellowship within the 6th European Community Framework Programme. The authors would like to thank Jan Sacha for an implementation of CRL in Java on which the experiments in this paper are based.

9. References

Bernstein, D.; Zilberstein, S.; Immerman, N. (2002). The Complexity of Decentralized Control of Markov Decision Processes. *Proceedings of the Conference on Uncertainty in Artificial Intelligence,* pp. 32-27, Morgan Kaufmann Publishers Inc., San Francisco.

Bowling, M.; Veloso, M.. (2000). An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning. Technical Report, Carnegie Mellon University, (CMU-CS-00-165), January 2000.

Chang, Y-H; Ho, T; Kaebling, L. (2003). All learning is local: Multi-agent learning in global reward games , *Advances in Neural Information Processing Systems*, MIT Press, ISBN .

Cogill, R; Rotkowitz, M; Van Roy, B.;Lall, S. (2006). An Approximate Dynamic Programming Approach to Decentralized Control of Stochastic Systems, *Control of Uncertain Systems*, pp. 243–256, LNCIS 329, Springer-Verlag, Berlin.

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. American Association for Artificial Intelligence.

Curran, E. (2003). Swarm: Cooperative Reinforcement Learning for Routing Ad-hoc Networks, *MSc. Thesis*, Dept. of Computer Science, Trinity College Dublin, October 2003.

Dorigo, M; Stützle, T. (2004). Ant Colony Optimization, MIT Press, ISBN-10: 0262042193.

Dowling, J.; Curran, E.; Cunningham, R.; Cahill, V. (2005). Using Feedback in Collaborative Reinforcement Learning to Adaptively Optimise MANET Routing, *IEEE Transactions on Systems, Man and Cybernetics (Part A), Special Issue on Engineering Self-Orangized Distributed Systems*, pp. 360-372, Vol. 35, No. 3, ISSN 0018-9472.

Dowling, J. (2004). The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems, *PhD Thesis*, Dept of Computer Science, Trinity College Dublin.

Ghasemaghaei, R.; Rahman, Md. A.; Gueaieb, W.; El Saddik, A. (2007). Ant Colony-Based Reinforcement Learning Algorithm for Routing in Wireless Sensor Networks. *In Proceedings of IEEE Conference on* Instrumentation and Measurement Technology Conference, pp. 1-6, Warsaw, Poland, ISBN 1-4244-0588-2.

Goldman, C.; Zilberstein, S. (2004). Decentralized Control of Cooperative Systems: Categorization and Complexity Analysis, *Journal of Artificial Intelligence Research*, pp. 143-174, Vol. 22, ISSN 11076 – 9757.

Guestrin, C; Koller, D.; and Parr, R.; (2002). Multiagent Planning with Factored MDPs, *In Advances in Neural Information Processing Systems*, pp. 1523-1530, Vancouver, Canada.

Guestrin, C.; Koller, D.; Parr, R.; Venkataraman, S. (2003). Efficient Solution Algorithms for Factored MDPs, *Journal of Artificial Intelligence Research,* pp. 399-468, Vol. 19, ISSN 11076 – 9757.

Hu, J; Wellman, M. (2003). Nash q-learning for general-sum stochastic games, *Journal of Machine Language Research*, pp. 1039-1069, vol. 4, MIT Press (Cambridge), ISSN 1533-7928.

Kennedy, J; Eberhart, R. (2001). Swarm Intelligence, Morgan Kaufman, ISBN:1-55860-595-9.

Kok, J.; Vlassis, N. (2006). Collaborative Multiagent Reinforcement Learning by Payoff Propagation, *Journal of Machine Language Research,* pp. 1789-1828, Vol. 7, MIT Press, ISSN 1533-7928.

Lauer, M.; Riedmiller, M. (2000). *An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems.* In Proceedings of the 17th International Conf. on Machine Learning, pp. 535-54, Morgan Kaufmann Publishers Inc., San Francisco.

Lawson, J.; Wolpert, D. (2002). The Design of Collectives of Agents to Control Non-Markovian Systems, *In Proceedings of the National Conference on Artificial intelligence*, pp. 332-337, American Association for Artificial Intelligence, ISBN: 0-262-51129-0.

Martin, J.A.;De Lope, H. (2007). A Distributed Reinforcement Learning Architecture for Multi-Link Robots, *4th International Conference on Informatics in Control, Automation and Robotics*, pp. 192-197, INSTICC Press 2007, ISBN 978-972-8865-82-5.

Ng, A; Kim, H-J.; Jordan, M.; Sastry, S. (2004). Autonomous helicopter flight via reinforcement learning, *Advances in Neural Information Processing Systems*, MIT Press, ISBN 0-262-20152-6.

Peshkin, L.; Kim, K.; Meuleau, N.; Kaelbling, L. (2000). Learning to Cooperate via Policy Search. *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence,* pp. 489-496, Morgan Kaufmann Publishers Inc., San Francisco.

Rigole, P; (2006). Task- and resource-aware component deployment in ambient intelligence environments, *Ph.D. Thesis*, Department of Computer Science, K.U. Leuven, Leuven, Belgium, November, 2006.

Schneider, J.; Wong. W; Moore, A.; Riedmiller, M. (1999). Proceedings of the Sixteenth International Conference on Machine Learning, pp. 371-378, Morgan Kaufmann Publishers Inc., San Francisco, ISBN:1-55860-612-2.

Shoham, Y. ; Powers, R.; Grenager, T. (2003). Multi-agent reinforcement learning: a critical survey, Technical report, Computer Science Department, Stanford University.

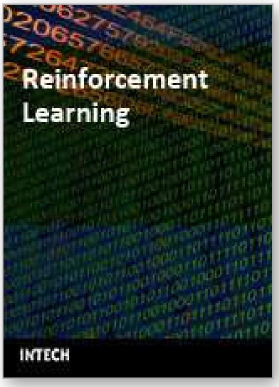
Sutton, R.; Barto, A. (1998). Reinforcement Learning: An Introduction, *MIT Press*, ISBN 0-262-19398, Cambridge MA.

Tesauro, G. (2007). Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies, *IEEE Internet Computing*, pp. 22-30, Vol. 11, No. 2, ISSN 1089-7801.

Tsitsiklis, J; Athans, M.. (1985). On the complexity of decentralized decision making and detection problems," *IEEE Trans. Automatic Control*, vol. 30, no. 5, pp. 440–446.

Van Renesse, R., Birman, K., Vogels, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computing Systems,* pp.164–206, Vol. *21*, No. 2, ISSN: 0734-2071.

Zhang, Y; Liu, J.; Zhao, F. (2006). Information-Directed Routing in Sensor Networks Using Real-Time Reinforcement Learning, In Combinatorial Optimization in Communication Networks, pp. 259-289, Kluwer Academic Publishers.



Reinforcement Learning

Edited by Cornelius Weber, Mark Elshaw and Norbert Michael Mayer

ISBN 978-3-902613-14-1
Hard cover, 424 pages
Publisher I-Tech Education and Publishing
Published online 01, January, 2008
Published in print edition January, 2008

Brains rule the world, and brain-like computation is increasingly used in computers and electronic devices. Brain-like computation is about processing and interpreting data or directly putting forward and performing actions. Learning is a very important aspect. This book is on reinforcement learning which involves performing actions to achieve a goal. The first 11 chapters of this book describe and extend the scope of reinforcement learning. The remaining 11 chapters show that there is already wide usage in numerous fields. Reinforcement learning can tackle control tasks that are too complex for traditional, hand-designed, non-learning controllers. As learning computers can deal with technical complexities, the tasks of human operators remain to specify goals on increasingly higher levels. This book shows that reinforcement learning is a very dynamic area in terms of theory and applications and it shall stimulate and encourage new research in this field.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821