Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Μηχανισμοί ελαστικής κατανομής πόρων σε περιβάλλοντα υπολογιστικών νεφών με χρήση τεχνικών ενισχυτικής μάθησης.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Χ. Κονταρίνης

Επιβλέπων : **Ν**εκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάιος 2015

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Μηχανισμοί ελαστικής κατανομής πόρων σε περιβάλλοντα υπολογιστικών νεφών με χρήση τεχνικών ενισχυτικής μάθησης.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Χ. Κονταρίνης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Μαΐου 2015.

....	....	....
Νεκτάριος Κοζύρης	Κωνσταντίνος Κοντογιάννης	Βασιλική Καντερέ
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Maitre Assistante
		Université de Genève

Αθήνα, Μάιος 2015

....
Αλέξανδρος Χ. Κονταρίνης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Αλέξανδρος Κονταρίνης, 2015.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Το Cloud Computing, χάρις στην κεντρική του ιδέα περί κοινής χρήσης φυσικών πόρων από πολλαπλούς χρήστες, καθιστά δυνατή την κατά βούληση, ομαλά κλιμακούμενη, αυξομείωση των χρησιμοποιούμενων υπολογιστικών πόρων. Αυτή η ιδιότητα είναι η λεγόμενη “Ελαστικότητα”, η οποία δίνει την δυνατότητα, στους χρήστες ενός νέφους (cloud) να έχουν πρόσβαση σε φαινομενικά άπειρους διαμοιραζόμενους πόρους, ενώ στους παρόχους ενός νέφους να βελτιστοποιούν την χρησιμοποίηση των κέντρων δεδομένων (datacenters) τους και να μεγιστοποιούν τα κέρδη τους. Ο ακριβής έλεγχος των υπολογιστικών πόρων ανοίγει τον δρόμο για την εφαρμογή δυναμικών πολιτικών δέσμευσής τους, οι οποίες θα ανταποκρίνονται επακριβώς και ανά πάσα χρονική στιγμή, στον φόρτο εργασίας και στις ανάγκες των εκάστοτε εφαρμογών και υπηρεσιών που εκτελούνται στο νέφος. Δυστυχώς, στα περιβάλλοντα υπολογιστικών νεφών, τα φορτία είναι ως επί το πλείστον δυναμικά και ετερογενούς φύσεως, όπως άλλωστε και οι ίδιοι οι πόροι. Ως εκ τούτου, προκύπτουν ποικίλλα πολύπλοκα προβλήματα διαχείρισης των υπολογιστικών πόρων, συχνά σχετιζόμενα με την κατανομή και την χρονοδρομολόγησή τους. Εξαιτίας της υψηλής πολυπλοκότητας και των αυστηρών απαιτήσεων αυτών των προβλημάτων, οι παραδοσιακοί αλγόριθμοι επίλυσής τους δεν επαρκούν, και η ερευνητική κοινότητα εξετάζει καινοτόμες προσεγγίσεις από διαφορετικά επιστημονικά πεδία.

Στην παρούσα εργασία μελετούμε αλγορίθμους στους οποίους, ένας ευφυής πράκτορας αποφασίζει για την λήψη ενεργειών βασιζόμενος σε πολλαπλά κριτήρια, και ο οποίος παρατηρώντας τις επιβραβεύσεις που λαμβάνει ως απόκριση του περιβάλλοντος στις ενέργειές του, μαθαίνει να επιλέγει τις ενέργειες εκείνες που βελτιστοποιούν την απόδοσή του. Αλγόριθμοι αυτού του τύπου ενδείκνυνται εν γένει για προβλήματα ακολουθιακής λήψης αποφάσεων, σε πραγματικό χρόνο, υπό καθεστώς αβεβεβαιότητας. Συγκεκριμένα, εξετάζουμε μία σειρά από αλγορίθμους, μεθόδους, και τεχνικές, Ενισχυτικής Μάθησης (Reinforcement Learning) και μηχανημάτων τυχερών παιγνίων (Multi-Armed Bandits), για την επίτευξη αποδοτικής προσαρμοστικής δέσμευσης των υπολογιστικών πόρων ενός νέφους. Αντλώντας έμπνευση από ένα αντίστοιχο πλαίσιο διαχείρισης πόρων (TIRAMOLA), ειδικευμένο στην προσαρμογή του μεγέθους NoSQL συστοιχιών που τρέχουν πάνω από IaaS, εκτελούμε προσομοιώσεις σε περιβάλλον MATLAB, ώστε να μελετήσουμε την διαδικασία λήψης αποφάσεων, να εξερευνήσουμε την επίδραση διαφορετικών μοντέλων, και να πειραματιστούμε με τις τιμές διαφόρων παραμέτρων. Υποστηρίζουμε ότι οι προσομοιώσεις είναι ένα αναντικατάστατο εργαλείο κατά την σχεδίαση πραγματικών συστημάτων διαχείρισης πόρων υπολογιστικών νεφών, και ότι τα αποτελέσματά τους καταδεικνύουν σημαντικές πτυχές του προβλήματος που αξίζουν περαιτέρω μελέτης.

Λέξεις-Κλειδιά: Cloud Computing, Ελαστικότητα, Κατανομή / Χρονοδρομολόγηση / Δέσμευση / Διαχείριση Υπολογιστικών Πόρων, Ενισχυτική Μάθηση, Q-Learning, Multi-Armed Bandit Learning, UCB, Thompson sampling, MATLAB, Προσομοίωση

Abstract

Cloud Computing, thanks to its core idea of multiple users sharing the same physical resources, has made possible to scale computing resources up and down, at will, and with minimal friction. This property, known as resource “Elasticity”, enables cloud clients to effortlessly access a seemingly infinite shared pool of such resources, while cloud providers can precisely optimize resource utilization in their data-centers, as well as maximize their profits. This improved resource control paves the way for dynamic provisioning policies, in an effort to precisely meet the actual workload and requirements of the running applications and services, at any given point in time. Unfortunately, in a cloud environment, workloads are usually of a dynamic and heterogeneous nature. As a result, a variety of complex resource management problems has emerged, often related to resource allocation and scheduling. There is on-going research aimed at tackling those problems, using state-of-the-art methods originating from diverse scientific fields, especially since traditional algorithms do not cope well with the higher complexity and stronger requirements of those problems.

In this thesis, we study algorithms in which, an intelligent agent decides which actions to take based on multiple criteria, and by observing the rewards it receives as an environmental response to those actions, learns – at runtime - to improve its decision-making in a way that optimizes its performance. Online learning algorithms of that type are a seemingly good fit for real-time decision-making problems under uncertainty. In specific, we examine a range of Reinforcement Learning and Multi-Armed Bandit algorithms, methods, and techniques, for achieving efficient adaptive cloud resource provisioning. We draw inspiration from an existing resource management framework (TIRAMOLA), specialized in the resizing of NoSQL clusters that run over IaaS, and run simulations in MATLAB in order to, delve deeper into the various aspects of the decision-making procedure, explore the effect of different modeling approaches, and experiment with the algorithms' learning parameters' values. We argue that simulation tools are imperative for designing a cloud-based resource management system, and that simulation results highlight important issues, worthy of further study when designing a similar real-world system.

Keywords: Cloud Computing, Elasticity, Computational Resource Allocation / Scheduling / Provisioning / Management, Reinforcement Learning, Q-Learning, Multi-Armed Bandit Learning, UCB, Thompson sampling, MATLAB, Simulation

Contents

Cover i

Abstract v

Contents vii

Abbreviations x

1 Introduction 1
- 1.1 The Cloud . . . 1
  - 1.1.1 Cloud computing . . . 1
  - 1.1.2 Cloud elasticity . . . 3
  - 1.1.3 Cloud resource types . . . 5
  - 1.1.4 Related Work . . . 6
- 1.2 Motivation . . . 9

2 Reinforcement Learning 12
- 2.1 Inspiration . . . 12
- 2.2 Mathematical origins . . . 15
  - 2.2.1 Markov Decision Processes . . . 15
  - 2.2.2 Markov Decision Problem . . . 17
  - 2.2.3 Dynamic Programming . . . 21
  - 2.2.4 Value Iteration . . . 22
  - 2.2.5 Policy Iteration . . . 23
  - 2.2.6 Optimistic Policy Iteration . . . 24
  - 2.2.7 Value Iteration versus Policy Iteration . . . 25
- 2.3 Formal Setting . . . 26
- 2.4 From Dynamic Programming to Reinforcement Learning . . . 29
  - 2.4.1 Sample-Based Learning . . . 30
  - 2.4.2 Sample-Based Policy Evaluation . . . 32
- 2.5 Algorithms and Techniques . . . 35
  - 2.5.1 Q-Learning . . . 35
  - 2.5.2 SARSA . . . 37
  - 2.5.3 R-max . . . 38
- 2.6 Applications . . . 40

3 Multi-Armed Bandits 41
- 3.1 Inspiration . . . 41

3.2 Mathematical Origins . . . 42
3.2.1 Probabilistic Models . . . 42
3.2.2 Confidence versus Credibility . . . 43
3.2.3 Relation to Markov Decision Processes . . . 45
3.3 Variations . . . 46
3.4 Formal setting . . . 47
3.5 Exploration versus Exploitation . . . 50
3.6 Algorithms and techniques . . . 53
3.6.1 Gittins Index . . . 53
3.6.2 Thompson Sampling . . . 53
3.6.3 Upper Confidence Bound . . . 55
3.6.4 Bayes Upper Confidence Bound . . . 56
3.6.5 Minimum Empirical Divergence . . . 57
3.6.6 Kullback-Leibler Upper Confidence Bound . . . 58
3.7 Applications . . . 60

4 Resource Allocation Modeling 61
4.1 Why Reinforcement Learning? . . . 61
4.2 Why Multi-Armed Bandits? . . . 62
4.3 TIRAMOLA . . . 63
4.3.1 TIRAMOLA's decision-making . . . 63
4.3.2 TIRAMOLA's decision-making proposal . . . 67
4.4 Problem Formulation . . . 68
4.4.1 General Remarks . . . 68
4.4.2 General Setting . . . 69
4.4.3 Full Reinforcement Learning Setting . . . 70
4.4.4 Multi-Armed Bandit Setting . . . 72

5 Simulation Results 74
5.1 RL Experiment Sets . . . 74
5.1.1 RL Experiment Set 1: The effect of the time-horizon under a small static workload . . . 75
5.1.2 RL Experiment Set 2: The effect of the time-horizon under a big static workload . . . 79
5.1.3 RL Experiment Set 3: The effect of optimistic initial conditions . . . 84
5.1.4 RL Experiment Set 4: The effect of the learning rate . . . 88
5.1.5 RL Experiment Set 5: The effect of the time-horizon under a dynamic workload . . . 92
5.1.6 RL Experiment Set 6: The effect of the learning rate under a dynamic workload . . . 97
5.1.7 RL Experiment Set 7: The effect of the workload amplitude and frequency . . . 102
5.1.8 RL Experiment Set 8: Adding a state dimension under a dynamic workload . . . 105
5.1.9 RL Experiment Set 9: The effect of the workload amplitude and frequency . . . 109
5.2 MAB Experiment Sets . . . 112
5.2.1 MAB Experiment Set 1: The effect of greediness . . . 112

5.2.2 Experiment Set 2: Gradually increasing greediness 117
5.2.3 Experiment Set 3: Taking into account the reward variance 124

6 Conclusions and Future Work **130**
6.1 Conclusions 130
6.2 Future Work 133
6.2.1 Approximate Q-Learning 134
6.2.2 Contextual Multi-Armed Bandits 134

Bibliography **136**

Abbreviations

SaaS	**S**oftware **as a** **S**ervice
PaaS	**P**latform **as a** **S**ervice
IaaS	**I**nfrastructure **as a** **S**ervice
OS	**O**perating **S**ystem
PM	**P**hysical **M**achine
VM	**V**irtual **M**achine
DBMS	**D**atabase **M**anagement **S**ystem
SLA	**S**ervice **L**evel **A**greement
RL	**R**einforcement **L**earning
ADP	**A**pproximate **D**ynamic **P**rogramming
ML	**M**achine **L**earning
SL	**S**upervised **L**earning
UL	**U**nsupervised **L**earning
DP	**D**ynamic **P**rogramming
VI	**V**alue **I**teration
PI	**P**olicy **I**teration
Q-VI	**Q** **V**alue **I**teration
TDL	**T**emporal **D**ifference **L**earning
MDP	**M**arkov **D**ecision **P**rocess
BOE	**B**ellman **O**ptimality **E**quation
BPE	**B**ellman **P**olicy **E**quation
MAB	**M**ulti **A**rmed **B**andit
SS-MAB	**S**tandard **S**tochastic **M**ulti **A**rmed **B**andit
RS-MAB	**R**estless **S**tochastic **M**ulti **A**rmed **B**andit
CS-MAB	**C**ontextual **S**tochastic **M**ulti **A**rmed **B**andit
CoI	**Co**nfidence **I**nterval
CrI	**Cr**edibility **I**nterval

Chapter 1

Introduction

1.1 The Cloud

Cloud computing[1] is in a way the culmination of Joseph Carl Robnett Licklider's early vision of an "intergalactic computer network" that dates back to 1963, and of John McCarthy's idea of "computing organized as a public utility" that dates back to 1961. A long time had to pass however before cloud computing started taking its modern form. Its main driving force was the Internet explosion and subsequent offering of enough bandwidth in the 1990s.

1.1.1 Cloud computing

Cloud computing is an over-the-Internet delivery model of computing resources and services, and "the Cloud" itself refers to the hardware infrastructure and software platforms of data centers that can turn this model into a reality. Cloud computing is still an evolving paradigm and therefore not easily defined, but it is intertwined with the notion of ubiquitous, on-demand, effortless access to a shared pool of resources, such as networks, servers, software applications, and many more. During the last decade, cloud computing has shaped a big part of the Information Technology industry, by virtually eliminating capital costs (e.g. building a server farm), by allowing businesses to outsource hardware and software maintenance, by freeing users from having to install and update all kinds of software on their local devices, by bringing High-Performance Computing power to consumer-oriented applications, etc. This is despite also giving rise to legitimate new

[1] The term was probably originally coined for marketing purposes, in a November 1996 Compaq Computer analysis entitled "Internet Solutions Division Strategy for Cloud Computing", and later popularized by Google in 2006.

risks, both for the organizations (production halting when access to the cloud is lost, getting locked-in to a specific cloud vendor, etc.) and for the individual user (personal data and information physically residing in remote servers, the cloud provider having complete access to them, dependence on the reliability of the Internet connection, etc.).

In addition, the Cloud has enabled delivery not just of applications, as the term "Software as a Service (SaaS)" alludes to, but of lower-level services as well, evidently suggested by the related nomenclature: "Platform as a Service (PaaS)" is a model that provides the computing platforms (such as programming languages, services, and tools) over which cloud customers can run their own high-level applications (whether developed by them or acquired from third-parties), essentially outsourcing network, server, operating system, and storage maintenance to the cloud provider, and focusing only on deploying and configuring their code and data. Similarly, "Infrastructure as a Service (IaaS)" is the "self-service" offering of physical or virtual machines and other resources, so that cloud customers may not only manage their applications, but also the middleware and Operating System (OS) over which those will run, and so that they may install and run any arbitrary software they deem fit. "Bare-metal" clouds are an emerging type of IaaS clouds, which lease physical machines lacking any virtualization layers. This is beneficial for some resource-intensive applications, but sacrifices traditional IaaS features.

In reality, all cloud resources emanate from the hundreds or thousands of machines that are, usually in physical proximity, connected via high speed networks, and operated by a single administrative entity (comprising a cluster), or sometimes even geographically distributed and connected via networks of varying speed[2]. Actually, the creation of extremely large-scale data centers based on commodity-hardware might itself have been the main reason why cloud computing has grown so wide beyond its original community, because it brought forth the benefits of the economies of scale [1].

Only data centers of a significant size can be considered to constitute a cloud. Moreover, there are public clouds and private clouds, depending on who has access to them: the general public or some specific organization. The distinction is irrelevant to who is operating and managing the cloud, and only describes who is using it. For example, a private cloud can be managed by a third-party company for exclusive use by one of its client enterprises. As of late, hybrid clouds have been gaining popularity, a model that

[2] In this case cloud computing is related to grid computing, a type of distributed and parallel computing, which uses loosely coupled heterogeneous computers working in synchronization to perform large common tasks.

combines private data centers with public cloud services into a single unified computing environment, usually involving frequent data movement between the two.

1.1.2 Cloud elasticity

An IaaS cloud provider hosts a whole range of infrastructure components (and also handles tasks such as system backup, resiliency planning etc.) for its clients. The monetary cost ultimately incurred to the cloud clients reflects the exact amount of resources allocated and consumed. Cloud computing in general has adopted this "pay-as-you-go"[3] paradigm, wherein the clients pay a fee corresponding to the number of transactions realized, or corresponding to the duration of utilization in days, hours, or even minutes.

The main motive for having such charging schemes is that clients obviously prefer to pay only for when resources are actually handling workloads (and not while they are remaining idle). Unfortunately, workloads are not always known in advance, nor are they necessarily uniform in time either. In general, cloud clients have two major objectives: to minimize the monetary cost and the execution time of their requests.
Providers on the other hand, certainly wish to offer competitive pricing, but at the same time they also want to maximize their return on investment. This leads to a need for sharing the cloud's physical resources between multiple cloud clients, because if resources are indeed only paid for when being used, then there is strong incentive for providers to minimize their idle time, which roughly means to efficiently "squeeze" many clients into each Physical Machine (PM). In general, cloud providers have two major objectives: to maximize resource utilization and to minimize maintenance costs.

In order to satisfy both providers and clients in the above respect, scalable resources should be allocated and re-allocated dynamically, even during client application execution. Thus, the notion of cloud "elasticity" was introduced: the capability of a cloud system to autonomously adapt to workload changes and specific requirements, as they arise, by accordingly provisioning and de-provisioning its resources. Indeed, cloud providers are moving towards offering the so-called elasticity "on-the-fly". Especially when the switching overhead is insignificant, seamless re-allocation of resources could be even considered in a scale of seconds. The ultimate goal of elasticity is to match (at any given time), as closely as possible, the actual computational demands of the system

[3] Also known as "pay-as-you-use" or "pay-on-demand".

clients[4]. These adjustments ought to depend on as little human intervention as possible, hence, well defined resource management strategies are needed to automate them. Elasticity from the customers' perspective relates to being able to increase or decrease at will the amount of resources requested and received. Again, ideally this procedure would be undertaken automatically by software agents.

To a large extent, this utilitarian view of computing has become possible by leveraging system virtualization technologies. Hardware virtualization is the creation of virtual guest machines that act as normal computational environments equipped with operating systems, when in reality some other real/physical host machine's hardware is being used. The software or firmware layer that makes this possible is called a hypervisor or a Virtual Machine (VM) manager/monitor. The hypervisor handles physical resources and separates them, in order to create dedicated virtual resources that can be dynamically applied whenever and wherever needed. This can eventually lead to improved hardware resource utilization and availability. Thanks to virtualization, several OSs can run in parallel on the same Central Processing Unit (CPU), and multiple VMs can co-exist on the same PM. Recently, containers have emerged as an OS-level virtualization alternative to full system virtualization, providing the application isolation that is required for mutually distrusting tenants of the same node, not by using discrete OS instances, but by relying on the host's kernel and its normal OS system call interface. This way, containers impose very little overhead and can be created within seconds, but on the other hand can not support any OS other than the one run by the host.

As for PaaS clouds, with the advent of Big Data analytics[5] leading to the creation of a plethora of execution engines and data stores[6], each better suited for specific types of computations over specific types of data, modern data centers are now capable of concurrently supporting a variety of heterogeneous platforms. They can host vast amounts of diverse data, spanning over multiple distributed data sources, residing in thousands of nodes. For example, a data center can hold relational row stores, columnar stores, unstructured raw data, semi-strucured XML data, document files, and more. Because mindlessly converting data from one format to another every time would be inefficient, a lot of effort is being put into developing unified data analytics frameworks that will leverage multiple collaborating platforms [2, 3], by truly integrating them and not just

[4]This is why elasticity is one element that differentiates cloud computing from grid computing.

[5]Big Data refers not only to the size of the data, but also to the diversity of their structure and representation.

[6]Repositories/Collections of sets of data objects.

"gluing them together". In such merger frameworks, all operations from query development and job execution to system management, should be handled in a unified manner.

In unified analytics systems, queries might target heterogeneous data residing in diverse data stores. This creates a need for a higher-level software layer to perform resource allocation and scheduling among multiple execution engines and data stores. That resource management layer is responsible for selecting between different data models and different computation models, with respect to the task or job submitted, taking into account performance, costs, restrictions etc. Therefore, it has to be both intelligent and adaptive to perform this kind of "match-making". For the user/programmer, the resource management layer should eventually result in frictionless switching from one platform to the other, without him having to think about which one to use.

The scheduling aspect of the resource sharing problem is related to choosing the order in which multiple user requests, such as jobs or tasks[7] that have been (or could be) mapped to the same resource, should be served. Queues might need to be formed and tasks or jobs might wait for others to finish, before they can start being served. Nevertheless, both "allocation" and "scheduling" are terms often used loosely by cloud engineers, to refer to the more general issue of distributing cloud resources.

To summarize, cloud computing elasticity has made possible to claim benefits previously unattainable (client customization, efficient resource utilization, etc.), but in order to achieve them, a variety of resource allocation and scheduling challenges need to be tackled first, which has yet to be done in a consistent manner.

1.1.3 Cloud resource types

In cloud-based utility computing, the resources provisioned span several different types, some offered more often than others. Compute resources can include: CPUs and CPU cores, threads, etc. Storage resources can include: disk space, Random-Access Memory (RAM) size, etc. Network resources can include: I/O bandwidth, IP addresses, TCP connections etc. Specialized resources can include: General-Purpose Graphics Processing Units (GPGPUs), Field-Programmable Gate Arrays (FPGAs), Solid-State Drives

[7] Each job is composed of one or more (even thousands of) tasks, which are programs requesting execution.

(SSDs), flash memory, elastic IP addresses etc. Higher-level resources can include: number of PMs, number of VMs, applications, services, web servers, database servers, etc.

Specifically when dealing with VMs, workloads are said to either "scale-out" horizontally, or "scale-up" vertically. The former refers to increasing the number of VMs when each is bearing some predetermined capacity, and the latter to increasing the capacity of each VM (perhaps with more CPU or RAM power), a process also known as "VM resizing". Each type of scaling comes with its advantages and disadvantages.

To be able to implement elasticity on-the-fly, the more the workload fluctuates, the smaller the temporal quantum of renting should be, or else a client might end up paying for longer periods of time than needed. For the same reasons, not only time slots, but also resources must be assigned in a fine granularity, or else excessive resources might be committed to a client who does not put them into good use. On the other hand, a wide range of fixed resource bundles provides "good-enough" solutions for the majority, and only a small portion of a cloud provider's clientele would currently pursue sophisticated techniques for optimization in continuous resource spaces. It is therefore not very obvious which exact type of resource offering will prevail in the future.

1.1.4 Related Work

Effectively handling resource provisioning is a multi-faceted challenge spanning across many different layers of abstraction [4]. In this subsection we summarize research work that is indicative of the wide variety of approaches currently under consideration by the research community. Of course, many more architectures, algorithms and techniques are being studied as well.

In [5], the authors examine infrastructure reconfiguration (and its temporal cost in particular) in response to workload variations, for virtualized data centers and clouds, using VM resizing and VM live migration. Having found that VM resizing has a negligible duration, they focus on the factors that make VM live migration costly, both in terms of the migration duration and of the running application's throughput drop. The authors do not propose any particular resource allocation policy, but their observations result in practical recommendations for designing one. They propose scaling up in a proactive way, actively predicting resource bottlenecks, instead of passively reacting to high CPU utilization (or else the migration takes much longer), prefering applications with small

active memory for migration (if there is no resource contention), not using VM live migration for dealing with short-term overloads, etc.

Another work that does not propose any particular resource allocation policy, but instead offers insight into designing one is [6], in which the authors analyze real-world trace data, collected over a one-month usage period, from a Google multi-purpose cluster of approximately 12500 machines. Interesting observations are made: that the workload is a mix of latency-sensitive tasks and less latency-sensitive programs, that new jobs are submitted all day, that 3 out of 4 jobs consist of only one task, that the majority of jobs run for only a few minutes, that the scheduler needs to make task-placing decisions tens or even hundreds of times per second, that task resubmissions account for almost half of all submission events, and more. The authors' general conclusion is that, heterogeneity in the resources offered and in the tasks executed, along with the restrictions that it brings, and the workload's dynamic variation, make common slot-based greedy schedulers (that treat machines or tasks equally) insufficient to cope with this wider variety of requirements. Instead, they advise on implementing direct sampling methods, because neither job duration nor total resource request can be well matched by any simple statistical distribution. The same goes for the number of tasks in jobs and the amount of resources requested for each task, both being parameters observed to take rough and discrete values (indicating manual setting), and unlikely to be accurately modeled.

In [7], the authors introduce a distributed architecture of autonomous agents, each residing in a Physical Machine (PM), monitoring local resource usage, and performing resource reconfigurations (assignment of new VMs and migration of existing VMs), when some over-utilization or under-utilization conditions are met. The choice of a configuration is performed using Multiple-Criteria Decision Analysis (MCDA) and the Preference Ranking Organization METHod for Enrichment of Evaluations (PROMETHEE), in particular.

In [8], the authors propose a dynamic resource allocation solution for cloud-based media processing, using Machine Learning (ML) and survival functions. Media clouds have particularly strict Quality of Service (QoS) requirements, because they deal with live media streams and can not afford many frame drops happening. In that work, resource allocation is controlled through the number of VMs being used, and the problem is modeled as an online bin-packing problem. This is a combinatorial optimization problem in which, given a set of items, each having a size and an arrival time (both unknown until

its arrival) and a departure time (unknown until its departure[8]), and the objective is to pack those items into bins, in a way that minimizes the upper bound on the number of bins used (at any point in time).

Dynamic Bin Packing settings are also examined in [9], a work motivated by cloud gaming systems using public clouds. For those systems, it is important to efficiently dispatch the playing requests of gamers to game servers (virtual machines), in a way that minimizes rental costs but at the same time keeps a smooth game-playing experience for the users.

In [10], the authors propose a cloud-based virtual Content Delivery Network (CDN) architecture, which multiplexes many CDN services with different Service Level Agreements (SLAs) into a single VM competing for resources, and allows the VM to be scaled according to the aggregate traffic demand from those services. In that work, dynamic resource allocation is implemented in a threshold-based manner, and dynamic resource scheduling is implemented in a priority-based manner.

In [11], the authors propose a dynamic resource allocation for cloud-based Free Viewpoint Video (FVV) services using two different time-scales. On a mid/long time-scale, the number of VMs allocated to each FVV channel is chosen with respect to cost minimization. On a finer-grained time-scale, VMs among FVV channels are reconfigured with respect to response time minimization, in order to adapt to the varying workload. The arrival of requests is predicted using an Auto-Regressive Integrated Moving Average (ARIMA) model, while assuming a Poisson arrival process allows to model each channel's service process as a $M/M/1$ queuing system.

In [12], the authors develop a generic modular system, that takes as input user queries and executes them in a Database Management System (DBMS) running over an IaaS cloud, based on user preferences and constraints and the IaaS provider's charging policies. The underlying model is an economical model of query services that selects the query plan to be executed, based on execution cost, but also on the amortized cost of building and maintaining data structures. The estimation of the prospective queries that will make use of a given data structure becomes possible through query traffic analysis.

[8]Because usage duration is not known in advance.

In [13], the authors develop an analytics query processing engine that runs over an IaaS cloud and exploits its elasticity by dynamically allocating or de-allocating VMs, depending on the query workload, so as to maximize the provider's net profit. Their solution takes into account the SLA that comes with each submitted query, modeled as a function of query execution with money as its output. The allocation problem is then dealt with as a multivariate function optimization problem, that takes into account the history within a prefixed moving time window, in order to predict the profit throughout a future time window. The predicted profit is calculated by means of a L-BFGS-B algorithm, a limited memory algorithm for solving optimization problems.

A work regarding how socio-economic factors might shape the design of cloud computing is [14], in which the authors, noticing emerging trends in cloud-based systems, express their belief that economic forces will unify IaaS cloud computing models into a single economic model, and that clients who wish to use it efficiently, will do so by deploying openly available economic agents to bid for them. They believe that for "mid-range" clients, providers will either precisely measure all the system's resources to quantify the real use each VM makes out of them, and then charge the clients accordingly, or they will switch to a market-driven model.
Indeed, the prevalence of a Sharing Economy (term coined in [15]) in Cloud Systems is becoming apparent, however one still has to wonder if deploying auction theory mechanisms is the main way forward, or if it is just a set of techniques only helpful in some cases. In any case, [14] also proposes ways in which bidding can be complemented with other mechanisms.

1.2 Motivation

As seen in the previous Section, sharing resources is an issue manifested in various ways for different cloud infrastructures, different interested parties, different application types, etc. The research community is considering many different allocation algorithms, some based on disciplined approaches, while others based on ad hoc methods. The former is a better path to follow, because resource allocation in cloud infrastructures includes subproblems that can - and probably should - be decoupled. For example, SLA mappings to resource requirements, performance monitoring, performance modeling, workload forecasting, application profiling, are five important topics that affect elasticity success in conjunction, but nonetheless each can be more deeply investigated assuming that the others are taken care of.

In that spirit, this work focuses only on the learning aspects of a decision-maker for cloud-based resource allocation. In other words, it considers as given the performance and/or cost models as well as any restrictions imposed by the SLAs, and looks into how - based on these - a software agent can intelligently decide if any provisioning adjustments would be beneficial.

Again in the previous section, a recurring theme is the implementation of dynamic algorithms. This is to be expected in a very complex and continuously evolving system, with ever-evolving heterogeneous hardware, with dynamic workloads, with many exoteric variables affecting its function, as well as many nuisance parameters, because static approaches and planning ahead of time would consistently break down in this context. Obtaining a statistical model and solving it would be very troublesome in such a setting. Static models can still be obtained offline, through benchmarking and profiling, but in order to obtain good results online methods are also needed.

Therefore, there is strong incentive to use Online learning algorithms, and especially algorithms running in real-time, since elasticity actions must account for newly monitored data, as they are being generated and observed by the agent. When long-term predictions are difficult to make, it is even more important to use adaptive algorithms.

Moreover, as previously stated, the resource sharing problem can be viewed, either from a cloud provider perspective, trying to decide how to allocate owned resources among different client workloads, or from a cloud client perspective, trying to decide which resources to ask for. Despite who the stakeholder is, there is a commonality in that both cases are dealing with decision-making problems[9].
This observation might seem trivial at first, but a lot of research work is looking at the issue strictly as a mathematical optimization problem, wherein the optimality criterion is usually some measurable quality of the service, even a different one depending on the particular case[10]. Of course, there is a duality between the two types of problems, but there are reasons for also taking into consideration the learning and approximation aspects of the issue, possibly under the context of sequential decision-making processes[11].
First of all, the problem's natural goal is to properly choose elasticity actions, rather than to come forth with arithmetic predictions. Hence, the problem is a decision-making

[9] Even though, the provider's perspective case is more complex than the client's perspective case.

[10] This is not something particularly new: in [16] the authors had already recognized that traditional resource allocation approaches were impractical for the large and complex distributed systems of the year 1996. Now, systems are even more decentralized and heterogeneous.

[11] Process models in general support predictions based on information observed in the entities of interest and their behaviors.

one by nature. Secondly, cloud system operation is too complex to even consider that meaningful arithmetical solutions can be derived. It is adequate to strive for simpler evaluations of a comparative nature (between available choices), rather than specific but meaningless scalar values. Thirdly, under such dynamic conditions amounting to uncertainty, even if arithmetical solutions were meaningful, optimal solutions would still be impossible to calculate, hence, the unavoidable errors should ideally affect the sorting/ranking of alternatives as little as possible. Simply put, it does not matter if the agent makes bad value estimations, it only matters if these bad estimations lead to taking inappropriate actions by affecting the preference ordering over possible actions. And finally, decisions themselves (and not just the observed metrics) affect the system's knowledge/belief about the problem, therefore elasticity decisions can also have a learning value. For these reasons, the potential of inter-temporal multi-objective (utiity) optimization techniques, for dealing with cloud resource sharing issues, seems promising.

This work delves deeper into learning techniques for decision-making under uncertainty, of a "black-box" type, in other words, not assuming much about how the environment works (unlike gradient methods for example) and instead depending on (perhaps noisy) observable quantities and their variations over time. It presents research directions in the design of real-time adaptive techniques for dealing with cloud resource sharing. It is proposed that decision-making modules in cloud infrastructures or cloud clients, make use of findings in scientific fields such as Decision Theory and Statistical Learning, in the belief that more robust solutions will eventually spring that way.

Chapter 2

Reinforcement Learning

Reinforcement Learning (RL), also referred to as Approximate Dynamic Programming (ADP) in the Operations Research and Control Theory domains, is a scientific area very close to Machine Learning (ML) [1], that stems from classical Dynamic Programming (DP) (Value Iteration [VI] and Policy Iteration [PI]), Stochastic Approximation (Monte Carlo Methods [MCM]), Function Approximation (Regression and Artificial Neural Networks), and Artificial Intelligence (Temporal Difference Learning [TDL]). RL is a general approach to agent learning by receiving rewards that come as a result of interacting with a dynamic environment. RL does not require teachers, experts, or any type of domain-specific knowledge that specifies how a task is to be achieved, but at the same it can harness such assets if they are available.

2.1 Inspiration

The name Reinforcement Learning was introduced by Richard Sutton and Andrew Barto in [17], but the term already existed in branches of Psychology studying behavioral learning. Nowadays, with so many scientific subfields sharing ideas and techniques, RL does not make for a very precise topic description, however it is indicative of how the idea of intelligent adaptive learning algorithms that mimic how animals learn, was originally incepted.

[1] Reinforcement Learning can actually be considered as one of the three major Machine Learning approaches, the other two being: Supervised Learning (SL) in which the learning algorithm generalizes from labeled training examples to unseen data, and Unsupervised Learning (UL) in which the learning algorithm discovers hidden structure in unlabeled data without receiving any feedback. RL sets itself apart from both SL and UL, because learning comes from acting in an environment and receiving reward/error signals as a result, without ever being explicitly told which actions are optimal.

Quoting Christopher Watkins [18], a pioneer in the field of RL:
"A central problem of Artificial Intelligence is to understand how animals and people learn from experience to improve their skills, to achieve new goals in new ways, and to form new and more insightful representations of the world and of their own abilities."
In his PhD thesis [19] Watkins, inspired from experimental psychology and the Optimality Argument[2], introduced a model of learning from rewards in the form of incrementally optimizing control of a Markov Decision Process (MDP), and proposed a new algorithm that learns the optimal control directly, without modeling neither the transition probabilities nor the expected rewards of the MDP, namely Q-Learning. MDPs and Q-Learning are explained in Sections 2.2.1 and 2.5.1 respectively.

More specifically, Watkins was inspired by animal conditioning experiments, in which an animal is kept in an artificial environment and is subject to reinforcements and punishments, in order to study its ability to learn how to behave. In the behavioral literature, a positive reinforcer is a stimulus that observably strengthens (makes it more probable to be repeated) an organism's future behavior preceding it. A major distinction is between the "classical" or "Pavlovian" or "respondent" conditioning and "instrumental" or "operant" conditioning. Watkins' work was influenced by the latter type, but both are briefly presented.

In classical conditioning [20], an animal is exposed to a sequence of pairs that consist of a neutral Conditioned Stimulus (CS) and a potent Unconditioned Stimulus (US). The stimuli correlation is controlled by the experimenter and does not depend on the animal's actions. In these types of experiments, one can study for instance how the animal learns to expect the US as a result of receiving the CS. For example, suppose that the animal is a dog, the CS is a bell ringing, and the US is a morsel of food being offered to the dog. Initially, the US (food) makes the dog salivate more, in other words, the US elicits an automatic natural Unlearned/Unconditioned Response (UR), while the CS (bell) does not make the dog salivate more, in other words, the CS does not elicit the same UR as the US does. Then, the experimenter starts always ringing the bell just before feeding the dog, in other words, in each session the CS precedes the US. After a while, the dog will start salivating more just by hearing the bell ring (without foraging yet), which means that the CS will eventually elicit the same response as the US, only now it is a Conditioned Response (CR). So the dog will have learned the association of the two stimuli. Of course, conditioning can be reversed: if the bell starts ringing without the food offering following it, then the dog will eventually cease salivating more

[2] More generally in various scientific domains, optimality arguments explain empirical regularities through the maximization of some objective function.

in response to just hearing the bell. This experiment has come to be known as "Pavlov's dogs", and is indicative of learning through responding rather than goal setting.

In instrumental conditioning [21], the animal again learns to behave in a certain way, only this time not simply as a response to certain stimuli, but specifically because behaving in that way will lead to positive reinforcements taking place (and/or avoid negative reinforcements). This is due to the rewards no longer being under the control of the experimenter, and instead being produced by the animal's own choice of actions. A pioneer of modern behaviorism, Burrhus Frederic Skinner believed that behavior is a consequence of environmental histories of reinforcement, not a product of free will. In [22], he and Charles Bohris Ferster proposed designs of experiments for evaluating the extent to which an organism's own behavior determines its subsequent behavior, with the main dependent experimental variable being the frequency of occurrence of behavior. The famous "Skinner Box" is a research method that employs the "free operant"[3]. More specifically, it is a chamber including levers, bars, keys, or other parts that the animal manipulates in its quest to receive pleasant stimuli (positive reinforcement) such as food and water, or even to avoid noxious stimuli (negative reinforcement) such as electric shocks. Additional stimuli can also be included in the box, in the form of lights, sounds, images, etc. The Skinner box was meant to be used for determining the reinforcement schedule that leads to the highest rate of response from an animal.

In short, instrumental conditioning is based on the antecedents and consequences of the organism's behavior, while classical conditioning is based on antecedent conditions and the organism's reflexive behavior, not on behavioral consequences.

In [19], Watkins associated the analysis of instrumental conditioning with the Optimality Model for animal behavior [23], and argued that: "In many cases, an animal need not always perform its critical skills with maximal efficiency: it is the capacity to perform with maximal efficiency when necessary that is valuable." This idea is critical in Reinforcement Learning as will become obvious in the following Sections.

[3]Use of the free operant in Ferster's own words refers to: "...any apparatus that generates a response which takes a short time to occur and leaves the animal in the same place ready to respond again.".

2.2 Mathematical origins

After having introduced the founding idea of RL, in this Section an overview is given of how RL practically came to exist as a mixture of methods and techniques. The motivation for this is that in order to get a grasp of how RL is meant to be applied in general or to dynamic resource allocation problems in specific, one has to first understand the basic math behind it.

2.2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework for modeling sequential decision-making, widely used in disciplines such as Control Theory, Operations Research, Robotics, Economics, Communications, and others. More specifically, a MDP is a stochastic control process driven by underlying Markov chains, which are sequences of random variables. The space of possible value combinations of those random variables forms a countable set, the "state-space" of the chain. The word "process" refers to the state being a function of time. A Markov process makes stochastic transitions between states of the state-space. In this thesis, we start by considering discrete-time MDPs and discrete-time Markov chains, but their continuous-time counterparts also exist.

What differentiates a MDP from a Markov chain is the addition of actions and rewards. In a MDP, each state transition produces an immediate reward for the system (to be controlled), which can be either positive, negative (viewed as a cost), or zero. Some or all of the states are decision-making states, and their corresponding points in time are called "decision epochs". Whenever the system is in a decision-making state, the decision-maker is required to choose one among a set of possible actions. Together, all possible actions for all states, form the "action-space" of the MDP. The selected action will influence the probability distribution of the system's next state, and these transition probabilities will only depend on the current state and the action taken at the current state, not on past states nor on past actions. This lack of memory is known as the Markov Property, and denotes a serial dependence only between adjacent decision epochs. Following every transition, the decision-maker observes the outcome of his choice of action, in other words, the new state of the system and the immediate reward earned.

Formally, the setting of a MDP consists of:

1. the set of states (or state-space):
   S (possibly including initial and terminal/goal states[4])

2. the set of actions (or action-space):
   $A \equiv \bigcup_{s \in S} A(s)$

3. the transition probabilities (or transition function):
   $T(s, a, s') = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a)$

4. the transition rewards (or reward function):
   $R(s, a, s')$

where:

- s,$s' \in S$
- $a \in A(s)$
- S_t a random variable denoting the state at discrete time step t.
- A_t a random variable denoting the action taken at discrete time step t.

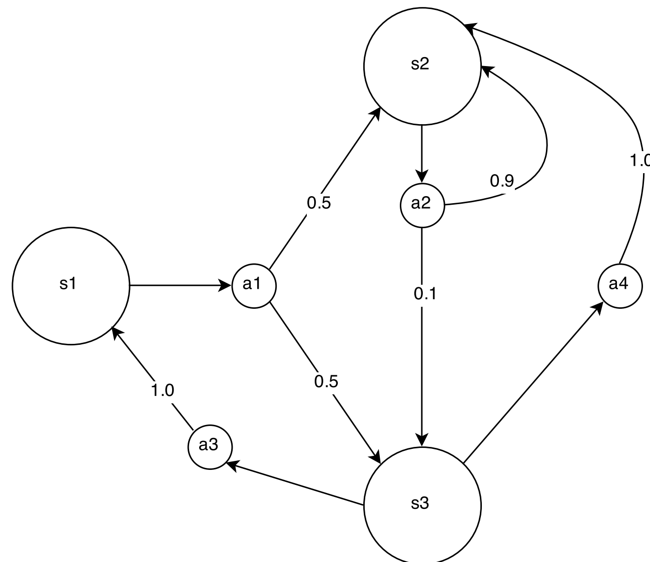

FIGURE 2.1: A simple 3-state Markov Decision Process with 4 different actions. Edge numberings represent the transition probabilities. Rewards are not represented.

Rewards are probabilistic. It is important however, to determine if this is due to a stochastic nature in their materialization process, or if it is simply a result of the transitions being probabilistic and the rewards deterministically depending on them. For

[4]Sink states are states that can never lead to another state, no matter which action is taken. Terminal/goal states are sink states which are not problematic, but rather highlight the end of some interaction session.

simplicity, the latter is assumed, in other words, that for a given transition or "state-action-state" triple (s, a, s'), taking place at a given moment in time t, the immediate reward is fixed: $R : S_t \times A_t \times S_{t+1} \to \mathbb{R}$. Therefore, one only needs to consider the probability distribution of transitions, because the reward distribution depends on it alone. It is also assumed that transition probabilities do not change with time.

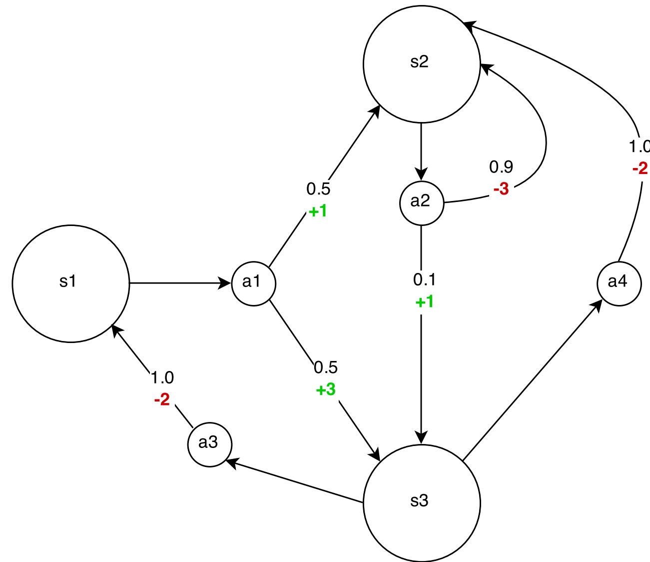

FIGURE 2.2: Having assumed that the immediate reward of every transition is fixed, rewards can now be represented in the graph of figure 2.1.

The Markov Property can be expressed as follows:
$\Pr(S_{t+1}=s' \mid S_t=s_t, A_t=a_t, S_{t-1}=s_{t-1}, A_{t-1}=a_{t-1}, ..., S_0=s_0, A_0=a_0) =$
$= \Pr(S_{t+1}=s' \mid S_t=s_t, A_t=a_t),$
indicating that the history of states and actions does not affect the next state's probabilities.
As a result of the Markov Property, the description of the current state (which in essence is the situation that the system faces) must be detailed enough to allow for the MDP to closely model the real-world process' evolvement. In other words, a state should capture all real world aspects that might play a role in determining an action's outcome, or else the decision-making problem can not be adequately modeled.

2.2.2 Markov Decision Problem

A policy (for a MDP) is a sequence of functions: $\pi = \{\mu_0, \mu_1, ...\}$, where function μ_t corresponds to time step t, and is a mapping from the set of all "state-action" pairs

to their corresponding probabilities (of the decision-maker taking each of the available actions when being in each state): $\mu_t : S \times A \to [0,1]$. A policy is called stationary if it does not involve any time-dependence: $\pi = \{\mu, \mu, ...\}$, in which case, this unique mapping can itself be considered to be the policy: $\pi : S \times A \to [0,1]$ and $\pi(s,a) = \Pr(A_t = a \mid S_t = s), \forall s \in S, \forall a \in A(s)$. In other words, under a stationary policy the distribution of actions only depends on the current state. Additionally, in the frequent case that a stationary policy is deterministic, then it is a mapping from the set of states to the set of actions: $\pi : S \to A$ and $\pi(s) \in A(s), \forall s \in S$. Therefore, a deterministic stationary policy prescribes which action to choose in every decision-making state, independent of time. The objective (or utility or return) function is a performance metric[5] for distinguishing "good" from "bad" policies, usually a mathematical function of the immediate rewards earned over a pre-determined finite or infinite time-horizon. In short, it is the mathematical expression of the agent's goal. The Markov Decision Problem is defined as the problem of finding the policy which optimizes such a predetermined objective, given a MDP. The objective function in Decision Analysis in general, provides a way to incorporate the decision-maker's preference information, and thus deal with the multiple and often conflicting criteria of optimality.

The value function is the optimal value of the objective function written as a function of the state. In other words, $V^*(s)$ is the expected utility, for starting in a state $s \in S$, and acting optimally ever after. Similarly, the Q-value function is the optimal value of the objective function written as a function of the Q-state (the state-action pair). In other words, $Q^*(s,a)$ is the expected utility, for starting in a state $s \in S$, taking an action $a \in A(s)$ available in that state, and acting optimally ever after.[6] It follows from the two definitions that: $V^*(s) = \max_{a \in A(s)} Q^*(s,a)$ and $Q^*(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$. Substituting one into the other gives a recursive formula for the value function: $V^*(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$, known as the Bellman Optimality Equation (BOE), and a recursive formula for the Q-value function: $Q^*(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma \max_{a' \in A(s')} Q^*(s',a')]$. These functional equations are of the utmost importance for the Dynamic Programming (DP) algorithms that will be presented in Section 2.2.3. Finally, the policy function is the optimal action,

[5] The term "performance" is used here in a general sense, not referring to any particular type of domain-specific performance.

[6] On the occasion of introducing specific terminology, it is worthwhile to note that depending on the type of literature, one may encounter different terms having the same signification, such as: "cost" instead of "reward" and "cost-to-go" instead of "value" (since the optimization problem can be seen as a minimization problem of a cost objective), "Q-factors" instead of "Q-values", "reinforcers" instead of "rewards", "controls" instead of "actions", "strategies" instead of "policies", "transformations" instead of "transitions", etc.

written as a function of the state: $\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s, a)$.

There is a duality between the space of value functions and the space of policies. More specifically, following a fixed policy in a given MDP setting, produces a sequence of actions, each yielding an immediate reward. Obviously, the goal of optimizing the objective function leads to preferences over these streams of rewards. For example, in figure 2.3, a decision maker with the objective function being the sum of rewards, would prefer to follow a policy that earns him $(7, 6, 2)$ rather than a policy that earns him $(3, 4, 2)$, in a 3-step finite decision-making problem, because it would lead to a total return of 15 compared to 9 for the second one.

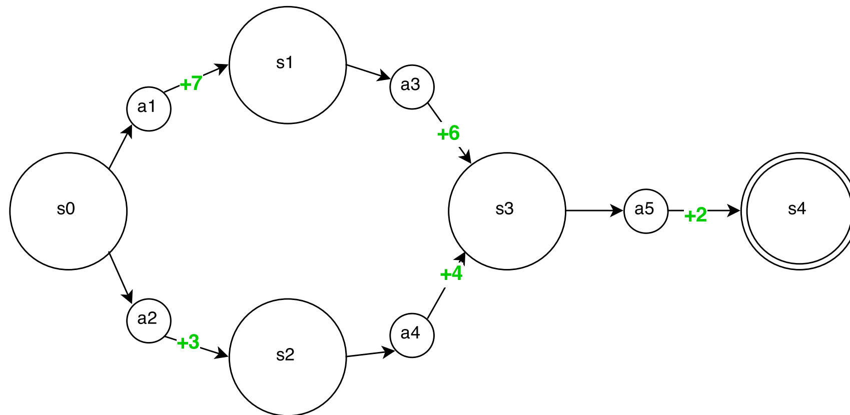

FIGURE 2.3: A 5-state Markov Decision Process with 5 different actions. Only 2 possible deterministic policies exist, corresponding to the 2 actions available in s_0.

Hence, preferences over policies can be viewed as preferences over streams of rewards. Formally, these are stationary over time if, for any two periodic time streams of rewards earned: $(r_1, r_2, ..., r_T)$ and $(r_1, r'_2, ..., r'_T)$, that yield the same reward at period 1 but possibly different rewards at the rest of the periods, it holds that:
$(r_1, r_2, ..., r_T) \succeq (r_1, r'_2, ..., r'_T) \Leftrightarrow (r_2, ..., r_T) \succeq (r'_2, ..., r'_T)$.
Equivalently, with respect to the utility function, they are stationary over time if, for any two periodic time streams of rewards earned: $(r_1, r_2, ..., r_T)$ and $(r_1, r'_2, ..., r'_T)$, that yield the same reward at period 1 but possibly different rewards at the rest of the periods, it holds that:
$U(r_1, r_2, ..., r_T) \geq U(r_1, r'_2, ..., r'_T) \Leftrightarrow U(r_2, ..., r_T) \geq U(r'_2, ..., r'_T)$.
Intuitively, stationarity means that if the first n elements of two streams (being compared) are the same, then their utility difference (and preference ordering) can be evaluated from their remaining elements alone. An implication of stationarity is that $U_t(r_1, r_2, ..., r_T) = U(r_1, r_2, ..., r_T) = V(r_1, U(r_2, ..., r_T))$, which means that preference

ordering is independent of its place in time, and as a result, choices are history independent. Simply put, if the decision maker prefers some policy π_1 over another policy π_2, then he will always prefer it (no matter the passage of time).

As previously described, if each time a state is visited, a policy deterministically specifies the same action, then it is a deterministic stationary policy. On the other hand, it is a stochastic stationary policy, if each time a state is visited, the policy specifies the same probability distribution over all possible actions. Stationary policies are important for the following reasons: In [24], David Blackwell has shown that if the same actions are available in every state, then there exists a deterministic stationary optimal policy (i.e. having an expected total return equal to the supremum of the total returns of all possible policies, for all initial states, irrespective of time). In his doctoral dissertation (supervised by Blackwell) [25], Ralph Strauch has shown that in any Dynamic Programming (DP) problem with negative or discounted total return, if there is an optimal policy, then there is an optimal policy which is also stationary. Blackwell later generalized this for all DP problems [26]. Sheldon Ross proved that for every MDP there is a stationary optimal policy [27]. This last result stems from the fact that - due to the Markov property being satisfied - the current state's description holds all the necessary information for deciding what to do next. Therefore, in a given state the same decision will always be the optimal one, independent of time. And because this holds true for every state, the optimal policy will always be the same regardless of the decision epoch.

The importance of the above conclusion, is that for solving a Markov Decision Problem, there is no loss of generality if the search for the optimal policy is restricted only in the set of stationary policies. Hence, it is convenient to choose an objective function that leads to preferences being stationary over time. Two types of objective functions support stationary preferences:
Additive utility: $U(r_0, r_1, r_2, ..., r_n) = r_0 + r_1 + r_2 + ... + r_n$
Discounted utility: $U(r_0, r_1, r_2, ..., r_n) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... + \gamma^n r_n$[7]
The additive utility is a special case of the discounted utility (for $\gamma = 1$). The discounted utility [28] increases at a decreasing rate, and is therefore bounded, which leads to a favorable structure and a more "well-behaved" model, compared to the (un-discounted) additive utility, or to any other type of infinite time-horizon stochastic DP problem in general [29, 30].

[7]The reason why discounted utility leads to stationary preferences over policies, is because a delay has the same impact on utility, regardless of the point in the stream at which it occurs.

More specifically, assuming that starting at state s, a given policy π is followed over an infinite time-horizon, then two performance metrics are often used as the utility function:

1. the expected sum of discounted rewards:
   $\psi_\pi(s) = \lim_{l\to\infty} E[\sum_{k=1}^{l} \gamma^{k-1} r(x_k, \pi(x_k), x_{k+1}) \mid_{x_1=s}]$
2. the expected average reward[8]:
   $\rho_\pi(s) = \lim_{l\to\infty} \frac{E[\sum_{k=1}^{l} r(x_k,\pi(x_k),x_{k+1}) |_{x_1=s}]}{l}$

where:

- $0 < \gamma < 1$ is the discount factor.
- x_k is the state occupied before the kth transition
- E is the expectation operator.

Therefore, the Markov Decision Problem for discounted rewards, consists in finding a policy π^* such that $\psi_{\pi^*}(s) \geq \psi_\pi(s), \forall s \in S, \forall \pi$. While the Markov Decision Problem for the average reward, consists in finding a policy π^* such that $\rho_{\pi^*}(s) \geq \rho_\pi(s), \forall s \in S, \forall \pi$.

Average reward metrics equally weight every single reward, whether earned in near future transitions or in distant future transitions. Discounted reward metrics on the other hand, consider rewards as being of less importance the longer in the future they are earned. This stems from the financial concept of discount yield, equivalent to the rate of return, or simply put, from the idea that having money in the future is worth less than having the same amount of money right now, due to the opportunity cost of not being able to invest it until actually receiving it. Specifically, the discount factor γ denotes the factor by which a future reward must be multiplied, in order to obtain its present significance. For the reader interested in delving deeper into alternative models of temporal discounting, a recent survey is [31]. In this Chapter, only discounted rewards are considered.

2.2.3 Dynamic Programming

Richard Bellman made an important contribution in solving Markov Decision Problems, by introducing Dynamic Programming (DP) in the 1940s. DP is an algorithmic process

[8] If the Markov chains are regular, then the starting state does not affect the expected average reward so that $\rho_\pi(s) = \rho_\pi$ holds.

of solving decision-making problems (in the form of mathematical optimization problems) that exhibit the properties of "optimal substructure" and "overlapping subproblems" (satisfying Bellman's "Principle Of Optimality"). Optimal substructure means that the problem's structure is such, that its (globally) optimal solution can be easily obtained by combining (locally) optimal solutions of subproblems (smaller instances) of the original problem. Overlapping subproblems means that the space of subproblems is small enough, so that any recursive algorithm that tries to solve the problem, will need to solve the same subproblems many times, instead of just generating and solving only subproblems that don't share anything in common.

As the optimal substructure property suggests, DP tries to solve subproblems of the original problem and then combine their solutions into an overall solution. For multi-step decision problems, this means breaking them into simpler decision steps over time. But, in contrast to other more naive algorithmic methods that break a problem into subproblems (i.e. Divide and Conquer), due to the overlapping subproblems property, DP seeks to solve each subproblem only once, store its solution, and simply look it up whenever needed, thus reducing the total number of computations.

The two classical DP algorithms are Value Iteration (VI) and Policy Iteration (PI).

2.2.4 Value Iteration

VI starts with a guessed estimate of the value function $V_0(s), \forall s \in S$ and finds ϵ-optimal estimates of it, using the Bellman Optimality Equation (BOE)[9] (mentioned in Section 2.2.2):

$V^*(s) = \max_{a \in A(s)} [\overline{r}(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s')], \forall s \in S \Leftrightarrow$

$V^*(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')], \forall s \in S$

where $\overline{r}(s,a) = \sum_{s' \in S} T(s,a,s') R(s,a,s')$ is the expected immediate reward for being in state s and taking action a, calculated as an average of all the possible outcomes of action a.

[9] In continuous-time optimization problems, which we do not consider in this work, the analogous is a partial differential equation, called the Hamilton−Jacobi−Bellman equation, which involves the gradient of the value function.

The BOE is a recurrence relation describing the maximum expected utility, when the system is in a state s. Therefore, it comprises one equation for each state of the state-space. Solving it as a system of non-linear equations can be difficult, this is why VI instead uses a fixed-point iteration to find the ϵ-optimal estimates of the value function, from which the ϵ-optimal policy can then be derived. Simply put, VI successively implements the BOE as an update rule, rather than a set of equations that must be satisfied:
$V_{k+1}(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V_k(s')], \forall s \in S$
If the update is run infinitely long, then this estimate will converge to the (true) value function: $\lim_{k \to \infty} V_k(s) = V^*(s)$
The proof of convergence is based on the contraction property of the VI operator [32].

Of course, VI can also calculate the Q-value function (instead of the value function), if for each state it stores every possible action's value (instead of only the best action's value). This version of VI is referred to as Q-Value Iteration (Q-VI):
$Q_{k+1}(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma \max_{a' \in A(s')} Q_k(s',a')], \forall s \in S, \forall a \in A(s)$
Again, if the update is run infinitely long, then the estimate will converge to the (true) Q-value function: $\lim_{k \to \infty} Q_k(s,a) = Q^*(s,a)$.

After obtaining the state or Q-state values, with VI or Q-VI respectively, Policy Extraction follows to provide the answer to the decision problem. Policy Extraction is the process of finding the optimal policy, either from a given value function (through a one-step look-ahead), or equivalently from a given Q-value function (instantly):
$\pi^*(s) = \arg \max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')], \forall s \in S \Leftrightarrow$
$\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s,a), \forall s \in S$
Obviously, although storing Q-state values is more memory wasteful than storing state values (because there are more state-action pairs than there are states), it leads to a simpler Policy Extraction step.

2.2.5 Policy Iteration

PI starts with a fixed policy $\pi_0(s), \forall s \in S$ and repeats the following two steps: Policy Evaluation and Policy Improvement.

Policy Evaluation uses the Bellman Policy Equation (BPE) to find the state (or Q-state) values from a given fixed policy (almost the inverse of what the Policy Extraction step

of the previous Section 2.2.4 does):

$V^{\pi}(s) = [\overline{r}(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s')V^{\pi}(s')], \forall s \in S \Leftrightarrow$

$V^{\pi}(s) = \sum_{s' \in S} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(s')], \forall s \in S$

The BPE is actually mathematically the same as the BOE, but a is replaced by $\pi(s)$, and the notation $*$ is dropped in favor of π, to highlight that BPE involves a specific policy π and not the optimal policy (as the BOE does).

The BPE is therefore also a recurrence relation, only this time describing the given policy's expected utility, when being in a state s. Thus, Policy Evaluation successively implements the BPE:

$V_{k+1}(s) = \sum_{s' \in S} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k(s')], \forall s \in S$

If the update is run infinitely long, then the estimate will converge to the given policy's (true) state values: $\lim_{k \to \infty} V_k(s) = V^{\pi}(s)$.[10]

Contrary to VI (which includes a *max* operator), Policy Evaluation can alternatively be achieved by solving the linear system of equations that result from the BPE for every state. This approach is helpful whenever the discount factor γ is very close to 1 which would cause the iterative estimate to take too long to converge.

Having calculated the above function $V^{\pi}(s)$ in Policy Evaluation - now denoted $V^{\pi_m}(s)$ - PI proceeds to Policy Improvement. Policy Improvement (the equivalent of VI's Policy Extraction) finds the best action according to a one-step look-ahead, and thus updates the given policy π_m:

$\pi_{m+1}(s) = \arg\max_{a \in A(s)} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_m}(s')], \forall s \in S$

Now that the policy has been updated, PI repeats the first step which is Policy Evaluation. The loop terminates when the policy π_m has converged to the optimal policy π^*.[11]

2.2.6 Optimistic Policy Iteration

Optimistic Policy Iteration (OPI) is PI where the Policy Evaluation step implements the BPE update only a finite number of times. OPI falls between VI and PI. This is

[10]Throughout this work, we abide by the formal definition reserving the term "value function" for $V^*(s)$, always related to the optimal policy (π^*). However, $V^{\pi}(s)$ is by all practical means also a "value function", albeit under a fixed policy (π). Obviously: $V^*(s) = \max_{\pi} V^{\pi}(s)$

[11]It is worth noting that the discount factor γ - apart from its original purpose in modeling the decision process - also plays an important role in achieving convergence, when calculating the above recursive value functions.

more easily understood if one considers its two extreme cases.
On one extreme, if the BPE update is implemented only once in the Policy Evaluation step, then the Policy Improvement step that follows, will improve the given policy, not based on its true value function (as in PI), but rather on a rough approximation of it. This way, the policy is being constantly refined through a Bellman update based on its approximated value function. This is exactly what VI does, except VI does not bother to explicitly derive the policy itself until the very end of this procedure.
On the other extreme, if the BPE update is implemented an infinite number of times in the Policy Evaluation step, then the exact value function of the given policy is obtained from the Policy Evaluation step, and OPI is the same as PI.

2.2.7 Value Iteration versus Policy Iteration

The computational complexities of the two algorithms' core components are outlined:
VI is composed of: X VI-iterations + 1 Policy Extraction.
cost for 1 VI-iteration: $\mathcal{O}(S^2A)$
cost for 1 Policy Extraction: $\mathcal{O}(S^2A)$
PI is composed of: Y PI-iterations,
and each PI-iteration is composed of: Z Policy Evaluation iterations + 1 Policy Improvement.
cost for 1 Policy Evaluation iteration: $\mathcal{O}(S^2)$
cost for 1 Policy Improvement: $\mathcal{O}(S^2A)$

One key difference made apparent by the above complexities, is that a Policy Evaluation iteration, does not look at all possible actions like a VI-iteration does, but instead drops the *max* operator and tries to evaluate the given policy, not the optimal policy. This is particularly appealing in MDPs in which updating the state (or Q-state) values only rarely leads to a policy update. VI only generates values from which actions are implicitly updated, while PI explicitly and successively generates and updates values and actions[12].

The advantage that PI has over VI is that, due to the monotonicity of PI-iterations, if the policy does not change in a PI-iteration, then it is known to be optimal, and thus the method is known to have converged [30]. This is contrary to VI, where values are continuously updated without knowing for sure if they are accurate enough to end the

[12] This is actually why, PI could alternatively start at the Policy Improvement step given some initial values $V^{\pi_0}(s)$, instead of starting at the Policy Evaluation step given an initial policy π_0.

algorithm and provide the optimal policy. It might just be the case that had the VI algorithm run a little longer, the updated values being a little more accurate, would nevertheless have lead to a different policy as the optimal answer.
On the other hand, the advantage of VI over PI is that, the decision-maker is free to behave according to his own will, and still eventually learn the optimal policy (related to the concept of Active Learning explained in Section 2.3).

It is worth noting however, that DP - whether in the form of VI or PI - is intractable in the case of very large state-spaces. This is a typical manifestation of the "curse of dimensionality" (term coined in [33]): the exponential growth of hyper-volume as a function of the dimensionality. Consequently, even though both algorithms will still complete in a finite number of steps[13] (because a finite state-space and a finite action-space lead to a finite number of possible policies), that number will be prohibitively large. This infeasibility of the "exact" DP paradigm has been a motivating factor for later introducing Reinforcement Learning (RL).

2.3 Formal Setting

Mathematically, in RL the setting is the same as that of an MDP, however - contrary to Probabilistic Planning - the acting agent[14] is located in the real world and is unaware of the underlying MDP.
More precisely, the agent is unaware of the 3rd and 4th elements of the MDP, namely:

- the transition probabilities (or transition function/model): $T(s, a, s')$
- the transition rewards (or reward function): $R(s, a, s')$

Not knowing the dynamics of the environment, makes the agent effectively unaware of the model of the world.

As a result of only knowing the 1st and 2nd elements of the MDP, namely:

- the set of states (or state-space): S

[13] For infinite state-spaces, DP can be shown to converge asymptotically.
[14] The term "decision-maker" used in the general sequential decision-making context, is now replaced by the term "acting agent" used in Reinforcement Learning, because the former can refer to anything ranging from a dedicated controller to a physical moving entity, while the latter more narrowly implies an intelligent and autonomous entity. Moreover, because in this work the interest is on software agents in particular, and not on persons, we will refer to the acting agent as an "it".

- the set of actions (or action-space): A

the agent is no longer able to apply the Bellman Equations as presented in Sections 2.2.4 and 2.2.5, and needs some other way of assigning values to states.

However, it is assumed that the agent can at any point in time observe:

- the immediate reward that a chosen action grants it: r
- the environmental state that it is currently found in: s

This is known as full observability[15] by part of the agent.

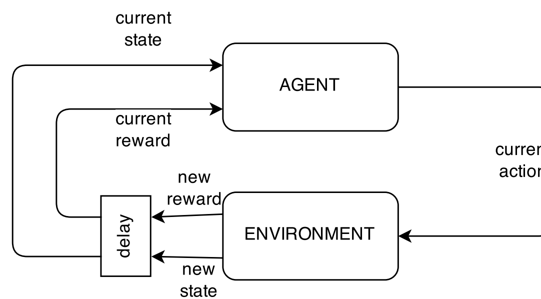

FIGURE 2.4: The interaction between the learning agent and its environment, which is basically everything the agent can not arbitrarily change. The agent takes an action, the environment responds, and the agent receives the (possibly delayed) feedback.

Due to the learning agent not knowing the governing model of its environment (because reward and transition probabilities are ignored), there exist two main RL approaches:

In "Model-Based RL", the agent first learns an approximate model of the world, based on its experiences in it. The more experiences it collects during this phase, the more accurate its estimation of the true model will be. When this learning phase stops, then the agent solves its estimated model of the world as if it were correct, in other words it performs Planning for that empirical model and then implements its solution.[16] For example, after learning the model, the agent can apply the two classical DP algorithms previously examined: VI or PI.

[15]Partially Observable Markov Decision Processes (POMDPs) are MDPs where the agent does not have perfect knowledge of the current system state. In this work, we will not consider POMDPs.

[16]This is still considered an Online learning approach, because the model has been learned by acting in the real world.

In “Model-Free RL”, the agent never explicitly learns the model of the world. Instead, it right away starts calculating some quantities of interest to guide its decision-making, again by acting in the real world. These quantities usually take the form of a value function (for example the value function of a Markov Decision Problem in Section 2.2.2). The present work is more interested in Model-Free RL techniques, which in turn can be implemented in two different ways:
In “Passive RL”, the agent follows a given fixed policy, and while doing so, learns the evaluations of the states and possibly of the actions. This is as if the agent was being remotely operated by some controller, instead of free-willingly choosing its own actions[17]. After following the specified policy and learning the values or Q-values, the agent can then use them to improve its policy, and run again in the real world with the updated policy. Hence, there is a repeating cycle of acting in the world along a fixed policy, and then updating that policy to account for actions found to be sub-optimal.
In “Active RL”, the agent is not following any particular policy. Equivalently, its policy is evolving in real-time, with every move through the state-space accounting for the collection of more knowledge (updating state or Q-state values). There are many ways in which the agent can use estimated values to shape its continuously changing policy, and no matter how it does it, hopefully its policy will eventually converge at some point.

Model-Free RL has to specifically deal with how the agent can assign values to the various states or Q-states (state-action pairs), so as to distinguish between a “good” state and a “bad” state or between a “good” action and a “bad” action for a given state, based on its observations and without any a priori model of - or knowledge about - the world. These values are usually scalar and implemented through a so-called objective (or utility or return) function, which accounts for the stakeholder’s personal notion of optimality. The objective function possibly aggregates many conflicting criteria that require trade-offs to be made.

Initially, the agent might either not have any assumption at all concerning the world, or it might already have some estimated state (or Q-state) values. Either way, the agent will learn more as it goes along acting in the real world, instead of computing ahead of time what to do (which constitutes Planning). This means that a RL agent will visit states, take actions, experience the accompanying transition rewards, and while doing so will calculate and re-calculate the aforementioned values. The hope is for the agent to learn behaviors that will eventually converge to a near-optimal policy.

[17] Also called a “reflex agent”.

This is why RL is an "Online Learning" solution, while Dynamic Programming (DP) is an "Offline Learning" solution. Online refers to learning by collecting experiences in the real world, and using them to update the plan being followed. Offline refers to first planning ahead of time based on a given model of the world, and after obtaining a complete plan, only then implementing it in the real world. Another example of Offline Learning is any Machine Learning (ML) algorithm whose knowledge is not updated once the initial training phase has been completed.
Naturally, Online Learning involves a lot of failing by part of the learning agent, at least in the beginning, until sufficient feedback from the world has been obtained, because the agent does not have any domain-specific knowledge, and can only avoid failures if it first experiences them.

This raises an important question: can the agent progressively learn an optimal policy while following other sub-optimal policies? or is the agent required to adopt the optimal policy in order to verify that it is indeed the optimal one?
The answer is that it is not necessary for the agent to apply the best policy in order to learn which one it is. This idea again has its conceptual roots in the Optimality Argument concerning animal learning, which suggests that after sufficient experience in its environment, an animal will eventually learn to exploit that environment with maximal efficiency, even if it behaves with less than optimal efficiency while learning to do so. And it is exactly this idea that differentiates "On-policy" from "Off-policy" Learning. Techniques that belong to the former category involve evaluating the policy being followed, while techniques that belong to the latter category involve directly evaluating the optimal policy (irrespective of the policy being followed).

2.4 From Dynamic Programming to Reinforcement Learning

In the previous Section, the Reinforcement Learning (RL) setting was introduced. It has become clear by now, that the only way for a learning agent to solve a decision-making problem under this setting, is by acting and observing the consequences, because its lack of understanding of how the world functions makes it impossible for him to plan in advance. In this Section, we look at how this "learning by doing" approach can be combined with Dynamic Programming (DP).

2.4.1 Sample-Based Learning

It was mentioned before that there exists a utility function in the context of RL, which plays the role of a performance criterion for ranking policies. However, this utility function has not yet been specified. To do so, inspiration will be drawn from the utility functions of MDPs presented in Section 2.2.2, keeping in mind that now the agent does not know $T(s, a, s')$ nor $R(s, a, s')$.

Let us consider a passive RL agent who is learning in an episodic manner. In a given learning episode, the agent has a fixed policy and tries to evaluate the states. When the learning episode is completed, the agent might modify his policy according to the updated state values, before proceeding to the next learning episode.

One way to estimate state values is through Direct Evaluation (DE). DE generally implies that each state's value should directly depend on how "good" the agent did from the moment he visited that state and onwards, not caring at all about how "good" the agent did before reaching that state. Of course, due to its ignorance of the model of the world, the agent can not calculate this in advance. Instead, every time a new learning episode ends, the agent calculates for each state, the average - across all learning episodes - sum of rewards, that it received after visiting that state and until the end of its interaction with the environment for that episode (when having reached a terminal/goal state, or simply after a finite number of decision epochs). If during a learning episode, the agent did not pass from some states, then those states' values will remain unchanged, entering the next episode.
As a simple example of how DE works, in the MDP of figure 2.5, suppose that s_0 is the initial state and that the agent has already gone through five learning episodes. In four of the episodes, taking action a_1 has resulted in transitioning to terminal state s_3 via state s_1. In one of the episodes, taking action a_1 has resulted in transitioning to terminal state s_4[18] via state s_2. Hence, in four episodes the agent has gone from s_0 on to collect a total return of 15, while in one episode the agent has gone from s_0 on to collect a total return of 0. As a result of implementing DE, after those five learning episodes, state s_0 has an estimated value of $V(s_0) = \frac{15+15+15+15+0}{5} = 12$, while states s_1 and s_2 have a value of 10 and -1 respectively, and states s_3 and s_4 both have a value of 0 by

[18] Representing a terminal/goal state is actually a matter of convention. It could be represented as a normal state having a vague transition arrow (imagined to lead to the next learning episode and producing a reward). It could also be represented as a sink state with a fixed state value. But more usually, a terminal/goal state is represented as an (extra) sink state having a fixed zero value that never gets updated, and thus the benefit of reaching it is attributed to the transition that landed in it.

convention (because they are terminal states and do not lead anywhere).

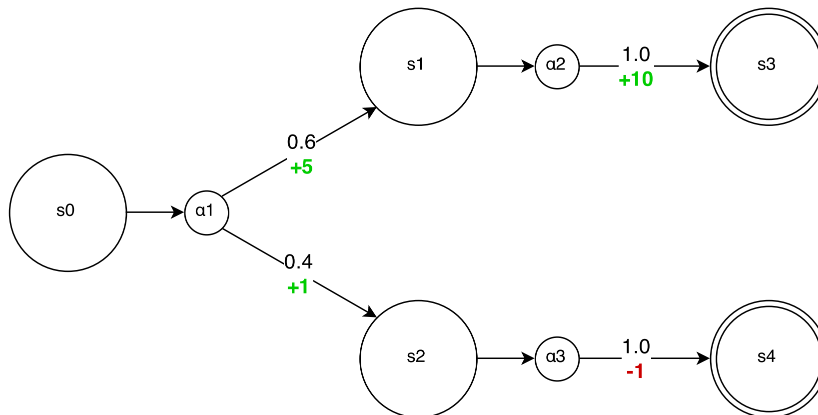

FIGURE 2.5: A 5-state Markov Decision Process. Although transition probabilities are represented, the agent is unaware of them in the Reinforcement Learning setting.

Unfortunately, DE depends too much on stochasticity. Even though in the limit, it will converge to the correct/true state values, for a finite learning time-horizon it can easily result in bad estimations (and therefore bad policies).

For example, in figure 2.6, state s_1 and state s_2 both lead to state s_3. In specific, the only action available in s_1 is a_2 and the only action available in s_2 is a_3, and they both always deterministically lead to s_3. In addition, both transitions $s_1 \rightarrow s_3$ and $s_2 \rightarrow s_3$, always carry the same reward of -1. As a result, it would be appropriate for the value estimations of states s_1 and s_2 to be equal, indicating two equally good states, for indeed whether the agent finds himself in s_1 or in s_2, makes absolutely no difference on the total return that the agent will eventually achieve.

However, if it so happens that in the learning episodes in which the agent has passed from s_1, he got unlucky[19] and received many -10s by ending up in terminal state s_5, then this will lower s_1's value, because DE will remember that passing from state s_1 the agent eventually did "bad". On the other hand, if it so happens that in the episodes in which the agent has passed from s_2, he luckily received many $+10$s later on by ending up in terminal state s_4, then this will similarly raise s_2's value, because DE will remember that passing from state s_2 the agent eventually did "good". As a result, DE will miss the fact that whether passing through s_1 or passing through s_2, has no effect whatsoever on subsequent rewards, and will misinterpret the causality of the different state-space trajectories.

[19] Or in some other example, followed a bad policy from then onwards, instead of getting unlucky.

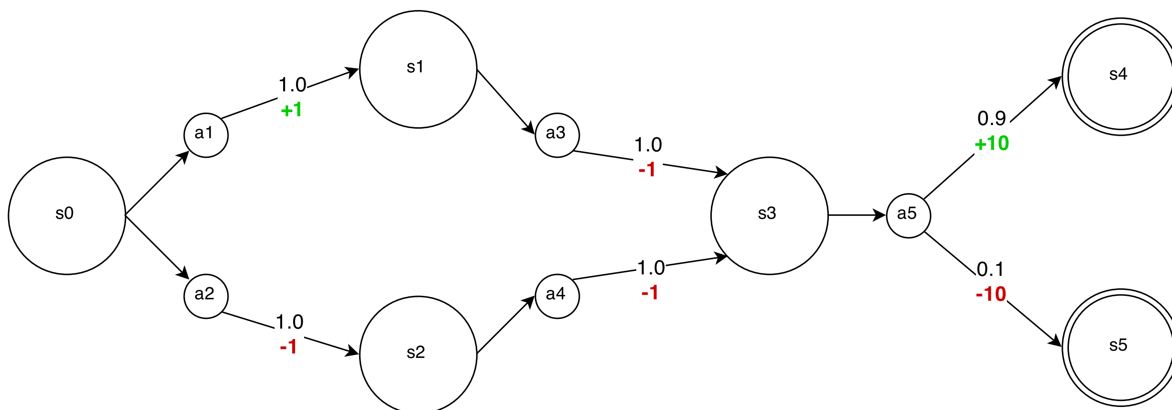

FIGURE 2.6: A 6-state Markov Decision Process. States s_1 and s_2 are obviously co-equal. However, lucky streaks will confuse a learning agent applying Direct Evaluation.

In the above example, miscalculating the value of s_2 as being greater than the value of s_1 will mislead the agent into adopting the corresponding policy which, in state s_0 prefers action a_2 over a_1. This problematic behavior will be corrected once enough learning episodes have been run, and the lucky or unlucky streaks have faded, but it shows that DE is not an efficient way to gather knowledge about a state-space that is interconnected in particular ways. What is more, as a result of taking action a_2 instead of a_1, the agent will be getting a -1 instead of a $+1$ when leaving the initial state s_0. This is not only due to the belief that state s_2 is better than state s_1, but also a result of evaluating states instead of actions, because the immediate reward of $+1$ or -1 has no effect on anything other than the unimportant value of the starting state s_0.

2.4.2 Sample-Based Policy Evaluation

A better way for evaluating states would be to use Policy Evaluation from the PI algorithm, which takes into account how states are interrelated. As a reminder from 2.2.5, in Policy Evaluation, the agent iteratively updates the state values, with respect to the following estimation rule:

$V_0(s) = 0$ (or some other initial value)

$V_{k+1}(s) = \sum_{s' \in S} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k(s')], \forall s \in S$

where the estimates - if run infinitely long - converge to the state values of the fixed policy:

$\lim_{k \to \infty} V_k(s) = V^{\pi}(s)$.

The above rule embodies the idea of a one-step look ahead, which means to evaluate a state s based on two separate quantities:

1. the immediate reward observed when transitioning from state s to state s' through action $\pi(s)$: $R(s, \pi(s), s')$
2. how good the landing state s' is currently considered: $V_k(s')$

Because of the probabilistic nature of the transition model, this idea is formulated using a linearly weighted sum over all possible outcomes of an action. This is exactly why Policy Evaluation can not be used in this form for RL purposes, because both $R(s, \pi(s), s')$ and $T(s, \pi(s), s')$ are not known to the learning agent, who is unaware of the transitions' and rewards' governing distribution, even though one does exist.

This issue can be addressed using a sample-based version of Policy Evaluation. This means that state values are estimated with an empirical (instead of analytical) average computation, in other words, that the agent keeps a running average of samples:
$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + (\alpha)sample_k \Leftrightarrow$
$V_{k+1}(s) \leftarrow V_k(s) + (\alpha)(sample_k - V_k(s))$
where:

- $sample_1 = R(s, \pi(s), s'_1) + \gamma V_k(s'_1)$
- $sample_2 = R(s, \pi(s), s'_2) + \gamma V_k(s'_2)$
- ...
- $sample_n = R(s, \pi(s), s'_n) + \gamma V_k(s'_n)$
- α is the "learning rate" parameter.

The reasoning behind this approach is simple: even though the agent does not know the transition probability distribution, he does observe each reward that comes with every transition, at the moment he earns it. Therefore, he can estimate state values by averaging samples rather than expected rewards, akin to Monte Carlo Methods (MCM). What is more, instead of obtaining (and storing) all the samples first and then computing their average (batch method), the agent can simply keep an exponential moving average (incremental method). This allows efficient episodic learning even for long state-space trajectories, because state values are updated one at a time, instead of all together as is the case in classical DP.

This type of value estimation is implemented through Temporal Difference Learning (TDL). TDL a class of prediction methods useful in sample-based incremental learning settings, conceived around the idea that different predictions are often correlated and could therefore be adjusted according to the more accurate among those predictions. In other words, whereas in Supervised Learning (SL) the learning adjustments rely on the error between the predicted outcome and the actual outcome, in TDL they also rely on the difference between temporally successive predictions, meaning that learning takes place whenever predictions change (instead of only when the final outcome is reached). Updating estimates based on other estimates is known as "bootstrapping" and it is essentially the same technique that DP implements in order to solve optimization problems, as seen in 2.2.3. TDL methods were first studied independently in [34], and were later mostly applied as a fundamental part of RL methods.

In relation to the specific update rule presented above, the TDL element lies in that a state value update is completely based on the difference between the newly observed sample and the current estimation: $(sample_k - V_k(s))$, where the newly observed sample $sample_k$ itself contains a prediction, namely $V_k(s'_k)$. Consequently, whenever updating the present state's value estimation, the next state's value estimation is required (again the notion of a one-step look-ahead)[20]. This is how knowledge about the interconnection of states is taken into account. Moreover, this TDL aspect is why the agent can learn before yet finding out the true total return (at the end of its interaction with the world), which is essential for Online learning. It is also why there is no need for the agent to explore states in any particular order, which in turn allows to drop the index k from the update rule (i.e. no longer keep count of how many updates each value has gone through):

$V(s) \leftarrow (1-\alpha)V(s) + (\alpha)sample_{new} \Leftrightarrow$

$V(s) \leftarrow V(s) + (\alpha)(sample_{new} - V(s))$

Even after unbinding the update procedure from any specific state-visiting order, the agent still needs to visit each state infinitely often, for estimations to converge to the true real-world values. However, the approximations are now built in a more efficient way, because this one-step look-ahead allows the agent to learn from transitions asynchronously. For example, it is perfectly acceptable if the agent at some point during learning, has collected many samples for updating state's s_1 value and no samples at all for updating state's s_2 value, because it just so happened that he did not yet pass from state s_2. This exempts the learning algorithm from having to wait until a specific state

[20]Technically this poses no problem to the agent, since it is just another element of the evolving vector V of state values, that the agent has access to. Simply put, the learning agent keeps a value corresponding to each state at all times, and whenever leaving a state, he updates that state's estimated value, using the landing state's estimated value.

is reached before applying a value update, which would make learning in the real world quite more difficult.

After obtaining the state values, PI moves on to the Policy Improvement step:
$\pi_{m+1}(s) = \arg\max_{a \in A(s)} [R(s, a, s') + \gamma V^{\pi_m}(s')], \forall s \in S$

A key observation here is that, to implement Policy Improvement, the agent needs to know - for each state - which specific action is the one that seems to be leading to the maximal expected utility, because ultimately a policy is about choosing actions not states. Obviously, if the agent only knew which states were good, but had no idea about which actions were good, then he would not be able to evolve its current policy in any other way than by using a model-based approach. Using an estimated model, it would try to predict how often each available action leads to a profitable - according to the calculated values - state. Moreover, some actions might be even better than others simply thanks to their immediate rewards and not because they lead to better states, so the agent would have to model rewards as well. One straightforward way to take these factors into account in a model-free way, is to keep track of Q-values instead of state values. Ranking preferences over actions for every state (instead of ranking preferences over states), is a cornerstone idea and practice in the theory of RL [35].

2.5 Algorithms and Techniques

In this Section some of the most exemplary algorithms of RL are presented. These are deceptively simple and are typically extended in various ways before effectively tackling real-world problems. However, in their most basic form they already implement the core concepts that characterize RL.

2.5.1 Q-Learning

After seeing how a passive RL agent - learning in episodes - can use sample-based Policy Iteration, let us now consider an active RL agent - learning with every transition - who will similarly use a sample-based version of Value Iteration (VI). Again, the values will be estimated in a TDL fashion. The main difference now is that the agent gets to choose each of his actions, and is not blindly following a fixed policy[21]. Therefore, it is

[21]This is the difference between Active Learning and Passive Learning in general, as described in 2.3.

even more imperative for him to keep track of Q-values rather than simple state values, because he will be potentially redefining his policy at each time step. Hence the agent should implement a sampling version of Q-VI (instead of VI).

The sample-based Q-Value Iteration (Q-VI) is called Q-Learning and is based on the following update:
$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + (\alpha)sample_t \Leftrightarrow$
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + (\alpha)(sample_t - Q_t(s,a))$
where:

- $sample_t = R(s,a,s') + \gamma \max\limits_{a' \in A(s')} Q_t(s',a')$[22]
- t is the time step in which the agent is in state s and chooses action a.
- $t+1$ is the next time step in which the agent is in state s'.

If we roll out the iterative form of the update we get the equation:
$Q_{t+1}(s,a) = (1-\alpha)^t Q_1(s,a) + \sum_{i=1}^{t} [\alpha(1-\alpha)^{t-i} sample_i]$

Thanks to the learning rate α, the old Q-value estimates are not completely replaced but rather updated, which means that past samples are not thrown away in favor of the newest sample. On the contrary, their effect is still preserved within the current estimate $Q_t(s,a)$ with a weight factor of $1-\alpha$. More specifically, as is obvious from its rolled out form, the above recursive equation is an exponentially weighted average[23], which leads to newer samples being more important than older samples in a one-to-one comparison. This is a beneficial feature of TDL, because as learning progresses, the estimated values get closer and closer to the real values. And since they make part of the updates of other values, the more accurate they are, the more accurate those updates will also be. This is why, if the agent trusts recent “correct” values more than earlier “wrong” values, convergence will be faster. On the other hand, the learning rate α must continuously decrease in order to attain convergence, or else the estimations could keep fluctuating. These two requirements do not conflict. The former requires the new sample to weight more than any older sample weights right now: $\alpha > o\alpha(1-\alpha)^{t-i}$ which is true for $0 < \alpha \leq 1$. The latter requires the new sample to weight less than any older sample weighted back when it was new: $\alpha_{now} > \alpha_{then}$ which is true for a monotonically decreasing α with $\lim\limits_{t \to \infty} \alpha_t = 0$.

[22]It is reminded that immediate rewards were assumed to be, not only deterministic, but also stationary for any given transition. This is why in the Q-Learning transformation the reward is written as $R(s,a,s')$ instead of $R_t(s,a,s')$ or $R_{t+1}(s,a,s')$, which would roughly correspond to assuming that a reward is determined when the associated transition begins or when the associated transition ends respectively.

[23]Exactly like the equation in 2.4.2.

Algorithm 1 Q-Learning

Require: S, A, s_{term}, E (# training episodes), T (time-horizon, can be ∞)

1: Initialize $Q(s,a)$, $\forall s \in S$, $\forall a \in A(s)$.
2: $Q(s_{term}, \sim) = 0$
3: **for** $episode = 1, 2, ..., E$ **do**
4: Choose s_{init} (perhaps arbitrarily).
5: $s = s_{init}$
6: $time_step = 1$
7: **while** $time_step < T$ and $s \neq s_{term}$ **do**
8: Choose an action $a \in A(s)$ based on the policy derived from the Q-values.
9: Implement a and observe s' and $R(s,a,s')$.
10: $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a' \in A(s')} Q(s',a')]$
11: $s = s'$
12: $time_step = time_step + 1$
13: **end while**
14: **end for**

Identically to the episodic learning of the previous Section 2.4.2, if the update is run infinitely long, then again the estimate will converge to the true Q-value function: $\lim_{k \to \infty} Q^k(s,a) = Q^*(s,a)$. The conditions that guarantee this convergence are ignored here (albeit being simple), because there is little practical interest in asymptotic convergence. Finding good-enough sub-optimal policies fast-enough is of greater concern.

In addition, Q-Learning is an Off-policy learning algorithm, meaning that the agent eventually converges to the optimal policy, no matter which policy it is following (even a completely random one). This happens because of the max operator, which allows only the best choice's value to propagate back through the recursion. The On-policy counterpart of Q-Learning is the "State-Action-Reward-State-Action" algorithm, better known by its acronym SARSA.

2.5.2 SARSA

SARSA is a RL algorithm applying essentially the same update rule as Q-Learning:
$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + (\alpha)sample_t \Leftrightarrow$
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + (\alpha)(sample_t - Q_t(s,a))$
but now:
$sample_t = R(s,a,s') + \gamma Q_t(s', \pi(s'))$

The difference between Q-Learning and Sarsa is that the max operator is now gone, because the agent no longer cares about how "good" the next state can be, but rather about how "good" it is actually going to be (given the current policy). More specifically, the difference is found in the estimation of the next state's "worthiness" that is included in the newest sample, because Q-Learning ideally assumes that the agent will take an exploitative choice in that state, while Sarsa assumes that the agent might as well take an exploratory choice in that state, if that is what its current policy will dictate. As a consequence, the agent continuously estimates the Q-values that correspond to the current policy π, and as these evolve over time, the policy π itself also changes accordingly, at every time step[24], since different Q-values might surpass the previously biggest Q-values, and thus different actions might be chosen when behaving greedily. Therefore, the only difference between the two algorithms is in how Q-values are updated (Off-policy for Q-Learning, On-policy for Sarsa).

Algorithm 2 SARSA

Require: S, A, s_{term}, E (# training episodes), T (time-horizon, can be ∞)
1: Initialize $Q(s, a)$, $\forall s \in S$, $\forall a \in A(s)$.
2: $Q(s_{term}, \sim) = 0$
3: **for** $episode = 1, 2, ..., E$ **do**
4: Choose s_{init} (perhaps arbitrarily).
5: $s = s_{init}$
6: $time_step = 1$
7: **while** $time_step < T$ and $s \neq s_{term}$ **do**
8: Choose an action $a = \pi(s) \in A(s)$ based on policy π derived from the Q-values.
9: Implement a and observe s' and $R(s, a, s')$.
10: Choose an action $\pi(s') \in A(s')$ again based on the policy π.
11: $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[R(s, a, s') + \gamma Q(s', \pi(s'))]$
12: $s = s'$
13: $time_step = time_step + 1$
14: **end while**
15: **end for**

2.5.3 R-max

R-max [36] follows a radically different approach compared to Q-Learning and Sarsa. First of all, R-max is a model-based RL algorithm, meaning that it does not engage in value estimation, but rather tries to figure out the underlying model of the world, and make predictions based on its optimal solution (even if the model is inaccurate). Secondly, R-max is defined in the context of Stochastic Games (SGs), a more general (albeit similar) model than Markov Decision Processes (MDPs). As a result, R-max is

[24] Just like the policy constantly changes in Q-Learning.

able to generalize to adversarial settings, apart from applying to the stochastic settings as the ones examined so far. R-max polynomially (in time) converges to a near-optimal solution.

Games are actually models of Multi-Agent interactions, wherein each player (the learning agent and its opponents) gets to choose from a given set of actions, and receives a reward depending not only on his choice but on the combined choices of all players. R-max is concerned in particular with fixed-sum games, in which the sum of the payoffs of all players must be a fixed constant. In a Stochastic Game (SG), the players participate in a (possibly infinite) sequence of standard/stage games (out of a set of potential games). In other words, each state of the SG is associated with a fixed-sum game (represented as a matrix[25]), and after completing it, the players receive their payoffs and stochastically move on to a new game. The rewards and transitions are associated with the joint actions of each game, meaning with how the previous game was played by every player. An MDP, is a special case of a degenerate SG, in which state transitions depend on the actions of the learning agent alone. For example in the case of a 2-player game, if the adversary only has a single action available at each state, then the problem can be modeled by an MDP.

R-max implicitly deals with the Exploration-versus-Exploitation trade-off issue (ref for more information), by implementing an "optimism in the face of uncertainty" heuristic[26]. More specifically, R-max initializes the rewards of every game matrix to an upper bound value of R_{max}, from which its name stems. At the same time, R-max initializes all transition probabilities in a way that, all joint actions in every game deterministically (with probability of 1) lead to an additional fictitious game G_0. When enough feedback has been collected about some (stage-game, joint-action) pair, then R-max updates the corresponding entries in its estimated model. Whenever this happens, the algorithm then solves the updated model, and repeats the whole learning procedure. This results in the learning agent showing a preference towards unexplored games and actions over explored ones. The agent spends a polynomial number of steps exploring, and the rest exploiting. The optimal policy coming of the agent's fictitious model is (by design) always, either optimal or efficiently learning, with respect to the real model. The agent does not know which one of the two holds at every time step, because that depends on the opponent decisions as well.

[25] Also known as the "strategic" form of the game.

[26] This is also known as "optimistic initial values/conditions" or the "optimism under uncertainty" bias, and has been implemented in other RL algorithms as well.

This work is not interested in adversarial settings, because cloud resource allocation is generally better represented in a stochastic setting, and therefore does not go into the formal definition of the R-max algorithm. However, it was deemed important to mention how R-max works in principle, because it goes on to show the wealth of techniques that are involved in RL.

2.6 Applications

Every problem that can be mapped to a sequential decision-making process, irrespective of its specific domain, is a potential candidate for implementing RL. Therefore, RL has been applied for dealing with various tasks such as: robot acquisition of motor skills (e.g. robotic arms control and object manipulation, humanoid robot balancing and walking, robot sports, etc.), playing repeated games (poker, backgammon, Othello, chess, etc.), playing video-games, autonomous navigation and collision avoidance for unmanned land and airborne vehicles (self-driving cars, etc.), reducing energy waste and pollution (building energy conservation, coordination of urban traffic stop lights, dynamic power usage of devices, etc.), healthcare (optimizing treatments for dynamic diseases), control and risk-management in complex systems (network intrusion detection, catastrophic failure detection and restoration, etc.). RL has also received a limited but growing attention in economic problems, such as targeted marketing, product pricing, financial trading, and more.

Chapter 3

Multi-Armed Bandits

The Multi-Armed Bandit problem is a sequential decision-making problem archetype, defined in a rather simple setting of slot machines, also known as one-armed bandits. In a nutshell, it is the cumulative sum of rewards (i.e. the total return) maximization problem, that a gambler faces when many slot machines are available for him to play in turns. Choosing 1 one-armed bandit among K different one-armed bandits can also be regarded as choosing 1 arm among K different arms of a single K-armed bandit. Thus, the names Multi-Armed Bandit (MAB) and K-armed Bandit (KAB) are well justified. The gambler can only observe the materialized rewards resulting from his actual plays[1], and does not know in advance the slot machines' statistical properties.

3.1 Inspiration

Although the Multi-Armed Bandit (MAB) problem naturally has a "casino flavor" instilled in it, it has not been originally inspired by the slot machine context suggested by its name. In fact, in [37] Robbins already mentions a few practical implementations of improved population selection strategies in statistical experiments, namely comparing: manufacturing processes in industrial quality control, seed varieties in plant breeding, or treatment efficiency in medicine. In addition, he was also inspired by previous work regarding yet another practical application: accurately estimating India's jute crop [38]. This is in stark contrast to how other gambling games started being probabilistically modeled, such as the Red-and-Black game for example, which was first studied purely out of mathematical curiosity for the respective roulette game in [39]. In this regard,

[1] This is referred to as an Incomplete Feedback setting, in contrast to a Complete Feedback setting in which all of the arms are played and the gambler gets to observe, not only the reward that he has earned, but also the reward produced by every arm.

Odell's quote from [40] is characteristic: "Briefly, the book is a well-written treatise on the abstract theory of gambling which is in all honesty probably of little value as a working reference for the industrial mathematician and statistician."

3.2 Mathematical Origins

The Multi-Armed Bandit (MAB) problem dates back to 1933 [41] but was more formally introduced in [37], a seminal work in Statistics Theory dealing with the design of sampling experiments whose sample size and composition are both dependent on the observations themselves, rather than being fixed in advance of the experiment.

3.2.1 Probabilistic Models

The Multi-Armed Bandit (MAB) problem is essentially a whole category of problems that can be tackled with Probability Theory methods, since it concerns making the right choices in the face of uncertainty, based on knowledge extracted from measured quantities (using data as a means of reducing that uncertainty). As a result, it deals with both inference and prediction tasks for potentially very complex systems (even without full state-knowledge) and involves random variables and stochastic processes. But in order to understand how such tasks can be accomplished, it is a prerequisite to first realize that each MAB algorithm represents one of the two fundamentally different interpretations of the very nature of probability:

The first approach is known as the "Physical" or "Objective" or "Frequency" Probability, and claims that the probability of a random event denotes the relative frequency of that event's occurrence "in the long run", or more formally as the number of well defined random experimental trials approaches infinity. As a consequence of this mindset, Probability has nothing to do with the notion of opinion nor belief. This approach was formalized in 1866 [42].

The second approach is known as the "Evidential" or "Subjective" or "Bayesian" Probability, and claims that the probability of a random event denotes the degree of belief, or perhaps the measure of likeliness, of that event occurring, ranging from impossibility (0) to certainty (1). The higher the probability, the more certain one is that the event

will occur. But even more generally, probability can be the subjective plausibility of any statement at all. Thus, this mindset suggests that Probability is a representation of uncertainty and ignorance. Bayesian Probability Theory is sometimes even viewed as an extension to Aristotelian Logic [43].

These are clearly two very different answers to the philosophical question of "What is Probability?", but at the same time are of great practical importance, because they lead into two almost opposite approaches to Statistical Learning. In some problems the distinction might be negligible, but in the case of the MAB problem, where the gambler needs to draw conclusions from limited sample data to make informed playing decisions, the chosen type of inference can greatly affect the solution.

Under the Frequency Probability viewpoint or the "frequentist" approach as it is often called, the analysis is on variations of the data for a given model. The main idea is that the underlying parameters of the probability distribution are fixed and do not change with each repetition of the process. Variation is due to variability in the random sample, not in the probability distribution. As more and more samples are collected, the relative frequency of an event better approximates its true probability.

Under the Bayesian Probability viewpoint or the "Bayesian" approach, the analysis is on variations of the model for some given data. The main idea is now the reverse of the above: the underlying parameters of the probability distribution are not fixed and do indeed change with each new sample obtained. Bayesian Probability, combines expert knowledge with empirical data. The former is represented by a subjective prior probability distribution, in other words a prior assumption as to how the world really is. The latter is incorporated into a likelihood function. The normalized product of the prior and the likelihood results in the posterior probability distribution, which incorporates all the information there is to know about the environment.

3.2.2 Confidence versus Credibility

With concern to the MAB problem in specific, the most interesting difference between the "frequentist" approach and the "Bayesian" approach lies in their handling of uncertainty, and indeed results in two big families of algorithms as seen in Section 3.6.

"Frequentism" makes probabilistic statements about creating Confidence Intervals, given an unknown but fixed population/model parameter of interest. A Confidence Interval (CoI) is an observed range of values[2], that represents a good estimate of that population parameter [44]. Specifically, the CoI frequently includes the model parameter value if a well defined experiment is repeated. To quantify how frequently it does so, is the role of the Confidence Level (CoL)[3]: if CoIs are constructed across separate data analyses of repeated (possibly different) experiments, then the proportion of CoIs containing the true value of the model parameter will be equal to the CoL. However, it can also infrequently happen that none of these CoIs covers the value of the model parameter.
In other words, the CoL of the CoI indicates the probability that the CoI captures the true population parameter given a distribution of samples. It does not describe any single sample. For example, to be "99% confident that the true value of the parameter lies inside the CoI" means that 99% of the observed CoIs will indeed hold the true value of the parameter[4]. It is worth noting that after a single sample is observed, the parameter either resides in the newly constructed CoI or it does not, without the notion of "chance" ever entering into play. Greater levels of variance yield larger CoIs and less precise estimates of the model parameter.

"Bayesianism" makes probabilistic statements about a varying unknown population/-model parameter, given a fixed Credible Interval[5]. A Credible Interval (CrI), also called a "Bayesian CoI", by postulating a prior distribution for the parameter of interest, predicts that the true value of the parameter has a particular probability of residing in the CrI, given the data actually obtained. The size of the sample data-set, the required level of confidence, the population variability, all play a role in the size of the CrI.

To sum up the subtle yet crucial difference between the two approaches:
In "frequentist" statistics, a 95% CoI means that for a large number of repeated samples (picked from the same population), that calculated CoI will include the true value of the parameter 95% of the times. The probability that the non-random unknown parameter is inside the given interval is either 0 or 1 (either in or out), in other words, the parameter is fixed (does not have a distribution of possible values) while the CoI is a random variable (actually random variables since it is comprised of its bounds). Quoting Gordon Antelman, a 95% CoI is "an interval generated by a procedure that will give correct intervals 95% of the time" [45].

[2] The Lower or Upper Confidence Bounds (LCB and UCB) are the corresponding one-sided limits.
[3] Also known as the Confidence Coefficient.
[4] Typically Confidence Intervals are stated at a 95% Confidence Level.
[5] The Credible Region is the generalization of CrIs to multivariate problems.

In "Bayesian statistics" on the other hand, a 95% CrI, defined in the domain of a posterior probability distribution, means that indeed there is a 95% probability that the parameter falls inside that interval. CrIs are analogous to CoIs but differ on a philosophical basis: they treat their bounds as being fixed, while the estimated parameter as being a random variable.

3.2.3 Relation to Markov Decision Processes

The MAB problem can be viewed as a Reinforcement Learning problem defined over a degenerate MDP, which includes only a single state. In other words, the learning agent is always in the same state, but still gets to choose among different possible actions and receive a corresponding immediate reward, in each decision epoch. Therefore, the only main difference is that now the agent does not need to account for the interconnection of states, because there is none. Hence there is no long-term reward due to the state-space traversal, for the agent to consider. As a result, it is forced to take decisions only based on the immediate rewards produced[6].

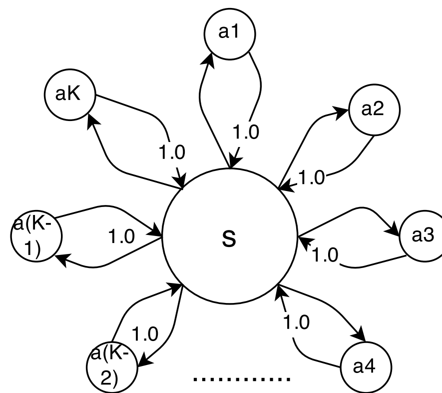

FIGURE 3.1: The Multi-Armed Bandit problem's setting can be viewed as a Markov Decision Process with only one state.

The relationship of a MAB problem to MDPs does not end there. Sometimes each arm is represented as an MDP itself, and every pull of the arm results in a state transition of the related MDP. This work however does not adopt that type of problem modeling.

[6]As mentioned in [46], there are two distinct types of action feedback: "evaluative" feedback indicating how good the action taken is, but not whether it is the best or the worst action possible and "instructive" feedback, indicating the correct action to take, independently of the action actually taken (typically used in Supervised Learning methods). Feedback in the MAB problem is of the former type.

3.3 Variations

Perhaps the most basic distinction of the MAB problem (and of all On-line problems in general), is the one between the Stochastic setting and the Adversarial setting.

In the Stochastic setting, each reward is drawn from a probability distribution characteristic of that specific arm. Usually, the same distribution type is used for all arms, albeit perhaps with different parameter values for each one. However, this is not a prohibition but rather a frequent simplifying assumption. Thus, each arm can potentially even represent a completely different distribution, as long as the rewards drawn from it are compatible with the rewards drawn from the rest.

In the Adversarial setting, the arms' rewards are not generated by stochastic processes. Instead, they are chosen by an adversary simultaneously with the gambler's choice of an arm. Two types of adversary are of interest: the oblivious adversary who knows the gambler's algorithm but does not observe the outcomes of any randomness in it, and the adaptive adversary who not only knows the gambler's algorithm but is also aware of the randomized results in it (only after they have happened). In either case, statistical estimation of the rewards is no longer adequate for forming an efficient policy, because rewards no longer come from stationary probabilistic distributions, but rather depend arbitrarily on the past. Simply put, rewards might adapt to the gambler's playing style in a malicious fashion. The Adversarial setting is therefore quite more difficult for the gambler and brings a shift of focus to the worst-case performance analysis of gambling strategies. Naturally, deterministic strategies tend to perform poorly in it, and randomized strategies are preferred [47–49].
The Adversarial setting exempts the modeler from the risk of choosing distributions that do not represent the real-world well-enough, but at the same time it is only suitable for particular kinds of problems. Due cloud resource allocation problem's nature, this work only considers the Stochastic setting, but the Adversarial setting could also be helpful, if for example cloud clients were assumed to be actively competing for resources, and each client's satisfaction depended on the actions of the rest.

The Stochastic Multi-Armed Bandit (S-MAB) problem's core idea presented above has sprung a long line of setting extensions:
In the Standard Stochastic Multi-Arm Bandit (SS-MAB) problem, the probability distributions of the rewards are fixed. We will later present this particular problem in more detail in Section 4.3.

In the Restless Stochastic Multi-Arm Bandit (RS-MAB) problem, the probability distributions of the rewards are not stationary. This is more appropriate when modeling problems such as network routing where transmission costs change over time, etc.
In the Contextual Stochastic Multi-Arm Bandit (CS-MAB) problem, the gambler uses features to encode domain context. This is a classical dimensionality reduction technique borrowed from supervised learning, useful in cases where the number of arms is so large that the gambler can not cope with all of them, and can only explore a relatively small portion.

There are also other parameters of the MAB problem that call for different solution approaches:

1. the number of arms (possibly infinite).
2. the number of trials (possibly infinite).
3. the existence or not of negative rewards.
4. the existence or not of switching costs.
5. the existence or not of delayed reward observations.
6. the existence or not of risk-aversion.

Each parameter comprises the focal point of a separate body of research work, and thus can not be deeply studied within the limits of this work. Their mentioning however serves as a hint to realizing that, despite its suspiciously simple setting, the MAB problem can indeed be used to model real-life situations and serve real-world needs.

3.4 Formal setting

Formally, the setting of the SS-MAB problem consists of:

1. the set of K arms (or levers):
   $L = \{1, 2, ..., K\}$, each representing a probabilistic distribution of rewards.
2. the set of actions (or action-space):
   $A = \{a_1, a_2, ..., a_K\}$, each corresponding to choosing the respective arm.

3. the normalized reward (or gain) vectors:
   $r_t = (r_{1,t}, r_{2,t}, ..., r_{K,t}) \in [0, 1]^K$

where:

- K is a finite integer
- $t \in [1, T]$
- T the number of discrete time steps (or trials or rounds).
- $l_t \in 1, 2, ..., K$, the arm chosen at time step t.

The random rewards produced by the arms are independent and identically distributed (i.i.d.), which means that their respective random variables all have the same distribution but are also mutually independent. This is a very common simplifying assumption in statistical inference applications.

Asymptotically optimal solutions (within an infinite time-horizon) to the SS-MAB problem are easy to obtain. For example, a naive strategy of equally choosing all arms, or a "play-the-winner" strategy of cycling through arms in a round-robin fashion but repeating a choice if its reward was above some "success" value, as well as many other simple deterministic heuristics, will indeed lead to eventually identifying the optimal arm with probability of one, as a result of the strong law of large numbers suggesting that the arms' parameters will overcome any "lucky" or "unlucky" streaks. Naturally, identifying the optimal arm also means achieving the optimal expected reward ever after, because the gambler can choose to commit to only playing that arm. In fact, any "Greedy in the Limit of Infinite Exploration (GLIE)" strategy will be optimal. In practice however, there will be a certain threshold that has to be achieved for the gambler to commit to an arm, and therefore a positive (albeit potentially small) probability of playing the wrong arm forever after.

In any case, given that the gambler can always eventually reach the optimal expected payoffs, the goal of maximizing the total return (cumulative discounted reward) reduces to the goal of minimizing the opportunity cost suffered while still trying to identify which arm is optimal [50]. Another way of viewing this is that, for the SS-MAB problem optimality still refers to maximizing the total return but only over a finite time-horizon. To this end, the metric of "regret" is often used to evaluate gambling strategies.

Regret[7] is the expected loss of the gambler due to him not always playing the best arm, and can be more formally defined as the difference between:
either
1) the total sum of the expected rewards of the arms actually played, and the total sum of the expected rewards had the optimal arm always been played instead[8].
or
2) the total sum of the materialized rewards of the arms actually played, and the total sum of the expected rewards had the optimal arm always been played instead.
Depending on if the regret is assumed to be, parameter-dependent or averaged over the parameters, it is called a frequentist regret or a Bayesian regret, respectively. Frequentist regrets are more often encountered in the related literature since they can be measured both for frequentist and Bayesian MAB algorithms.

The observation can be made that, in the SS-MAB problem, the interest shifts from "learning to behave optimally" to "optimally learning to behave optimally". This was not the case in the full Reinforcement Learning problem studied in Chapter 2, where state-space connectivity and transition non-determinism would have made it very difficult to strive for optimal learning. This does not mean that full RL techniques like Q-Learning do not care about converging fast (in fact regret can be defined in that setting too), just that they do not focus so much on optimal learning, whereas MAB techniques have the luxury to do so, thanks to the simpler setting over which they operate (see Section 3.5).

Since the gambler - given enough trials - will eventually find and commit to the optimal arm, regret per trial asymptotically approaches zero. Lately, a lot of research work has been looking into how regret accumulates in time, mainly inspired by findings in [50], where the following asymptotic inequalities, on the expected number of sub-optimal draws[9] $\mathbb{E}[T_n]$ and on the expected regret $\mathbb{E}[R_n]$ respectively, were proven to hold for single-parameter families of reward distributions:

$$\lim_{n\to\infty} \inf \frac{\mathbb{E}[T_n]}{log(n)} \geq \frac{1}{\inf\limits_{\theta\in\Theta_a:E|p_\theta|>\mu_{a*}} KL(p_{\theta_a},p_\theta)} = \frac{1}{KL_{inf}(\nu_a,\mu_{a*})}$$

$$\lim_{n\to\infty} \inf \frac{\mathbb{E}[R_n]}{log(n)} \geq \sum_{a:\mu_a<\mu_{a*}} \inf_{\theta\in\Theta_a:E|p_\theta|>\mu_{a*}} \frac{\mu_{a*}-\mu_a}{KL(p_{\theta_a},p_\theta)} = \sum_{a:\mu_a<\mu_{a*}} \frac{\mu_{a*}-\mu_a}{KL_{inf}(\nu_a,\mu_{a*})} \Rightarrow$$

Thus also proving that for any optimal - in terms of minimizing the gambler's regret -

[7]For dynamic problems where the the expected rewards of arms change over time, "Strong Regret" compares to the total sum of the expected rewards had each round's optimal arm been played (potentially a different arm in each round), and "Weak" Regret compares to the total sum of the expected rewards had the overall optimal arm been always played (the same arm in each round). In our simpler case of rewards produced by fixed i.i.d. processes, Weak Regret and Strong Regret coincide.

[8]Sometimes called the "expected" Regret.

[9]As well as that the optimal arm is played exponentially more often than any other, in the limit.

strategy, that minimal regret asymptotically grows logarithmically with the number of trials n:
$\Rightarrow R_n(\theta_1, \theta_2, ..., \theta_k) \sim [\sum_{a:\mu_a<\mu_{a^*}} \frac{\mu_{a^*}-\mu_a}{KL_{inf}(\nu_a,\mu_{a^*})}] * logn$, as $n \to \infty$
where μ_a is the expected mean of the distribution ν_a, μ_{a^*} is the expected best mean out of all the distributions of the arms, and $KL_{inf}(\nu, \mu)$ is the Kullback-Leibler divergence between distribution ν and the distributions (in the model of the problem) with expectations greater than μ.
Strategies that reach the lower bound of the aforementioned inequalities are considered to be asymptotically optimal.

To address the SS-MAB problem in a Bayesian way, the setting must be extended to include:

1. the set of past observed data:
   $D(t) = \{d_1, d_2, ..., d_t\}$, $d_t = (x_t, a_t, r_t)$
2. the likelihood function:
   $\Pr(r \mid a, x, \theta)$

where:

- $t \in [1, T]$
- T the number of discrete time steps (or trials or rounds).
- θ the parameters of the distribution.

The posterior distribution of these parameters is given by the Bayes rule:
$P(\theta|\mathbf{D}) = P(\theta)\frac{P(\mathbf{D}|\theta)}{P(\mathbf{D})}$

3.5 Exploration versus Exploitation

In Chapter 2, the Exploration versus Exploitation (EvE) trade-off in Reinforcement Learning problems was briefly discussed. Thanks to its simple non-associative setting, the Multi-Armed Bandit (MAB) problem even more effectively highlights the need for balancing this trade-off (which concerns all sequential decision-making problems under uncertainty). On the one hand, the gambler needs to explore all arms, because he does not know for sure which ones produce the biggest rewards, while on the other hand, he also needs to exploit the specific arms that he has come to expect yield the best

immediate results.

This relates to the idea of optimally learning the optimal arm, defined previously as a way of learning that minimizes the regret, or equivalently that maximizes the total payoff for a finite number of trials[10]. However it is as of yet unclear how different arm selection strategies can affect performance in that respect. In the rest of this Chapter, such arm selection strategies are presented, that try to balance the Exploration versus Exploitation (EvE) trade-off in different ways.

The first and simplest type is "undirected" exploration, which is based on sample averages. When following undirected exploration methods, the gambler assigns a value to each arm, usually some numerical average of the rewards observed from playing that arm in the past, or some other aggregate value that he deems fit. Accordingly, in every decision step, the gambler chooses an action based on the arms' current values. If the gambler exclusively considers these values, then the selection of an arm can only be exploitative, completely ignorant of the underlying uncertainty.

Perhaps the simplest undirected exploration method is for the gambler to always take the purely greedy action, meaning to choose an arm with value equal to the maximum out of all arm values. Another method is the hybrid that comes out of forcing some exploration into a greedy method, as is the case with the "ϵ-greedy" strategy, under which the gambler takes the greedy action with probability $1 - \epsilon$, and a random action with probability ϵ. Of course, ϵ-greedy has poor asymptotic efficiency since it keeps doing the exact same amount of exploration long after the optimal solution has become obvious. With concern to this, a viable update is to make the ϵ parameter decrease over time, thus enabling convergence to the optimal choice. For cases in which sub-optimal arms strongly differ in their rewards, a better alternative is to implement stratified sampling, for example a "soft-max" strategy, where the gambler chooses every arm with probability equal to a graded function of its value, so that the better an arm is considered to be, the more often it will be pulled[11]. The typical soft-max function is: $\sigma(\alpha) = \frac{\exp^{V(\alpha)/\tau}}{\sum\limits_{k=1}^{K} \exp^{V(\alpha_k)/\tau}}$, $\alpha \in A$. The so-called "temperature" τ is a positive tuning parameter that imposes how alike or different the action probabilities will be. For $\tau \to 0^+$ soft-max becomes greedy, just like ϵ-greedy does for $\epsilon \to 0$. Thus, again for reasons of asymptotic efficiency, a promising modification is to gradually decrease τ over time.

[10] In certain cases, one might care more about minimizing the number of trials, or some other metric other than regret.

[11] This type of exploration strategies is sometimes called "value-driven" or "utility-driven" exploration.

A second type is "myopic" exploration. When following myopic exploration methods, the selection of an action takes into account the uncertainty about the arms' value estimates. It does so by trying to reduce uncertainty wherever it exists, irrelevant of the long-term impact of actions on future rewards.

For example, Thompson Sampling and Upper Confidence Bound techniques (see Section 3.6) implement myopic exploration methods. Thompson Sampling in specific, implements randomized Probability Matching, which allocates an observation to an arm α with a probability w_α specific to that arm. These algorithms and the corresponding exploration methods are examined in more detail in the next Section.

A third type is "belief-lookahead" or "fully-Bayesian" exploration. When following belief-lookahead exploration methods, the decision criterion is maximization of the expected cumulative discounted reward (over the rest of the decision sequence) based on model assumptions and posterior approximations.

For example, the Gittins Index algorithm implements belief-lookahead exploration.

As previously argued, strategies that try to balance the EvE trade-off, even simple ones like ϵ-greedy, are asymptotically optimal because they eventually try each arm an infinite number of times. On the contrary, strategies that always choose greedily (right from the start) completely favor exploitation over exploration, which means that they are susceptible to getting stuck to local optima.
But even in the short run, it is not preferable to only exploit and never explore. For example, if rewards are noisy then the gambler needs to explore more in order to find the optimal arm. Only if the reward variances are all zero, will a purely greedy strategy always be optimal, because in that case it will only need to try each arm just once to find out its true value. In addition, for deterministic (and noiseless) but non-stationary rewards, exploration is again necessary to detect the temporal changes in the optimality of arms.

3.6 Algorithms and techniques

3.6.1 Gittins Index

The Gittins Index is a classical lookahead approach at balancing the EvE trade-off in the MAB problem. The corresponding policy proposed by this method is known as the Dynamic Allocation Index (DAI) policy, or simply the "Gittins Index rule", and the original (out of the many that followed) proof of its optimality is known as the DAI Theorem [51]. The DAI policy makes use of a forwards induction policy in which, an index is kept and updated for each arm, and the arm chosen is the one having the maximal index. That index represents the expected cumulative reward over the rest of the decision-making process and depends only on the history of draws of that specific arm. In [52], the authors showed that the Gittins Index rule samples only one action infinitely often and that this action is sub-optimal with positive probability, meaning that learning is incomplete. However, the real reason why Gittins indices are rarely used in practice, is that they are computationally very demanding (they repeatedly perform Dynamic Programming recursions on reduced models of a Markov Decision Problem), and apply only to specific reward distributions. In general, all lookahead approaches quickly become intractable as the setting of the problem grows.

3.6.2 Thompson Sampling

Thompson Sampling (TS) is one of the oldest exploration strategies proposed for sequential sampling [41], yet it has only very recently again received attention [53], mostly thanks to its low computational cost and practical effectiveness [54]. The basic idea of TS is the so-called Probability Matching: each arm is drawn according to its probability of being the optimal arm, which implies that "bad" arms are picked decreasingly often, but are not completely discarded. Specifically, TS implements randomized Probability Matching as its heuristic.

TS takes a Bayesian approach at doing that, since it requires a prior distribution on the parameters of every arm's reward distribution. At any time step, the gambler plays an arm according to its posterior probability of it having the biggest mean, or equivalently of being the best arm. In other words, TS draws a sample from each arm's Bayesian posterior distribution, and then compares those samples to each other to decide which arm to play next. However, contrary to "full-Bayesian" (or "Bayes-optimal" as they are called) methods, like Gittins Indices, that directly maximize the expected cumulative

reward, TS learns myopically, and can therefore be implemented much more efficiently.

Even though the concept of Probability Matching is a more general one, TS is typically used to tackle specifically the Bernoulli SS-MAB problem, where the k-th arm's reward follows a Bernoulli distribution with mean θ_k^*. The Bernoulli setting was the original consideration in [41], but its popularity among Bayesian MAB algorithms is mostly due to the mean reward of each arm being conveniently modeled by means of a Beta distribution, which is the conjugate distribution of the Binomial distribution. This property makes implementing the Bayes Rule computationally efficient. The learning agent keeps a different Beta distribution for each arm, as its prior, and gradually fits it to the data received whenever the corresponding arm gets played. Nowadays, thanks to the computational power of modern systems the class of reward distributions that can be efficiently modeled has broadened significantly.

Algorithm 3 Thompson Sampling for Bernoulli reward distributions

Require: K (number of arms), α and β (Beta distribution parameters)
1: $S_a = 0,\ a = 1, 2, ..., K$ (each arm's current number of successes)
2: $F_a = 0,\ a = 1, 2, ..., K$ (each arm's current number of failures)
3: **for** $t = 1, 2, ..., T$ **do** ▷ Pull the arm that draws the maximal posterior distribution sample
4:     **for** $a = 1, 2, ..., K$ **do**
5:         Draw θ_a from $Beta(S_a + \alpha, F_a + \beta)$.
6:     **end for**
7:     Play the arm $\arg\max_{a\in\{1,2,...,K\}} \theta_a$.
8:     Observe the reward $r = R(arm = a)$.
9:     **if** r=1 **then**
10:         $S_a = S_a + 1$
11:     **else**
12:         $F_a = F_a + 1$
13:     **end if**
14: **end for**

As previously stated, TS has only very recently been more deeply investigated, both numerically and theoretically.

In 2011 [55], the first non-trivial asymptotic upper bound for the expected regret was given, for the SS-MAB problem with Beta priors:
$\mathcal{O}(\frac{\ln T}{\Delta} + \frac{1}{\Delta^3})$, for $K = 2$
$\mathcal{O}([(\sum_{i=2}^{n} \frac{1}{\Delta_i^2})^2] \ln T)$, for $K > 2$
where Δ is the difference between the optimal and sub-optimal expected rewards.

In 2012 [56], inspired by standard analyses of frequentist index policies, the first finite-time regret bound for the same problem was derived:
$R(T) \leq (1+\epsilon) \sum\limits_{\alpha \in A:\mu_\alpha \neq \mu^*} \frac{\Delta_\alpha(\ln(T)+\ln(\ln(T)))}{K(\mu_\alpha,\mu^*)} + C(\epsilon, \mu_1, \mu_2, ..., \mu_K)$
where $C(\epsilon, \mu)$ is a problem-dependent constant $\forall \epsilon > 0$.
Moreover, asymptotic optimality follows from the above bound.

In general, Bayesian methods have also proven to be efficient even with respect to frequentist measures of performance, such as the cumulative regret [57]. Thompson Sampling is also particularly good for batch learning, where the gambler plays many times before seeing the rewards produced by his plays and making an update. This is actually true for any randomized action-selection method, and in contrast to deterministic methods, which will prefer the same arm for the whole duration of an update cycle, thereby adding variance to the total return.

3.6.3 Upper Confidence Bound

The Upper Confidence Bound (UCB) algorithm [58] was the first in a series of algorithms that achieved good finite-time regret, meaning that while those algorithms are running, the accumulated regret presents a logarithmic behavior uniformly over time (instead of only asymptotically). UCB uses an estimation of the variance of rewards, and in specific, the upper confidence bound (hence the name) of the empirical mean of each arm's rewards. What this means is that, UCB calculates one side of the Confidence Interval (CoI) into which the average reward falls with overwhelming confidence. There are various ways to secure a good enough Confidence Level (CoL). The original UCB algorithm is a "distribution-free" algorithm that uses Chernoff-Hoeffding bounds (derived from Hoeffding's inequality [59]). This is implemented through means of an index representing how "good" each arm is considered to be. The UCB Index is akin to the Gittins Index in the sense that both are quantities indicating the optimality of the corresponding arm, but apart from that they are completely different, because UCB does not adopt a Bayesian probability perspective and prefers the frequentist approach explained in the above paragraph.

Even though UCB is efficient, its regret is shown in [58] to be upper bounded by:
$\mathbb{E}[R_T] \leq C(\sum\limits_{a:\mu_a<\mu^*} \frac{1}{\mu^*-\mu_a}) \log(T) + o(\log(T))$, where C is a constant, which does not guarantee optimality in the sense of [50].

Algorithm 4 Upper Confidence Bound

Require: K (number of arms)
1: $S_a = 0,\ a = 1, 2, ..., K$ (each arm's current accumulated reward)
2: $T_a = 0,\ a = 1, 2, ..., K$ (each arm's current number of pulls)
3: **for** $a = 1$ to K **do** ▷ Pull each arm once
4: $T_a = 1$
5: $S_a = R(arm = a)$
6: **end for**
7: **for** $t = K + 1$ to T **do** ▷ Pull the arm that has the maximal UCB Index
8: **for** $a = 1$ to K **do**
9: $U_a(t) = \hat{x}_j + \sqrt{\frac{2\ln(t)}{T_a}}$
10: **end for**
11: Play the arm $a = \arg\max_{a \in \{1,...,K\}} U_a(t)$
12: Observe the reward $r = R(arm = a)$
13: $T_a = T_a + 1$
14: $S_a = S_a + r$
15: **end for**

There exist quite a few variants of the UCB algorithm, depending on how the confidence interval is chosen, with some being more promising than others. In [60], the authors proposed such a policy, called Minimax Optimal Strategy in the Stochastic case (MOSS), which achieves the distribution-free optimal rate[12] and a good distribution-dependent rate[13]. In particular, MOSS sets the index of an arm that has been drawn more than $\frac{n}{K}$ times equal to the empirical mean of its rewards, and the index of any other arm equal to the upper confidence bound: $B_{j,s} = \hat{X}_{j,s} + \sqrt{\frac{\max(log(\frac{n}{Ks}),0)}{s}}$, where $s = T_j(t-1) \geq 1$ is the number of times arm j has been pulled up until time step t. All indices are initially set to $+\infty$.

3.6.4 Bayes Upper Confidence Bound

In addition to Thompson Sampling (TS), another Bayesian index policy is Bayes Upper Confidence Bound (Bayes-UCB) [57]. Bayes UCB relies on calculating quantiles of the posterior distribution, a procedure which is more costly than simply drawing a sample from the posterior for each arm, as TS does. For binary bandits, Bayes-UCB is proven to be asymptotically optimal as a consequence of its finite-time regret bounds.

[12]In specific: $\sup R_n \leq 49\sqrt{nK}$

[13]In specific: $R_n \leq 23K \sum_{j:(\mu^* - \mu_j)>0} \frac{\max(log(\frac{n(\mu^*-\mu_j)^2}{K}),1)}{(\mu^*-\mu_j)}$

Algorithm 5 Bayesian Upper Confidence Bound

Require: K (number of arms), T (time-horizon), Π^0 (initial prior), c (quantile parameters)

1: $S_a = 0$, $a = 1, 2, ..., K$ (each arm's current accumulated reward)
2: $T_a = 0$, $a = 1, 2, ..., K$ (each arm's current number of pulls)
3: **for** $t = 1$ to T **do** ▷ Pull the arm that
4: **for** $a = 1$ to K **do**
5: $q_a(t) = Q(1 - \frac{1}{t(\log(T))^c}, \lambda_a^{t-1})$
6: **end for**
7: Play the arm $a = \arg\max_{a \in \{1,...,K\}} q_a(t)$
8: Observe the reward $r = R(arm = a)$
9: Update posterior Π^t in a Bayesian fashion: $\pi_a^t(\theta_a) \propto \nu_{\theta_a}(X_t)\pi_a^{t-1}(\theta_a)$.
10: **end for**

As in all Bayesian MAB algorithms, if the initial prior is improperly chosen, then Bayes-UCB will need more time to learn and be effective.

3.6.5 Minimum Empirical Divergence

In [61], the Minimal Empirical Divergence (MED) algorithm for the SS-MAB problem was presented. MED is another probability matching selection method, in which each arm is pulled with a probability reflecting how likely it is to be the optimal arm, here interpreted as a maximum likelihood. Therefore, MED is not an index policy (like Gittins Indices, UCB, etc.), and adopts a Bayesian approach to the MAB problem, however, it is unlike Thompson Sampling which interprets the probability of optimality as a posterior probability and thus requires the specification of a priori probabilities and derivations of distributions. MED on the contrary, uses a univariate convex optimization method to efficiently compute the minimum empirical divergence $D_{min}(\hat{F}_j(n), \hat{\mu}^*(n), \mathcal{A})$ in every decision round, where more specifically:

- $\mathcal{A} \equiv \{F : |supp(F)| < \infty, supp(F) \subset [a, b]\}$ is the family of distributions F with a finite bounded support $supp(F)$, and a, b are known constants that form an interval.
- $D_{min}(F, \mu, \mathcal{A}) \equiv \min_{G \in A: E(G) \geq \mu} D(F \| G)$, with $D(F \| G) \equiv \begin{cases} E_F & \text{if } \frac{dF}{dG} \text{ exists,} \\ +\infty & \text{otherwise .} \end{cases}$ being the Kullback-Leibler divergence[14] of F and G, a quantity that represents how distinguishable distribution F is from a distribution G with an expectated value

[14]More generally, the divergence $D(\cdot \| \cdot)$ is a function that quantifies the distance between two statistical objects.

$E(G)$ larger than μ, and with G^* being the distribution that minimizes this right-hand side quantity.

- $F \equiv (F_1, F_2, ..., F_j, ..., F_K)$ is the set of all the probability distributions F_j, each corresponding to the reward produced by arm j.
- $\hat{F}_j(n)$ is the empirical distribution of rewards produced by arm j, at the n-th decision step.
- $\mu_j \equiv E(F_j)$ is the expected value of the reward produced by arm j.
- $\mu^* = \max_j \mu_j$ is the maximum - out of all the arms - expected value of the reward produced.
- $\hat{\mu}^*(n)$ is the current (at the n-th decision step) maximum - out of all the arms - sample mean of the reward produced.

The rationale behind MED's minimizing of $D(F\|G)$ is the fact, that the maximum likelihood of an arm j being the optimal arm is proportional to $\exp(T_j(n)D_{min}(\hat{F}_j, \hat{\mu}^*))$, where $T_j(n)$ is the number of times that arm j has been pulled in the first n time steps, and that it is desired to minimize the posterior expectation of regret, which involves terms of the form $n * \exp(T_j(n)D_{min}(\hat{F}_j, \hat{\mu}^*))$. However, it is out of this work's scope to go any deeper into the mathematics of the convex optimization problem.

A deterministic version of MED, namely DMED, was introduced in [62], while the original MED proposal was a randomizing policy. DMED pulls the arms in loops[15]. Throughout a loop, the arms that will be pulled in the next loop are chosen.

3.6.6 Kullback-Leibler Upper Confidence Bound

In [63], the authors presented the Kullback-Leibler Upper Confidence Bound (KL UCB) algorithm. KL-UCB's regret satisfies: $\limsup_{n\to\infty} \frac{\mathbb{E}[R_n]}{\log(n)} \leq \sum_{a:\mu_a<\mu_{a^*}} \frac{\mu_{a^*}-\mu_a}{d(\mu_a,\mu_{a^*})}$, where $d(p,q) = p\log(p/q) + (1-p)\log((1-p/(1-q))$ is the Kullback-Leibler divergence between two Bernoulli distributions with respective parameters p and q. However, this bound holds for every reward distribution bounded in $[0,1]$[16], not only for the Bernoulli distribution. KL-UCB is proven to be strictly better than UCB, and the first asymptotically optimal (with respect to [50], as mentioned in Section 3.4) index solution for binary rewards.

[15] In no decision epoch, can the list L_C be empty, because at least the current best arm will satisfy the event $J'_t(a) \equiv \{T_a(t)D_{min}(\hat{F}_a(t), \hat{\mu}^*(t)) \leq \log(t) - \log(T_a(t))\}$ since it has $D_{min}(\hat{F}_a(t), \hat{\mu}^*(t)) = 0$.

[16] And thus all bounded reward distributions in general.

Algorithm 6 Deterministic Minimum Empirical Divergence

Require: K (number of arms), Π^0 (initial prior), c (quantile parameters)
1: $L_C = \{1, 2, ..., K\}$ (list of all arms to be pulled in the current loop)
2: $L_R = \{1, 2, ..., K\}$ (list of arms remaining to be pulled in the current loop $L_R \subset L_C$)
3: $L_N = \emptyset$ (list of arms to be pulled in the next loop, ordered according to how many times it has holded for each that $T_a(t)D_{min}(\hat{F}_a(t), \hat{\mu}^*(t)) \leq \log(t) - \log(T_a(t))$)
4: **for** Each Loop **do**
5:     **for** $a \in L_C$ (in ascending order) **do**
6:         $t = t + 1$
7:         Play the arm a.
8:         $L_R = L_R \setminus \{a\}$
9:         **for** $a' \notin L_R$ for which $T_{a'}(t)D_{min}(\hat{F}_{a'}(t), \hat{\mu}^*(t)) \leq \log(t) - \log(T_{a'}(t))$ **do**
10:             $L_N = L_N \cup \{a'\}$ (no duplicates)
11:         **end for**
12:         $L_C = L_N$
13:         $L_R = L_N$
14:         $L_N = \emptyset$
15:     **end for**
16: **end for**

As seen in the following algorithm, KL-UCB involves an optimization problem, just like MED does, because both deal with the problem of finding the KL-confidence region as if it were a convex optimization problem, and in specific the problem of maximizing a linear function on the probability simplex under Kullback-Leibler constraints. In mathematical terms, at each time step t, KL-UCB associates each ucb $U_a(t)$ with the expectation μ_a of the distribution ν_a that corresponds to arm a, and plays the arm with the highest $U_a(t)$.

Algorithm 7 Kullback-Leibler Upper Confidence Bound

Require: T (time-horizon), K (number of arms)
1: **for** $t = 1$ to K **do** ▷ Pull each arm once
2:     $N[t] = 1$
3:     $S[t] = R(arm = t)$
4: **end for**
5: **for** $t = K + 1$ to T **do** ▷ Pull
6:     **for** $i = 1$ to K **do**
7:         $U_a(t) = \sup\{E(\nu) : \nu \in \mathcal{D}, KL(\Pi_{\mathcal{D}}(\hat{\nu}_a(t)), \nu) \leq \frac{f(t)}{N_a(t)}\}$
8:     **end for**
9:     $\alpha = \arg \max_{a \in \{1,...,K\}} U_a(t)$
10:     $r = R(arm = a)$
11:     $N[a] = N[a] + 1$
12:     $S[a] = S[a] + r$
13: **end for**

An adaptation of KL-UCB to the specific case of Bernoulli case was also given in [63], known as the Clopper-Pearson Upper Confidence Bound (CP-UCB), which as suggested

by its name, simply computes the ucb in a different manner. CP-UCB is only marginally better than KL-UCB for Bernoulli rewards.

Algorithm 8 Clopper-Pearson Upper Confidence Bound

Require: n (time-horizon), K (number of arms)
1: **for** $t = 1$ to K **do**
2: $N[t] = 1$
3: $S[t] = R(arm = t)$
4: **end for**
5: **for** $t = K + 1$ to n **do**
6: $\alpha = \arg\max_{1 \leq a \leq K} u^{CP}(S[a], N[a], \frac{1}{t \log(t)^c})$
7: $r = R(arm = a)$
8: $N[a] = N[a] + 1$
9: $S[a] = S[a] + r$
10: **end for**

In [64], two versions of KL-UCB are distinguished, the first aimed at one-parameter exponential families of distributions, and the second, called Empirical KL-UCB, for bounded and finitely supported distributions.

3.7 Applications

Many sampling experiments can be effectively represented as MAB problems. Therefore, MABs have been applied for various tasks such as: comparing treatments in clinical trials, advertisement display, news article recommendation, and commenting systems over the Internet, adaptive network routing, channel selection in multi-channel wireless (and other communication) systems, sensor management, manufacturing systems and product improvement experiments in general, portfolio design.

Chapter 4

Resource Allocation Modeling

In order to achieve efficient dynamic cloud resource expansion/contraction, at any level of granularity, there is a need for automating the corresponding decision-making procedures, because manual provisioning is neither rapid nor exact. Reinforcement Learning and Multi-Armed Bandits are two closely related fields, both presenting some potential to aid in this respect.

4.1 Why Reinforcement Learning?

As explained in Chapter 2, RL provides online real-time learning, which is useful for controlling systems whose model is either unknown (or just too complex and therefore chosen to be ignored). In addition, it provides both model-based and model-free learning solutions, meaning that the agent can either try and learn the model or completely ignore it, and still optimize its behavior over that particular model. In our simulations, we will examine a model-free active learning approach, by implementing the classical tabular Q-Learning algorithm.

Depending on the specific provisioning problem, RL alone might not suffice. For example, if economic models are adopted, then perhaps a more game-theoretic approach might be useful, which will take into account conflicting or compatible personal incentives in a Multi-Agent System setting, or if the problem is precisely how to allocate Virtual Machines (VMs) into Physical Machines (PMs) then usually bin-packing algorithms are helpful, and so on. This does not mean that learning becomes irrelevant to these problems, only that RL will need to implemented in specialized ways (for example learning in an adversarial setting is dealt with differently than in a stochastic setting)

and in synergy with various other methods.

4.2 Why Multi-Armed Bandits?

As seen in Chapter 3, MABs too provide methods for online real-time learning, just like RL does. Actually, a MAB problem is considered to be a type of RL problem. The classical MAB setting however, does not deal with delayed rewards nor does it account for any state interconnection. Actually, it does not involve any states at all. Therefore, it is better suited for when the consequence of each action is discretely quantifiable and immediately attributed to a particular action, whereas RL is better suited for more complex environments where behavioral patterns might exist that sacrifice short-term rewards to achieve greater future rewards. Multi-armed Bandit research has made some very important breakthroughs over the last years, and state-of-the-art MAB algorithms are more promising with respect to real-world applicability than MAB algorithms ever were before.

By not accounting for the synergistic effect of multiple actions, and instead assuming an immediate reward is clearly attributed to a specific action, the MAB setting is free to focus on the finer details of the decision-making mechanism, which is why the MAB literature is usually more mathematically oriented than the RL literature.

It is very important to note that, for a realistic cloud resource provisioning system, an SS-MAB setting would be inadequate. This is because a real cloud environment is dynamic even in the simplest of cases. For example, for a system that is scheduling client requests (tasks, jobs, queries, etc), the current incoming request might be better served if a particular action is taken or equivalently if a particular "arm" is "pulled", but the next request might be better served by another arm, and the request after that by yet another arm, depending on the exact type of request. Thus, the "gambler" needs more data upon which to base its decisions, similarly to how an RL agent bases its decisions on the current state, because the type of the request (among other things) defines the optimality of a decision. In this work, Contextual Bandits have not been implemented, but as for most real-world applications, a system based on a CS-MAB setting is expected to perform better, by exploiting any helpful specific cloud-domain information.

4.3 TIRAMOLA

TIRAMOLA [65] is a modular cloud-enabled framework for automatic and real-time resizing of NoSQL clusters. As explained in Chapter 1, this is a very important task considering that IaaS providers offer VMs at a monetary cost, which gives a NoSQL cluster's administrators (the IaaS clients) strong incentive to avoid overconsumption of resources. Using TIRAMOLA, they can automatically ask the IaaS provider for the exact number of resources that the applications running in their cluster need, at any given point in time.

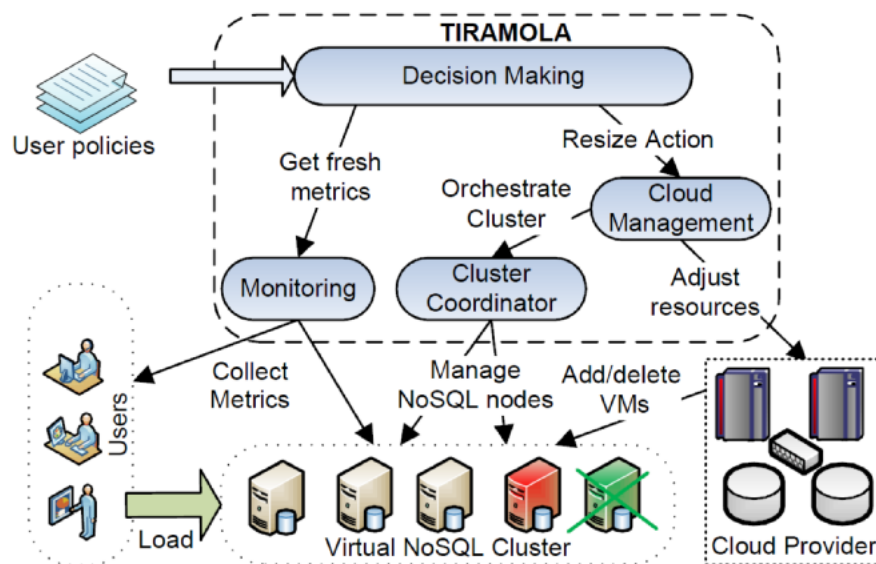

FIGURE 4.1: TIRAMOLA's architecture.

4.3.1 TIRAMOLA's decision-making

TIRAMOLA's decision-making module is responsible for deciding which cluster size optimizes certain performance criteria under some user-defined policy[1], in order for TIRAMOLA's cloud management module to proceed into requesting or releasing the appropriate number of VMs from the provider. Put simply, the decision-making module is the "brains" of the management system. Learning is implemented over a Markov Decision Process (MDP) model of the environment, a standard choice for applying Dynamic Programming (DP) or Reinforcement Learning (RL) algorithms and techniques that was covered in Section 2.2.1. In order to make informed decisions, TIRAMOLA's

[1] Not to be confused with a decision-making policy as in Section 2.2.2.

monitoring module periodically (and passively) observes various metrics, both server-related and user-related, and stores them in a log.

More specifically, TIRAMOLA models the world as a tuple $M(S, A, T, R)$ comprised of:

The set of states (or state-space):
$S = \{s_1, s_2, ..., s_n, ...\}$
Each state s_n represents the respective current number n of running VMs.

The set of actions (or action-space):
$A = \{no_op, add_1, remove_1, add_2, remove_2, ..., add_n, remove_n, ...\}$
Each action represents the respective control decision of adding or releasing a particular number of VMs, or keeping the current number of VMs intact.

Naturally, at every decision epoch the system might deem fit to prohibit some of the actions typically available. For example, it might be proper that - for a certain time period during which many cloud clients use the cloud provider's infrastructure - a particular client (like the NoSQL cluster under consideration) can only occupy between 4 and 7 VMs. In that case, the learning agent's set of allowed actions for each state permitted would be:
$A_{s_4} = \{no_op, add_1, add_2, add_3\}$,
$A_{s_5} = \{no_op, remove_1, add_1, add_2\}$,
$A_{s_6} = \{no_op, remove_1, remove_2, add_1\}$, and
$A_{s_7} = \{no_op, remove_1, remove_2, remove_3\}$ respectively.
This is enough to ensure that the system will not cross any boundaries imposed to it, because transitions are considered to be deterministic (as seen next), and therefore any action targeting an inbound state will not be at risk of landing outside the enforced state-zone[2].

The transition probabilities (or transition function):
$T(s, a, s') \in \{0, 1\}$, $\forall s, s' \in S$, $\forall a \in A$
$T(s_n, remove_x, s_{n-x}) = T(s_n, add_x, s_{n+x}) = T(s_n, no_op, s_n) = 1$
$T(s, \sim, s') = 0$ for all other transitions.

[2] On a side note, TIRAMOLA's action-space could equivalently be defined as:
$A = \{goto_s1, goto_s2, goto_s3, ..., goto_sn, ...\}$
The reason for this being that such an action-set is intuitively more straightforward for applying restrictions that directly refer to the number of VMs, while the original formulation is more suitable for limiting extreme transitions through the state-space. For example, the previous example of 4 to 7 VMs allowed, would now conveniently result in all states having the exact same set of permissible actions:
$A_{s_4} = A_{s_5} = A_{s_6} = A_{s_7} = \{goto_s4, goto_s5, goto_s6, goto_s7\}$.
In the rest of this Section we will alternate at will between the two formulations.

TIRAMOLA assumes deterministic transitions, but it can easily support stochastic ones too, by making the desired transition probabilities reside inside the $(0, 1)$ range, thus modeling uncertainty in the actions' outcomes (e.g. ask for 4 VMs but faultily receive only 3 VMs).

The transition rewards (or reward function):
$R(s, a, s') = 0$, $\forall s, s' \in S$, $\forall a \in A$
TIRAMOLA currently does not implement immediate rewards characterizing the transitions. Instead, it keeps a log of all the periodically monitored metrics, forming an incremental set of past experiences used for re-estimating state values.

So, how does TIRAMOLA act and learn over this MDP setting? At every decision epoch, the learning agent compares all the actions currently allowed, and (knowing that transitions are deterministic) greedily chooses the one that leads to the (seemingly) best next state. Certainly, the agent can only try to estimate which state will be best, simply because one can not predict the future, especially in complex systems, such as a running NoSQL cluster, where anomalies and randomness in actual performance will always be present (e.g. it might be the case that for no immediately obvious reason, 5 VMs perform much worse than both 4 or 6 VMs). This is exactly why, an adaptive control procedure is needed, because the environment is just too complex for the software to plan everything in advance, and it is instead perhaps better to constantly re-calibrate its "beliefs", based on what is actually going on in the world.
To quantify the notion of "best" in scalar values, at each decision point in time, the agent goes through the detailed history log of all past system "probings", and aggregates the experiences residing in it. This way, the agent calculates the estimated value $V(s')$ of each potential target state $s' \in S$. The aggregation is not based on some system-defined average, but rather on a utility function $U(s) = \phi(gains, costs)$ provided by the user, taking as parameters some measured quantities of interest, and producing a single scalar value indicating optimality. It is not the system's responsibility to define the utility function, because each user might be interested in optimizing different performance (or other) criteria. However, it is the system's responsibility to store all relevant metrics with which it will feed that utility function.
It is worth noting that the decision-making module functions at a higher level of abstraction and does not care about what these metrics really represent. This is an attribute of MDPs in general. As long as the user provides a proper utility function, and as long as the monitoring module correctly reports the measurements needed to evaluate that function, then the appropriate decisions will be taken. In any other case, the decision-maker can not be expected to choose wisely, no matter the nature of the problem. This shows the importance of issues other than the "pure" learning one, in achieving efficient

resource provisioning, as mentioned in Section 1.2. It also means that if monitoring is ever changed to include additional metrics (such as memory utilization or disk space utilization), then the decision-making module will not need to change its core logic, as long as it acquires a new suitable utility function taking into account the new metrics.

Naturally, the aforementioned collection of measurements forms a multi-dimensional dataset, because for each discrete monitoring time step, TIRAMOLA tracks down various metrics. However, the value of a potential target state is not simply an aggregation of all past metric records corresponding to that state. TIRAMOLA additionally keeps track of stimuli with every log entry. Stimuli are characteristics of the environment that are assumed to affect performance considerably when present. The idea behind this practice is simply to take advantage of the correlation between observed stimuli and measured performance, for better estimating the state values. For example, one such stimulus is the current incoming cluster workload, and thus the assumption made is that temporal variation of the workload is not exceptionally big. Therefore, whenever the agent finds itself in a decision epoch with an observable workload value of λ, it will only use a portion of its past experiences that involved similar rates of workload (close to λ), rather than all of them.
More specifically, for each potential next state, the agent will distinguish the subset of past experiences that has occurred in that state and under similar - to the currently observed - stimuli. Then, it will use a multi-dimensional clustering algorithm (k-means) on those measurements alone, in order to derive representative centroid data-points, and use them as the metric evaluations in calculating the utility. Finally, TIRAMOLA will compare the values that the utility function produced given the centroids (each representing a potential target state), and will choose the action that leads to the state with the maximal (estimated) utility.

In [66], TIRAMOLA was extended to account for one more stimulus, namely the type of workload applied to the NoSQL cluster. With this extension, TIRAMOLA monitors and uses the following metrics: cluster read throughput "thr_r", cluster write throughput "thr_w", read query latency "lat_r", write query latency "lat_w", cost per VM "$cost_{VM}$", and sure enough the number of VMs "$|VMs|$" (the MDP state itself). Other metrics are also monitored and could potentially be taken into consideration as well, such as cluster CPU load for example. Some metrics are considered to be gains while others losses in terms of their effect in the utility values, and a typical load-aware utility function would be: $A_r thr_r + A_w thr_w - B_r lat_r - B_w lat_w - C|VMs|$. By considering the type of load, the decision-making module can make more precise predictions, because it either considers

data related to past “read experiences” or to past “write experiences”, not both.

4.3.2 TIRAMOLA’s decision-making proposal

TIRAMOLA’s usage of unsupervised learning for approximating the values of potential landing states works well in practice [67], however, in this Section we propose a similar implementation that might scale better with the learning model.

TIRAMOLA’s MDP has a one-dimensional state-space which reflects the number of running VMs, but at the same time its decisions are based on more than one system state dimensions. The way in which TIRAMOLA deals with this, is by keeping the history of all observed metrics data, then choosing only a portion of these that is relevant to the current system observation (in specific the current amount and type of cluster workload), and subsequently only using that portion to calculate how good a potential target state is.
A k-means clustering algorithm is implemented for the above purposes. K-means clustering in general, partitions n observations into k clusters, with each observation being a d-dimensional real vector and belonging to the cluster with the nearest mean. This clustering problem is NP-hard, therefore K-means clustering algorithms usually implement heuristics and have a $\mathcal{O}(nkdi)$ running time complexity, where i is the number of iterations needed until convergence. TIRAMOLA’s implementation in specific is $\mathcal{O}(nk)$ [68], a low constant factor linear running time, which might still however, present scaling issues in a real-time system, in case n is allowed to grow large as observations are being gathered throughout the system’s life-time. Simply put, whenever TIRAMOLA makes a new observation, it goes through all similar past observations and recalculates “from scratch” an aggregation of the corresponding measurements, a process which might hurt real-time performance.

The alternative explored here is to include the various stimuli as part of the MDP state (in the form of additional random variables, apart from the number of VMs). This abides to the classical tabular Q-Learning algorithm archetype, where all the information necessary for making a decision is included in the current state. As a result, the MDP now has more states than before, and therefore the Q-Learning algorithm has to keep track of more Q-values. Exactly how many depends on how many dimensions are added and how each dimension’s values are quantized. For example, if only the cluster workload is added into the MDP state (along with the number of VMs), the agent will

store a separate Q-value, no longer for each (number of VMs, resizing action) pair, but now for each (number of VMs, amount of workload, resizing action pair) triple. The new model has two advantages, the first being that every decision-making iteration now takes constant time $\mathcal{O}(1)$, because it is as simple as performing a typical Q-value update and comparing a few[3] Q-values to each other. This is possible because now all of the past experiences are being conveniently integrated as they come, into the various Q-values that correspond to the different situations encountered. Secondly, it enables the modeler to use the typical Q-learning parameters: learning rate, discount factor, epsilon (in the case of ϵ-greedy), for fine-tuning the learning process according to the particular system's needs, instead of devising ad-hoc techniques on top of the clustering procedure. A disadvantage is that the new state dimensions will have to be discretized in advance, if value function approximation is to be avoided.

4.4 Problem Formulation

Matlab simulations were designed and run for the purpose of highlighting the potential that specific algorithms and techniques might hold for improving TIRAMOLA's decision-making performance. More importantly however, the main motivation is to explore "learning from experience" approaches as a new direction for dealing with cloud elasticity decision-making problems in general.

4.4.1 General Remarks

TIRAMOLA provides an ideal test-case for this work's purpose, because it adopts the cloud client's viewpoint, and thus deals with a simpler decision-making problem (when compared to the cloud resource scheduling problems that cloud providers face) which more accurately fits into a "light-weight" simulation. If on the other hand, the problem is tackled from the provider's perspective, then the setting would have to change accordingly to account for thousands of VMs (insted of just 20), for different types of clients or client applications, for interrelations between those, for synchronizing conflicting decisions (in case the provider uses two or more schedulers), and many other factors.

To elaborate on the generality of the learning techniques studied, TIRAMOLA controls the system through the number of running VMs, but some other framework could for

[3] Actually, equal to the number of actions allowed.

example control VM resizing instead. The exact same learning methods could be used in either case, because they do not depend on the nature of the controls. Similarly, when using Multi-Armed Bandits the learning agent views the arms as an abstract notion related to the "solution-space" of the problem, and asks itself which one among the available "temporary solutions" should it deploy right now. What the arms really represent is of no interest to the agent. As long as they are one-dimensional, arms could equally represent any system element that is controllable and expected to affect performance: the number of VMs, the amount of memory allocated, the amount of processing power allocated, a mapping of VMs to PMs, etc. This means that the same real-time learning algorithms can potentially be used for learning at different levels of granularity, even in a hierarchical way. Therefore, the same simulation tools can prove to be helpful for more than one problems.

Another important remark is that simulated performance and real-system performance will definitely differ, simply because any model in general is an abstraction of reality, and therefore will - on occasion - insufficiently represent it. This does not mean however that useful conclusions can not be drawn from simulations, but rather that not all results would be replicated in the real world. In fact, testing resource allocation policies on a simulator has some concrete advantages over testing on real clusters.
First of all, real-system testing is costly, and often prohibitively long-running in terms of natural time. Secondly, partial reward observability would render policy comparisons less direct. This means that, in the real world nobody will ever know the rewards that would have been produced, had the learning agent taken different actions. In contrary, simulations run over a model of the world that is completely known to the experimenter, and as a result, different methods can be precisely compared, at identical points in time and under identical conditions, if so desired. Moreover, many simulation runs can be statistically analyzed, whereas real-world trials can not be repeated many times and are therefore more prone to "luck" affecting their performance.

4.4.2 General Setting

In this thesis, scenarios were examined both for modeling the problem as a full Reinforcement Learning problem and as a Multi-Armed Bandit problem. The former keeps TIRAMOLA's underlying MDP formulation but explores it in new ways. The latter drops the MDP setting in favor of a simpler bandit setting.

In both types of simulations, the control is the number of VMs, with values ranging from 1 to 20.[4]

Discrete-time models are being used and therefore time in the x-axis is not measured in any particular natural time units, but rather in discrete time-steps[5]. This is helpful because as previously stated, different resource provisioning decisions might need to be made at different temporal scales. For example, TIRAMOLA makes a decision every minute or so, but some other framework might work in a finer-grained temporal scale that requires a decision to be taken every second. The same decision-making logic can be applied in both types of decision-making, especially since each decision iteration takes only a couple of msec (natural time) as measured from the simulations, providing enough temporal resolution to adapt to any real-time requirements.

4.4.3 Full Reinforcement Learning Setting

Throughout all of the full-RL simulations, increases or decreases by 1 VM are the only resizing actions allowed (contrary to TIRAMOLA which also allows larger resizing actions), and only non-negative rewards produced.

At first, the MDP includes a 1-dimensional state-space and transition rewards are fixed. More specifically:

1. The set of states (or state-space):
   $S = \{s_1, s_2, ..., s_n, ..., s_N\}$
   Each state s_n represents the respective number n of running VMs.
2. The set of actions (or action-space):
   $A = \{no_op, add_1, remove_1\}$
3. The transition probabilities (or transition function):
   $T(s_n, remove_1, s_{n-1}) = T(s_n, no_op, s_n) = T(s_n, add_1, s_{n+1}) = 1, \forall s_n \in S \setminus \{s_1, s_N\}$
   $T(s_1, remove_1, s_1) = T(s_1, no_op, s_1) = T(s_1, add_1, s_2) = 1$
   $T(s_N, remove_1, s_{N-1}) = T(s_N, no_op, s_N) = T(s_N, add_1, s_N) = 1$
   $T(s, a, s') = 0$, for all other transitions with $s, s' \in S$ and $a \in A$.

[4]Inspired by experiments in [65, 66, 69], where the number of VMs allowed ranged, from 9 to 24, from 1 to 16, and from 4 to 10 VMs, respectively.

[5]Viewed as "pulls" of an arm in the MAB case.

4. The transition rewards (or reward function):
   $R(s_n, a, s') = \max(\min(10 * n, load) - vm_cost * n, 0)$, $\forall n \in [1, N-1]$
   $R(s_1, remove_1, s_1) = 0$
   $R(s_1, no_op, s_1) = R(s_1, add_1, s_2) = \max(\min(10 * 1, load) - vm_cost * 1, 0)$
   $R(s_N, remove_1, s_{N-1}) = R(s_N, no_op, s_N) = \max(\min(10 * N, load) - vm_cost * N, 0)$
   $R(s_N, add_1, s_N) = 0$
   $R(s, a, s') = 0$, for all other transitions with $s, s' \in S$ and $a \in A$ [6].
   $load = 50$
   $vm_cost = 5$

All transitions are deterministic in the case of the one-dimensional state-space, meaning that the agent will always definitely land to the desired next state that it has chosen, or in other words, that it will get the desired number of VMs. Of course, there is no point in taking a decrease action when being in the state corresponding to only 1 VM, nor in taking an increase action when all N VMs are being used. This should ideally be taken care of, by disallowing the corresponding actions whenever the agent finds itself in those two states. On the contrary, in this simulator the two actions remain available, but result in no state change and achieve zero immediate reward, thereby not getting chosen at all. However, the former approach is more sound and should be preferred in production applications.
The immediate rewards are produced by a simple linear function of throughput and cost, where increases in the former give higher rewards while increases in the latter give lower rewards. This function should be user-specified, meaning that depending on the cloud user, it should be comprised of more variables, have different coefficients, and even be non-linear and very complex, if the user so desires. A simplistic assumption was made for producing the immediate rewards, that each VM can deal with 10K req/sec, and that therefore throughput is either equal to the number of VMs times 10K req/sec, or equal to the current incoming workload, whichever of the two is smaller. A random noise signal could have been overimposed for the observed throughput to better represent a real-world system, but this is not necessary for studying the learning performance in general. In addition, specialized scenarios can be implemented by setting the immediate rewards by hand, bypassing the throughput and reward formulas (to represent for example that a particular number of VMs behaves abnormally bad).
With respect to this, it is reminded that a RL learning agent simply observes each materialized immediate reward at the moment it earns it, not in advance. More specifically, in a real-system nobody can know what the throughput will be until it is actually observed straight from the monitoring tools, while in a simulation the throughput has to

[6] It does not really matter, since the probabilities of those transitions happening are all zero anyway.

be artificially produced. However, the agent is still unaware of the throughput, and as a result of the immediate reward as well, just like it would be in reality.

In the second part of the simulations, the MDP is extended to include a second dimension of states, namely the workload level. Hence, the set of states becomes:
$S = \{s_{1,1}, s_{1,2}, ..., s_{1,m}, ..., s_{1,M}, ..., s_{N,1}, s_{N,2}, ...s_{N,m}, ..., s_{N,M}\}$
Each state $s_{n,m}$ represents the respective number n of running VMs and the respective level m of the amount of workload currently observed.
The rationale is that even though workload is dynamic, it does have a temporal continuity and does not completely change from one moment to the next.

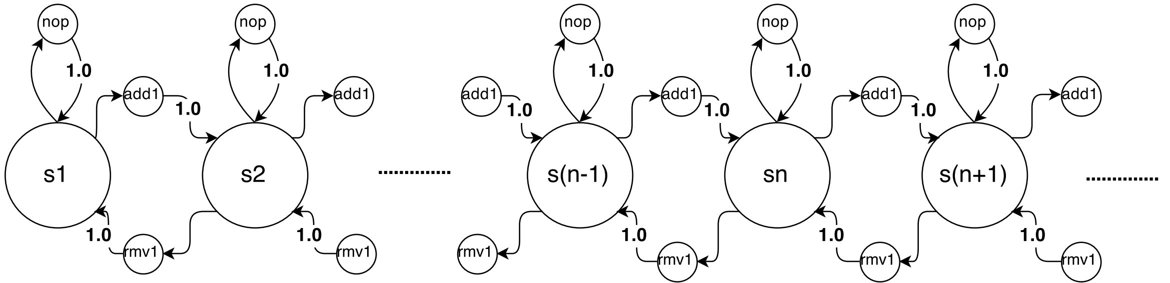

FIGURE 4.2: The simulation MDP model with 1-dimensional states. Each action leads to a deterministic transition and either adds or removes at most 1 VM.

4.4.4 Multi-Armed Bandit Setting

As previously stated, sometimes restrictions are applied over the extent of scaling allowed as a result of a single resize action. However, if it is most often the case that the decision-maker is able to resize as much as desired, then there is no point in trying to leverage a MDP state connectivity, and a MAB formulation of the problem is perhaps preferred, than a full RL formulation, and the occasional resizing restrictions can be modeled on top of the MAB setting, since they are not the rule but rather the exception.

As explained in Section 3.4, the main metric of interest in a MAB problem is the regret. In the following MAB simulations, the actual regret was measured, rather than the expected regret.

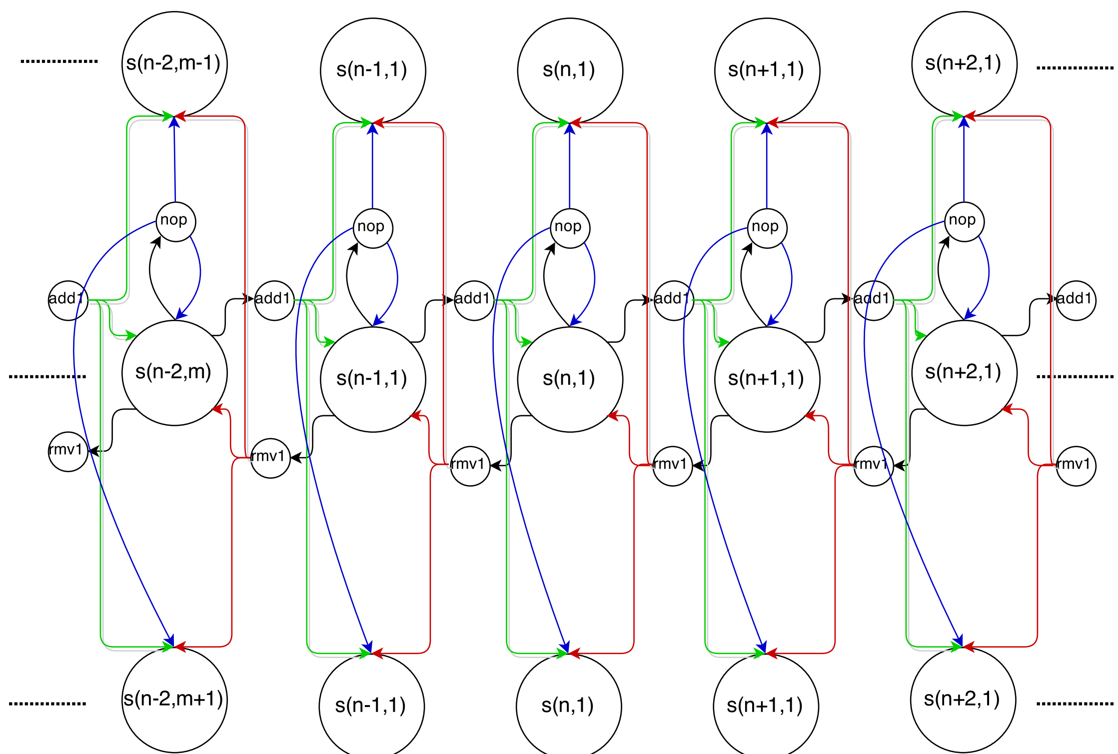

Figure 4.3: The extended simulation MDP model with 2-dimensional states. Transitions are no longer deterministic.

Chapter 5

Simulation Results

In this Section, simulation results[1] are presented that shed some light into aspects of dealing with cloud resource allocation problems by means of Reinforcement Learning (RL) and Multi-Armed Bandit (MAB) methods. The simulations were implemented in Matlab R2012b, based on the code in [70] for RL and on the code in [71] for MABs, with numerous extensions and modifications with respect to: the decision-making logic, the performance metrics monitored, and the results visualization.

5.1 RL Experiment Sets

Since the workload initially considered is both static and deterministic, so are the immediate rewards produced with each state transition of the RL algorithm (or similarly with each arm pull of the MAB algorithm). One might thus think that it is futile to experiment with a deterministic and static model, because all that it would take to perform optimally would be to sample every possible transition/arm just once. However, this argument misses the fact that, as explained in Section 4.4.3, despite the reward formula being specified by the user in advance, the immediate reward gained can not be calculated by the agent before taking an action, because the actual throughput is still unknown (albeit deterministically produced for simulation purposes). Thus, the agent can not arbitrarily assume that rewards are fixed. There are certainly specialized techniques and tricks for optimizing learning under fixed rewards, like setting a Q-value equal to the immediate reward observed the very first time the agent experiences it, or setting the learning rate to its maximal value, but it is still useful to draw conclusions about the more typical learning procedures. In real clouds, workloads and therefore rewards will

[1] The simulation models were presented in Section 4.4.2.

be dynamic, stochastic, and quite complex. However, simpler models with static rewards (whether deterministic or stochastic) provide a more informative simulation testbed for easily gaining insight into the different aspects of the learning algorithm's behavior, whereas, a more realistic dynamic model can be implemented as part of a "heavier" simulation that migth even take into account specific anomalies in a particular system's functioning.

5.1.1 RL Experiment Set 1: The effect of the time-horizon under a small static workload

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 1000, 10000, and 100000 # time steps (min)
- workload: 50 (K requests/second)
- rewards: $R(s, \sim, \sim) = \frac{1}{load}$ * [5; 10; 15; 20; 25; 20; 15; 10; 5; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0]

It is expected that the longer Q-Learning runs, the better its proposed policy will get, because Q-values will become more accurate with the passage of time.

It is obvious from figure 5.1 that 1000 time steps are not enough for Q-Learning to converge to a good policy, given that the learning agent starts with no prior knowledge. This is also precisely why, after 1000 time steps the agent has only explored the first 6 states, instead of uniformly exploring all 20 states, because with the Q-values being initialized to zero, whichever actions are taken first will automatically obtain bigger Q-values than the rest (which still have zero Q-values), and therefore will be preferred again and again over the unexplored actions. Simply put, the agent thinks that unexplored behaviors are "bad", so it tends to prefer the few same actions that it has already experienced,

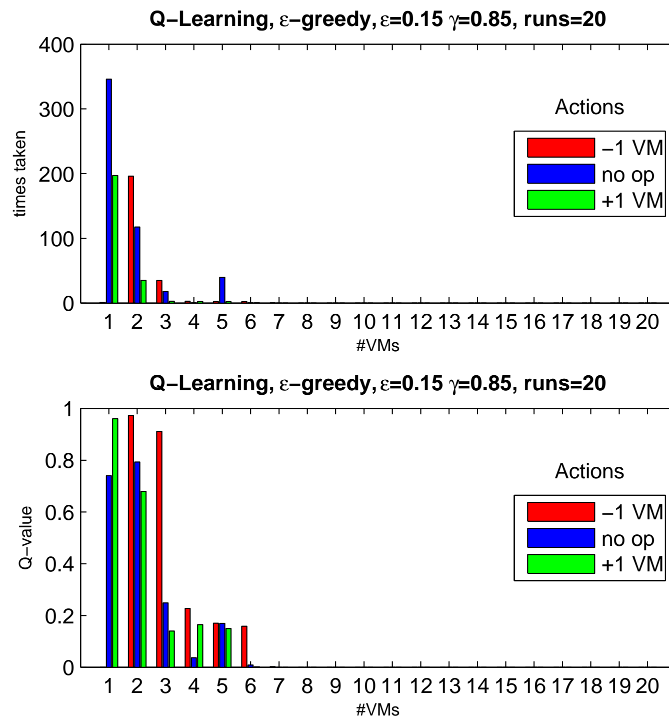

FIGURE 5.1: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

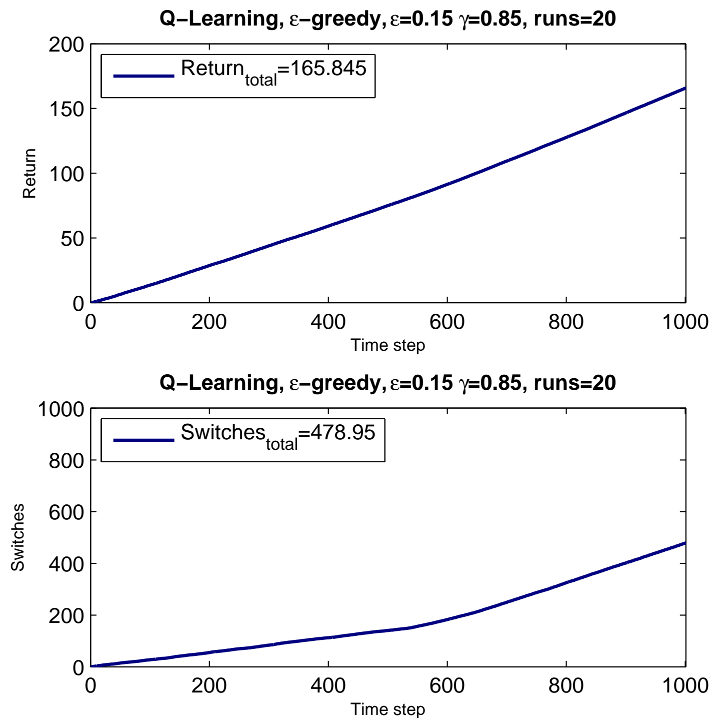

FIGURE 5.2: The mean total return (sum of rewards) and the mean total number of state switches, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

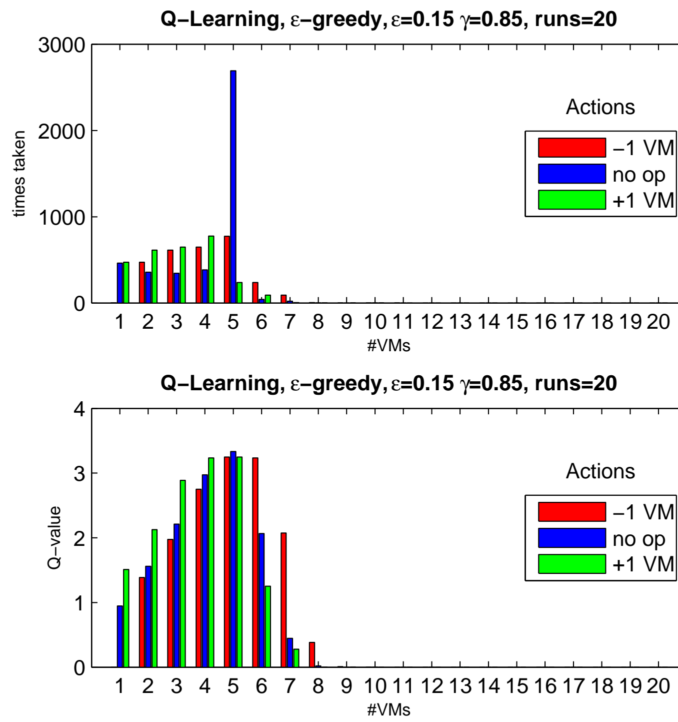

FIGURE 5.3: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

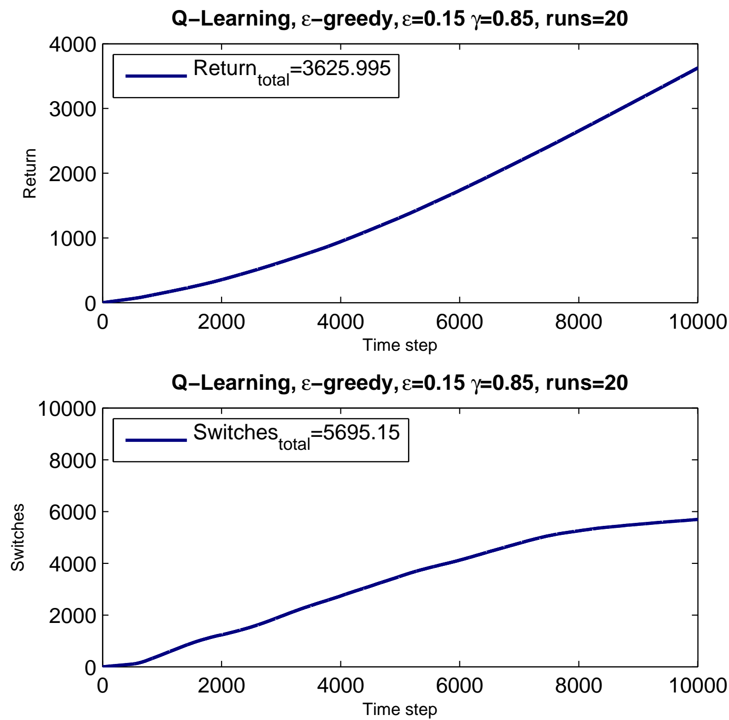

FIGURE 5.4: The mean total return (sum of rewards) and the mean total number of state switches, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

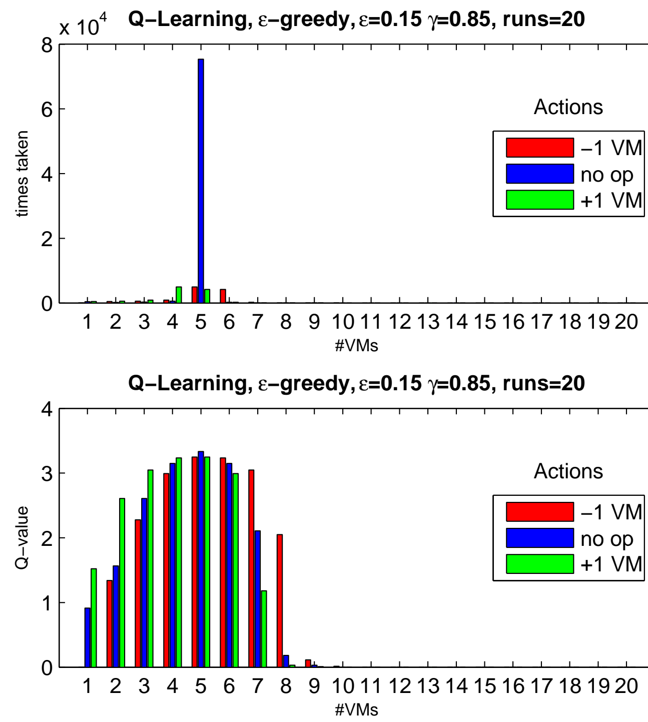

FIGURE 5.5: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

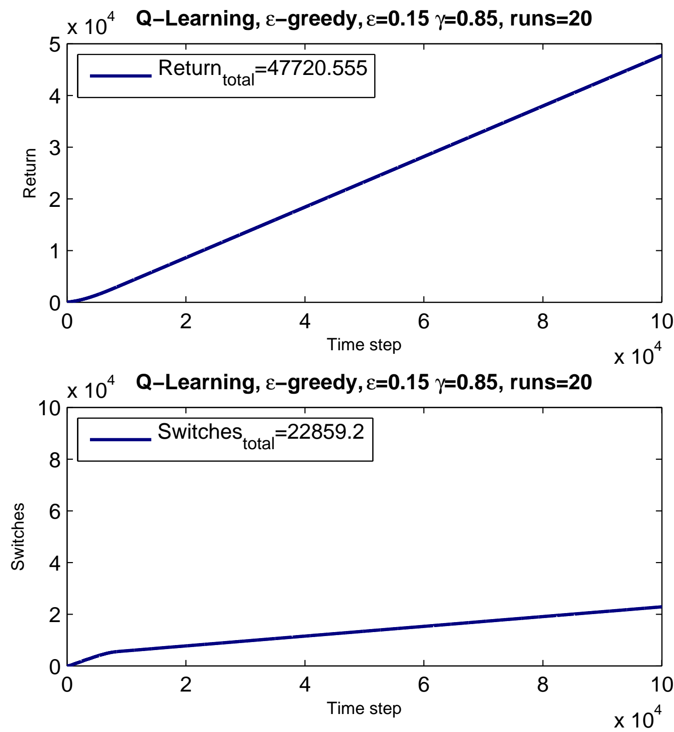

FIGURE 5.6: The mean total return (sum of rewards) and the mean total number of state switches, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 50$.

especially since the -1 VM and +1 VM states are the only potential transition targets, preventing him from randomly ending up in one of the upper states due to an exploration step. As can be seen in 5.3, at some point before the 10000 time step mark, the ϵ-greedy strategy will have explored enough new actions and converged to the optimal action of remaining in the state of 5 VMs. Ideally, the ϵ parameter should steadily decrease and fade to zero at that point, because taking random actions no longer serves the agent any good.

5.1.2 RL Experiment Set 2: The effect of the time-horizon under a big static workload

This time, a bigger fixed workload of 200K req/sec is selected, which as implied from the corresponding unnormalized reward vector [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 55; 60; 65; 70; 75; 80; 85; 90; 95; 100], is optimally dealt with the state of all 20 VMs (compared to 5 VMs being optimal in the previous experiment set). This scenario tests how efficiently rewards are propagated through the state-space, because the agent is initially set to have 1 VM and is only allowed to add 1 VM at a time, and therefore Q-Learning is expected to take longer to conclude that the optimal state is indeed s_{20}.

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 1000, 10000, 100000, and 500000 # time steps (min)
- workload: 200 (K requests/second)
- rewards: $R(s, \sim, \sim) = \frac{1}{load}$ * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 55; 60; 65; 70; 75; 80; 85; 90; 95; 100]

Indeed, from figures 5.7 and 5.9, it is verified that 1000 or even 10000 time steps are not enough for the agent to explore the state-space all the way through to state s_{20}.

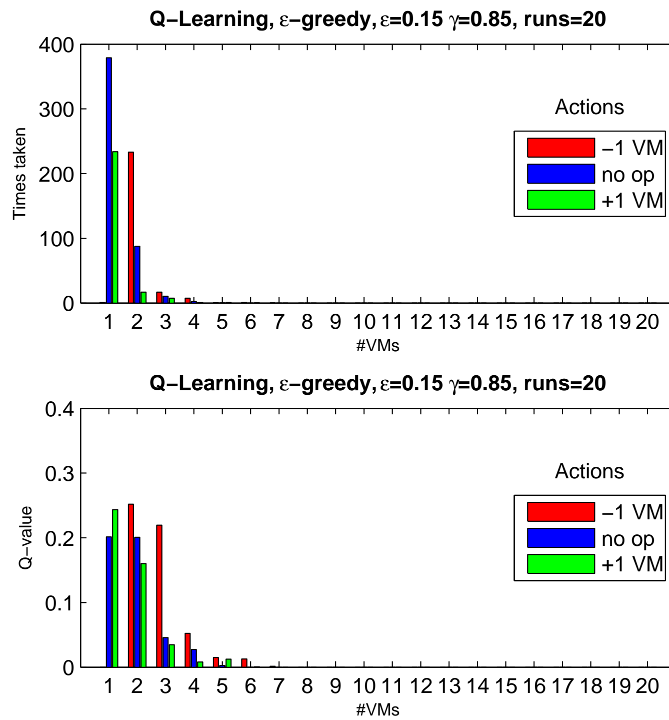

FIGURE 5.7: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

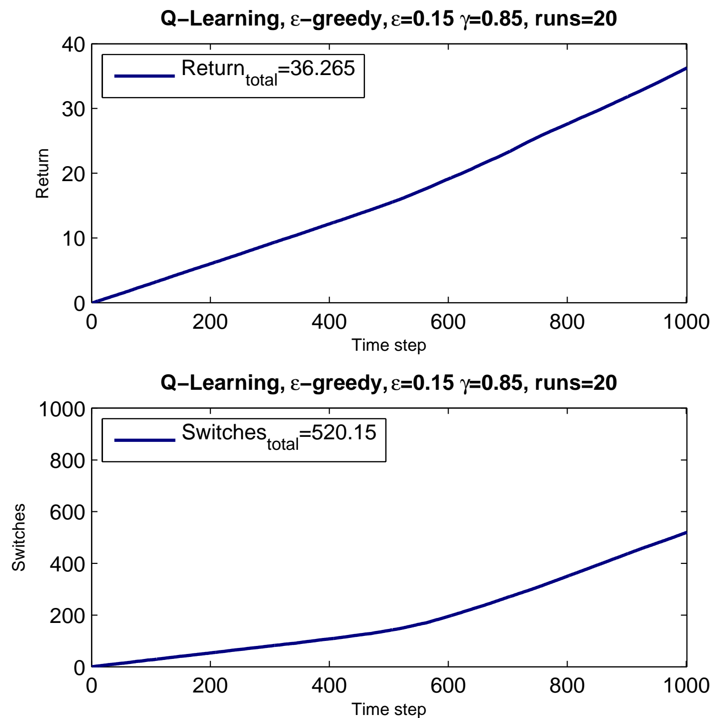

FIGURE 5.8: The mean total return (sum of rewards) and the mean total number of state switches, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

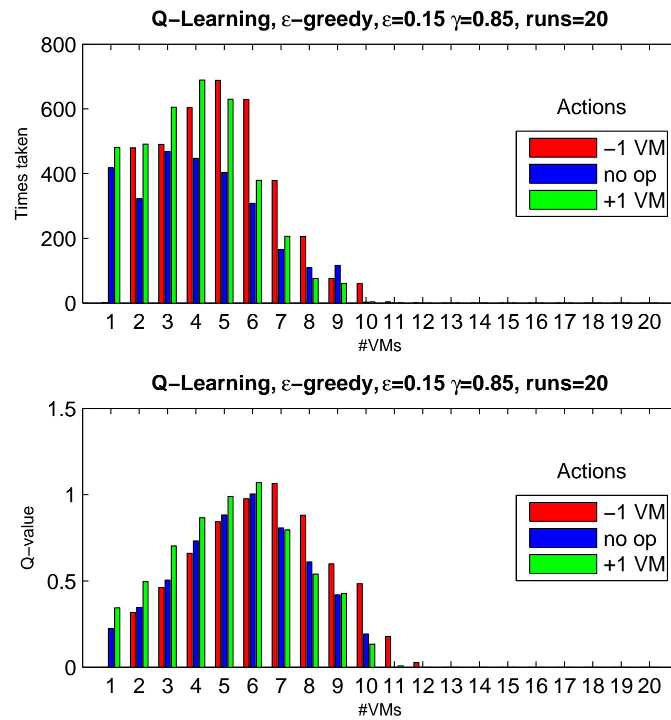

FIGURE 5.9: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

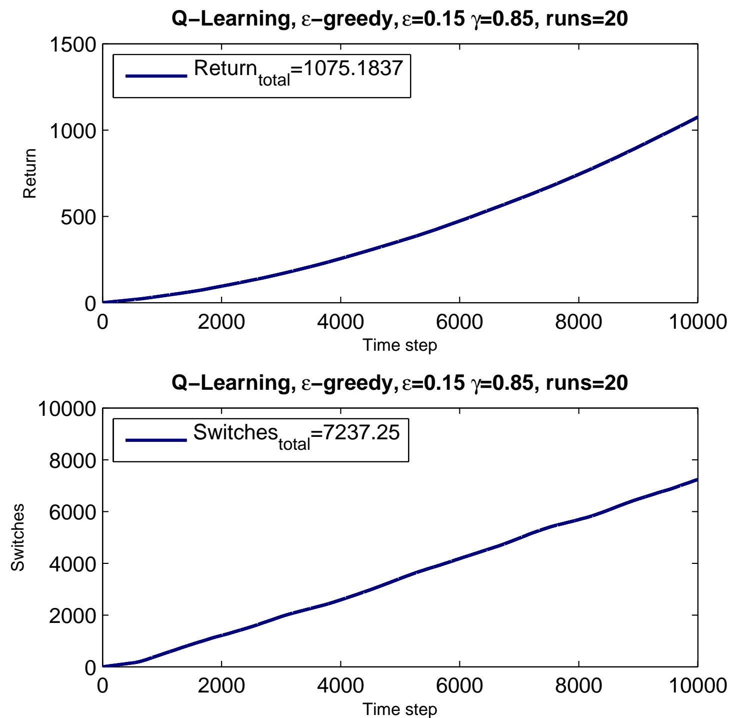

FIGURE 5.10: The mean total return (sum of rewards) and the mean total number of state switches, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

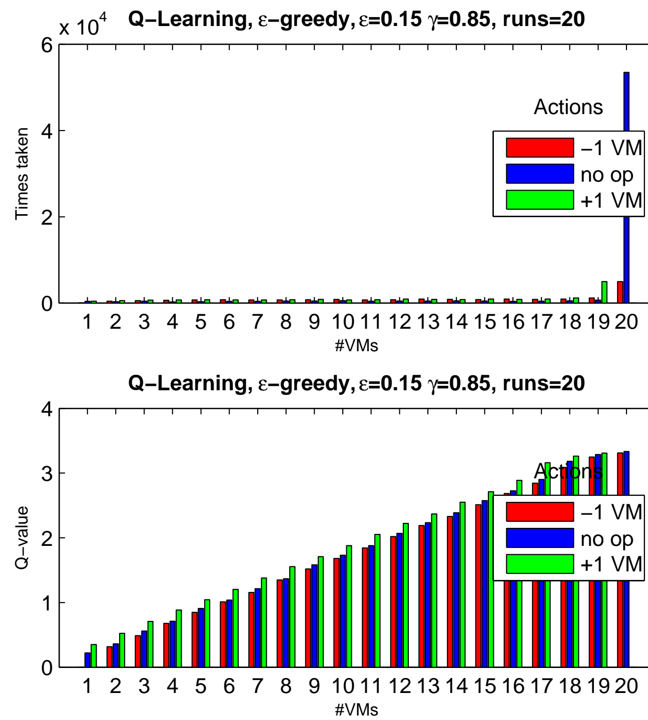

FIGURE 5.11: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

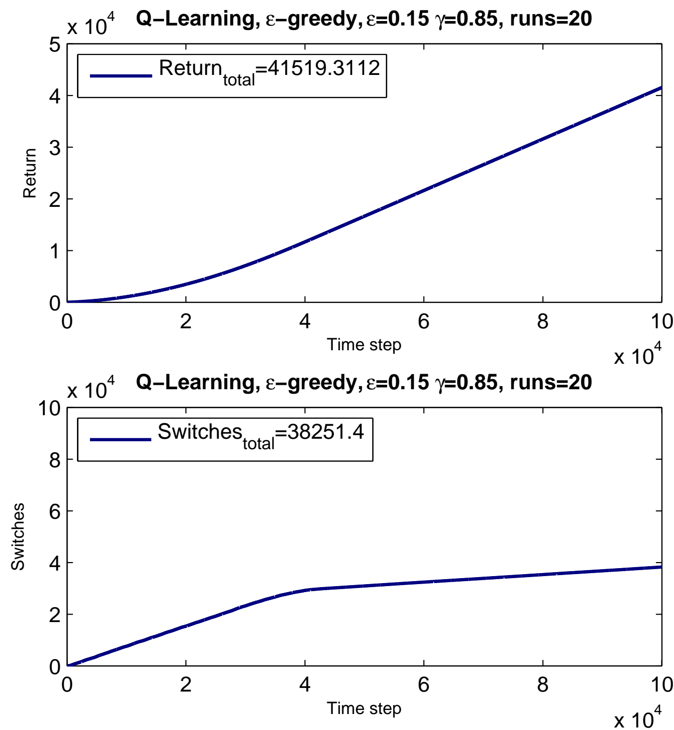

FIGURE 5.12: The mean total return (sum of rewards) and the mean total number of state switches, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

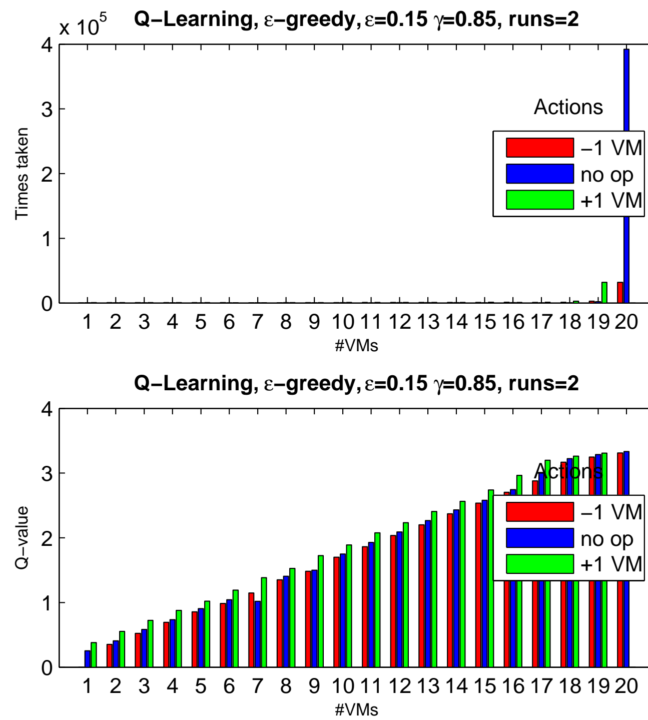

FIGURE 5.13: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 500000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

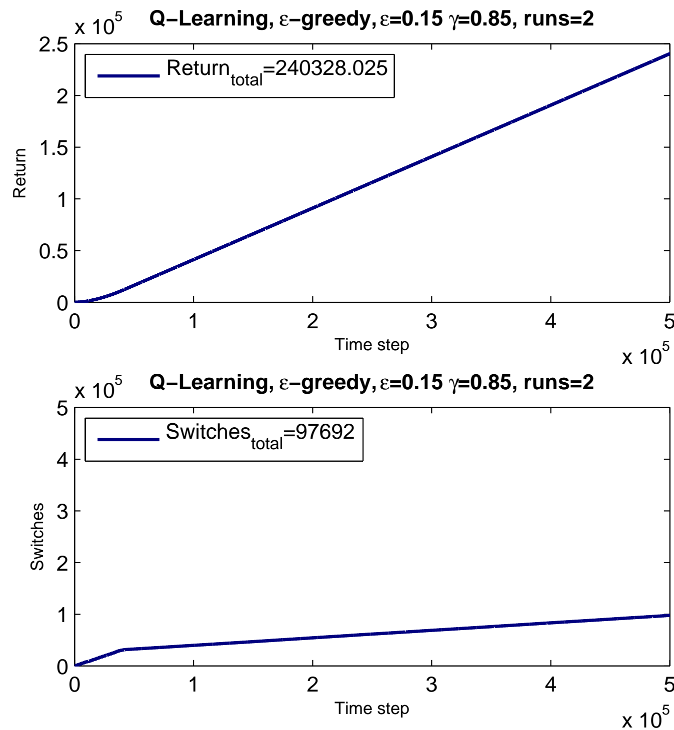

FIGURE 5.14: The mean total return (sum of rewards) and the mean total number of state switches, after 500000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$.

In the end however, the agent does converge to the optimal behavior as seen in 5.11, where more than half of the 100000 actions taken, indicate to the cluster to remain in s_{20}.

5.1.3 RL Experiment Set 3: The effect of optimistic initial conditions

The previous experiment set made it clear that by only allowing changes of ±1VM, there might be scenarios for which exploration is very time-consuming. In this set, instead of 0, the agent initially assumes that Q-values are equal to 5, a value greater than what they might reach in the time-horizons tested (but still not arbitrarily large). In this manner, exploration is implicitly favored through exploitation, because there is an exploration bonus integrated in the Q-values themselves, and therefore actually even propagated across states. This means for example that if state s_{20} has not been yet explored enough, then not only state s_{20}'s *no_op* action, but also state s_{19}'s *add_1* action will have a boosted Q-value because it leads to s_{20}'s unexplored *no_op* action, and similarly state s_{18}'s *add_1* action will also get a boosted Q-value because it leads to state s_{19}'s *add* action, and so on and so forth. This is all of course in addition to the random exploration steps taken due to the ϵ coefficient, so a faster overall convergence to the optimal behavior is to be expected.

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: optimistic (equal to 5)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 1000, 10000, and 100000 # time steps (min)
- workload: 200 (K requests/second)
- rewards: $R(s, \sim, \sim) = \frac{1}{load}$ * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 55; 60; 65; 70; 75; 80; 85; 90; 95; 100]

Interestingly enough, the optimistic approach substantially outperforms the pessimistic one, thanks to the exploration being accelerated in the early stages of operation. Comparing figure 5.15 to figure 5.7 and figure 5.16 to figure 5.8, it seems that after only 1000

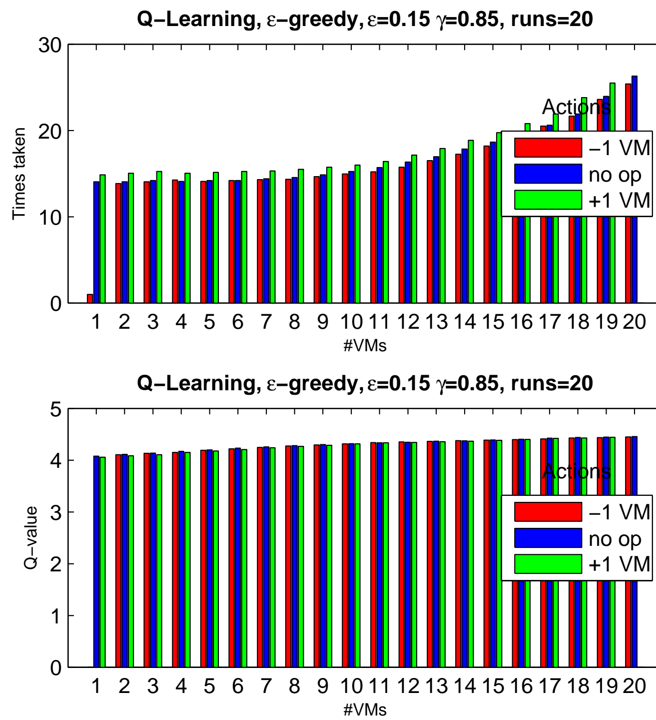

FIGURE 5.15: The mean number of state-action explorations and Q-values, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

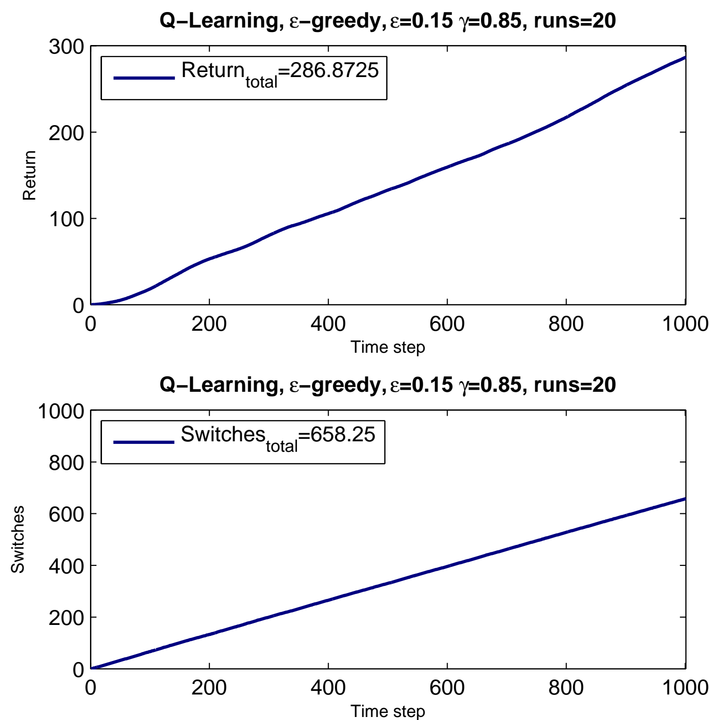

FIGURE 5.16: The mean total return and total number of state switches, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

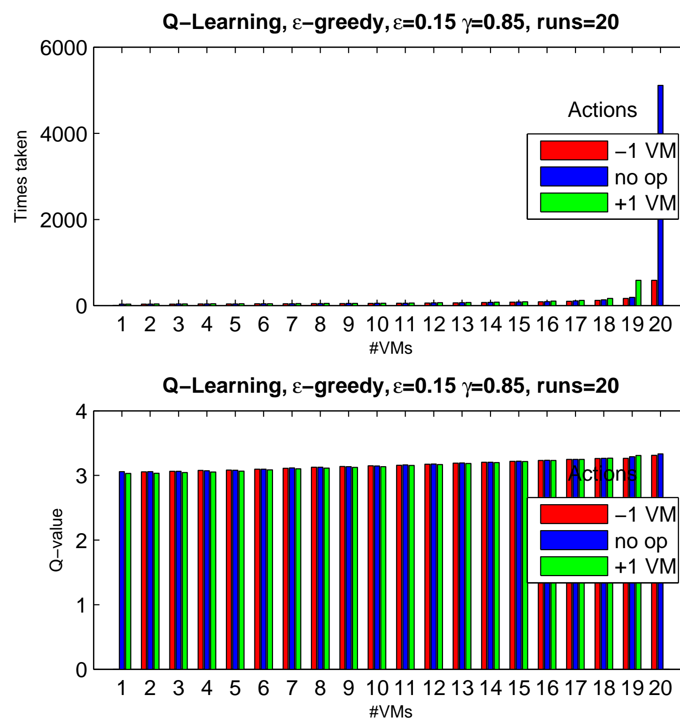

FIGURE 5.17: The mean number of state-action explorations and Q-values, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

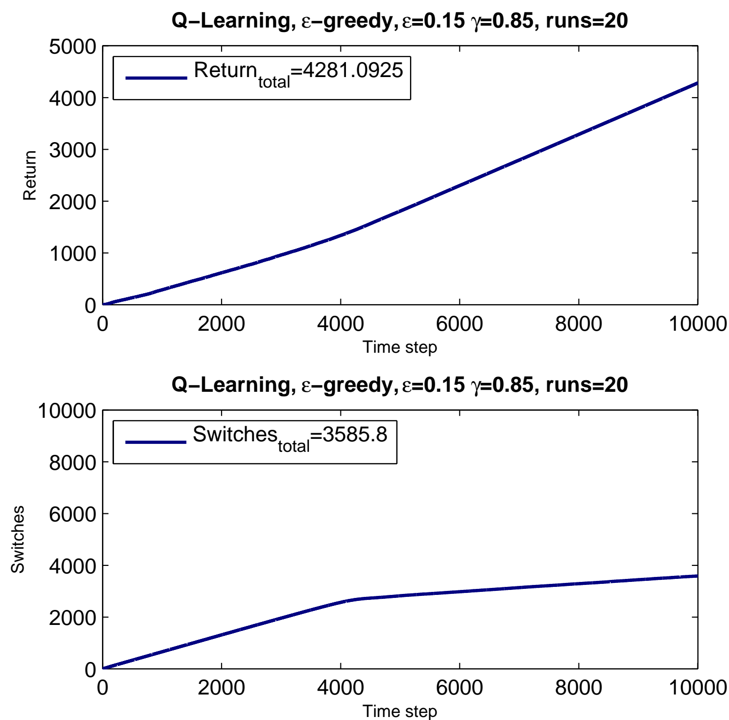

FIGURE 5.18: The mean total return and total number of state switches, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

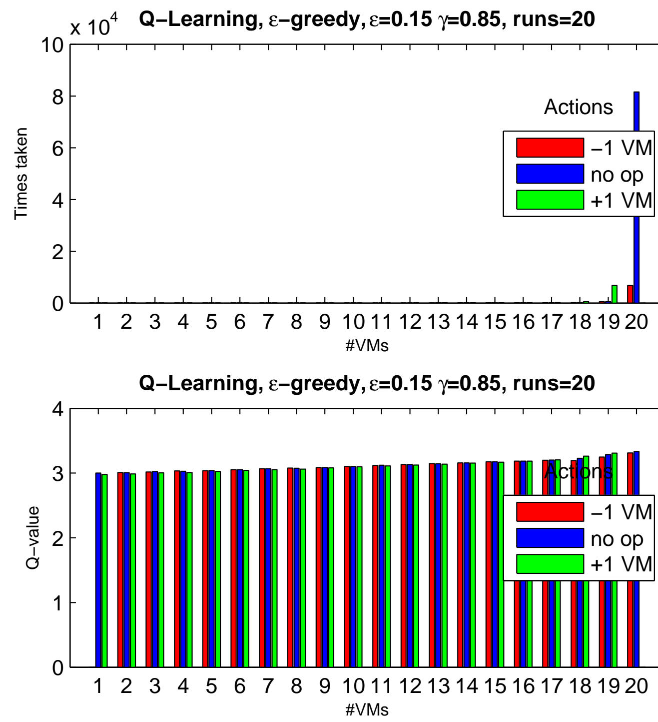

FIGURE 5.19: The mean number of state-action explorations and Q-values, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

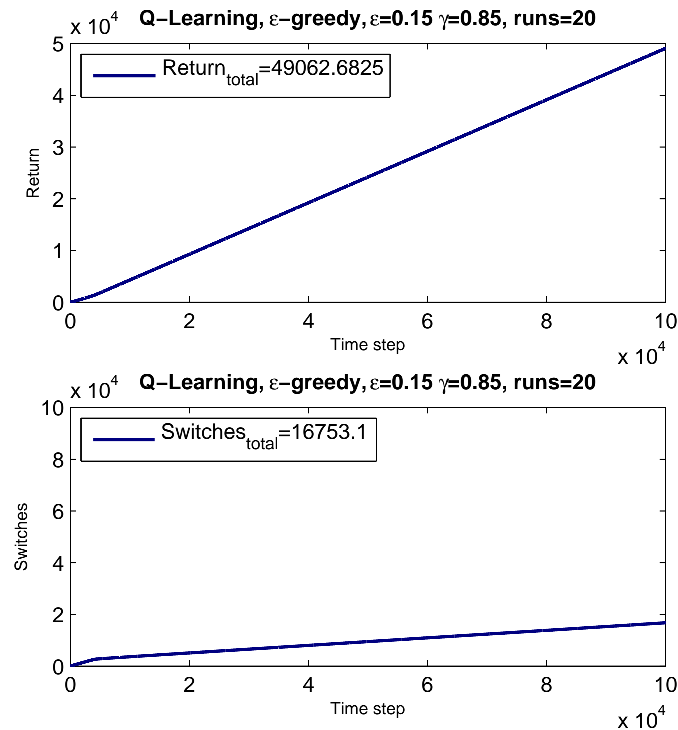

FIGURE 5.20: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = L = 200$, and optimistic initial Q-values.

time steps, the optimistic approach has already identified the optimal number of VMs (being 20), and collected a total return of 286 as opposed to only 36 in the previous experiment set. In the 10000 time step simulations of figure 5.17, the optimistic approach has more clearly converged to the optimal behavior, whereas the corresponding pessimistic approach, at the exact same temporal snapshot, had yet to even once try asking for 20 VMs. Finally, in figure 5.19, after 100000 time steps, the optimistic approach is still better, choosing to remain with 20 VMs in more or less 80% of the decision epochs and gathering a total return of 49062, as opposed to about 50% of the decision epochs and a total return of 41519 respectively for the pessimistic approach, which is of course asymptotically catching up with the optimistic one (but still falling behind). Finally, it should be noted that the initial Q-values should not be set exceptionally high either, because that would have the opposite effect, since (all other parameters being equal) it would take too long for the Q-values to drop and converge to their true levels.

5.1.4 RL Experiment Set 4: The effect of the learning rate

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 1$, $\gamma = 0.85$
- iterations: 1000 # time steps (min)
- workload: 50 and 200 (K requests/second)
- rewards: $R(s, \sim, \sim) = \frac{1}{load} * [5; 10; 15; 20; 25; 20; 15; 10; 5; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0]$ and $R(s, \sim, \sim) = \frac{1}{load} * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 55; 60; 65; 70; 75; 80; 85; 90; 95; 100]$

For static rewards, given that any one of the actions results in the exact same reward, every single time it is taken, one might expect that the learning agent will be better off, if it completely ignores the history of observations in favor of the last observation alone. With this in mind, in this experiment set, the learning rate is maximally set equal to 1.

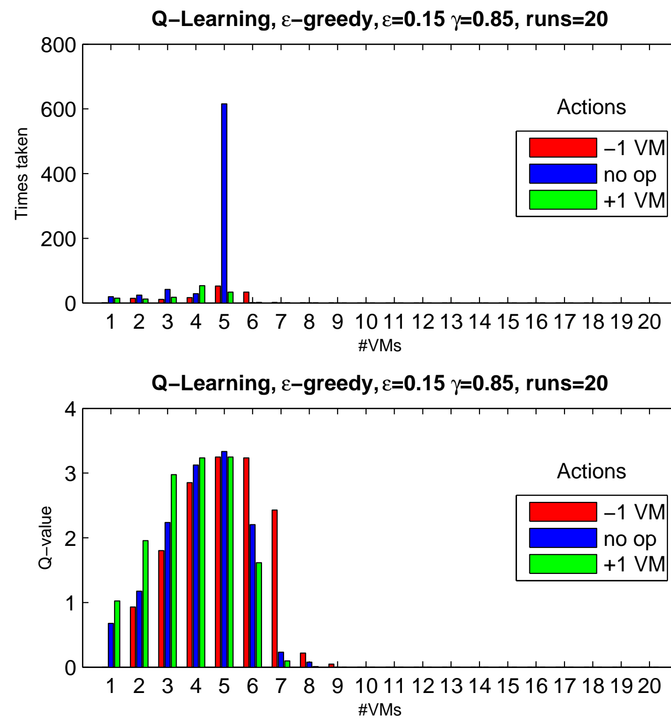

FIGURE 5.21: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 1000 decision steps, for $\alpha = 1$ and $L(t) = L = 50$.

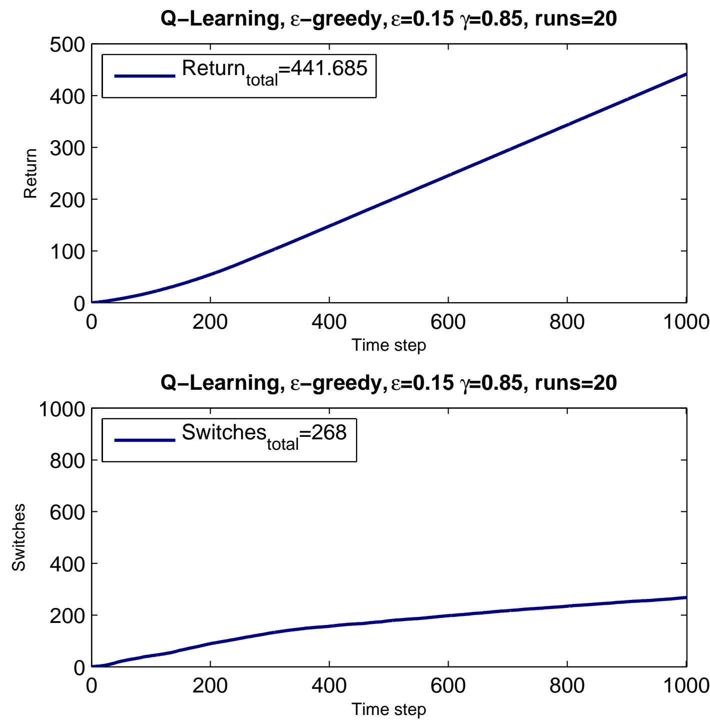

FIGURE 5.22: The mean total return (sum of rewards) and the mean total number of state switches, after 1000 decision steps, for $\alpha = 1$ and $L(t) = L = 50$.

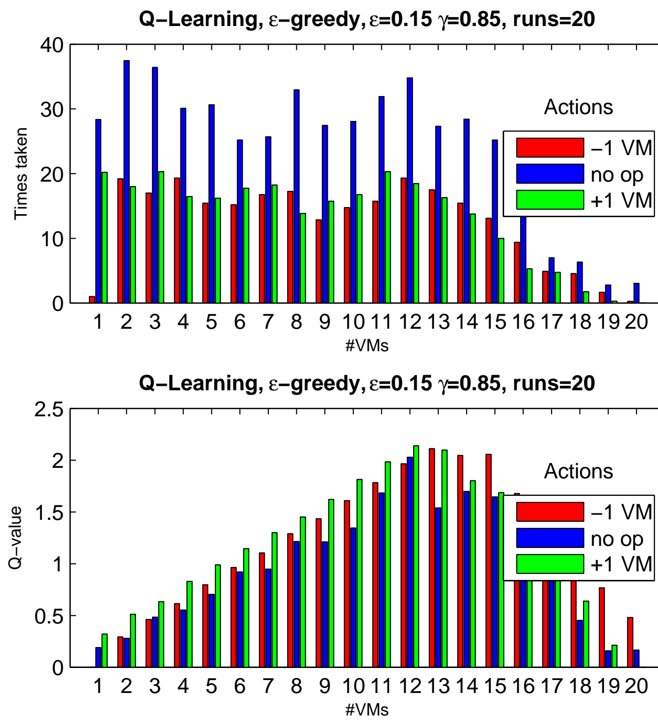

FIGURE 5.23: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 1000 decision steps, for $\alpha = 1$ and $L(t) = L = 200$.

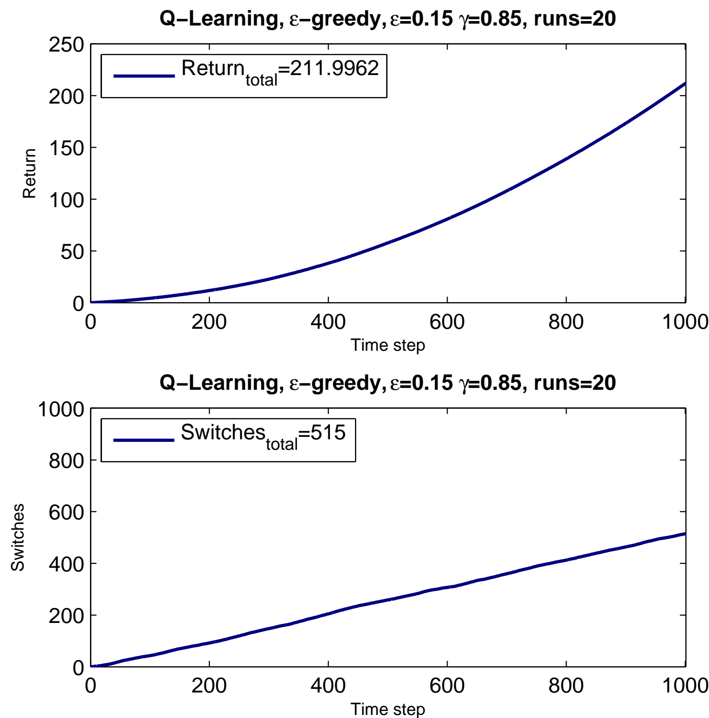

FIGURE 5.24: The mean total return (sum of rewards) and the mean total number of state switches, after 1000 decision steps, for $\alpha = 1$ and $L(t) = L = 200$.

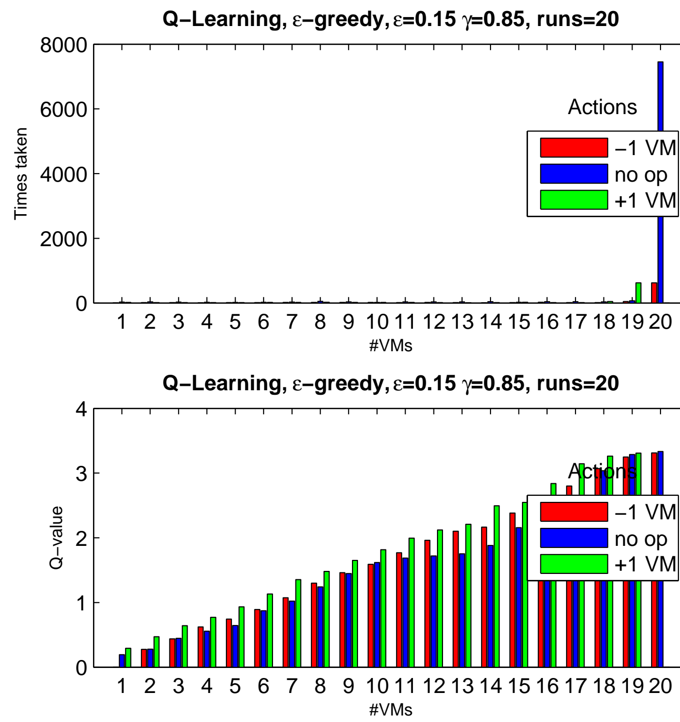

FIGURE 5.25: The mean number of times each action was chosen and the mean of the corresponding Q-values, after 10000 decision steps, for $\alpha = 1$ and $L(t) = L = 200$.

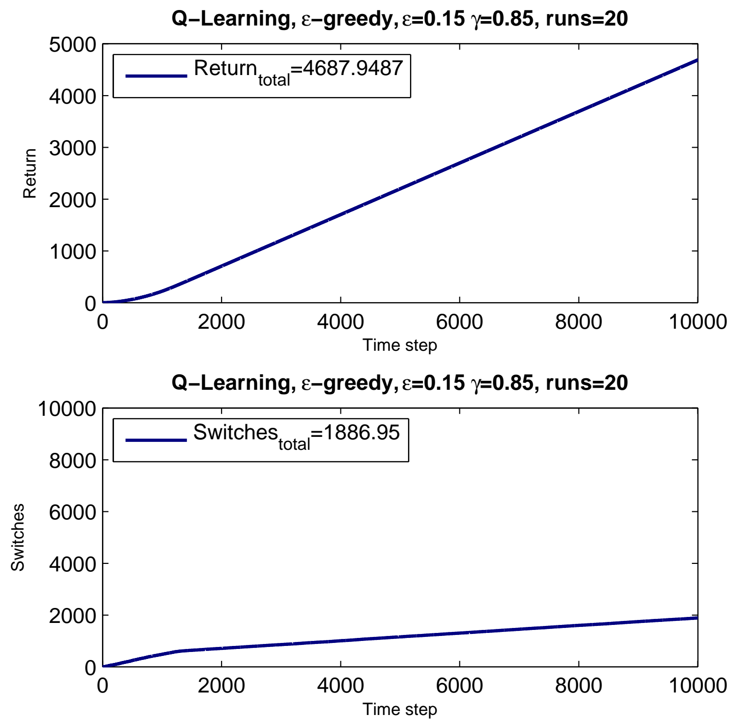

FIGURE 5.26: The mean total return (sum of rewards) and the mean total number of state switches, after 10000 decision steps, for $\alpha = 1$ and $L(t) = L = 200$.

Indeed, for both 50K and 200K workloads, it is best to simply "crank up" the learning rate for finding the optimal action values as fast as possible. Of course, this is a rather unrealistic scenario, because even if workloads were fixed, the decision-maker would have to know that in advance, or he would never set the learning rate equal to 1. In addition, in a real-world system there might exist random noise or irregularities in general, making extreme updates that depend too much on the very last experience quite risky. Aside from that, it can be observed by comparing figure 5.21 to figure 5.1, and figure 5.25 to figure 5.9, that even with a pessimistic initial ignorance, the agent can indeed converge to the optimal behavior in much fewer time steps, simply thanks to a greater learning rate value that leads to bold Q-value updates. In fact, the 10000 time step figures 5.26 and 5.18 indicate that, increasing the learning rate achieves an even greater amelioration in terms of total return with fewer state switches, than using optimistic initial conditions, but of course the two can always be used together.

5.1.5 RL Experiment Set 5: The effect of the time-horizon under a dynamic workload

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 1000, 10000, 100000, and 500000 # time steps (min)
- workload: $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$ (K requests/second)
- rewards: $R(s_n, \sim, \sim, t) = \frac{1}{load} * \max(\min(10 * n, L(t)) - vm_cost * n, 0)$, $\forall n \in [1, N-1]$

In this set of experiments, the workload is no longer static but changes in time. In particular, it is modeled after a simple sinusoidal signal. This serves for testing if the learning agent is able to adapt to a varying number of client requests, by scaling out to more VMs during the highs, and scaling in to less VMs during the lows, of demand.

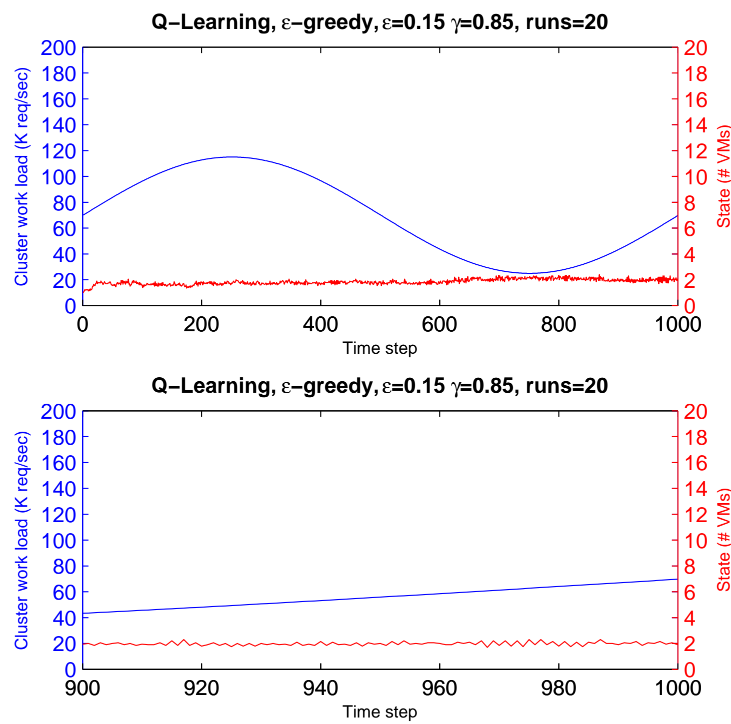

FIGURE 5.27: The current workload and the mean current number of VMs, during 1000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

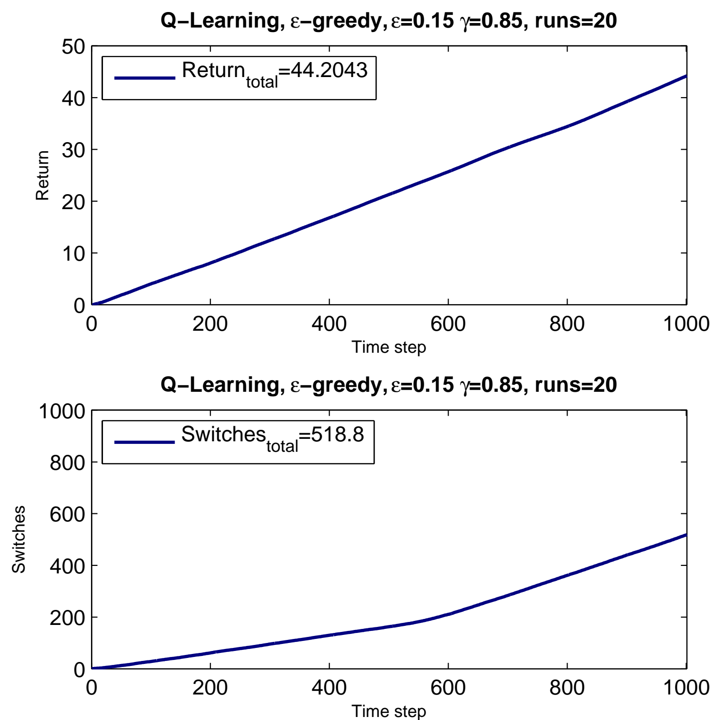

FIGURE 5.28: The mean total return and total number of state switches, after 1000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

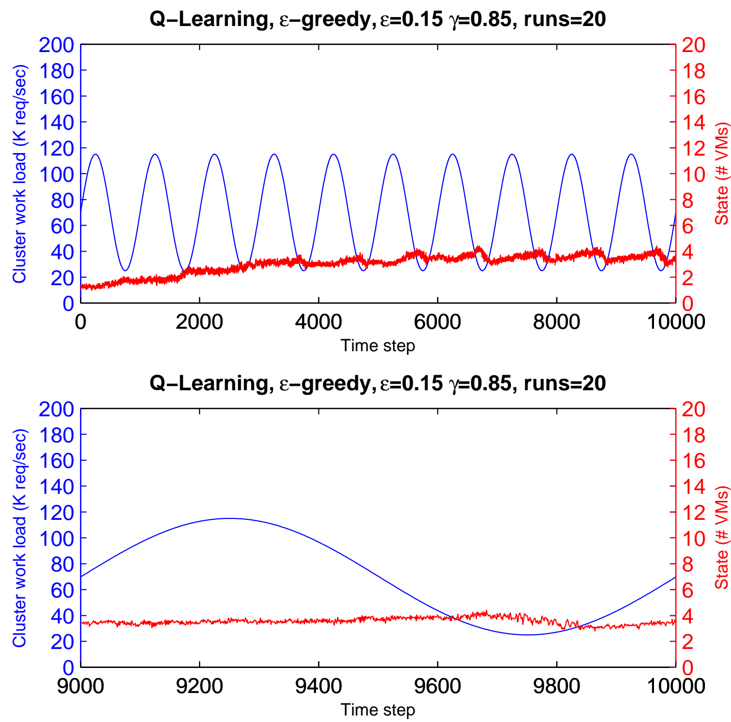

FIGURE 5.29: The current workload and the mean current number of VMs, during 10000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

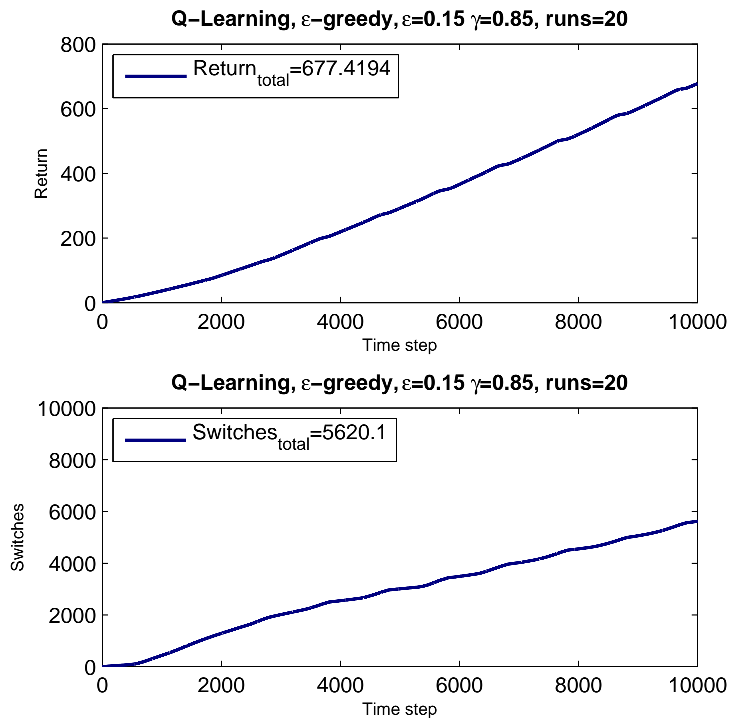

FIGURE 5.30: The mean total return and total number of state switches, after 10000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

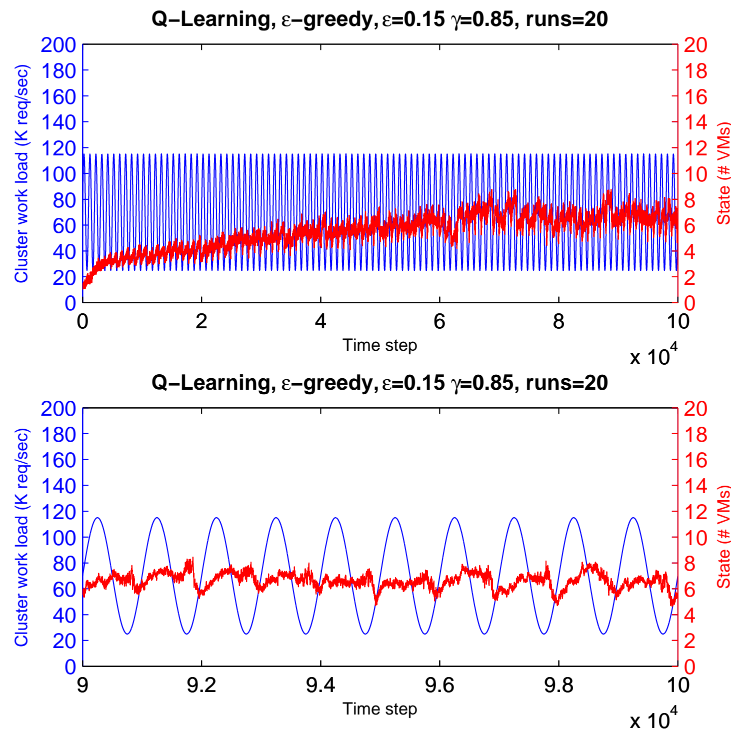

FIGURE 5.31: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

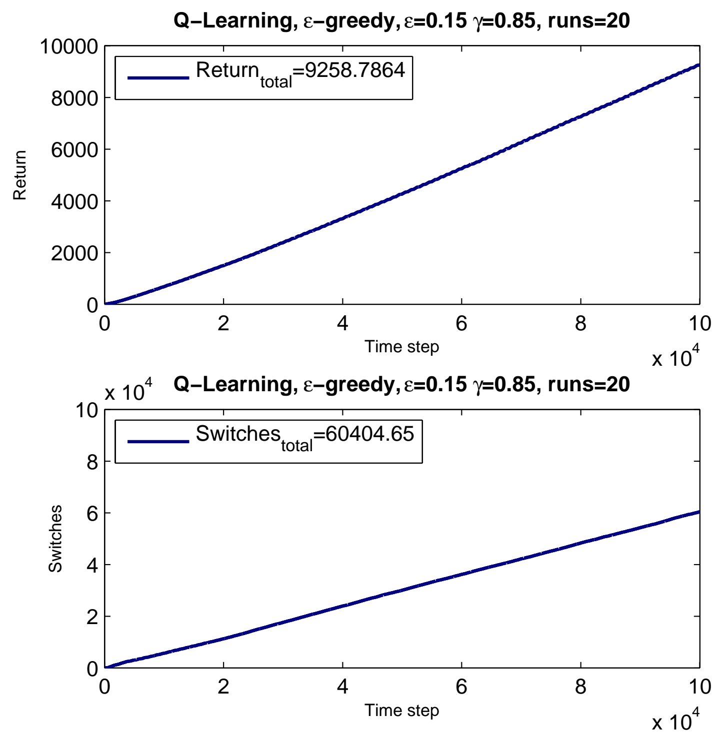

FIGURE 5.32: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

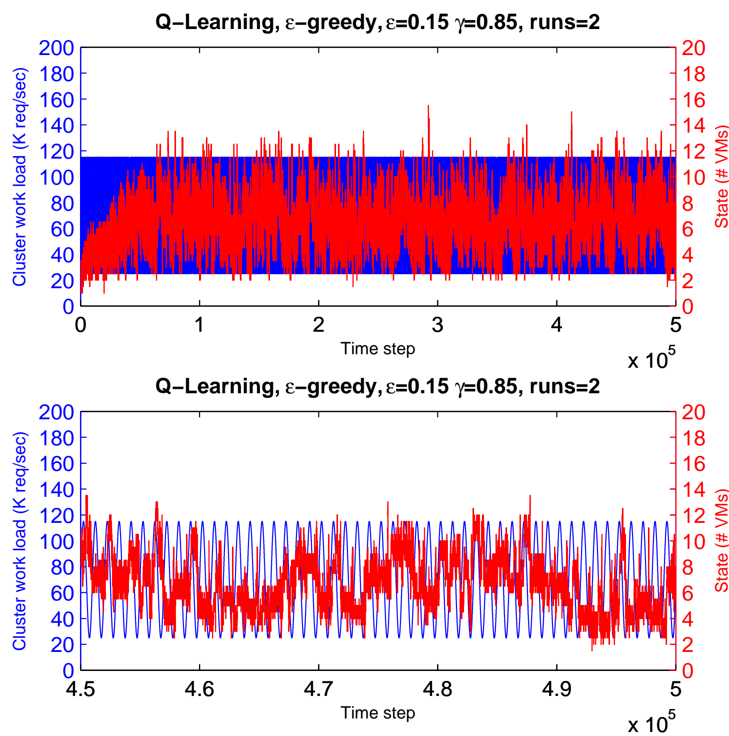

FIGURE 5.33: The current workload and the mean current number of VMs, during 500000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

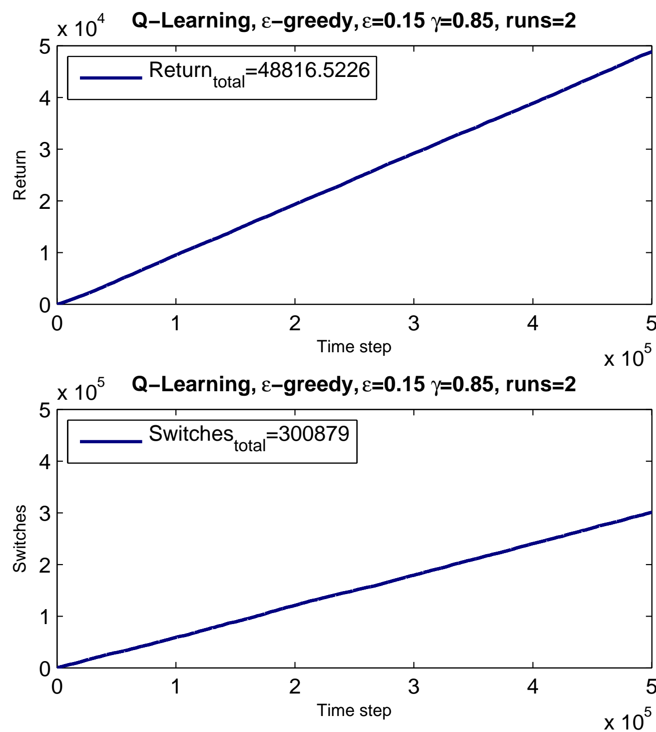

FIGURE 5.34: The mean total return and total number of state switches, after 500000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

The resulting graphs clearly show that while Q-Learning tries to adapt to the ever changing workload, it does so ineffectively. Looking at figures 5.31 and 5.33, it is evident that the problem is not related to the length of the time-horizon. The agent is not reacting fast enough to meet the fluctuations of demand, which is to be expected: the state-space is only 1-dimensional and therefore the agent only keeps 20*3[2] Q-values, comprising the totality of its knowledge. As a result, the constantly changing workload makes the same state-action pairs that are considered to be very rewarding, quickly turn into bad choices, and their corresponding Q-values must rapidly drop from being very high to being very low (and then again rise to their previous levels, due to the workload periodicity). Clearly, the Q-Learning algorithm needs to generalize less, and treat vastly different workloads as separate situations faced by the agent. TIRAMOLA achieves that by selectively choosing a portion of past experiences to be used in its calculations (as detailed in Section 4.3), but the most straightforward way is to simply add a second dimension to the MDP state-space. First however, a simpler solution shall be tested.

5.1.6 RL Experiment Set 6: The effect of the learning rate under a dynamic workload

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1,0.4,0.8$, and 1, $\gamma = 0.85$
- iterations: 100000 # time steps (min)
- workload: $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$ (K requests/second)
- rewards: $R(s_n, \sim, \sim, t) = \frac{1}{load} * \max(\min(10 * n, L(t)) - vm_cost * n, 0)$, $\forall n \in [1, N-1]$

By raising the learning rate, it is hoped that the learning agent will aim for greater Q-value updates, and thus better adapt to the workload's periodic changes, instead of lagging behind them as in the previous experiment set.

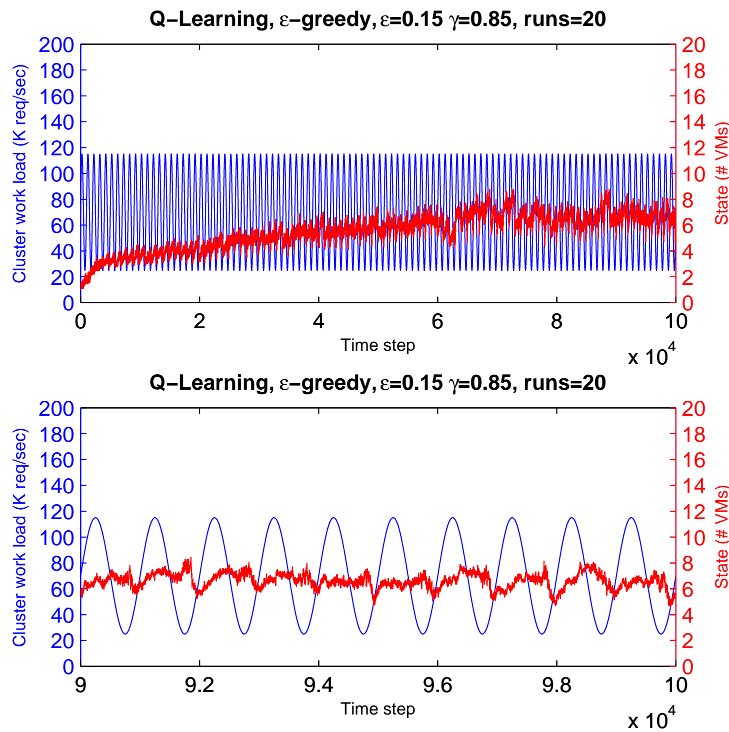

FIGURE 5.35: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

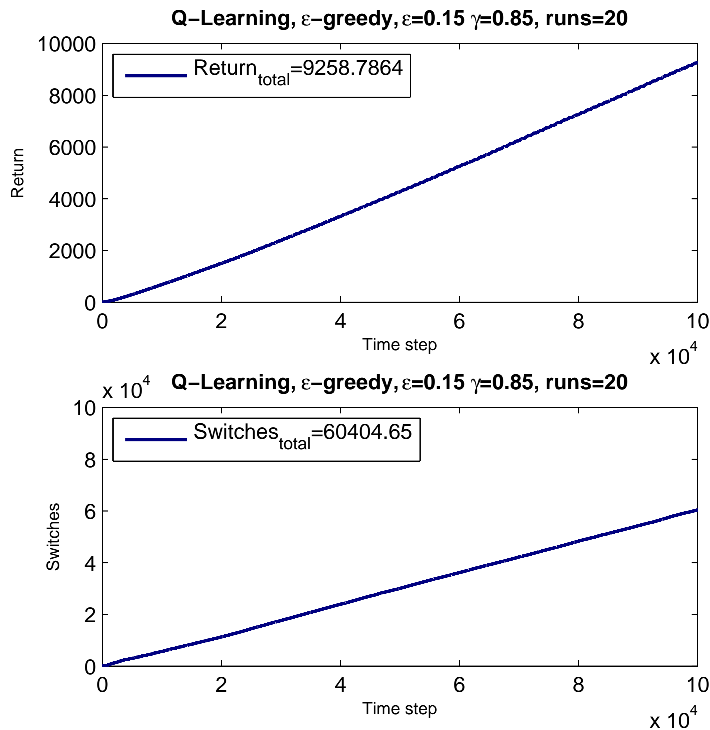

FIGURE 5.36: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

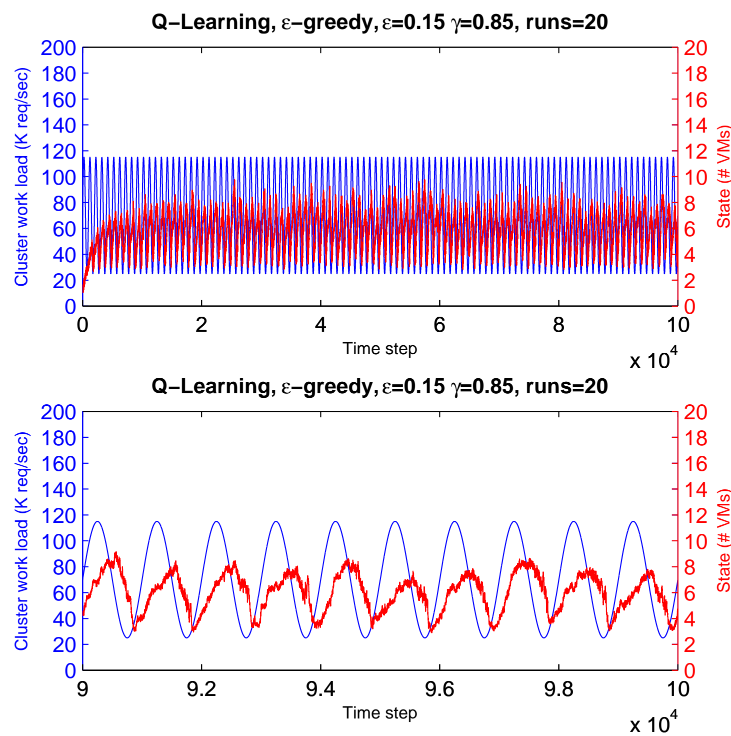

FIGURE 5.37: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.4$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

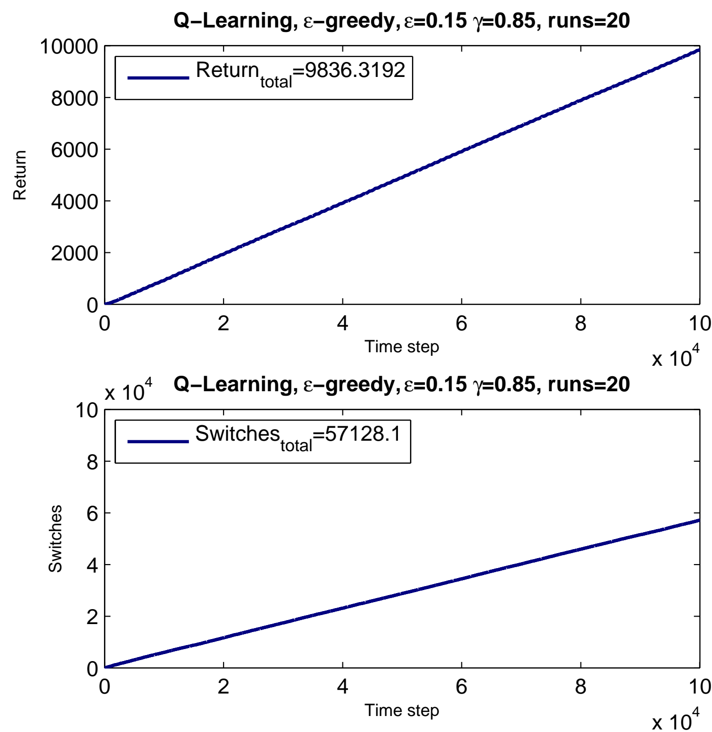

FIGURE 5.38: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.4$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

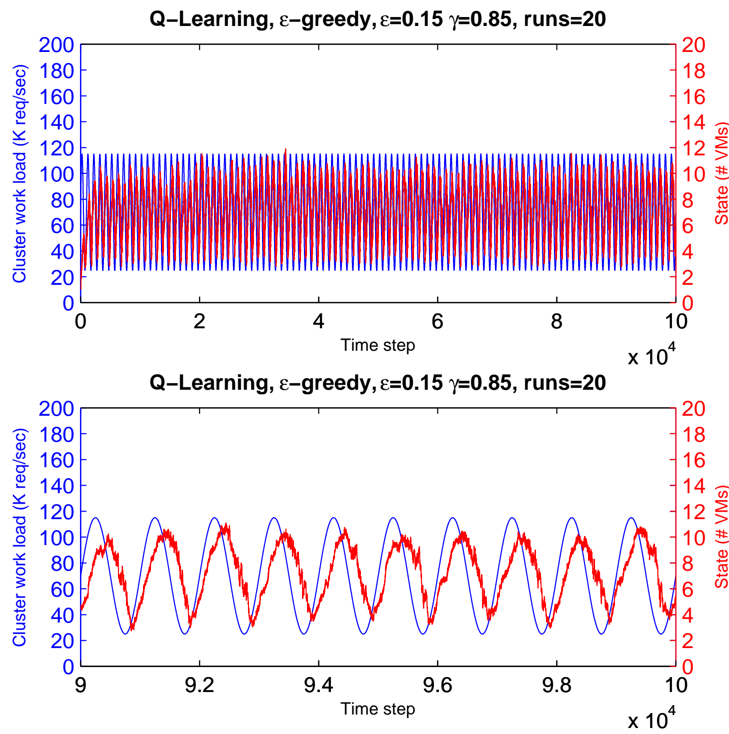

FIGURE 5.39: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.8$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

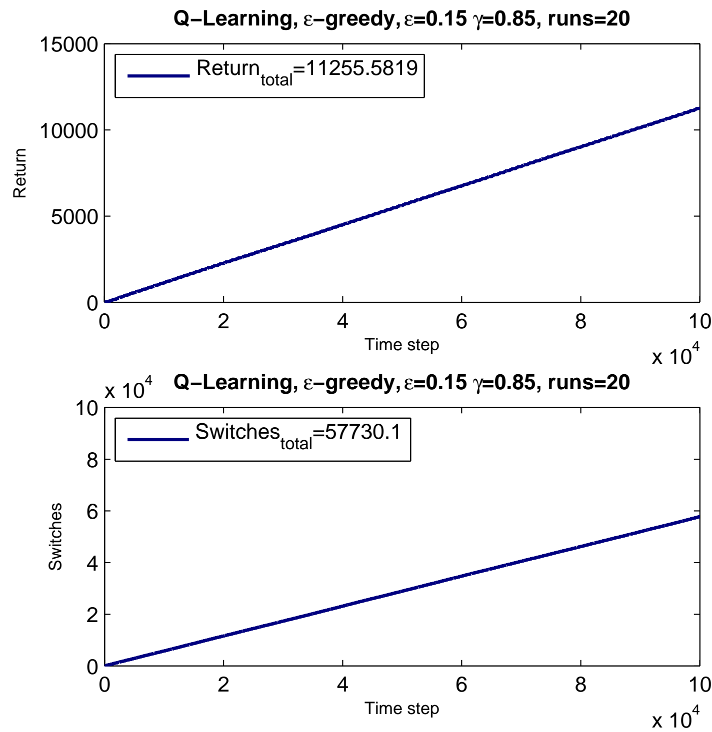

FIGURE 5.40: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.8$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

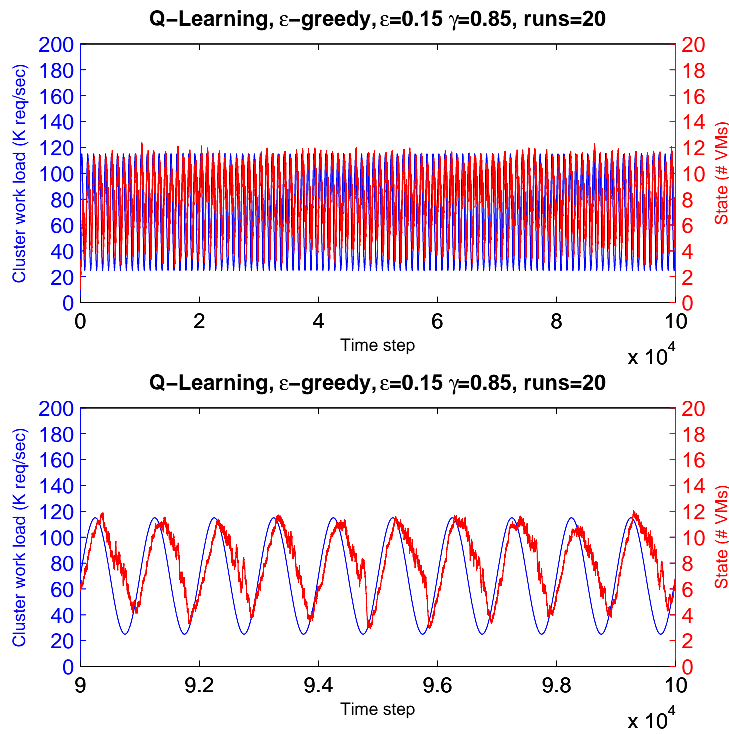

Figure 5.41: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

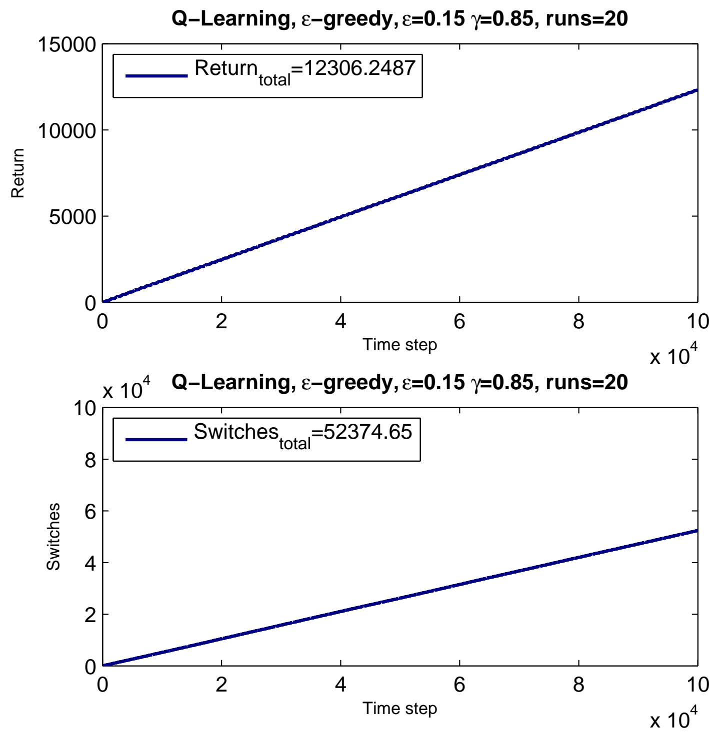

Figure 5.42: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$.

Indeed, judging from the graph in figure 5.39, with a learning rate of 0.8 the agent is able to quickly adapt to the sinusoidal workload, and in figure 5.41 where the learning rate is 1, this is even more the case. However, there are still two major problems: the agent's transitions still do not quite catch up with the workload signal, at least not enough to prevent wasteful provisioning of VMs when they are not really needed, or failing to serve a slight portion of the workload when more VMs would in fact be needed. It seems that simply raising the learning rate is not a trustworthy way for dealing with this problem.

5.1.7 RL Experiment Set 7: The effect of the workload amplitude and frequency

Settings:

- algorithm: Q-Learning
- state-space: 1-dimensional (number of VMs)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 1$, $\gamma = 0.85$
- iterations: 100000 # time steps (min)
- workload: $L(t) = 70 + 65 * \sin(\frac{2\pi t}{1000})$ (K requests/second) $L(t) = 70 + 45 * \sin(\frac{2\pi t}{300})$ (K requests/second)
- rewards: $R(s_n, \sim, \sim, t) = \frac{1}{load} * \max(\min(10 * n, L(t)) - vm_cost * n, 0),\ \forall n \in [1, N-1]$

Despite lacking extreme precision, the previous workload was dealt with satisfactorily by simply increasing the learning rate. In this set, the workload changes more aggressively, in order to test if that is enough to again provide respectable performance.

From the simulation results, it is apparent that even with the learning rate set to 1, Q-Learning has trouble keeping up with a very dynamic workload, especially beyond a certain amplitude (figure 5.43) or a certain frequency (figure 5.45). This can neither be dealt with by increasing the decision-making frequency, because it is already very high,

[2] -2 for the two unavailable actions.

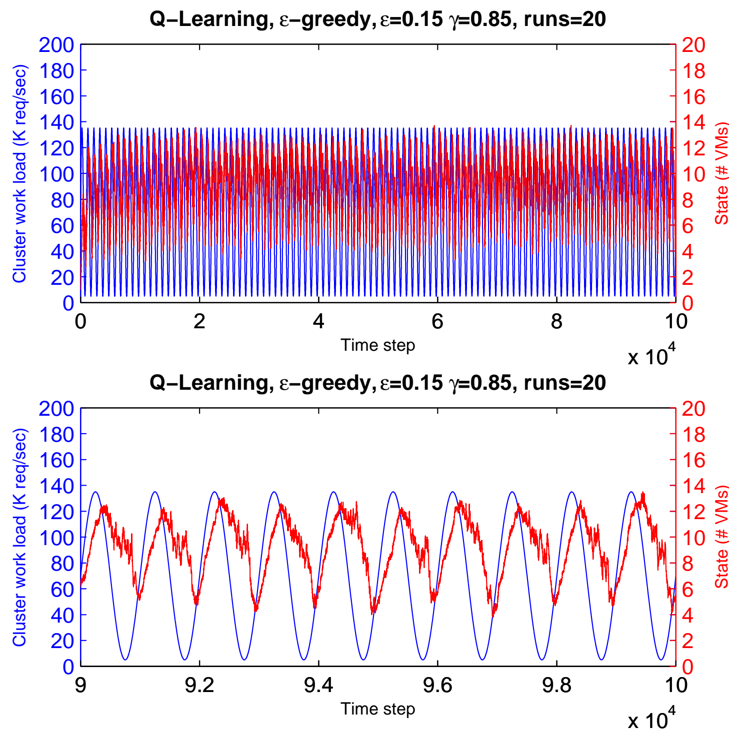

FIGURE 5.43: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 65 * \sin(\frac{2\pi t}{1000})$.

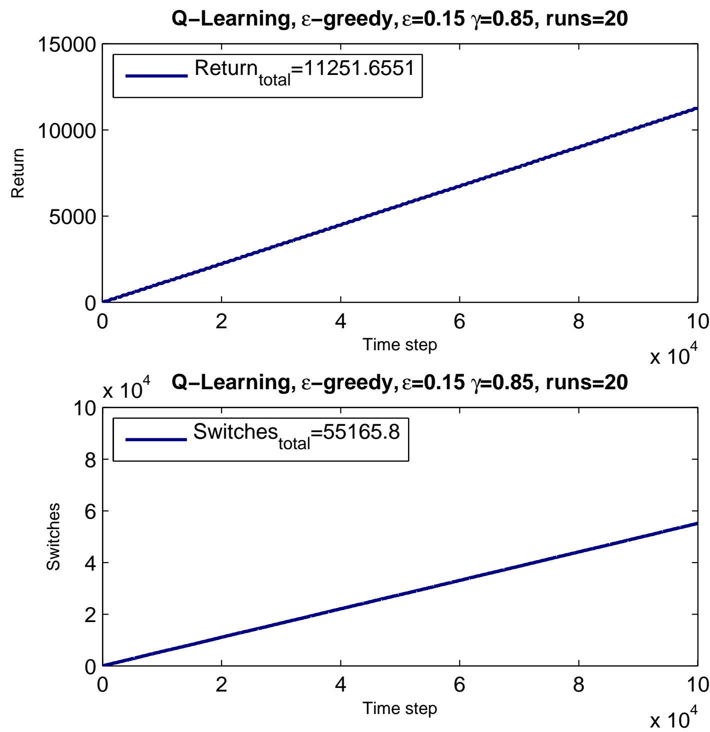

FIGURE 5.44: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 65 * \sin(\frac{2\pi t}{1000})$.

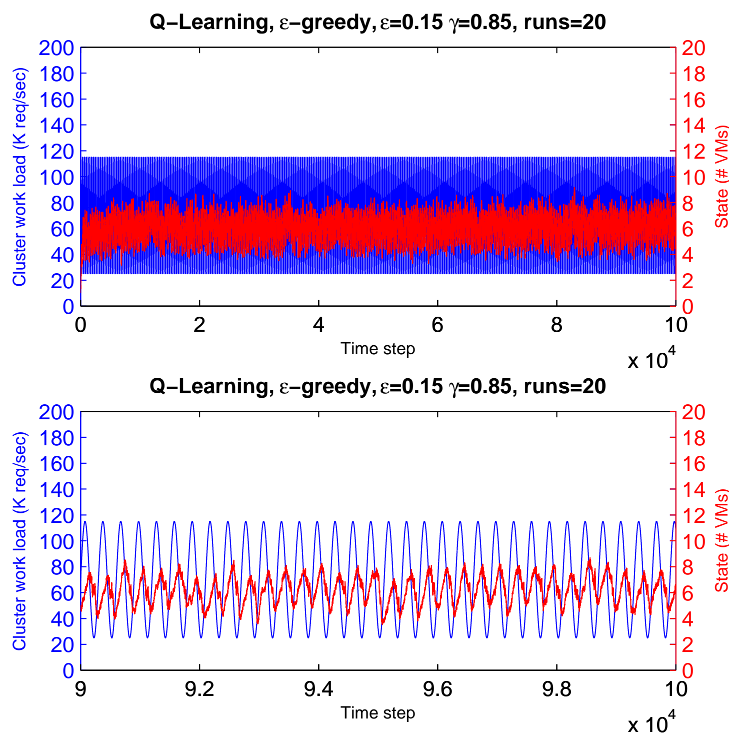

FIGURE 5.45: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{300})$.

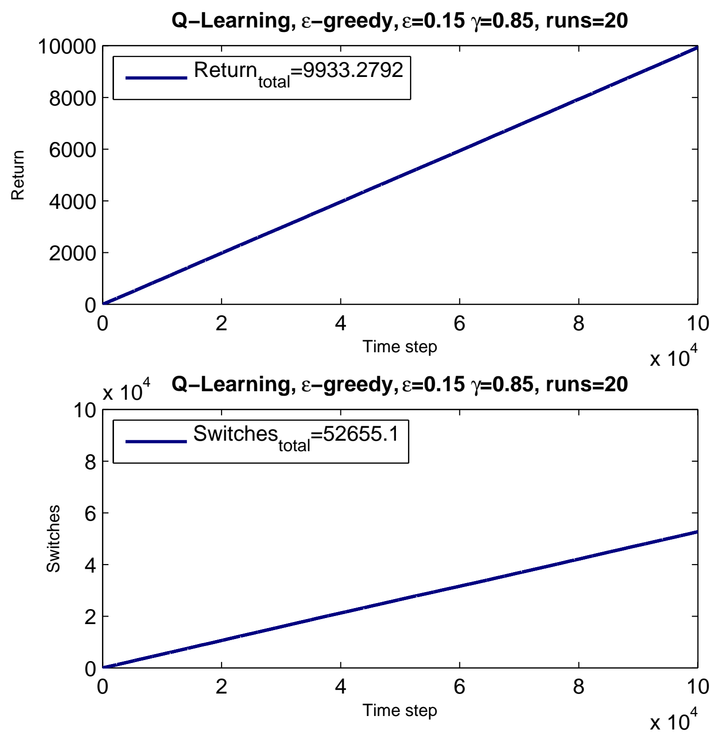

FIGURE 5.46: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 1$ and $L(t) = 70 + 45 * \sin(\frac{2\pi t}{300})$.

and in most real systems there would be a limit to the number of state switches allowed per time unit, due to the (mainly temporal) overhead of setting up new VMs. Another approach at this would be for states to encompass knowledge about the workload, so that the agent may keep more Q-values, and thus immediately distinguish between different situations, and no longer having to change the same few Q-values blazingly fast.

5.1.8 RL Experiment Set 8: Adding a state dimension under a dynamic workload

Settings:

- algorithm: Q-Learning
- state-space: 2-dimensional (number of VMs, workload level)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 1000, 10000, 100000, and 500000 # time steps (min)
- workload: $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$ (K requests/second)
- rewards: $R(s_n, \sim, \sim, t) = \frac{1}{load} * \max(\min(10 * n, L(t)) - vm_cost * n, 0),\ \forall n \in [1, N-1]$

By defining MDP states according, not only to the number of VMs, but to the current workload level as well, Q-Learning is expected to perform substantially better than before, because the observed workload will play a more direct role in distinguishing which action is best under those conditions.

Figure 5.51 displays the decision-making procedure throughout 100000 time steps of serving the same sinusoidal workload of: an offset of 70, an amplitude of 45, and a period of 1000. As a result of adding the second state-space dimension, Q-Learning needs more time than before to adjust to the workload, because there are now 20 times more Q-values that need to be independently updated. However, once all the useful Q-values are updated enough (at around 70000 time steps), the algorithm is then able to adapt much more precisely to the artificially created demand, than it was in the 1-dimensional case.

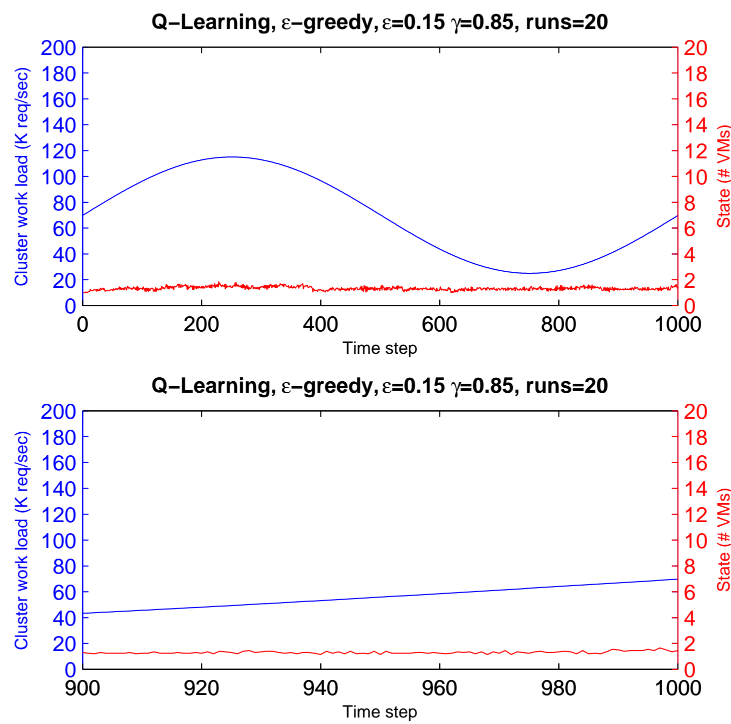

FIGURE 5.47: The current workload and the mean current number of VMs, during 1000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

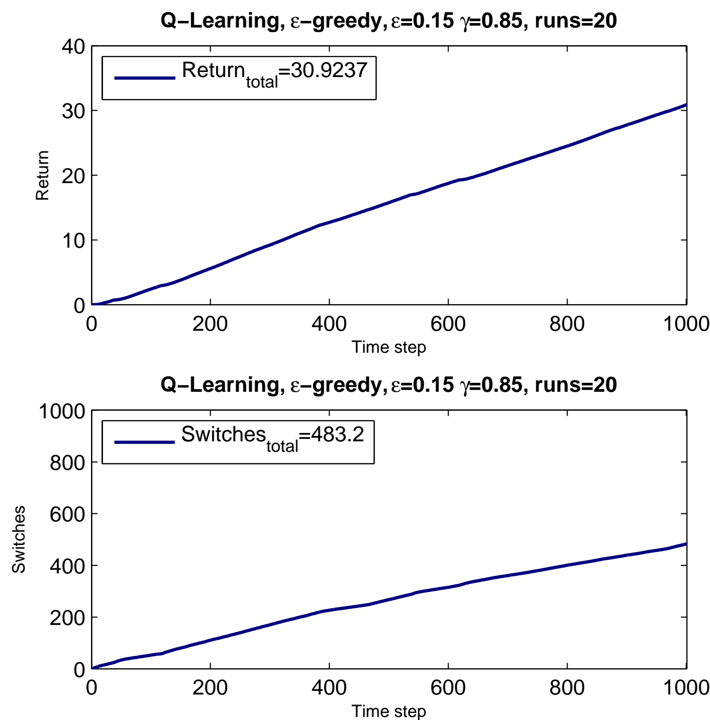

FIGURE 5.48: The mean total return and total number of state switches, after 1000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

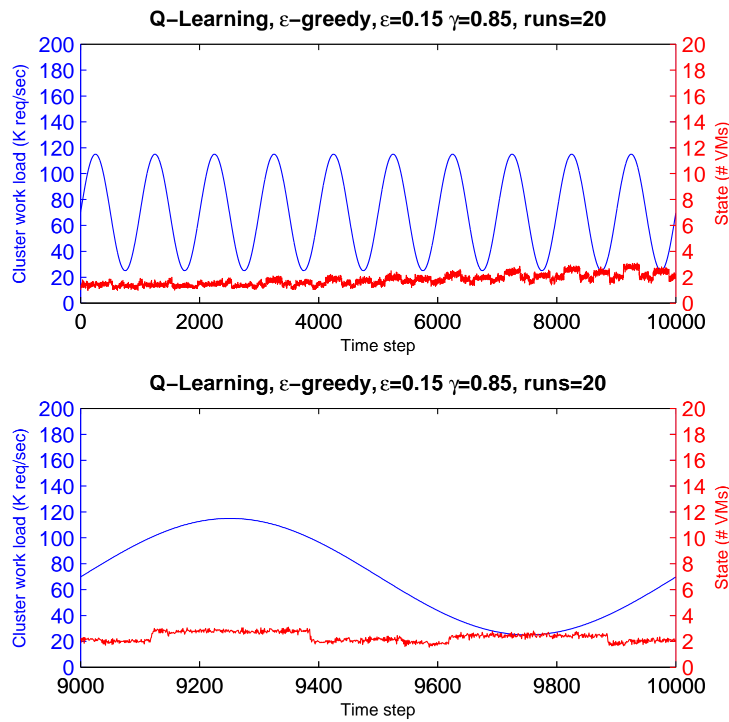

FIGURE 5.49: The current workload and the mean current number of VMs, during 10000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

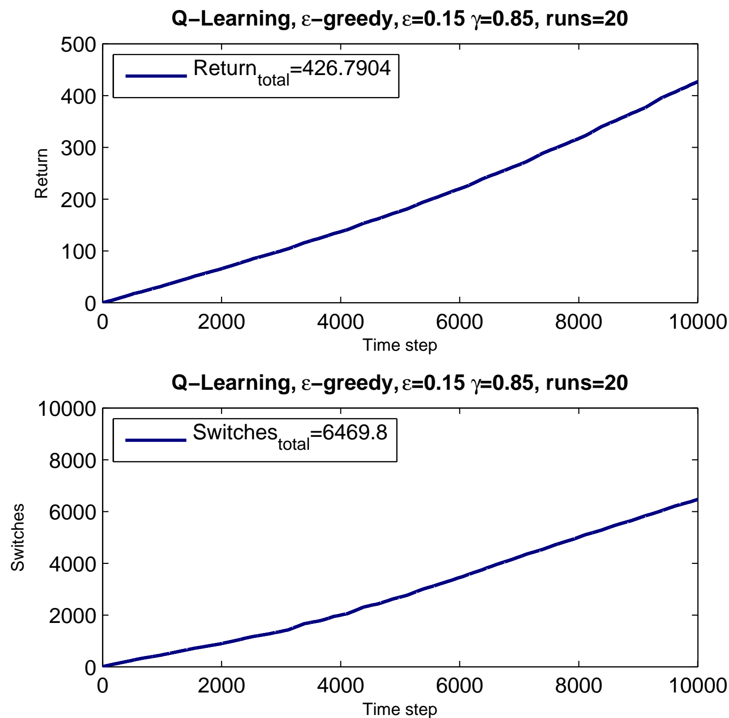

FIGURE 5.50: The mean total return and total number of state switches, after 10000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

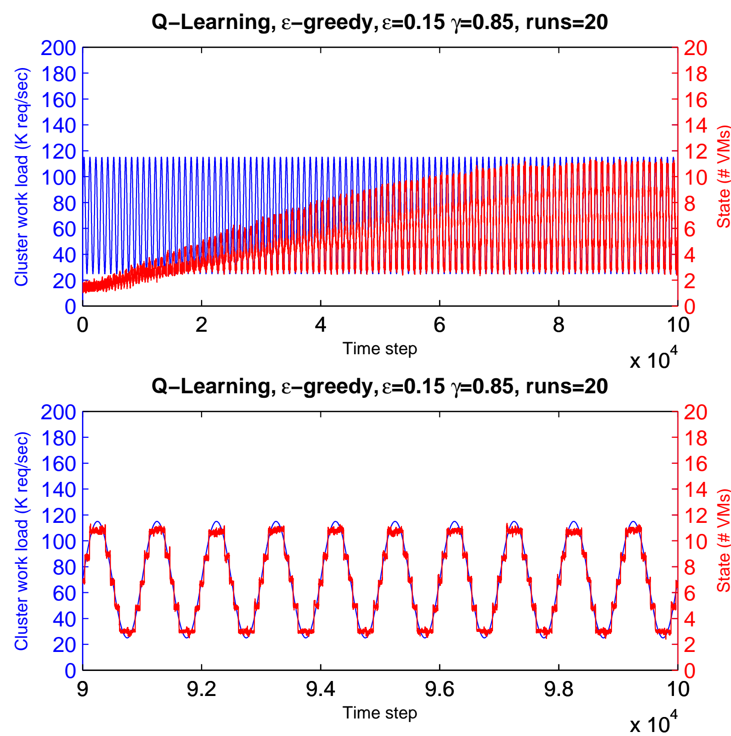

FIGURE 5.51: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

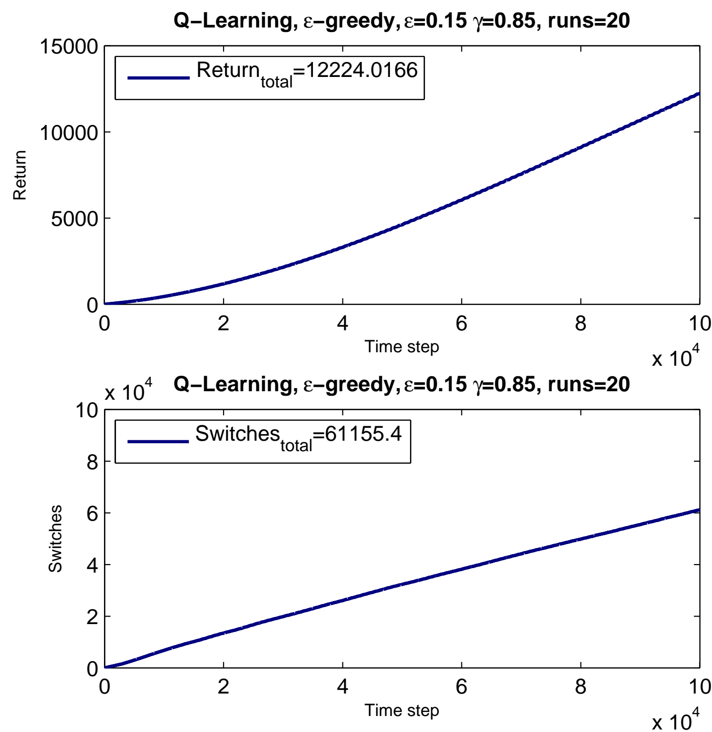

FIGURE 5.52: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

5.1.9 RL Experiment Set 9: The effect of the workload amplitude and frequency

Settings:

- algorithm: Q-Learning
- state-space: 2-dimensional (number of VMs, workload level)
- initial Q-values: pessimistic (equal to 0)
- strategy: ϵ-greedy, $\epsilon = 0.15$
- learning parameters: $\alpha = 0.1$, $\gamma = 0.85$
- iterations: 100000 # time steps (min)
- workload: $L(t) = 70 + 65 * \sin(\frac{2\pi t}{1000})$ (K requests/second) $L(t) = 70 + 45 * \sin(\frac{2\pi t}{300})$ (K requests/second)
- rewards: $R(s_n, \sim, \sim, t) = \frac{1}{load} * \max(\min(10 * n, L(t)) - vm_cost * n, 0), \forall n \in [1, N-1]$

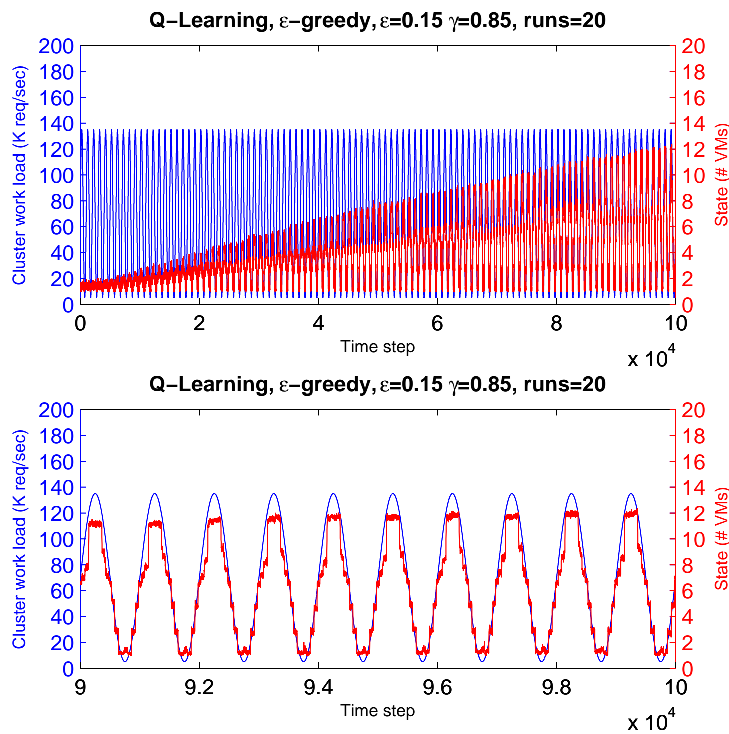

FIGURE 5.53: The current workload and the mean current number of VMs, during 10000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

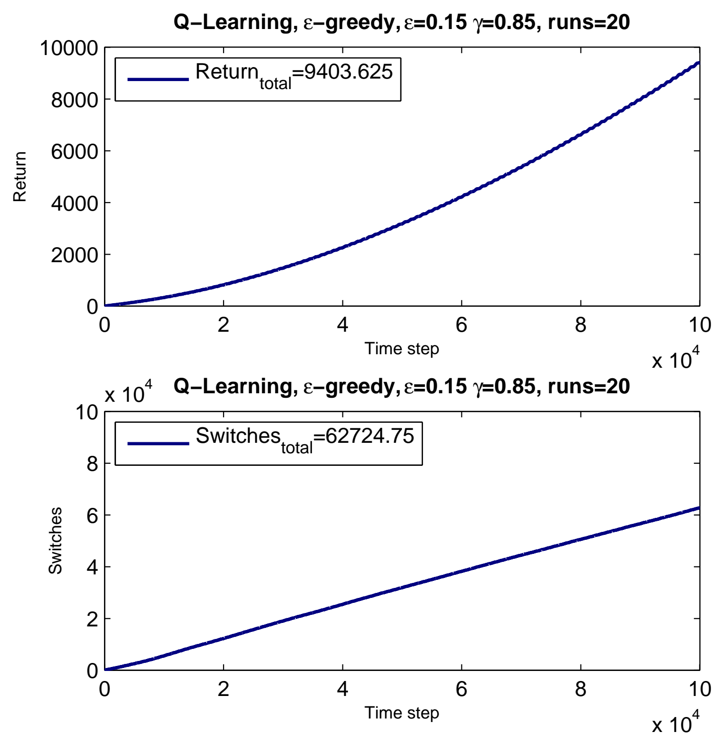

FIGURE 5.54: The mean total return and total number of state switches, after 10000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

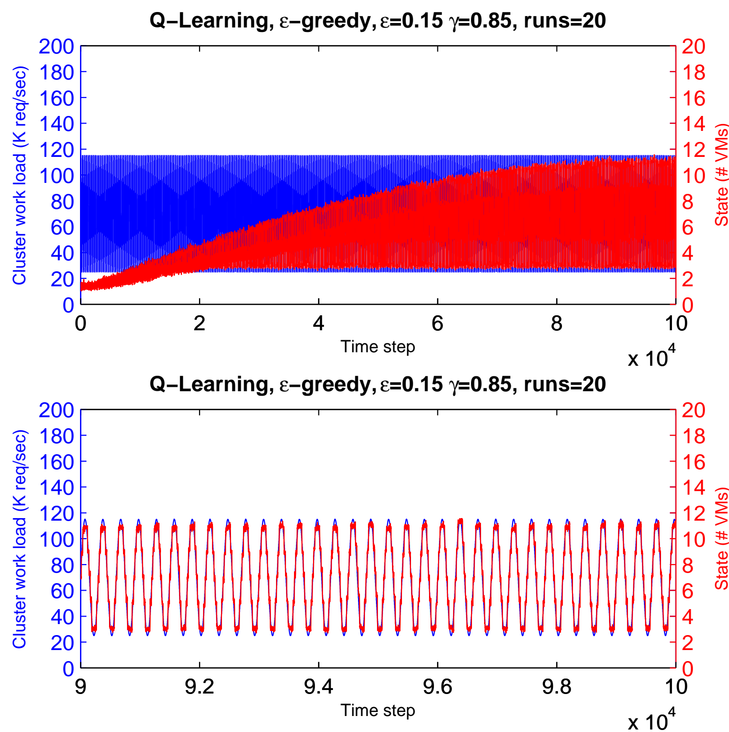

FIGURE 5.55: The current workload and the mean current number of VMs, during 100000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

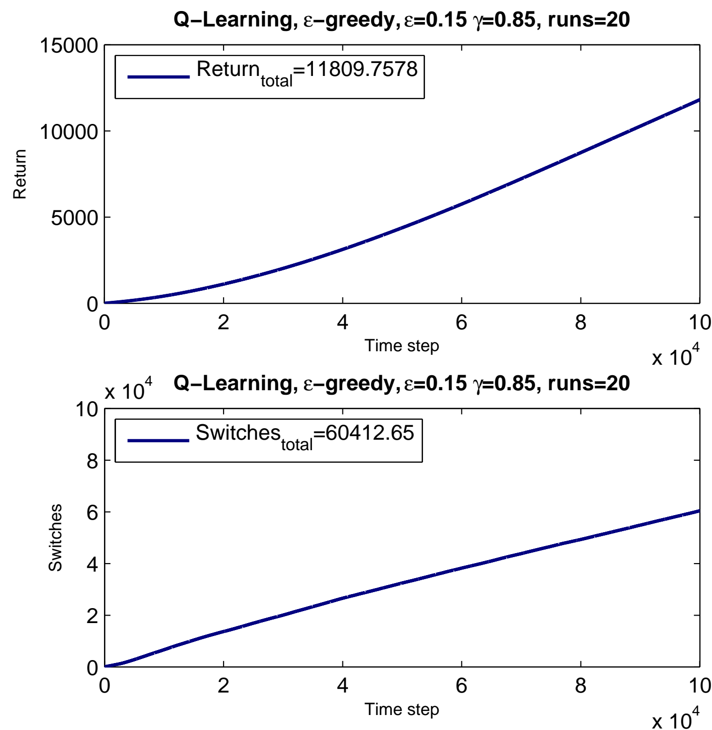

FIGURE 5.56: The mean total return and total number of state switches, after 100000 decision steps, for $\alpha = 0.1$, $L(t) = 70 + 45 * \sin(\frac{2\pi t}{1000})$, and a 2D state-space.

Now, the 2-dimensional MDP model is tested against two more difficult workloads of greater amplitude and of greater frequency respectively, and in specific the workloads that the 1-dimensional model could not keep up with in the experiment set of 5.1.7 (even with a learning rate of 1). Figures 5.53 and 5.54 display results when the workload's amplitude is increased to 65. Figures 5.55 and 5.56 display results when the workload's period is decreased to 300. All figures in this set represent 100000 decisions. The results indicate that a 2-dimensional state-space unlocks the true potential of Q-Learning.

More importantly, it is as easy to add more dimensions to the learning model, like the workload type mixture for example (that the extended version of TIRAMOLA takes into account). Beyond that, the curse of dimensionality takes effect, and an approximate Q-Learning algorithm becomes necessary because the number of Q-values grows prohibitively large.

5.2 MAB Experiment Sets

In this Section, results are presented for MATLAB simulations that were run over a MAB problem formulation. In contrast to the simulations of Section 5.1, the immediate rewards are now stochastic instead of deterministic, but unfortunately only static distributions have been tried out. It would be interesting to run similar experiments over a restless bandit setting, which would more closely resemble a real cloud-based system with its workload highs and lows. This is left for future work, and the focus is now turned into a more educational (useful for understanding the various MAB algorithms) but not so realistic static case of the problem.

5.2.1 MAB Experiment Set 1: The effect of greediness

Settings:

- setting: Standard Stochastic MAB
- arms: 20 i.i.d. Bernoulli arms
- initial knowledge/belief: none
- strategy: ϵ-greedy (with respect to the actual average reward), $\epsilon = 0$, 0.1, 0.2, 0.3, 0.4. 0.5, 0.6. 0.7, 0.8. 0.9, 1
- iterations: 100, 1000, 10000, 100000 # time steps (min)
- workload: $L(t) = L = 100$ (K req/sec)
- rewards: Bernoulli distributed with a mean of: $\mu(s_n) = \frac{1}{load} * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 45; 40; 35; 30; 25; 20; 15; 10; 5; 0]$ respectively.

A comparison of the performance of ϵ-greedy exploration strategies follows, for different values of the ϵ parameter. In the full-RL setting, greediness was defined with respect to the Q-values. Here, greediness is defined with respect to the running average of an arm's actual materialized rewards (as a simple form of reward expectation). The two extreme cases of ϵ-greedy are: random exploration for $\epsilon = 0$ (thick black line), and no exploration (pure exploitation) for $\epsilon = 1$ (thick green line). The simulation was repeated for time-horizons of varying length, ranging from 100 time-steps to 100000 time-steps, in order to examine if and how much, exploring helps in the "long-run". For each case, the corresponding mean value is drawn from 50 identical simulation runs in the graphs below.

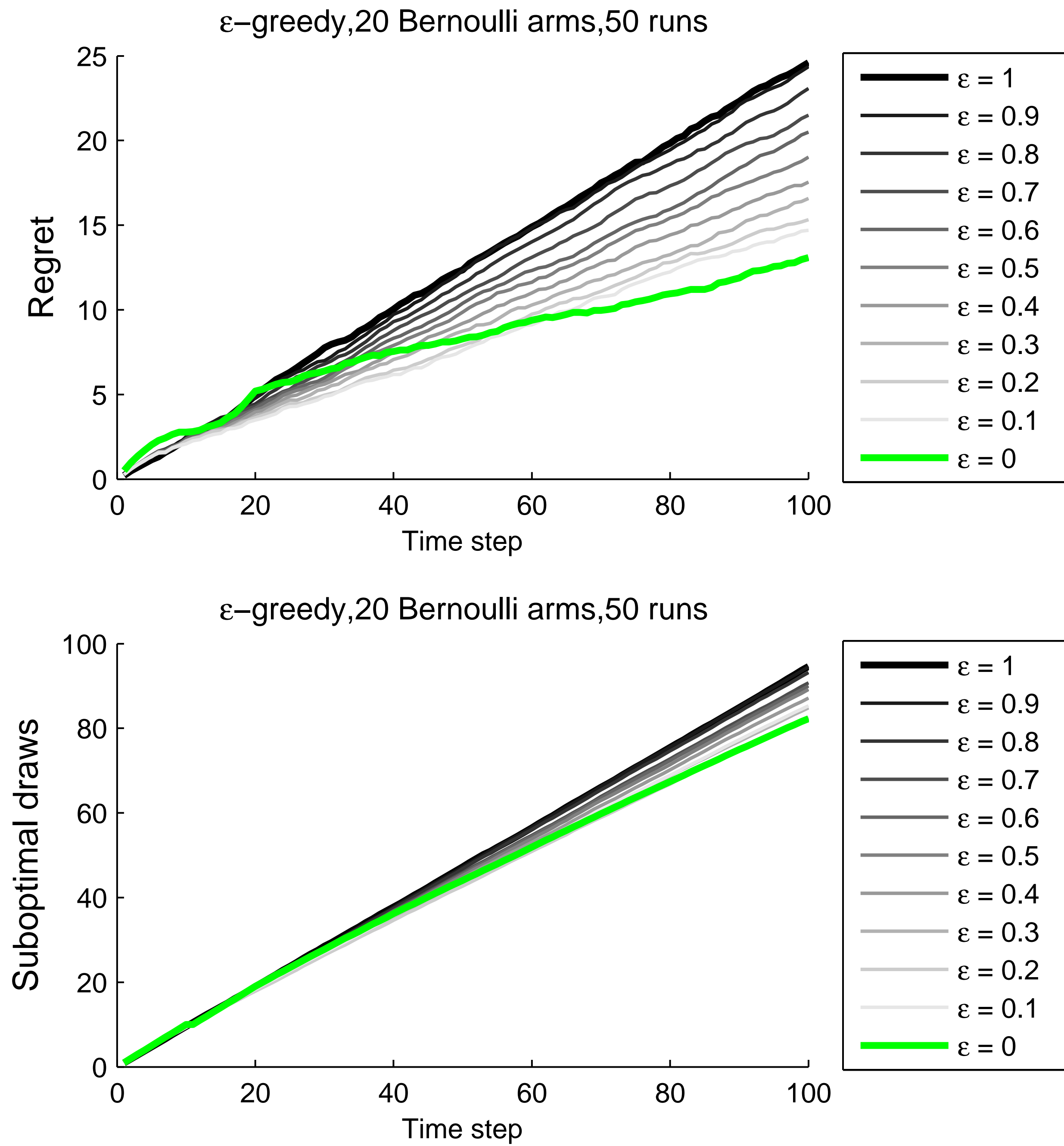

FIGURE 5.57: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 100 time steps.

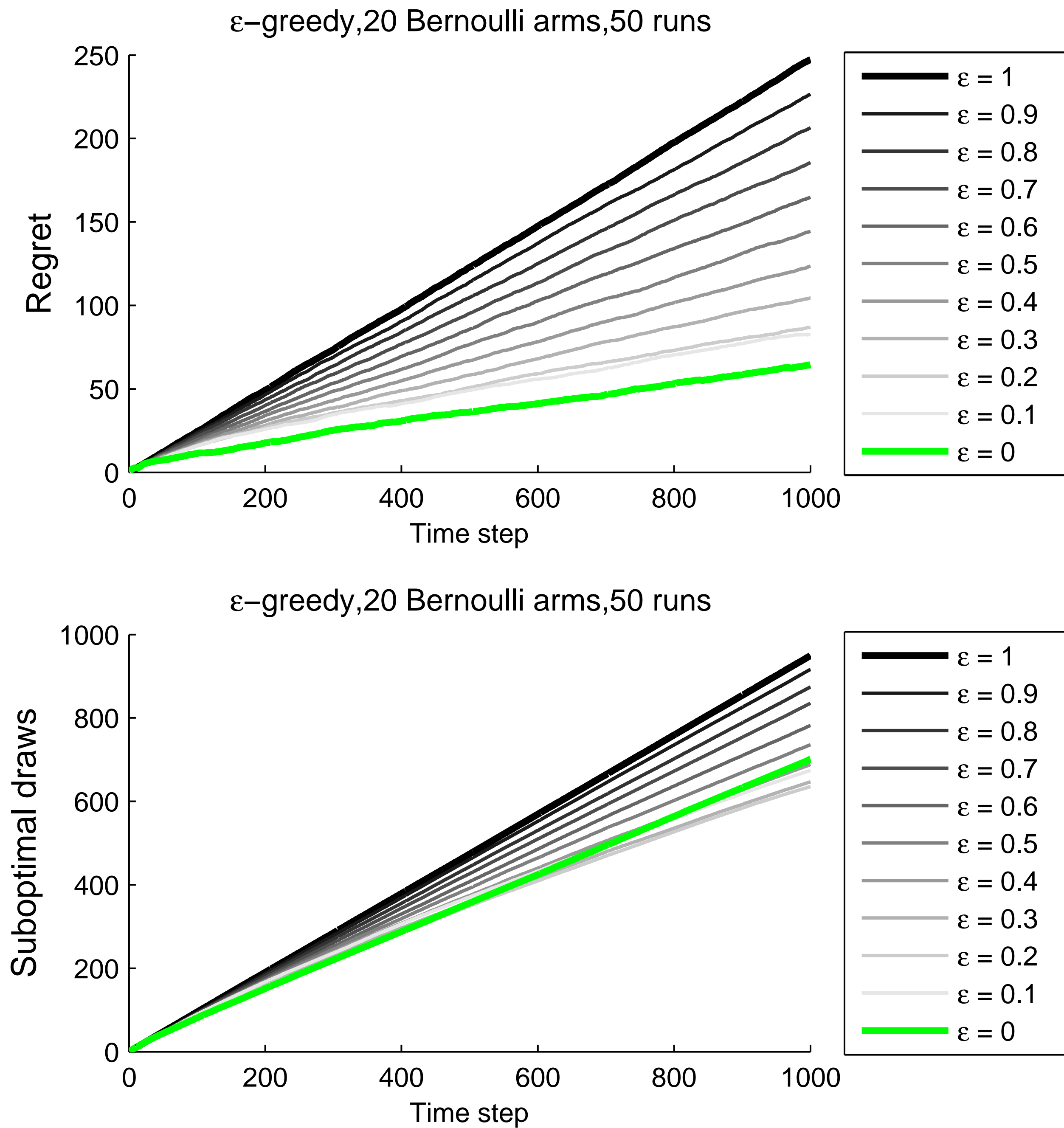

FIGURE 5.58: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 1000 time steps.

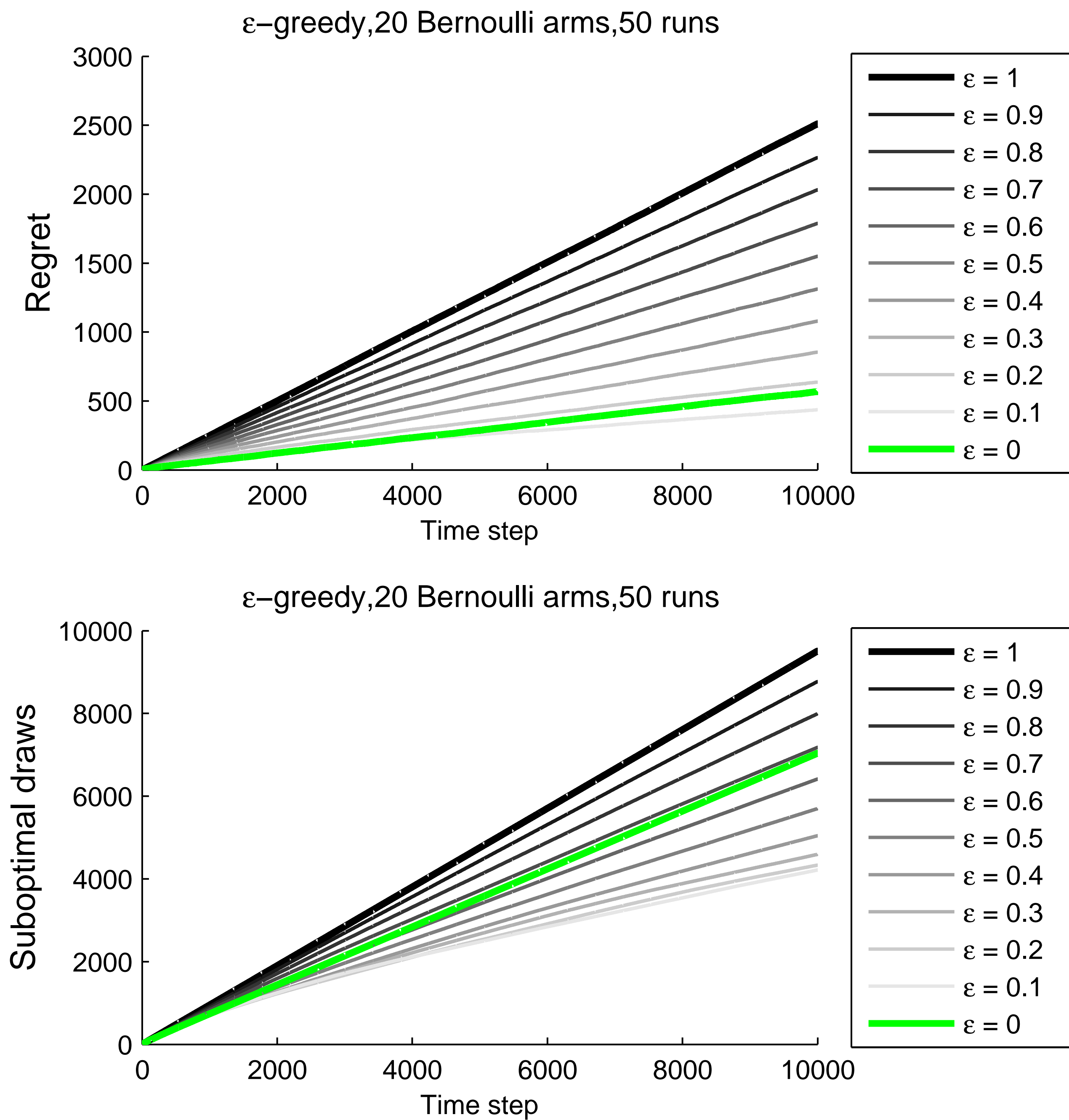

FIGURE 5.59: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 10000 time steps.

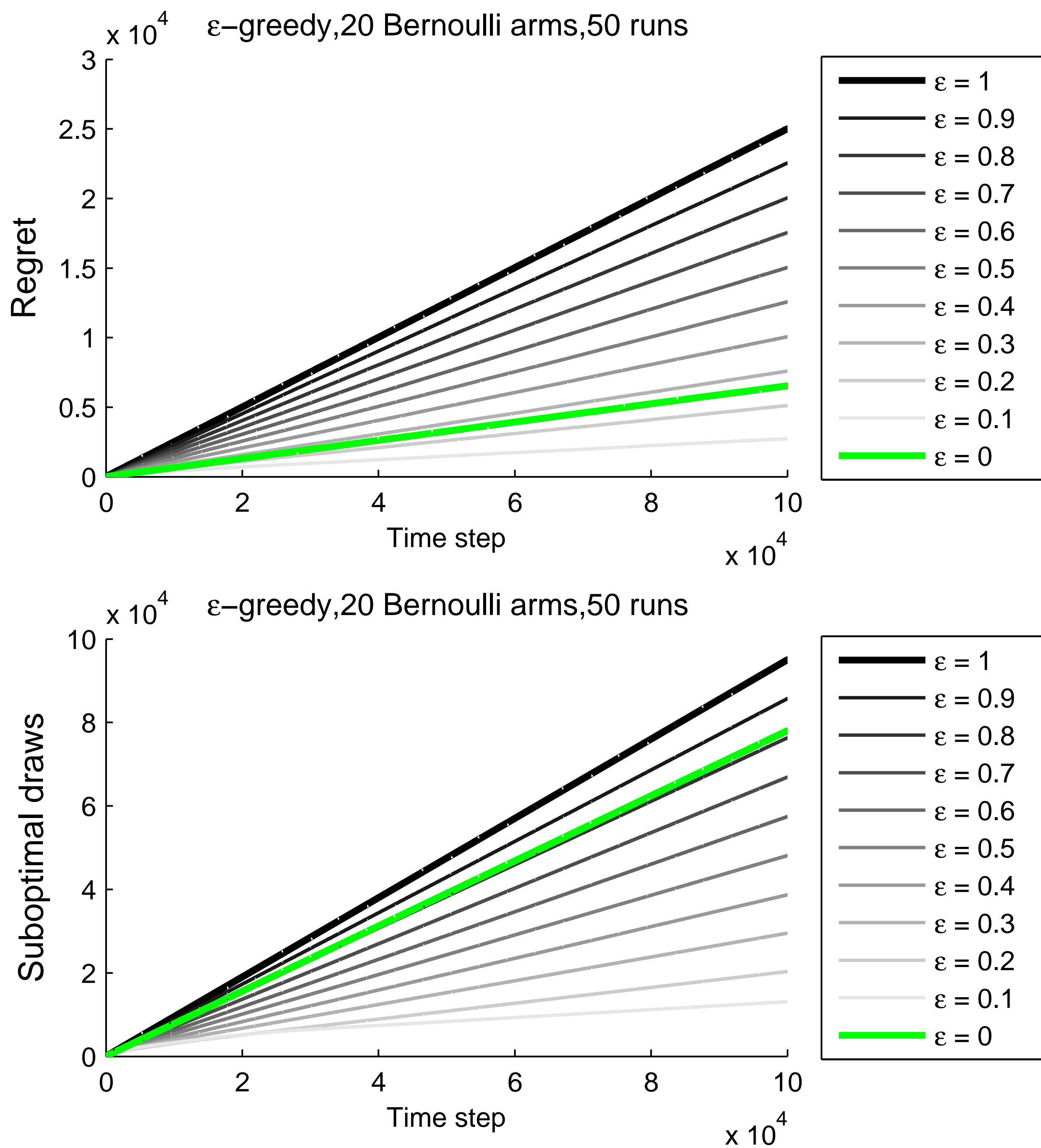

Figure 5.60: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 100000 time steps.

From the above diagrams, one can reach some interesting conclusions. From 5.57, it can be seen that despite the very small time-horizon (of 100 time steps), regret already starts to differentiate according to the value ot the ϵ parameter. In specific, the greedy policy of always playing the arm that with the highest empirical average is the best. This means that random choices do not help when the number of plays is small, because the extra knowledge that they collect about all of the arms, is not met with enough time to be actually put into good use, and statistically leads to more sub-optimal draws than pure exploitation. Actually, the effect is monotonic: the higher ϵ is, the worse performance after 100 rounds. Moving forward to 5.58, one notices that for 1000 time steps, a greedy policy is still the best (in terms of minimizing regret) but no longer achieves the most optimal draws. In fact, four other strategies, with ϵ equal to 0.1, 0.2, 0.3, and 0.4 respectively, all lead to less sub-optimal plays than $\epsilon = 0$. Then, in figure 5.59 corresponding to 10000 time steps, exploring with a probability of 10% finally achieves lower cumulative regret than no exploration at all. For a cloud resource allocation system, 10000 decisions is not really that much far into the future, therefore it is understood that it probably is worth alternating exploitation with exploration every once in a while. Finally, in figure 5.60, yet another strategy surpasses the purely greedy strategy, and it is obvious that even more than 10% exploration behaves better than no exploration at all. This experiment set is very informative since it shows that the ϵ parameter of a MAB (or even RL) algorithm should ideally be set to a value close to 0.1. This has been an example of how simulations can support engineering decisions in a resource provisioning system without the hassle of experimenting with different parameters in a real computational cluster.

5.2.2 Experiment Set 2: Gradually increasing greediness

Settings:

- setting: Standard Stochastic MAB
- arms: 20 i.i.d. Bernoulli arms
- initial knowledge/belief: none
- strategy: ϵ-greedy (with respect to the actual average reward), $\epsilon(t) = 0$, 0.1, 0.2, $\frac{1}{t}$, $\frac{1}{log(t+10)}$, $\frac{\log(t+1)}{t+1}$, $\frac{1}{\ln(t+10)}$, $\frac{\ln(t+1)}{t+1}$
- iterations: 100, 1000, 10000, 100000 # time steps (min)
- workload: $L(t) = L = 100$ (K req/sec)

- rewards: Bernoulli distributed with a mean of: $\mu(s_n) = \frac{1}{load} * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 45; 40; 35; 30; 25; 20; 15; 10; 5; 0]$ respectively.

A natural update to the ϵ-greedy policy is to gradually decrease the frequency of exploratory plays over time. The rationale of this is that exploration is valuable only as long as the knowledge gathered by it, is put into enough good use, therefore exploration should be done in the beginning of the system's life-time. To make this even clearer, if only a few time steps remain until the end of the gambler-casino (or agent-environment) interaction, then there is be no point in playing random arms in those last few steps, it is better to just play what is thought of as providing greater rewards. In the following diagrams we compare the (previously found to be) best cases of fixed amount of exploration, namely $\epsilon = 0$, 0.1, and 0.2, with five new dynamic ϵ-decreasing policies. One policy implements a linearly decreasing parameter, while the other four policies implement a logarithmically decreasing parameter for a smoother transition from exploration to exploitation. For referential purposes, the totally random policy of $\epsilon = 1$ is also re-run.

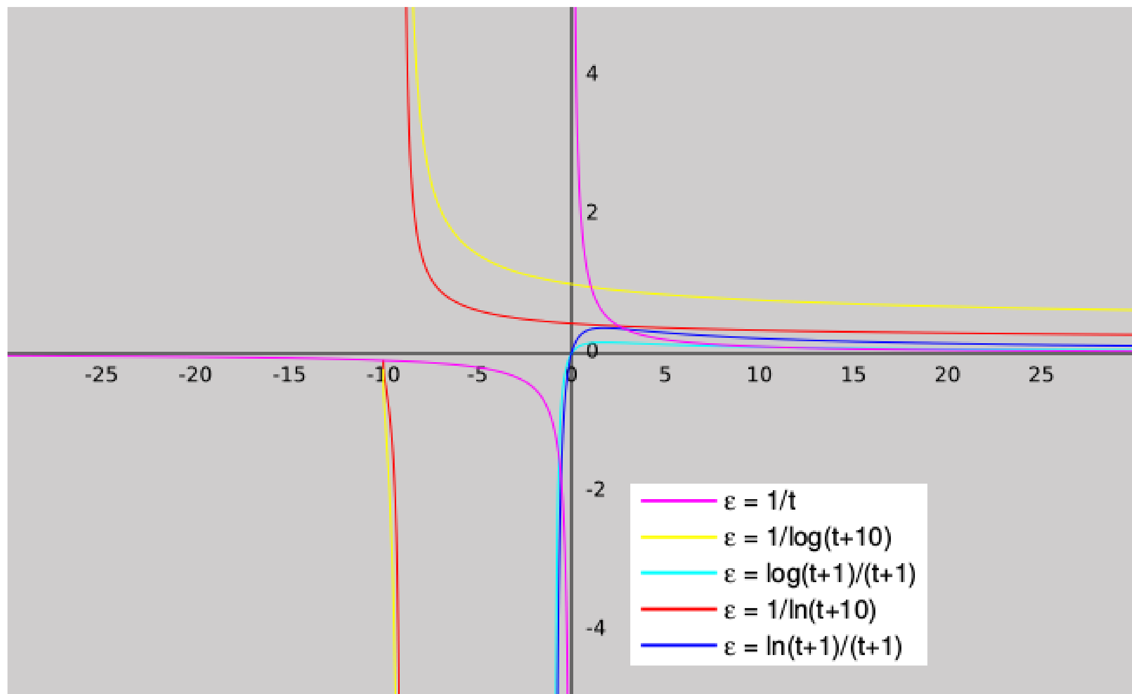

FIGURE 5.61: The five different functions used for defining a dynamic ϵ parameter.

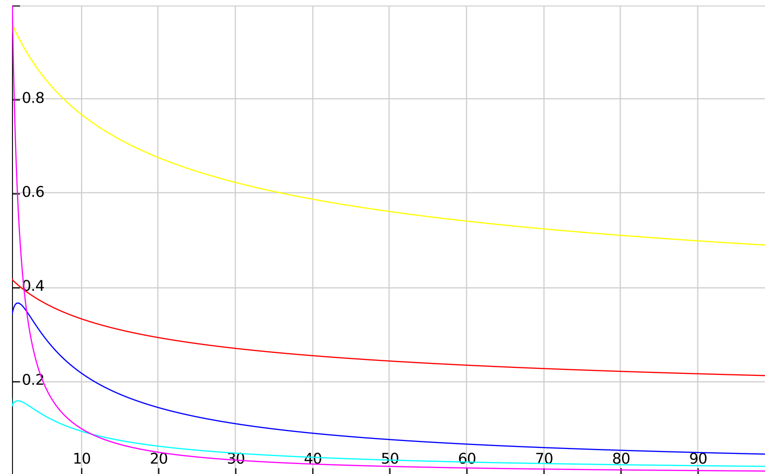

FIGURE 5.62: The five different functions used as a gradually decreasing $\epsilon(t)$ parameter, over 100 time steps.

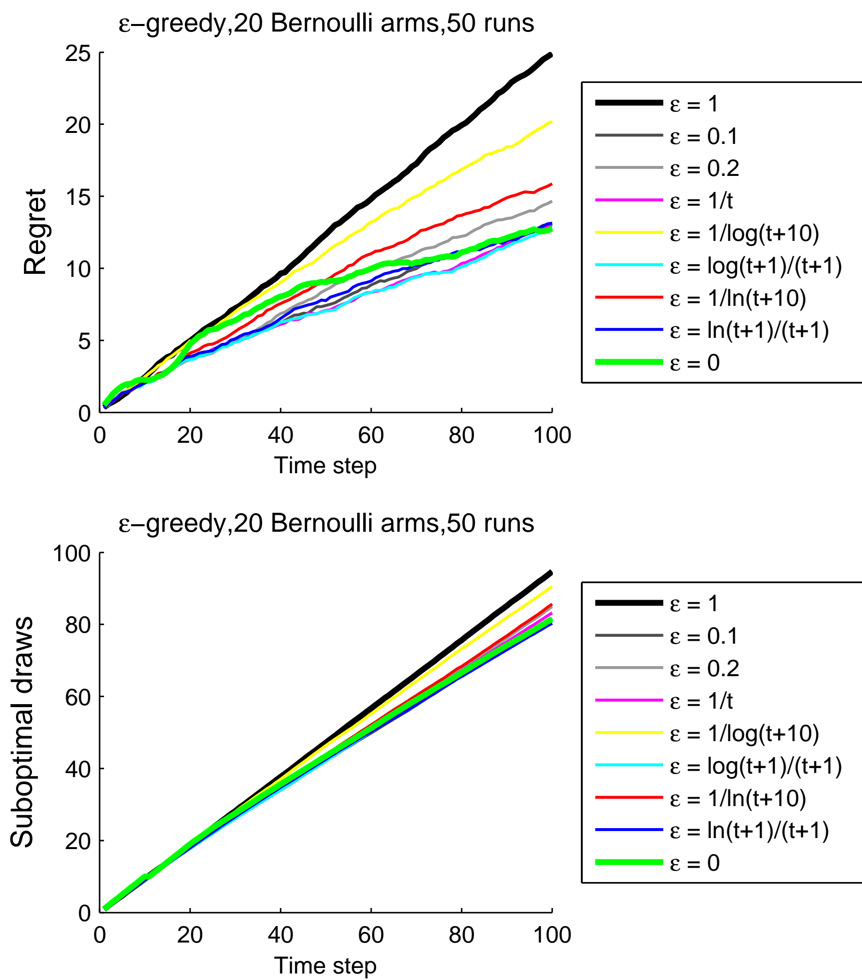

FIGURE 5.63: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 100 time steps.

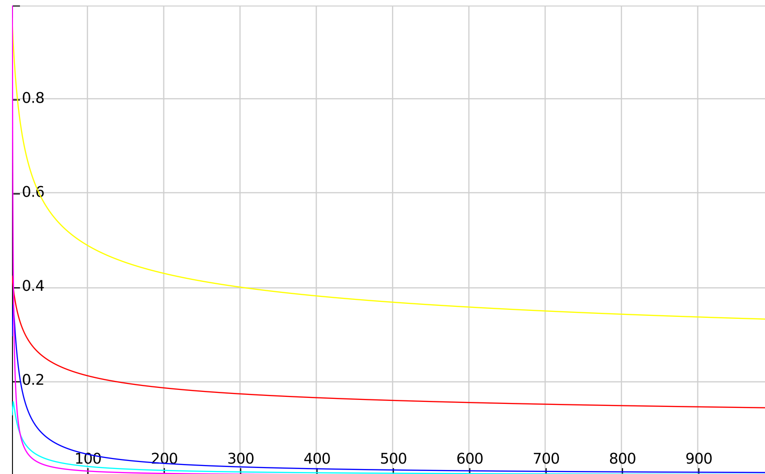

FIGURE 5.64: The five different functions used as a gradually decreasing $\epsilon(t)$ parameter, over 1000 time steps.

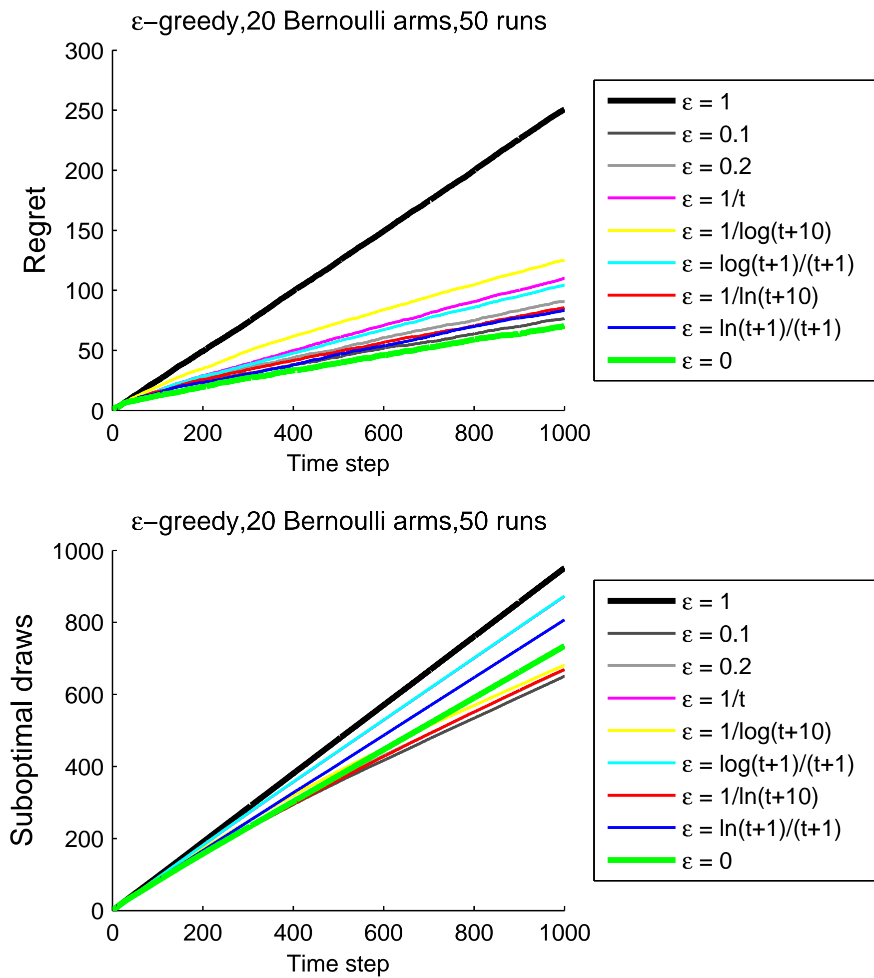

FIGURE 5.65: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 1000 time steps.

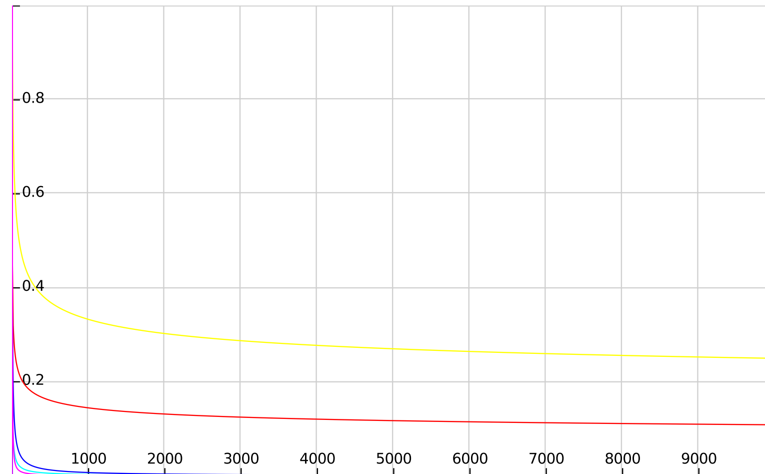

FIGURE 5.66: The five different functions used as a gradually decreasing $\epsilon(t)$ parameter, over 10000 time steps.

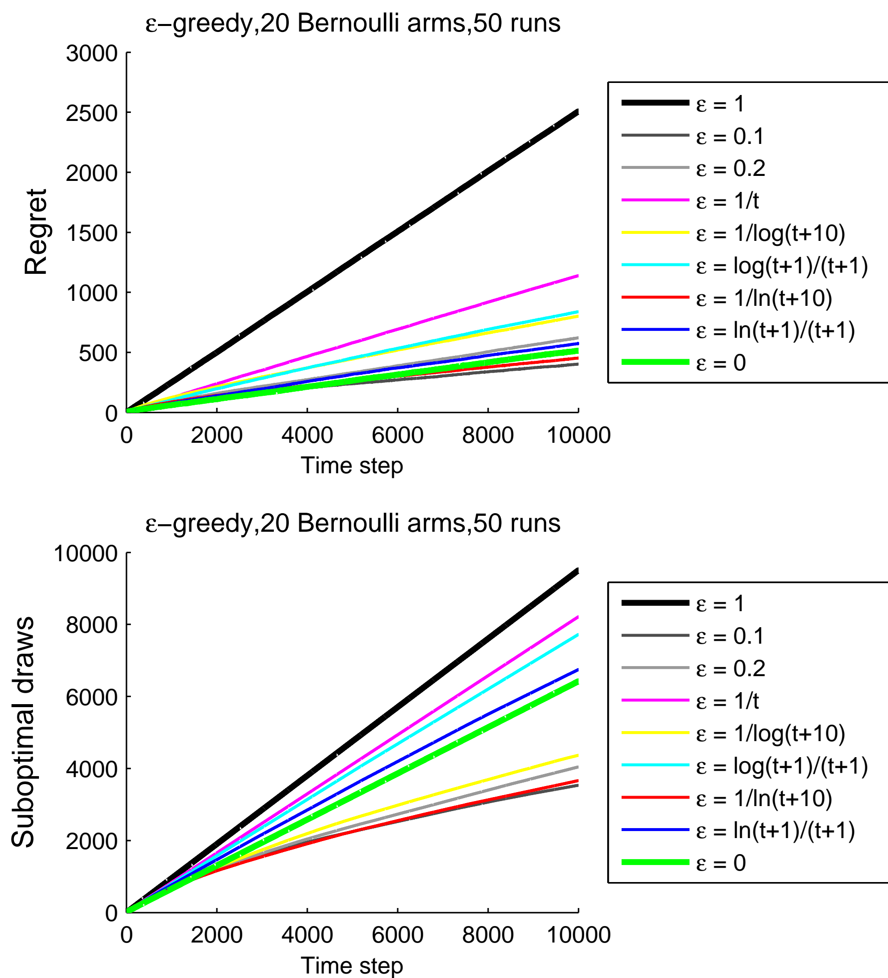

FIGURE 5.67: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 10000 time steps.

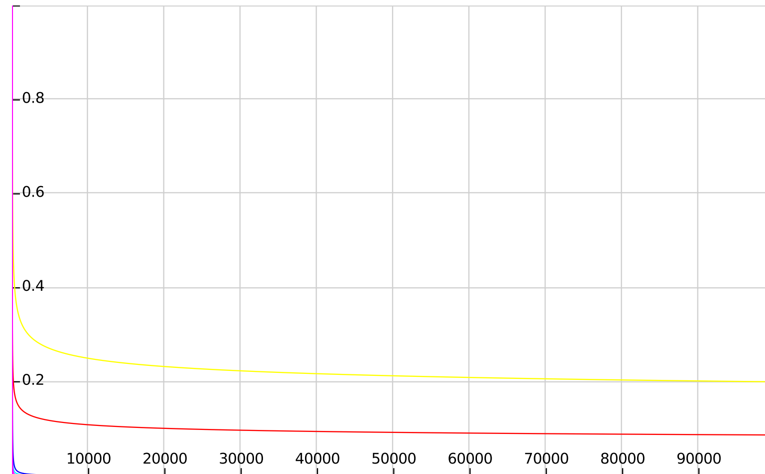

FIGURE 5.68: The five different functions used as a gradually decreasing $\epsilon(t)$ parameter, over 100000 time steps.

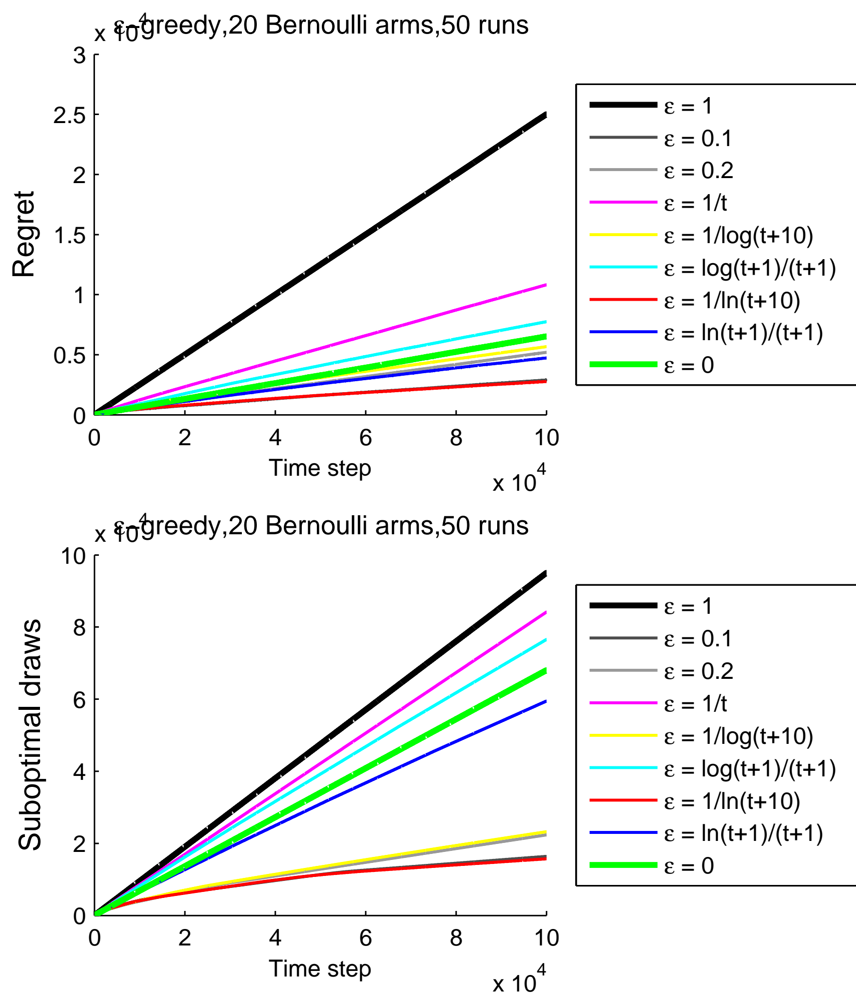

FIGURE 5.69: The mean performance of an ϵ-greedy exploration strategy over a time-horizon of 100000 time steps.

First of all, it must be noticed that the yellow and red functions are of the same exact form and are decreasing considerably slower than the other three functions. They actually need hundreds of thousands of time steps to approach zero. Also, notice that the red function $\frac{1}{\ln(t+10)}$ is the natural logarithm counterpart of the yellow function $\frac{1}{\log(t+10)}$, just like the blue function $\frac{\ln(t+1)}{(t+1)}$ is the natural logarithm counterpart of the ciel function $\frac{\log(t+1)}{(t+1)}$.

From figure 5.63 one understands that due to the small number of playing rounds (100), policies with lower ϵ perform best. More specifically, the green purely greedy policy performs very good as expected. However, the blue and ciel policies that start by exploring with almost 40% and 20% probability and then quickly drop to about 10% and lower, perform even better. The same is true for the purple policy of a linearly decreasing ϵ which again does better than having $\epsilon = 0$ right away from the start of the interaction. Of course, for such a short time-horizon more simulation repetitions should be run to verify if this is consistently true, or if simply the greedy policy was “unlucky” in those particular 50 simulation runs. On another note, policies that retain high ϵ values perform bad, with the totally random policy being the worst among them.

From figure 5.65, for 1000 time steps the same is true, however, the difference is not that great (except for the totally random policy which is very bad). In fact, the red policy of $\epsilon(t) = \frac{1}{\ln(t+10)}$ surpasses the purple and ciel policies and competes with the blue policy in terms of regret. This is interesting because the red policy never drops below a 10% exploration probability but neither does the yellow policy. It seems that as suggested from the previous experiment set, an ϵ value of 0.5 to 0.15 is generally a good choice.

From figure 5.67, for 10000 time steps it is again the case that the two best strategies (in terms of minimizing both regret and suboptimal draws) are, either to keep a fixed ϵ = 0.1, or gradually decrease ϵ from 0.2 to 0.1. Linearly decreasing ϵ as represented by the purple policy has already fallen behind from the logarithmic decreases in terms of performance.

From figure 5.69, the same two policies are still the best (red $\frac{1}{\ln(t+10)}$ and dark gray 0.1) in the longest-running case of 100000 time steps. It should be noted that for such a long time-horizon, all of the decreasing policies except the yellow and red ones, converge to $\epsilon \to 0$ quite fast, and therefore resemble to the purely greedy policy. However, interestingly enough, even if ϵ is very close to 0 all the time (5.68), it is still helpful not to make greedy choices in the very beginning, as demonstrated by the blue policy clearly beating the green policy both in regret and number of sub-optimal choices.

Finally, it can be seen that for the same formula type, the natural logarithm function always performed better than the common (base 10) logarithm.

5.2.3 Experiment Set 3: Taking into account the reward variance

Settings:

- setting: Standard Stochastic MAB
- arms: 20 i.i.d. Bernoulli arms
- initial knowledge/belief: none
- strategy: ϵ-greedy, Thompson Sampling, Bayes-UCB, KL-UCB, MOSS, DMED
- iterations: 100, 1000, 10000, 100000 # time steps (min)
- workload: $L(t) = L = 200$ (K req/sec)
- rewards: Bernoulli distributed with a mean of: $\mu(s_n) = \frac{1}{load} * [5; 10; 15; 20; 25; 30; 35; 40; 45; 50; 55; 60; 65; 70; 75; 80; 85; 90; 95; 100]$ respectively.

In this experiment set, five new real-time action-selection strategies are tested, with the best ϵ-greedy policies from the previous set re-run as a reference. These are five state-of-the-art MAB algorithms (presented in detail in Section 3.6), that take into account the variance of the rewards produced by the arms. In other words they not only consider which arm appears to be the best (as greedy strategies do, even if mixed with some randomness like ϵ-greedy), but also how certain or uncertain that seems to be. The first two: Thompson Sampling and Bayes-UCB, are Bayesian approaches, while the latter three: KL-UCB, MOSS, and DMED, are frequentist approaches.

In figure 5.72, the greedy strategy is no longer any good. The problem with always making a purely greedy choice is of course that, if in the beginning the best arm is under-estimated and some other sub-optimal arm is over-estimated, then the agent keeps playing the latter and neglecting the former. This is why it is important to have at least a small amount of exploration, or else it becomes impossible to ever detect the mistake and the losses perpetuate forever. It is up to chance (and to the specific problem instance) how bad that sub-optimal arm will be. Another remark is that after about 4000 decisions, the smarter approaches to the EvE trade-off issue start to shine. The curve in the regret graph can be attributed to the (proven to be optimal) logarithmic rate of regret accumulation. Among the five intelligent approaches, TS is the best in terms of minimizing regret, but the other four are closely following. Keeping a fixed ϵ parameter equal to 1 is still very good. More interestingly, with respect to the number of switches the difference is far greater. DMED makes the most switches, followed by TS.

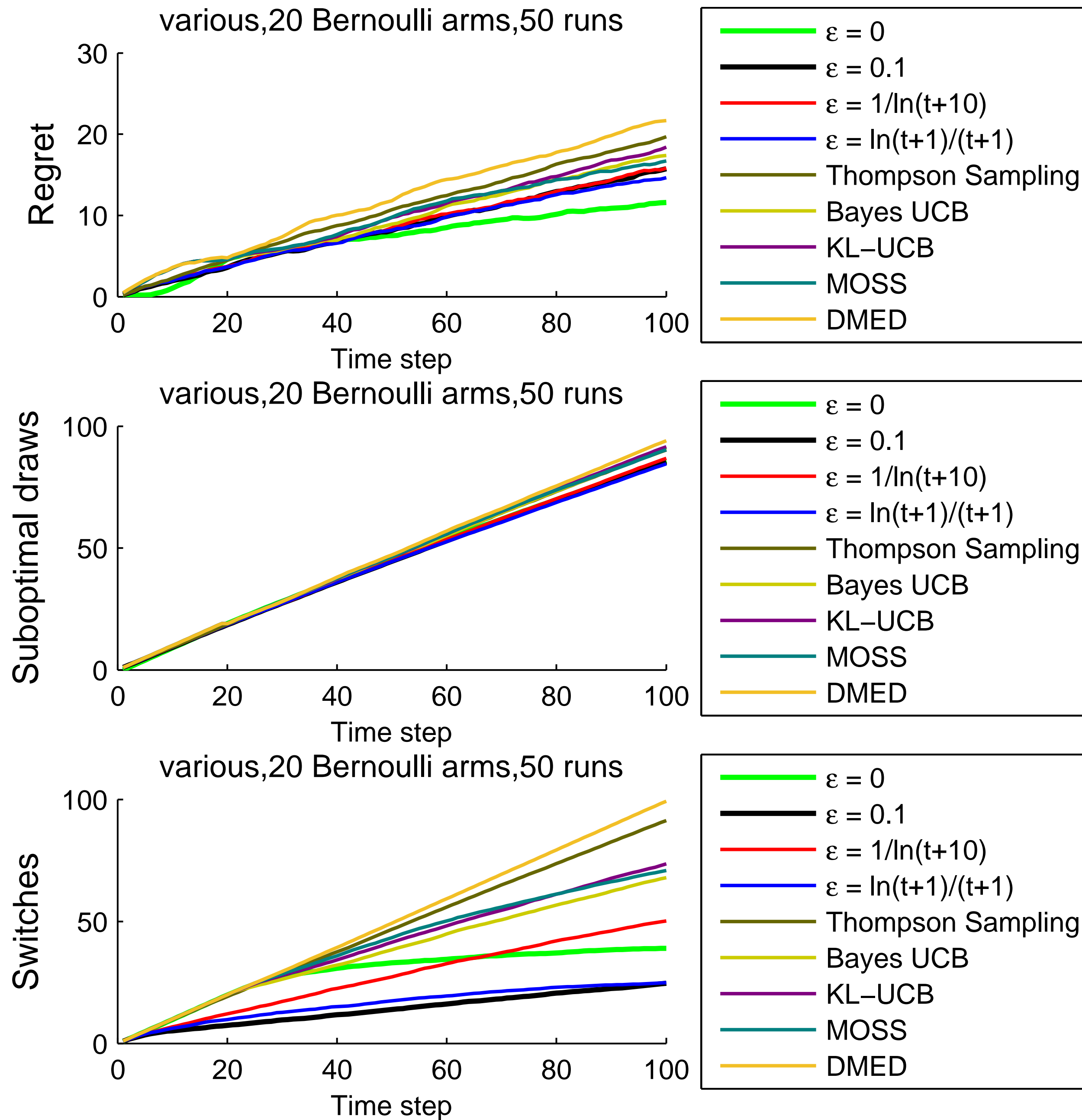

FIGURE 5.70: The mean performance of various MAB algorithms over a time-horizon of 100 time steps.

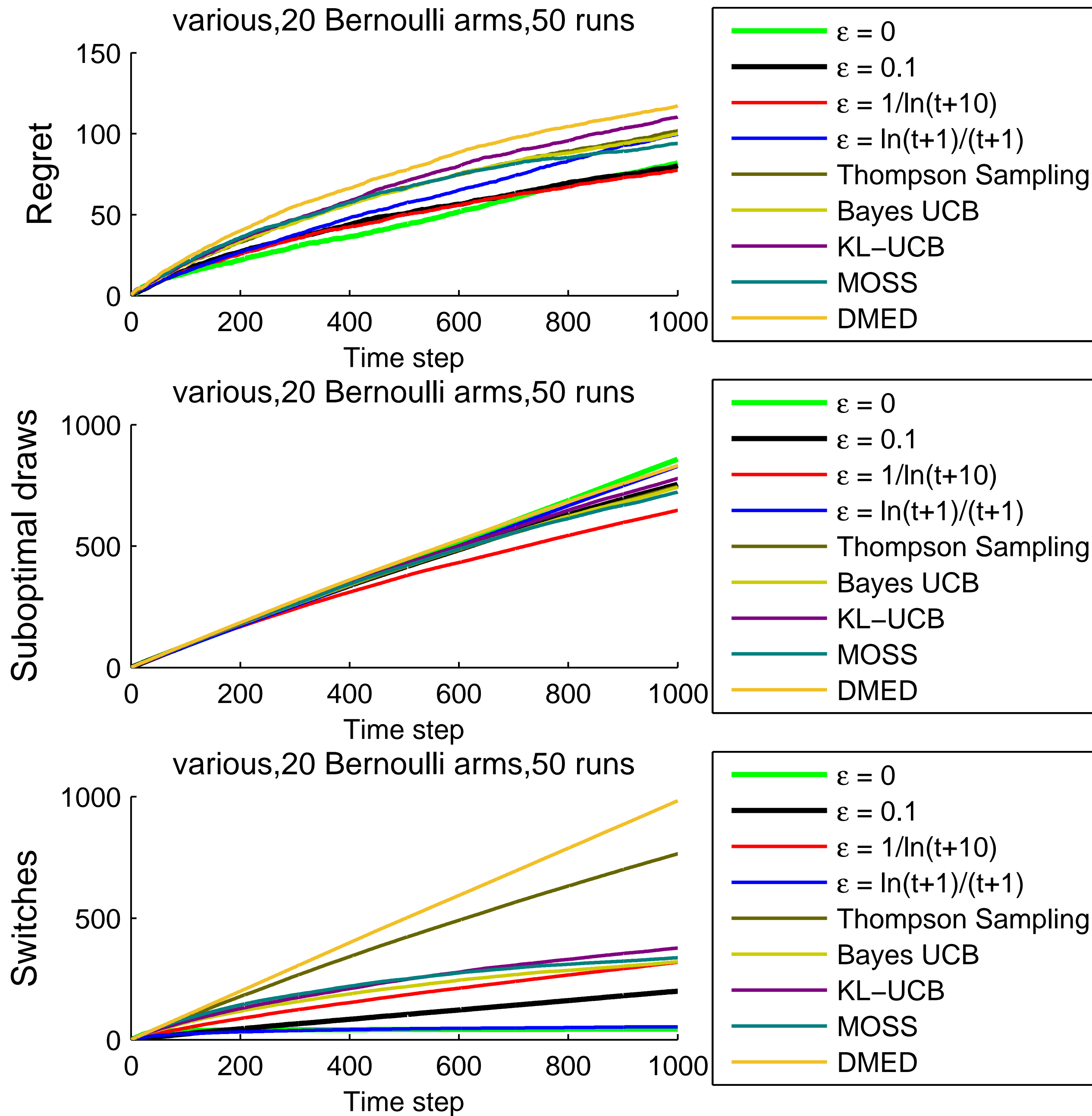

FIGURE 5.71: The mean performance of various MAB algorithms over a time-horizon of 1000 time steps.

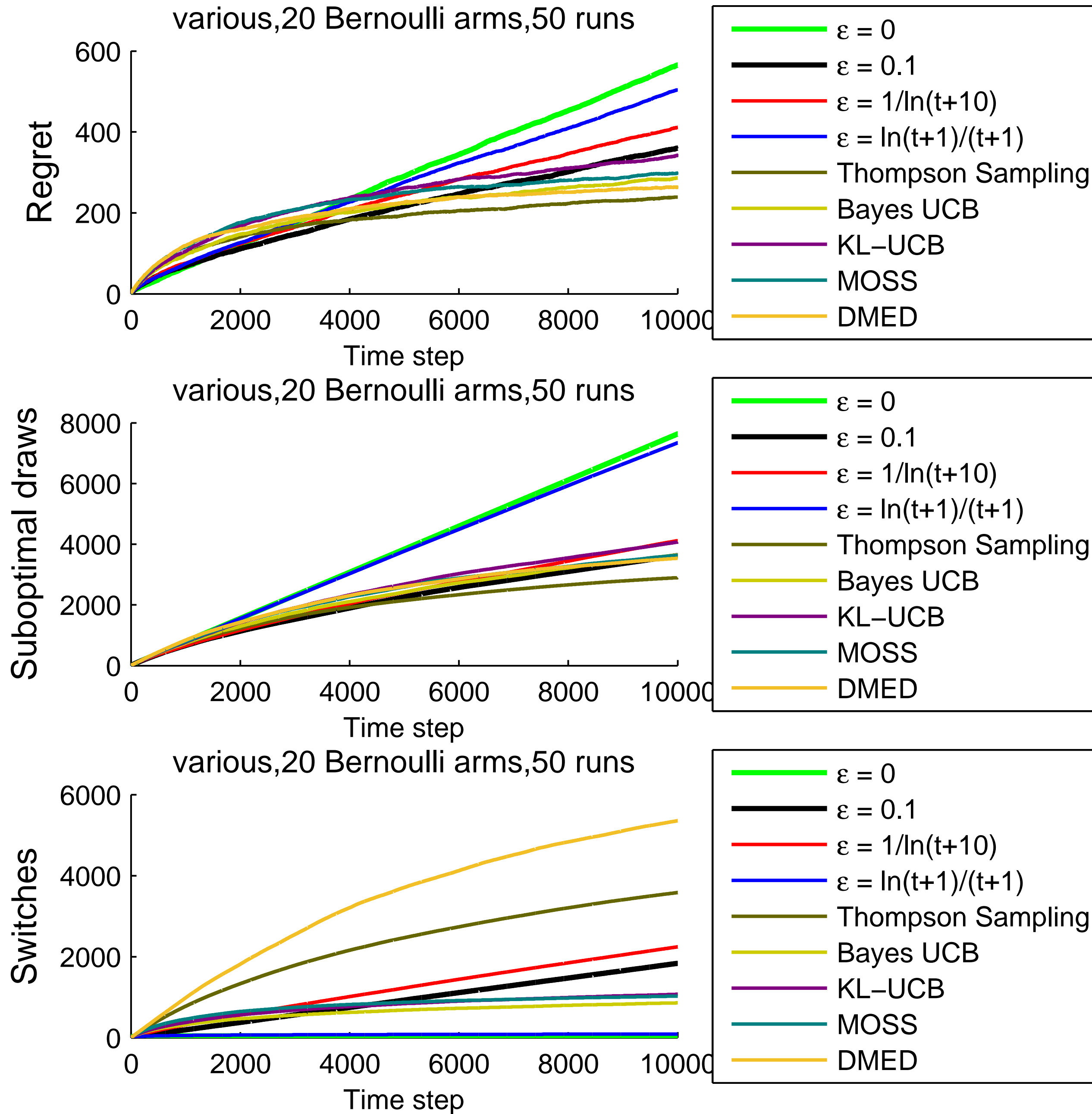

FIGURE 5.72: The mean performance of various MAB algorithms over a time-horizon of 10000 time steps.

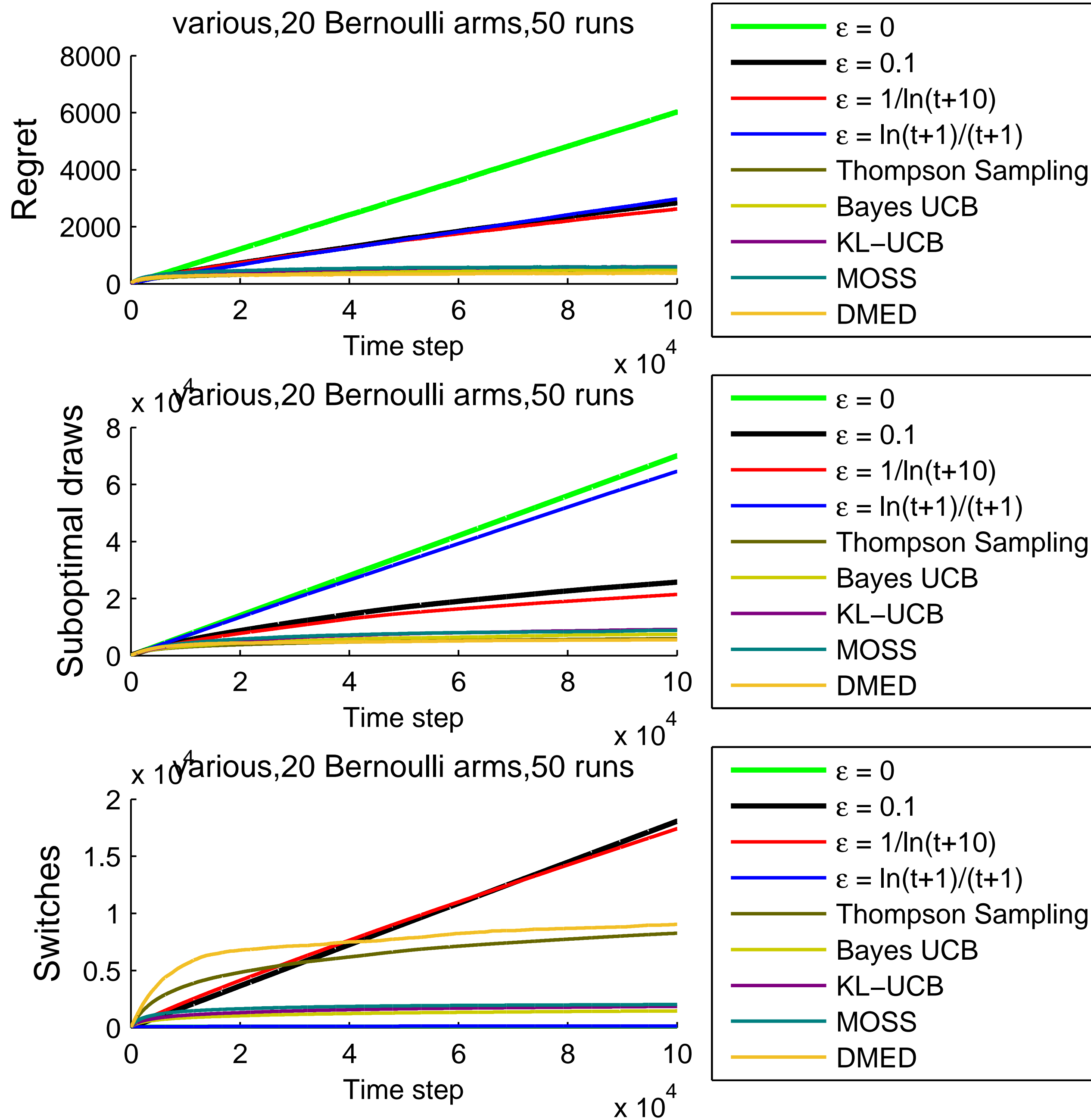

Figure 5.73: The mean performance of various MAB algorithms over a time-horizon of 100000 time steps.

This makes those two algorithms less attractive choices for controlling the VM cluster, because with each transition to another state, also comes a delay in starting up the new VMs. On the contrary, Bayes-UCB, MOSS, and KL-UCB are very attractive choices because they may have slightly worse regret performance, but they achieve so with in a much smaller number of state switches. Actually, after about 3500 decisions they already start beating the greedy approaches (whose switches increase linearly). Why this is the case probably has to do with the Bayesian viewpoint of TS and DMED, but is best left for a deeper theoretical analysis. The situation is the same after 100000 decisions as seen in 5.73: TS and DMED are slightly better but need to change the number of VMs more than 5000 times, while Bayes-UCB, MOSS, and KL-UCB need less than 2500 changes to be made.

Chapter 6

Conclusions and Future Work

In this thesis, we have explored the possibility of implementing Reinforcement Learning and Multi-Armed Bandit algorithms, in order to tackle an Elastic Cloud Resource Allocation problem, and potentially other more complex Resource Scheduling problems. There is currently limited research work on the use of such methods for the automatic control of resources in cloud-based systems. We have run various sets of representative "lightweight" MATLAB simulations, whose results helped us draw some initial (albeit important) conclusions. Here we will summarize the most important ones and propose a direction for future work.

6.1 Conclusions

Reinforcement Learning (RL) (Chapter 2) shows a lot of promise in helping an autonomous agent (or group of agents) improve its decisions over time. There are various RL approaches and even more RL algorithms to consider. In this work, we have mostly focused on Q-Learning, being the most iconic among them. We will therefore present our conclusions about, Q-Learning in specific, and RL in general. Some of the theoretical advantages of Q-Learning were verified in our discrete-time simulation scenarios, which leads us to believe that its use in real systems is worthy of consideration:

- Q-Learning is relatively easy to implement. At least in its (basic) tabular form, programming a Q-Learning algorithm is straightforward. The most time-consuming part is probably coding the Markov Decision Process (MDP) formulation over which it has to run, but the upside of this is that the modeler then possesses an environment over which he can implement other decision-making approaches as

well (for example Dynamic Programming algorithms). Moreover, the modeler can alternatively adapt his code into existing MDP frameworks, since MDPs have long been studied and used in Control Theory and other scientific fields[1].

- Q-Learning is very fast, which is of the utmost importance in cases where a system has to make informed decisions in a temporal resolution of seconds or even milliseconds. In our MATLAB experiments, Q-Learning chose an action in a matter of milliseconds and of course we would expect the algorithm to run even faster in a production system. This is possible thanks to the incremental nature of Q-Learning, where each experience is taken into account only at the moment it is observed and then is discarded. History-based RL approaches can be fast too (TIRAMOLA for example keeps a log of all past experiences and re-learns from some of them), but they start tipping the scales more to the favor of precision at the expense of speed.

- Q-Learning is a very generic algorithm, an important advantage in the cloud domain where there exist multiple stakeholders and multiple objectives, as well as a constantly changing environment over which it is virtually impossible to analytically plan a good solution (if any at all). This is actually a feature of all RL algorithms in general, because in the RL problem setting, the learning agent completely ignores the model of its world, and therefore "black-box" methods are required. For example, depending on the user, a different reward function might be specified, and the learning algorithm does not have to change at all to support that new function. Another example of fruitful generality, is that although in our simulations we assumed control over the number of VMs, even if the agent controlled another resource type, the decision-making logic would still be the same "trial and error" one, totally irrespective of the true nature of the actions and controls.

- Q-Learning is based on a well-structured algorithmic process, and thus provides some standard means for fine-tuning the learning procedure and hence better tackle each specific problem. Our simulations have shown that indeed the algorithm's performance is greatly affected, by the specific techniques being used (optimistic/pessimistic initial ignorance, exploration-exploitation balancing strategy, etc), as well as by the parameter values (learning rate, discount factor, epsilon parameter). Ad-hoc techniques or custom parameters can always be added to the basic algorithm if so desired (because in the end decisions will simply depend on the ordering of

[1] In this work we have modified an existing educational "bare bones" Matlab implementation [70], in which the agent was only taking random actions, and implemented ϵ-greedy strategies instead, among the many changes.

the estimated Q-values) but we would expect this to be rarely the case since the modeler already has enough tools to tune the learning procedure with.

- Q-Learning can even be modified in various (both subtle and radical) ways: implement batch-learning (cycles of multiple Q-value updates) to perhaps learn from log files, turn into an On-policy algorithm (SARSA), specialize for problems with a relational structure (Relational RL), change the amount of lookahead taking place at each step, change the type of Temporal Difference Learning being used, implement Probably Approximately Correct (PAC) learning, implement Supervised Learning methods for generalizing experiences over many states (see next Section), implement double estimators to avoid over-estimating Q-values, implement Genetic Algorithms for finding good parameter values, and many more. Thus, even though Q-Learning is suspiciously simple, it should not come as a surprise that it finds its way into real-life applications. What is more, it can be used in conjunction with non-learning methods and together deal with the Resource Allocation and Scheduling problems in a synergistic fashion. These methods can include: dynamic-bin packing problems, adversarial settings and game-theoretic models, economic models, and other methods that the cloud community is currently considering for automated Resource Management solutions.

As our simulations suggest, a key point in successfully implementing Q-Learning is to carefully design the MDP itself. Perhaps the most important part of this, is to come up with suitable MDP states, able to efficiently represent the variety of situations that the agent will face while learning, and which indeed should play a role in choosing the right action (because not every type of environmental distinction is worth taking into account). The available actions need to be properly chosen as well, but the modeler can be less scrupulous about it, since actions will more or less correspond to the elastic properties of the real system, known in advance of designing the learning model. In addition, if one wishes to build a simulator of the learning procedure, like we have done in this work, then he will need to additionally define the transition probabilities and rewards. These should be (at least roughly) realistic, which again is not very easy, because it depends almost entirely on domain-specific knowledge. This is why insight from domain experts (in this case cloud engineers) is invaluable for designing effective RL solutions for managing cloud-based systems.

Multi-Armed Bandits (MABs) (Chapter 3) on the other hand are also very promising for dealing with the same range of problems, however MABs should be preferred when the problem has a simpler, more straightforward structure, the reason for that being

that they can only accommodate multiple actions, not multiple states. For such uncomplicated settings (e.g. a clinical trial problem setting) most of the aforementioned advantages of Q-Learning still apply, but now the focus is on trying to perform the learning procedure itself in an optimal way. This is much easier without the full-RL state interconnection complexity, because it becomes more viable to perform statistical analyses by assuming independent reward distributions for the actions.

In our MAB simulations we have numerically and graphically verified the theoretical superiority of state-of-the-art MAB algorithms, greatly improving performance over greedy approaches, at least in long-running systems (tens of thousands of decisions or more). We have also witnessed the logarithmic (with respect to how regret increases over time) optimality of those algorithms. Additionally, we kept track of the number of arm switches as another metric of comparison. Some of those algorithms performed substantially less switched than the rest, and still optimally learned the optimal arm, with less variation in their choices. This leads us to the conclusion that for systems where switching costs are important, the modeler should at least keep track of them, if not provide for them within the model itself.

6.2 Future Work

Provisioning resources is directly connected to Service Level Agreements (SLAs) and has at least three relevant aspects: availability, performance, security. In this work, we only dealt with the performance aspect of resource allocation. Our goal was performance optimization and we did not provide for the other two aspects. However, a learning environment should cope with fault-tolerance requirements for example. These and other requirements are usually in the form of rules (i.e. anti-affinity rules). With respect to the learning procedure itself, they can be easily implemented as occasional restrictions over the available actions of the agent (both in the full-RL and in the MAB setting). It would be worth investigating how dynamic restrictions affect the agent's behavior and performance.

More importantly however, we suggest two variations (one for each setting) of the techniques studied in this work, that we believe are more suitable for a real system, thanks to their handling of scaling issues. These are Value-Function approximation and Contextual Bandits.

6.2.1 Approximate Q-Learning

The problem with tabular Q-Learning is that it does not scale well with the number of state dimensions: if we add many random variables to denote the MDP state, then there will be exponentially many different states, rendering Q-Learning unable to efficiently update their Q-values or even keep those Q-values stored in memory.

This is a major limitation for real-world applications, which is traditionally tackled by implementing classical Machine Learning methods to support the main Reinforcement Learning algorithm, in order to generalize each estimation update over many states and actions. In this way, each new experience allows the agent to learn more about other similar experiences. Machine Learning algorithms have long been doing exactly that type of generalization through the use of domain-specific features, characteristics of observed data that affect how each value should be updated. In our case, we would like the agent to update many Q-values at once, even when the particular experience only involved only one specific state. In other words, we would like a way to map a state to a set of feature values, in order to effectively reduce the model's dimensions.
Q-Learning is a value function algorithm and thus can apply this generalized learning scheme, by approximating its Q-value function. This means that the Q-value function will no longer be fixed as it was throughout this work. This is known as Approximate Q-Learning, and can be implemented with various Machine Learning techniques, such as a simple linear regressor, a neural network, etc. In practice the Q-value function is defined with respect to the features, so that each state will get its own Q-value updated according to the degree of its feature similarity with the original state experienced in the transition. Feature have been traditionally specified by domain experts, but recently there have been breakthroughs in automatic resource extraction, in which the learning system itself chooses those features.

6.2.2 Contextual Multi-Armed Bandits

Bandits do not have multiple states. This is very limiting for modeling real-world systems and phenomena. In future work, Contextual Bandits should be considered for coping with the exploding number of possible system configurations. Contextual Bandits exploit a generalization capability similar to Approximate Q-Learning, where an arm's reward estimation is updated by examining rewards from other arms as well, provided that similar stimuli (not necessarily related to the arms) have been observed at

the time of the decision.

Formally, the setting of the CS-MAB problem is extended to include context information:

1. the world's context:
   $x_t \in X$
2. the set of K arms (or levers):
   $L = \{1, 2, ..., K\}$, each representing a probabilistic distribution of rewards.
3. the set of actions (or action-space):
   $A = \{a_1, a_2, ..., a_K\}$, each corresponding to choosing the respective arm.
4. the normalized reward (or gain) vectors:
   $r_t = (r_{1,t}, r_{2,t}, ..., r_{K,t}) \in [0, 1]^K$

where:

- $t \in [1, T]$
- T the number of discrete time steps (or trials or rounds).
- $l_t \in 1, 2, ..., K$, the arm chosen at time step t.

The goal is then to choose the optimal arm given the currently observed context. The context corresponds to the features of the environment that are present at any particular time and affect the production of rewards (non-trivially). Therefore, the goal is now to find an optimal mapping from context to actions (arms), similarly to the full Reinforcement Learning's goal of finding an an optimal mapping of states to actions. We believe that Contextual Multi-Armed Bandits can perform well in problems similar to the one TIRAMOLA has to face.

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

[2] European Commission. An adaptive, highly scalable analytics platform. Contract No: 61970, 2014. URL http://www.asap-fp7.eu/.

[3] Martin Giese, Diego Calvanese, Peter Haase, Ian Horrocks, Yannis Ioannidis, Heralk Kllapi, Manolis Koubarakis, Maurizio Lenzerini, Ralf Möller, Mariano Rodriguez-Muro, Özgür Özcep, Riccardo Rosati, Rudolf Schlatte, Michael Schmidt, Ahmet Soylu, and Arild Waaler. Scalable end-user access to big data. In Rajendra Akerkar, editor, *Big Data Computing*. CRC Press, 2013.

[4] Georgiana Copil, Daniel Moldovan, Hong Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. 8274:429–436, 2013. doi: 10.1007/978-3-642-45005-1_31. URL http://dx.doi.org/10.1007/978-3-642-45005-1_31.

[5] Akshat Verma, Gautam Kumar, and Ricardo Koller. The cost of reconfiguration in a cloud. pages 11–16, 2010. doi: 10.1145/1891719.1891721. URL http://doi.acm.org/10.1145/1891719.1891721.

[6] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. pages 7:1–7:13, 2012. doi: 10.1145/2391229.2391236. URL http://doi.acm.org/10.1145/2391229.2391236.

[7] Y.O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. pages 91–98, July 2010. doi: 10.1109/CLOUD.2010.66.

[8] Krisantus Sembiring and Andreas Beyer. Dynamic resource allocation for cloud-based media processing. pages 49–54, 2013. doi: 10.1145/2460782.2460791. URL http://doi.acm.org/10.1145/2460782.2460791.

[9] Yusen Li, Xueyan Tang, and Wentong Cai. On dynamic bin packing for resource allocation in the cloud. pages 2–11, 2014. doi: 10.1145/2612669.2612675. URL http://doi.acm.org/10.1145/2612669.2612675.

[10] Tai-Won Um, Hyunwoo Lee, Won Ryu, and Jun Kyun Choi. Dynamic resource allocation and scheduling for cloud-based virtual content delivery networks. *ETRI Journal*, 36(2):197–205, April 2014. doi: 10.4218/etrij.14.2113.0085. URL http://etrij.etri.re.kr/etrij/journal/article/article.do?volume=36&issue=2&page=197.

[11] Xiaoming Nan, Yifeng He, and Ling Guan. Towards dynamic resource optimization for cloud-based free viewpoint video service. pages 3498–3502, Oct 2014. doi: 10.1109/ICIP.2014.7025710.

[12] Ioannis Konstantinou, Verena Kantere, Dimitrios Tsoumakos, and Nectarios Koziris. COCCUS: self-configured cost-based query services in the cloud. pages 1041–1044, 2013. doi: 10.1145/2463676.2465233. URL http://doi.acm.org/10.1145/2463676.2465233.

[13] Herald Kllapi, Panos Sakkos, Alex Delis, Dimitrios Gunopulos, and Yannis E. Ioannidis. Elastic processing of analytical query workloads on iaas clouds. *CoRR*, abs/1501.01070, 2015. URL http://arxiv.org/abs/1501.01070.

[14] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The rise of raas: The resource-as-a-service cloud. *Commun. ACM*, 57(7):76–84, July 2014. ISSN 0001-0782. doi: 10.1145/2627422. URL http://doi.acm.org/10.1145/2627422.

[15] Marcus Felson and Joe L. Spaeth. Community structure and collaborative consumption: A routine activity approach. *American Behavioral Scientist*, 21(4):614–624, 1978. doi: 10.1177/000276427802100411. URL http://abs.sagepub.com/content/21/4/614.short.

[16] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini X. Economic models for allocating resources in computer systems. pages 156–183, 1996.

[17] Barto and R. S. Sutton. Goal seeking components for adaptive intelligence: An initial assessment. Technical Report AFWAL-TR-81-1070, Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH, 1981.

[18] Christopher J. C. H. Watkins. Reinforcement learning — some history. website of the Department of Computer Science in Royal Holloway, University of London, 2010. URL http://www.cs.rhul.ac.uk/home/chrisw.

[19] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[20] IP Pavlov. *Conditioned Reflexesand Psychiatry*. International Publishers, NY, 1941.

[21] Kazimierz Zieliński. Jerzy konorski on brain associations. *Acta neurobiologiae experimentalis*, 66(1):75—84; discussion 85—90, 95—7, 2006. ISSN 0065-1400. URL http://europepmc.org/abstract/MED/16617679.

[22] Charles B Ferster and 1904-1990 Skinner, B. F. (Burrhus Frederic). *Schedules of reinforcement*. Englewood Cliffs, N. J. : Prentice-Hall, Inc, 1957. ISBN 0137923090. "Most of the work ... was carried out under contract to the Office of Naval Research under Contracts No. N5ori-07631 and N5ori-07656 with Harvard University ... between September 1, 1949 and June 30, 1955.".

[23] G. A. Parker and Maynard J. Smith. Optimality theory in evolutionary biology. *Nature*, 348(6296):27–33, November 1990. doi: 10.1038/348027a0. URL http://dx.doi.org/10.1038/348027a0.

[24] David Blackwell. Discounted dynamic programming. *Ann. Math. Statist.*, 36(1): 226–235, 02 1965. doi: 10.1214/aoms/1177700285. URL http://dx.doi.org/10.1214/aoms/1177700285.

[25] Ralph E. Strauch. Negative dynamic programming. *Ann. Math. Statist.*, 37(4): 871–890, 08 1966. doi: 10.1214/aoms/1177699369. URL http://dx.doi.org/10.1214/aoms/1177699369.

[26] David Blackwell. On stationary policies. *Journal Of The Royal Statistical Society. Series A (General)*, 133(1):33–37, 1970. URL http://www.jstor.org/discover/10.2307/2343810.

[27] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming: Probability and Mathematical*. Academic Press, Inc., Orlando, FL, USA, 1983. ISBN 0125984200. URL http://dl.acm.org/citation.cfm?id=538843.

[28] Paul A. Samuelson. A note on measurement of utility. *The Review of Economic Studies*, 4(2):155–161, 1937. doi: 10.2307/2967612. URL http://restud.oxfordjournals.org/content/4/2/155.short.

[29] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II*. Athena Scientific, 4th edition, 2012. ISBN 1886529442, 9781886529441.

[30] Dimitri P. Bertsekas. *Abstract Dynamic Programming*. Athena Scientific, 1st edition, 2013. ISBN 1886529426, 9781886529427.

[31] John R. Doyle. Survey of time preference, delay discounting models. *Judgment and Decision Making*, 8(2):116–135, March 2013. URL http://journal.sjdm.org/12/12309/.

[32] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. ISBN 0471619779.

[33] Richard E. Bellman. *Adaptive control processes - A guided tour*. Princeton University Press, Princeton, New Jersey, U.S.A., 1961. URL http://www.jstor.org/discover/10.2307/167876.

[34] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. doi: 10.1007/BF00115009. URL http://dx.doi.org/10.1007/BF00115009.

[35] Abhijit Gosavi. *Simulation-Based Optimization*, volume 55 of *Operations Research/Computer Science Interfaces Series*. Springer US, New York, 2015. ISBN 978-1-4899-7491-4. URL http://link.springer.com/book/10.1007%2F978-1-4899-7491-4.

[36] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3:213–231, March 2003. ISSN 1532-4435. doi: 10.1162/153244303765208377. URL http://dx.doi.org/10.1162/153244303765208377.

[37] Herbert Robbins. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.*, 58(5):527–535, 09 1952. URL http://www.ams.org/journals/bull/1952-58-05/S0002-9904-1952-09620-8.

[38] Prasanta Chandra Mahalanobis. A sample survey of the acreage under jute in bengal. *Sankhya: The Indian Journal of Statistics*, 4(4):33–37, January 1940. URL http://www.jstor.org/discover/10.2307/40383954.

[39] Lester E. Dubins and Savage Leonard J. *How to Gamble If You Must: Inequalities for Stochastic Processes*. McGraw-Hill Book Company, New York, 1965. URL http://dl.acm.org/citation.cfm?id=538843.

[40] Patrick L. Ode11. How to gamble if you must: Inequalities for stochastic processes. *Technometrics*, 8(4):713–713, 1966. doi: 10.1080/00401706.1966.10490419. URL http://www.tandfonline.com/doi/abs/10.1080/00401706.1966.10490419.

[41] W. R. Thompson. On the Likelihood that one Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25(3-4):285–294, 1933. doi: 10.1093/biomet/25.3-4.285. URL http://biomet.oxfordjournals.org/content/25/3-4/285.short.

[42] John Venn. *The Logic of Chance*. Macmillan, London, 3rd edition, 1888.

[43] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003. ISBN 0521592712. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0521592712.

[44] Jerzy Neyman. Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 236:333–380, 1937. URL http://www.jstor.org/stable/91337.

[45] Gordon Antelman. *Elementary Bayesian Statistics*. Edward Elgar Publishing, Cheltenham, UK, August 1997. ISBN 1858985048. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1858985048.

[46] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

[47] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997. ISSN 0022-0000. doi: http://dx.doi.org/10.1006/jcss.1997.1504. URL http://www.sciencedirect.com/science/article/pii/S002200009791504X.

[48] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. pages 322–331, Oct 1995. ISSN 0272-5428. doi: 10.1109/SFCS.1995.492488.

[49] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, January 2003. ISSN 0097-5397. doi: 10.1137/S0097539701398375. URL http://dx.doi.org/10.1137/S0097539701398375.

[50] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985. URL http://www.cs.utexas.edu/~shivaram.

[51] J. C. Gittins. Bandit Processes and Dynamic Allocation Indices. *J. Roy. Stat. Soc. B*, 41(2):148–177, 1979. URL http://www.jstor.org/stable/2985029.

[52] Monica Brezzi and Tze Leung Lai. Incomplete learning from endogenous data in dynamic allocation. *Econometrica*, 68(6):1511–1516, 2000. ISSN 1468-0262. doi: 10.1111/1468-0262.00170. URL http://dx.doi.org/10.1111/1468-0262.00170.

[53] Steven L. Scott. A modern bayesian look at the multi-armed bandit. *Appl. Stoch. Model. Bus. Ind.*, 26(6):639–658, November 2010. ISSN 1524-1904. doi: 10.1002/asmb.874. URL http://dx.doi.org/10.1002/asmb.874.

[54] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2249–2257. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4321-an-empirical-evaluation-of-thompson-sampling.pdf.

[55] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. *CoRR*, abs/1111.1797, 2011. URL http://dblp.uni-trier.de/db/journals/corr/corr1111.html#abs-1111-1797.

[56] Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. Thompson sampling: An asymptotically optimal finite-time analysis. pages 199–213, 2012. doi: 10.1007/978-3-642-34106-9_18. URL http://dx.doi.org/10.1007/978-3-642-34106-9_18.

[57] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On Bayesian Upper Confidence Bounds for Bandit Problems. *Journal of Machine Learning Research - Proceedings Track*, 22:592–600, 2012. URL http://dblp.uni-trier.de/db/journals/jmlr/jmlrp22.html#KaufmannCG12.

[58] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2002. URL http://www.jmlr.org/papers/v3/auer02a.html.

[59] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963. URL http://www.jstor.org/stable/2282952?

[60] Jean-Yves Audibert and Sébastien Bubeck. Minimax policies for adversarial and stochastic bandits. 2009. URL http://www.cs.mcgill.ca/~colt2009/papers/022.pdf#page=1.

[61] Junya Honda and Akimichi Takemura. An asymptotically optimal policy for finite support models in the multiarmed bandit problem. *Machine Learning*, 85(3):361–391, 2011. doi: 10.1007/s10994-011-5257-4. URL http://dx.doi.org/10.1007/s10994-011-5257-4.

[62] Junya Honda and Akimichi Takemura. An asymptotically optimal bandit algorithm for bounded support models. pages 67–79, 2010. URL http://www.colt2010.org/papers/29honda.pdf.

[63] Aurélien Garivier and Olivier Cappé. The KL-UCB algorithm for bounded stochastic bandits and beyond. pages 359–376, 2011. URL http://www.jmlr.org/proceedings/papers/v19/garivier11a/garivier11a.pdf.

[64] Olivier Cappé, Aurélien Garivier, Odalric-Ambrym Maillard, Rémi Munos, and Gilles Stoltz. Kullback-leibler upper confidence bounds for optimal sequential allocation. *Annals of Statistics*, 41(3):1516–1541, Jun. 2013.

[65] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, Christina Boumpouka, Nectarios Koziris, and Spyros Sioutas. TIRAMOLA: elastic nosql provisioning through a cloud management platform. pages 725–728, 2012. doi: 10.1145/2213836.2213943. URL http://doi.acm.org/10.1145/2213836.2213943.

[66] Evie Kassela, Christina Boumpouka, Ioannis Konstantinou, and Nectarios Koziris. Automated workload-aware elasticity of nosql clusters in the cloud. pages 195–200, 2014. doi: 10.1109/BigData.2014.7004232. URL http://dx.doi.org/10.1109/BigData.2014.7004232.

[67] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. Cloud elasticity using probabilistic model checking. *CoRR*, abs/1405.4699, 2014. URL http://arxiv.org/abs/1405.4699.

[68] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7): 881–892, July 2002. ISSN 0162-8828. doi: 10.1109/TPAMI.2002.1017616. URL http://dx.doi.org/10.1109/TPAMI.2002.1017616.

[69] Dimitrios Tsoumakos, Ioannis Konstantinou, Christina Boumpouka, Spyros Sioutas, and Nectarios Koziris. Automated, elastic resource provisioning for nosql clusters using TIRAMOLA. pages 34–41, 2013. doi: 10.1109/CCGrid.2013.45. URL http://doi.ieeecomputersociety.org/10.1109/CCGrid.2013.45.

[70] Abhijit Gosavi. Matlab codes for q-learning (with look-up tables and with neurons), r-smart, and q-value iteration, 2008. http://web.mst.edu/~gosavia/codes/wsccodes.html.

[71] Olivier Cappe, Aurelien Garivier, and Emilie Kaufmann. pymabandits, 2012. http://mloss.org/software/view/415/.