Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Προσαρμοστικός Διαμοιρασμός Χώρου Καταστάσεων Μαρκοβιανών Μοντέλων για Ελαστική Διαχείριση Πόρων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΛΩΛΟΣ Β. ΚΩΝΣΤΑΝΤΙΝΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Προσαρμοστικός Διαμοιρασμός Χώρου Καταστάσεων Μαρκοβιανών Μοντέλων για Ελαστική Διαχείριση Πόρων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΛΩΛΟΣ Β. ΚΩΝΣΤΑΝΤΙΝΟΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Ιανουαρίου 2016.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Επίκουρος Καθηγητής Ι.Π.	Γεώργιος Γκούμας Λέκτορας Ε.Μ.Π.

Αθήνα, Ιανουάριος 2016

Λώλος Β. Κωνσταντίνος

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

Περίληψη

Τα συστήματα υπολογιστικού νέφους (*cloud computing*) έχουν αποτελέσει έναν από τους ταχύτερα αναπτυσσόμενους κλάδους της πληροφορικής κατά τη διάρκεια των τελευταίων ετών. Με την ανάπτυξη τεχνολογιών όπως οι μη-σχεσιακές βάσεις δεδομένων, οι οποίες εξυπηρετούν σήμερα τεράστιους όγκους δεδομένων γνωστών ως *Big Data*, η ανάγκη για ανάπτυξη εργαλείων τα οποία να ελέγχουν και να συντονίζουν αυτά τα συστήματα είναι σημαντικότερη από ποτέ. Μία από τις μεγαλύτερες προκλήσεις σε αυτό το πεδίο, είναι η ανάπτυξη μεθόδων μέσω των οποίων να γίνεται δυναμική κατανομή πόρων σε αυτές τις εφαρμογές, μια ιδέα γνωστή ως *ελαστικότητα* (*elasticity*).

Εφόσον η ελαστικότητα είναι μια μορφή προβλήματος λήψης αποφάσεων, για τη λύση του στο παρελθόν έχει προταθεί η χρήση *Μαρκοβιανών Διαδικασιών Αποφάσεων* (*Markov Decision Processes*) και *Q-Learning* για τη μοντελοποίηση αυτών των συστημάτων. Όμως, το πλήθος των παραμέτρων οι οποίες επηρεάζουν τη συμπεριφορά ενός τέτοιου συστήματος είναι υπερβολικά μεγάλος, με αποτέλεσμα αυτές οι παραδοσιακές μέθοδοι να είναι ανεπαρκείς, αφού ακόμα και αν γίνει διακριτοποίηση των συνεχών μεταβλητών το πλήθος των καταστάσεων που θα απαιτούνταν για να αναπαραστήσουν όλους τους δυνατούς συνδυασμούς τους αυξάνει εκθετικά με το πλήθος των παραμέτρων.

Στα πλαίσια αυτής της εργασίας, προτείνουμε την χρήση τροποποιήσεων στα παραδοσιακά αυτά μοντέλα ενισχυτικής εκμάθησης, οι οποίες πραγματοποιούν δυναμικό διαμοιρασμό του χώρου καταστάσεων χρησιμοποιώντας Δέντρα Αποφάσεων. Υλοποιούμε και πειραματιζόμαστε με διαφορετικές υλοποιήσεις τέτοιων αλγορίθμων σε σενάρια προσομοίωσης εμπνευσμένα από το χώρο της διαχείρισης πόρων σε υπολογιστικά νέφη, και διαπιστώνουμε ότι οι λύσεις αυτές επιτυγχάνουν καλύτερες επιδόσεις από παραδοσιακές λύσεις σε τέτοιου είδους προβλήματα. Τέλος, δοκιμάζουμε την προτεινόμενη λύση μας σε ένα πραγματικό *HBase cluster* με τη χρήση του *TIRAMOLA*, ενός συστήματος διαχείρισης μη-σχεσιακών βάσεων δεδομένων.

Λέξεις κλειδιά

Ελαστικότητα, Διαχείριση Πόρων, Υπολογιστικό Νέφος, Μαρκοβιανές Διαδικασίες Αποφάσεων, Δέντρα Αποφάσεων, HBase, NoSQL, TIRAMOLA

Abstract

Cloud computing has been one of the fastest evolving industries over the last decade. With the introduction of *Big Data* and technologies such as distributed non-relational databases, the need for tools that can control and orchestrate those technologies is important as ever. One of the biggest challenges in the field, is developing methods of dynamically allocating resources for these applications, a concept known as *elasticity*.

Since in its core elasticity is a decision making problem, *Markov Decision Processes* and *Q-Learning* have been proposed in the past as methods of modeling those systems and making optimal decisions. However, the number of parameters that affect the behavior of these systems is exceedingly large, making traditional methods inadequate to model their full complexity, since even if the parameters are discretized the required number of states needed to model them grows exponentially with their number.

In this work, we propose using modifications of traditional reinforcement learning algorithms that partition the state space dynamically using *Decision Trees*. We implement and experiment with such algorithms in simulation scenarios inspired from the field of cloud computing, and find that they can outperform traditional solutions in these types of scenarios. Finally, we proceed to test our solution in a real *HBase* cluster running on top of an OpenStack IaaS provider with the help of *TIRAMOLA*, an open-source, cloud-enabled framework for the management of NoSQL clusters.

Key words

Elasticity, Resource Management, Cloud Computing, Markov Decision Process, Decision Tree, HBase, NoSQL, TIRAMOLA

Ευχαριστίες

Με την εκπόνηση της παρούσης εργασίας ολοκληρώνεται ο κύκλος σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου, ο οποίος αποτελεί το δεύτερο προπτυχιακό κύκλο σπουδών μου στο συγκεκριμένο ίδρυμα.

Θα ήθελα αρχικά να ευχαριστήσω τον Καθηγητή κ. Νεκτάριο Κοζύρη για τη δυνατότητα που μου έδωσε να ασχοληθώ με το σύγχρονο και ενδιαφέρον θέμα της παρούσης εργασίας. Επίσης, θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή Ιωάννη Κωνσταντίνου για τη συνεχή παρακολούθηση και το χρόνο που αφιέρωσε για να διευκολύνει την ολοκλήρωση της εργασίας, αλλά και το ανοιχτό μυαλό το οποίο επέδειξε σε κάθε τρελή ιδέα που είχα κατά τη διάρκεια της συνεργασίας μας. Θα ήθελα επιπλέον να ευχαριστήσω ολόκληρο το προσωπικό του Εργαστηρίου Υπολογιστικών Συστημάτων, και ιδιαίτερα τον υποψήφιο διδάκτορα Χρήστο Μαντά, για την άμεση ανταπόκρισή τους οποτεδήποτε κατέστη αναγκαία η βοήθειά τους.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου για την απεριόριστη στήριξη που μου παρείχαν και την υπομονή που επέδειξαν κατά την απόκτηση και των δύο τίτλων σπουδών μου.

Λώλος Β. Κωνσταντίνος,

Αθήνα, 11η Ιανουαρίου 2016

Contents

List of Figures

Chapter 1

Εισαγωγή

1.1 Κίνητρο

Η εκρηκτική ανάπτυξη των συστημάτων υπολογισμού νέφους (*cloud computing*) την τελευταία δεκαετία έχει αλλάξει ριζικά τη δομή και τον τρόπο λειτουργίας των εφαρμογών και των υπηρεσιών. Δεδομένα όπως έγγραφα εργασίας, εικόνα και ήχος, δεδομένα κοινωνικών δικτύων και πολλά ακόμη αποθηκεύονται σε υπηρεσίες νέφους και γίνονται διαθέσιμα παγκοσμίως μέσω υπηρεσιών διαδικτύου. Ο όγκος αυτών των δεδομένων μετριέται σε τρισεκατομμύρια Gigabytes (ή *Zetabytes*). Η ανάγκη ανάγκη για αποθήκευση και επεξεργασία αυτού του όγκου δεδομένων προκάλεσε τη δημιουργία μιας σειράς από νέες τεχνολογίες και αρχιτεκτονικές. Οι παραδοσιακές σχεσιακές βάσεις δεδομένων λόγω της αρχιτεκτονικής τους δεν ήταν δυνατόν να φιλοξενήσουν αυτό τον όγκο των δεδομένων, με αποτέλεσμα να δημιουργηθεί μια σειρά από νέες, μη σχεσιακές βάσεις για να εκπληρώσουν αυτό το σκοπό. Οι βάσεις δεδομένων αυτές είναι σχεδιασμένες ώστε να τρέχουν σε μεγάλα κατανεμημένα συστήματα, και όχι απλά διαχειρίζονται το διαμοιρασμό των δεδομένων και το συντονισμό των διαφορετικών υπολογιστών στους οποίους τρέχουν, αλλά έχουν τη δυνατότητα να χειριστούν και αστοχίες στο υλικό των υπολογιστών, κάτι που σε αυτή την κλίμακα είναι αναπόφευκτο.

Έχοντας τη δυνατότητα να εκτελεστούν σε χιλιάδες υπολογιστές, τα συστήματα αυτά συχνά τρέχουν σε εικονικά περιβάλλοντα, τα οποία παρέχονται από κάποιον πάροχο υπηρεσιών υποδομής (*Infrastructure as a Service*). Αυτοί οι πάροχοι έχουν τη δυνατότητα να υποστηρίξουν τη λειτουργία χιλιάδων εικονικών μηχανών, και συχνά μπορούν να αυξήσουν ή να μειώσουν δυναμικά την ποσότητα των πόρων που παρέχουν σε κάθε χρήστη, μια ιδέα γνωστή ως ελαστικότητα (*elasticity*). Παρόλα αυτά, στις περισσότερες περιπτώσεις, οι μέθοδοι μέσω των οποίων επιτυγχάνεται αυτή η ελαστικότητα είναι απλοϊκές, και απαιτούν από το χρήστη να ορίσει συγκεκριμένα κριτήρια με βάση τα οποία αυξάνεται ή μειώνεται η παρεχόμενη ποσότητα πόρων.

Πολύ συχνά όμως, αυτές οι μέθοδοι λήψης αποφάσεων αδυνατούν να επιτύχουν καλή απόδοση σε πολύπλοκα και δυναμικά περιβάλλοντα, αφού η απλοϊκή φύση τους δεν τους επιτρέπει να πάρουν στρατηγικές αποφάσεις. Μια πιο συστηματική προσέγγιση στο πρόβλημα μπορεί να γίνει με τη χρήση τεχνικών ενισχυτικής εκμάθησης όπως *Μαρκοβιανών Αλυσίδων Αποφάσεων* (*Markov Decision Processes*) ή *Q-Learning*. Αυτοί οι αλγόριθμοι αποτελούν κλασικές λύσεις σε προβλήματα λήψης αποφάσεων, και παρέχουν εγγυήσεις βελτιστότητας υπό λογικές προϋποθέσεις.

Ακόμα και αυτές οι λύσεις όμως έχουν τους περιορισμούς τους. Σε μια τυπική θεώρηση

ενισχυτικής εκμάθησης, ο κόσμος μπορεί να βρίσκεται σε μια σειρά από δυνατές καταστάσεις (*states*), και σε κάθε τέτοια κατάσταση μια σειρά από δράσεις (*actions*) είναι διαθέσιμες. Μετά την εκτέλεση κάποιας δράσης, ο δράστης επιβραβεύεται με μια βαθμωτή *ενίσχυση* (*reinforcement*), και ο κόσμος μεταβαίνει σε μια καινούρια κατάσταση. Οι αλγόριθμοι είναι βέλτιστοι υπό την έννοια ότι μεγιστοποιούν κάποιο μακροπρόθεσμο μέτρο αυτών των ενισχύσεων. Επιπλέον, αυτή η βελτιστότητα επιτυγχάνεται υπό την προϋπόθεση ότι η συμπεριφορά του κόσμου είναι ίδια κάθε φορά που βρίσκεται σε μια συγκεκριμένη κατάσταση (μια ιδιότητα γνωστή ως ιδιότητα *Markov*). Αυτό σημαίνει ότι οι καταστάσεις που θα επιλεγούν για τη μοντελοποίηση του κόσμου θα πρέπει είναι να επαρκώς λεπτομερείς ώστε να αντικατοπτρίζουν ολόκληρη την πολυπλοκότητα του συστήματος.

Στην περίπτωση της διαχείρισης κατανεμημένων, μη σχεσιακών βάσεων δεδομένων που εκτελούνται σε περιβάλλοντα νέφους όμως, ο αριθμός των παραμέτρων οι οποίες επηρεάζουν τη συμπεριφορά του συστήματος είναι εξαιρετικά μεγάλος (πλήθος και χαρακτηριστικά των μηχανών, δεδομένα απόδοσης πραγματικού χρόνου, χαρακτηριστικά του φορτίου κλπ). Ακόμη και αν οι τιμές των παραμέτρων αυτών διακριτοποιηθούν, ο ορισμός μιας διαφορετικής κατάστασης για κάθε έναν από τους συνδυασμούς διαφορετικών τιμών τους θα οδηγούσε σε εκθετικά μεγάλο αριθμό καταστάσεων. Ένα μοντέλο ενισχυτικής εκμάθησης αυτής της κλίμακας όχι απλά δεν θα μπορούσε να αναπαρασταθεί στη μνήμη, αλλά πολύ περισσότερο θα ήταν αδύνατο να εκπαιδευθεί, αφού το πλήθος των εμπειριών που θα απαιτούνταν θα ήταν επίσης εκθετικά μεγάλο. Το αντικείμενο αυτής της εργασίας συνεπώς, είναι η αναζήτηση μεθόδων οι οποίες μπορούν να ξεπεράσουν αυτή τη δυσκολία, ενώ ταυτόχρονα παρέχουν τα ίδια πλεονεκτήματα με τις παραδοσιακές μεθόδους ενισχυτικής εκμάθησης.

1.2 Σχετικές Εργασίες

Στο άρθρο [Chap91] οι συγγραφείς προτείνουν μια τροποποίηση του *Q-learning* που έχει δυνατότητα γενίκευσης επί της εισόδου. Το πρόβλημα το οποίο προσπαθούν να επιλύσουν είναι ο έλεγχος ενός χαρακτήρα σε ένα δισδιάστατο ηλεκτρονικό παιχνίδι, όπου η είσοδος η οποία παίζει το ρόλο της κατάστασης του κόσμου είναι μια σειρά από bits που αντιστοιχούν στην αναπαράσταση του παιχνιδιού στην οθόνη. Εφόσον το μήκος αυτής της συμβολοσειράς είναι μεγαλύτερο από μερικές εκατοντάδες bits, το μέγεθος του χώρου καταστάσεων που θα απαιτείτο θα ήταν μεγαλύτερο από 2^{100} καταστάσεις, και συνεπώς είναι αναγκαίος κάποιος τρόπος γενίκευσης επί της εισόδου. Ο προτεινόμενος αλγόριθμος διαμοιράζει σταδιακά το χώρο καταστάσεων με βάση τις τιμές μεμονωμένων bits της εισόδου. Ο έλεγχος για το ποιο bit πρέπει να χρησιμοποιηθεί κάθε φορά για το διαμοιρασμό αυτό γίνεται με τη χρήση ενός *t*-στατιστικού τεστ.

Στο άρθρο [Pyea01], προτείνεται ένας αλγόριθμος βασισμένος στο *Q-learning* ο οποίος χρησιμοποιεί ένα δέντρο αποφάσεων για το δυναμικό διαμοιρασμό ενός συνεχούς χώρου καταστάσεων. Το κίνητρο είναι η κατασκευή ελεγκτών για δύο ρομποτικές εφαρμογές όπου ο χώρος καταστάσεων είναι υπερβολικά μεγάλος για να διαμοιραστεί με κλασικές τεχνικές διακριτοποίησης. Ο αλγόριθμος κατασκευάζει ένα δέντρο αποφάσεων βασιζόμενος σε τιμές παραμέτρων της εισόδου, και διατηρεί ένα μοντέλο *Q-learning* στα φύλλα του δέντρου. Διαφορετικά κριτήρια ελέγχονται για το σπάσιμο των κόμβων, και η απόδοση του αλγορίθ-

μου συγκρίνεται με παραδοσιακές μεθόδους ενισχυτικής εκμάθησης και νευρωνικά δίκτυα.

Στο άρθρο [Uthe98], προτείνεται ένας αλγόριθμος πλήρους μοντέλου βασιζόμενος σε δέντρα αποφάσεων, ο οποίος ονομάζεται *Continuous U Tree*. Ο αλγόριθμος χωρίζεται σε δύο φάσεις. Κατά τη φάση *συλλογής δεδομένων*, οι καταστάσεις του μοντέλου παραμένουν σταθερές, αλλά εμπειρίες συλλέγονται και αποθηκεύονται για μελλοντική χρήση. Κατά τη φάση *επεξεργασίας*, οι αποθηκευμένες πληροφορίες χρησιμοποιούνται για τον καθορισμό των καταστάσεων του μοντέλου οι οποίες θα πρέπει να διαχωριστούν σε νέες. Όταν οι νέες καταστάσεις του μοντέλου έχουν αποφασισθεί, οι ίδιες πληροφορίες χρησιμοποιούνται για τον υπολογισμό των συναρτήσεων μετάβασης και αμοιβής του μοντέλου, και οι αξίες των καταστάσεων και των δράσεων υπολογίζονται εκ νέου. Ο αλγόριθμος συνεχώς εναλλάσσεται μεταξύ των δύο φάσεων, επεκτείνοντας περιοδικά το δέντρο αποφάσεων και υπολογίζοντας εκ νέου τις αξίες των καταστάσεων του μοντέλου.

Στο άρθρο [Tsou13], οι συγγραφείς παρουσιάζουν τον TIRAMOLA, ένα framework ανοιχτού κώδικα το οποίο εκτελείται σε υπολογιστικό περιβάλλον νέφους και τροποποιεί δυναμικά το μέγεθος μιας συστάδας υπολογιστών πάνω στους οποίους τρέχει μια μη σχεσιακή, κατανεμημένη βάση δεδομένων σύμφωνα με πολιτικές ορισμένες από το χρήστη. Το σύστημα αποφασίζει το βέλτιστο μέγεθος της συστάδας και προβαίνει αυτόματα στις κατάλληλες ενέργειες για να το τροποποιήσει δεσμεύοντας ή απελευθερώνοντας εικονικές μηχανές από τον πάροχο υποδομής, και ενσωματώνοντάς τες στην υπόλοιπη συστάδα. Η συστάδα μοντελοποιείται σαν μια Διαδικασία Λήψης Αποφάσεων Markov, όπου οι καταστάσεις αναπαριστούν διαφορετικά μεγέθη της συστάδας και οι δράσεις ενέργειες οι οποίες τροποποιούν το μέγεθός της. Για την απομόνωση των πιο σχετικών καταγεγραμμένων εμπειριών από τις οποίες να μπορεί να προβλεφθεί η κατάσταση στην οποία θα μεταβεί η συστάδα μετά την εκτέλεση μιας ενέργειας, χρησιμοποιείται ο αλγόριθμος συσταδοποίησης *k-means*, και η αναμενόμενη ανταμοιβή υπολογίζεται με βάση το κέντρο βάρους της προκύπτουσας περιοχής.

Στο άρθρο [Kass14], οι συγγραφείς επεκτείνουν τον TIRAMOLA ώστε να έχει τη δυνατότητα να αναγνωρίζει διαφορετικούς τύπους ερωτημάτων. Πραγματοποιείται μια ανάλυση της επίπτωσης των διαφόρων τύπων ερωτημάτων στην απόδοση μιας κατανεμημένης, μη σχεσιακής βάσης δεδομένων διαφόρων μεγεθών, και η αποκτηθείσα γνώση χρησιμοποιείται από τον TIRAMOLA ώστε να πραγματοποιήσει πιο ακριβείς αποφάσεις τροποποίησης του μεγέθους της συστάδας υπολογιστών που ελέγχει.

Στο άρθρο [Nask], παρουσιάζεται μια προσέγγιση εφαρμογής ελαστικότητας μέσω της δυναμικής πραγματοποίησης ενός ποσοτικοποιημένου ελέγχου μιας Διαδικασίας Αποφάσεων Markov, χρησιμοποιώντας πιθανοτικό έλεγχο μοντέλων. Μελετώνται μια σειρά από μοντέλα που αποσκοπούν στην υλοποίηση ελαστικότητας χρησιμοποιώντας μετρήσεις από μια πραγματική μη σχεσιακή βάση δεδομένων υπό συνεχώς μεταβαλλόμενο εξωτερικό φορτίο. Η συστάδα υπολογιστών μοντελοποιείται σαν μια Διαδικασία Αποφάσεων Markov, με πολλαπλές καταστάσεις για κάθε μέγεθος της συστάδας, και μη-ντετερμινιστικές μεταβάσεις μεταξύ των καταστάσεων. Οι τιμές μιας μετρικής καθορίζουν τον τρόπο ομαδοποίησης σε καταστάσεις, και οι πιθανότητες μετάβασης είναι ανάλογες με τον αριθμό των σημείων ανά κατάσταση.

Στο άρθρο [Maso15], οι συγγραφείς προτείνουν μια μέθοδο ομαδοποίησης εικονικών μηχανών. Ο γενικός σκοπός της μεθόδου είναι να χειριστεί φυσικούς κόμβους ώστε να

αποφύγει την υπερφόρτωση ή την αδράνεια της υποδομής, και να βελτιστοποιήσει την τοποθέτηση των εικονικών μηχανών σε αυτούς. Ο αλγόριθμος *Fuzzy Q-learning* χρησιμοποιείται σαν αντικατάσταση του *Q-learning*, για να περιορίσει το χώρο καταστάσεων και να επιταχύνει την εκμάθηση. Η εκμάθηση πραγματοποιείται με μια συνεργατική προσέγγιση, όπου πολλαπλοί δράστες μοιράζονται τη γνώση τους μέσω ενός σχήματος επικοινωνίας *blackboard*. Οι καταστάσεις καθορίζονται από ζεύγη μετρήσεων που αντιστοιχούν στη χρησιμοποίηση της κεντρικής μονάδας επεξεργασίας και του αριθμού των εικονικών μηχανών, και οι δράσεις αποφασίζουν τις οριακές τιμές μιας πολιτικής μετακίνησης εικονικών μηχανών, καθώς και το κριτήριο επιλογής τους. Η συνάρτηση ανταμοιβής υπολογίζεται από την κατανάλωση ενέργειας σε συνδυασμό με τον αριθμό των παραβιάσεων της SLA (*Service Level Agreement*).

1.3 Προτεινόμενη Λύση

Η εφαρμογή στην οποία στοχεύει η συγκεκριμένη εργασία είναι η κατασκευή ενός συστήματος το οποίο να παίρνει με δυναμικό τρόπο αποφάσεις αλλαγής των χαρακτηριστικών και του μεγέθους μιας συστάδας υπολογιστών που εκτελούνται στο εικονικό περιβάλλον ενός παρόχου υποδομής νέφους. Αυτό το πρόβλημα έχει δύο σημαντικές προκλήσεις που θα πρέπει να ληφθούν υπόψιν:

- Υπάρχει ένας μεγάλος αριθμός παραμέτρων που επηρεάζουν τη συμπεριφορά του συστήματος, πολλές από τις οποίες δεν είναι διακριτές. Με άλλα λόγια, η λύση θα πρέπει να έχει τη δυνατότητα να γενικεύσει επί ενός πολυδιάστατου και συνεχούς χώρου καταστάσεων.

- Το χρονικό διάστημα που μεσολαβεί μεταξύ δύο διαδοχικών αποφάσεων είναι της τάξεως των λίγων λεπτών. Αυτό έχει δύο σημαντικές συνέπειες. Πρώτον, η λήψη δεδομένων γίνεται με αργό τρόπο, πράγμα που σημαίνει ότι ο αλγόριθμος θα πρέπει να κάνει όσο γίνεται καλύτερη χρήση των δεδομένων που έχει στη διάθεσή του. Δεύτερον, υπάρχει ένα μεγάλο χρονικό διάστημα διαθέσιμο για λήψη αποφάσεων, που σημαίνει ότι είναι εφικτή η χρήση υπολογιστικά ακριβότερων λύσεων για το σκοπό αυτό.

Για την αντιμετώπιση των συγκεκριμένων προκλήσεων, στην παράγραφο 1.7 προτείνουμε μια λύση με τα παρακάτω χαρακτηριστικά:

- Υιοθετούμε μια προσέγγιση βασισμένη σε πλήρη μοντέλα Markov αντί για μια προσέγγιση βασισμένη στο *Q-Learning*. Έχοντας στη διάθεσή μας χρόνο της τάξης των μερικών λεπτών για λήψη αποφάσεων, είναι ρεαλιστικό να διατηρούμε ένα πλήρες μοντέλο Διαδικασίας Αποφάσεων Markov για το σύστημά μας, αποθηκεύοντας δεδομένα για τις ανταμοιβές και τις μεταβάσεις του συστήματος, και να χρησιμοποιούμε αλγορίθμους όπως *Prioritized Sweeping* και *Value Iteration* για την ενημέρωση των τιμών σε κάθε βήμα. Το γεγονός ότι οι εμπειρίες συλλέγονται με αργό ρυθμό περιορίζει το μέγεθος του μοντέλου και κάνει δυνατή την εκτέλεση πολύπλοκων υπολογισμών.

- Επιλέγουμε τη χρήση ενός αλγορίθμου βασιζόμενου σε δέντρα αποφάσεων για την πραγματοποίηση δυναμικού διαμοιρασμού του χώρου καταστάσεων. Οι αλγόριθμοι

που βασίζονται σε δέντρα αποφάσεων, εν αντιθέσει με τους παραδοσιακούς αλγορίθμους ενισχυτικής εκμάθησης, δεν περιορίζονται από ένα σταθερό αριθμό καταστάσεων ο οποίος πρέπει να καθοριστεί στην αρχή της εκτέλεσης, αλλά μπορούν δυναμικά να δημιουργούν νέες καταστάσεις όταν αυτό χρειάζεται, όπως επιτάσσει η συμπεριφορά του συστήματος. Αυτό όχι απλά τους επιτρέπει να λειτουργήσουν σε ένα πολυδιάστατο και συνεχή χώρο καταστάσεων, αλλά επίσης να προσαρμόζουν το μέγεθός τους ανάλογα με την ποσότητα των δεδομένων εκπαίδευσης που διαθέτουν. Εφόσον αυτά τα μοντέλα ξεκινούν με έναν μικρό αριθμό καταστάσεων, είναι ευκολότερο να εκπαιδευθούν με μια μικρή ποσότητα δεδομένων. Καθώς όμως περισσότερα δεδομένα γίνονται διαθέσιμα, ο αριθμός των καταστάσεων αυξάνεται, και μαζί του αυξάνεται και η ακρίβεια του μοντέλου.

- Είναι απαραίτητη προϋπόθεση για τη γρήγορη εκπαίδευση του μοντέλου το να μη σπαταλά ποτέ δεδομένα εκπαίδευσης. Για το σκοπό αυτό, όταν μια κατάσταση αντικαθίσταται από δύο νέες, τα δεδομένα που είχαν χρησιμοποιηθεί για την εκπαίδευση της παλιάς κατάστασης χρησιμοποιούνται ξανά για την εκπαίδευση της νέας. Με αυτό τον τρόπο, παρόλο που δημιουργούνται νέες καταστάσεις κατά τη διάρκεια της λειτουργίας του συστήματος, αυτές οι νέες καταστάσεις εισέρχονται στο σύστημα ήδη εκπαιδευμένες και οι αξίες τους αντικατοπτρίζουν όλες τις εμπειρίες που έχουν συλλεγεί από την αρχή της λειτουργίας του συστήματος. Για να επιτευχθεί αυτό, υλοποιούμε έναν αλγόριθμο ο οποίος πραγματοποιεί διαχωρισμό και επανεκπαίδευση καταστάσεων με τοπικό τρόπο, χωρίς να χρειάζεται η καθολική επανεκπαίδευση του μοντέλου. Αυτό μας επιτρέπει να δημιουργούμε νέες καταστάσεις αποδοτικά με κάθε νέα εμπειρία που συλλέγεται, πράγμα που όχι μόνο είναι υπολογιστικά γρηγορότερο, αλλά βελτιώνει και τη δυνατότητα λήψης αποφάσεων του αλγορίθμου.

1.4 Οργάνωση Κειμένου

Στα κεφάλαια 1.1 έως 1.3 ορίζουμε το πρόβλημα στο οποίο εστιάζει αυτή η εργασία, αναφέρουμε συνοπτικά σχετικές εργασίες πάνω σε παρεμφερή προβλήματα και περιγράφουμε συνοπτικά την προτεινόμενη λύση μας.

Στο κεφάλαιο 1.5 παρουσιάζουμε τα εργαλεία και τις τεχνολογίες που χρησιμοποιήσαμε κατά τη διάρκεια των πειραμάτων μας. Συγκεκριμένα, περιγράφεται το *HDFS* (*Hadoop Distributed File System*), ένα κατανεμημένο σύστημα αρχείων που δημιουργήθηκε από την Apache. Πάνω από το HDFS τρέχει η *HBase*, μια κατανεμημένη και μη σχεσιακή βάση δεδομένων. Ο πάροχος υποδομής επί του οποίου έτρεχαν οι εικονικές μηχανές μας χρησιμοποιούσε το *OpenStack*, ενώ οι μετρικές που ήταν απαραίτητες για τη λήψη αποφάσεων συλλέγονταν με τη βοήθεια του *Ganglia*. Όλα τα προηγούμενα ελέγχονταν και συντονίζονταν από τον TIRAMOLA, ένα framework ανοιχτού κώδικα το οποίο πραγματοποιεί δυναμική προσαρμογή των χαρακτηριστικών μιας συστάδας υπολογιστών επί της οποίας τρέχει μια κατανεμημένη βάση δεδομένων.

Στο κεφάλαιο 1.6, περιγράφουμε τις Διαδικασίες Αποφάσεων Markov, οι οποίες αποτελούν τη θεωρητική βάση της προσέγγισής μας. Αναλύουμε τη θεωρία πίσω από αυτές, και περιγράφουμε τα πλεονεκτήματα και μειονεκτήματα γνωστών προσεγγίσεων πάνω σε αυτές.

Στο κεφάλαιο 1.7 εστιάζουμε στα Δέντρα Αποφάσεων, και περιγράφουμε σε λεπτομέρεια την υλοποίησή μας, ένα πλήρες μοντέλο Διαδικασίας Αποφάσεων Markov το οποίο χρησιμοποιεί ένα Δέντρο Αποφάσεων για να επιτύχει γενίκευση επί της εισόδου του.

Στο κεφάλαιο 1.8 παρουσιάζουμε αποτελέσματα από μια σειρά από προσομοιώσεις. Στόχος αυτών των προσομοιώσεων είναι να διερευνήσουν τον τρόπο συμπεριφοράς των αλγορίθμων που συζητήθηκαν στο κεφάλαιο 1.6, και να αξιολογήσουν την απόδοση της πρότασής μας σε σύγκριση με παραδοσιακές προτάσεις στο χώρο της ενισχυτικής εκμάθησης.

Στο κεφάλαιο 1.9 παρουσιάζουμε πειραματικά αποτελέσματα από τη χρήση την πρότασής μας σε μια πραγματική συστάδα υπολογιστών που τρέχει σε έναν πάροχο υποδομής OpenStack, επί της οποίας εκτελείται μια κατανεμημένη βάση δεδομένων HBase.

Τέλος, στο κεφάλαιο 1.10 συνοψίζουμε και αξιολογούμε τα αποτελέσματα που προέκυψαν από τη διεξαγωγή της παρούσης εργασίας.

1.5 Τεχνολογικό Υπόβαθρο

1.5.1 HDFS

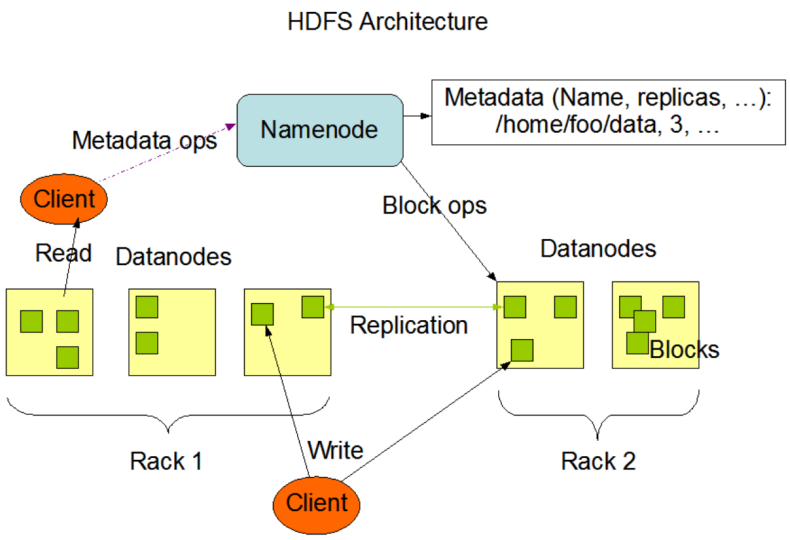


Figure 1.1: Η αρχιτεκτονική του HDFS

Το *HDFS* (Hadoop Distributed File System) είναι ένα κατανεμημένο σύστημα αρχείων το οποίο είναι σχεδιασμένο ώστε να είναι ανθεκτικό σε αστοχίες του υλικού πάνω στο οποίο εκτελείται. Αποτελεί μια υλοποίηση ανοιχτού κώδικα του *GFS* (Google File System), και αποκλίνει από το πρότυπο POSIX ώστε να παρέχει αποδοτικότερα τις υπηρεσίες του σε εφαρμογές που απαιτούν υψηλής ταχύτητας πρόσβαση σε μεγάλους όγκους διαδοχικά αποθηκευμένων δεδομένων.

Χρησιμοποιεί αρχιτεκτονική slave/master, όπου το ρόλο του master αναλαμβάνει ο Namenode, ο οποίος είναι υπεύθυνος για την οργάνωση του συστήματος αρχείων και την παροχή πρόσβασης σε αυτό στους πελάτες. Ο χώρος ονομάτων είναι ιεραρχικά οργανωμένος, και οι πελάτες του έχουν τη δυνατότητα δημιουργίας και χειρισμού καταλόγων και αρχείων με τρόπο ανάλογο αυτού των παραδοσιακών συστημάτων αρχείων. Το ρόλο των slaves

αναλαμβάνουν οι Datanodes, οι οποίοι φιλοξενούν τα δεδομένα που είναι αποθηκευμένα στο σύστημα, και εξυπηρετούν αιτήματα εγγραφής και ανάγνωσης από τους πελάτες. Ταυτόχρονα, κατόπιν εντολής του Namenode, δημιουργούν, καταστρέφουν, ή αντιγράφουν blocks, τα οποία είναι τα τμήματα στα οποία είναι αποθηκευμένα τα αρχεία που φιλοξενούνται στο σύστημα.

Ο τρόπος με τον οποίο ο Namenode ενημερώνεται για την τρέχουσα κατάσταση των blocks είναι με την αποστολή από τους Datanodes προς αυτόν ανά προκαθορισμένα χρονικά διαστήματα μηνυμάτων τα οποία ονομάζονται *Heartbeats*. Αν κάποιος Datanode αποτύχει να αποστείλει ένα τέτοιο μήνυμα προς το Namenode, ο τελευταίος υποθέτει ότι ο Datanode έχει τεθεί εκτός λειτουργίας, σταματά την προώθηση προς αυτός αιτημάτων εξυπηρέτησης και επιχειρεί να ανατοποθετήσει σε νέους κόμβους τα blocks τα οποία ήταν αποθηκευμένα σε αυτόν.

1.5.2 HBase

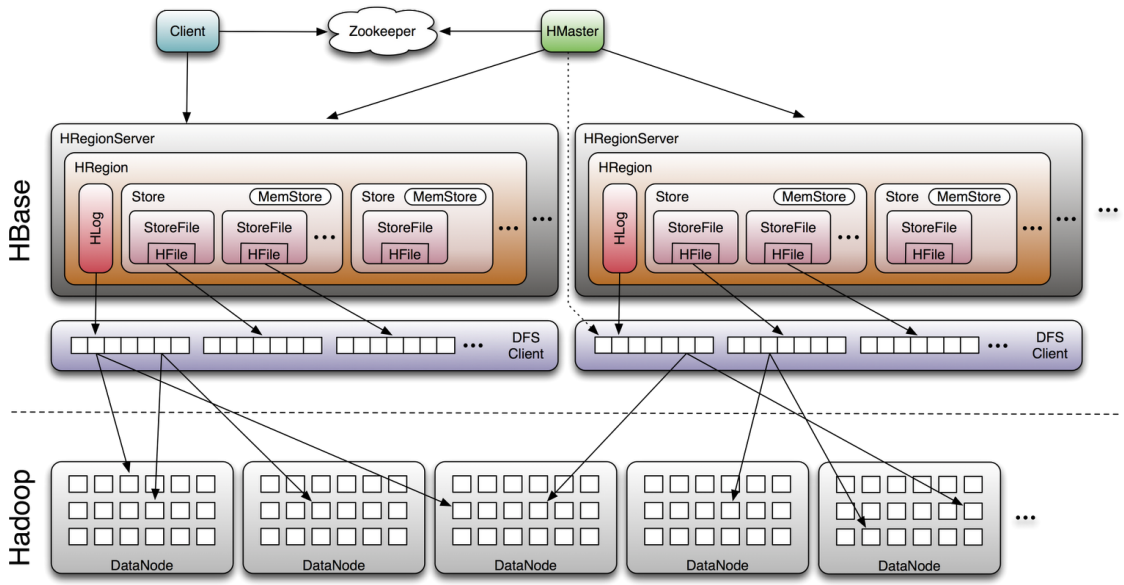


Figure 1.2: Η αρχιτεκτονική της HBase

Η HBase είναι μια κατανεμημένη βάση δεδομένων ανοιχτού κώδικα, η οποία εκτελείται πάνω από το HDFS και εξειδικεύεται στην αποθήκευση αραιών δεδομένων. Ο σχεδιασμός ακολουθεί το *BigTable* της Google, και το μοντέλο των δεδομένων της διαφοροποιείται από τις παραδοσιακές σχεσιακές βάσεις δεδομένων.

Τα δεδομένα της HBase αποθηκεύονται σε μια σειρά από πίνακες (*tables*). Κάθε πίνακας αποτελείται από σειρές (*rows*), σε κάθε μία από τις οποίες αντιστοιχεί ένα μοναδικό κλειδί. Οι σειρές ενός πίνακα είναι ταξινομημένες με βάση τα κλειδιά τους, πράγμα το οποίο επιτρέπει στον προγραμματιστή να ελέγξει τον τρόπο με τον οποίο είναι ταξινομημένα τα δεδομένα. Κάθε σειρά αποτελείται από ένα σύνολο από οικογένειες στηλών (*column families*), οι οποίες είναι ίδιες σε κάθε σειρά του πίνακα και ορίζονται κατά τη δημιουργία του. Οι οικογένειες στηλών αποτελούνται από στήλες (*columns*), οι οποίες είναι όμως δυνατόν να διαφοροποιούνται από σειρά σε σειρά. Ο συνδυασμός του κλειδιού μιας σειράς του πίνακα, μιας οικογένειας

στηλών και μιας στήλης αντιστοιχεί σε ένα μοναδικό κελί (*cell*) του πίνακα, το οποίο φιλοξενεί την τιμή του, που είναι μια σειρά από bits.

Η HBase χρησιμοποιεί αρχιτεκτονική master/slave. Ο *Master Server* φιλοξενεί όλα τα μεταδεδομένα για όλους τους πίνακες της βάσης, και πραγματοποιεί όλες τις αλλαγές στη μορφή τους. Τα δεδομένα είναι διαχωρισμένα σε μια σειρά από περιοχές (*regions*), και κάθε *Region Server* αναλαμβάνει να φιλοξενήσει κάποιες από αυτές. Η κατανομή των περιοχών στους Region Servers ελέγχεται από το Master Node, με κριτήριο την ισοκατανομή του φορτίου σε όλους τους κόμβους του συστήματος. Η αρχιτεκτονική αυτή της HBase της δίνει τη δυνατότητα να φιλοξενήσει πολύ μεγάλους όγκους αραιών δεδομένων, ενώ το γεγονός ότι εκτελείται πάνω από το HDFS της επιτρέπει να παρέχει υψηλή διαθεσιμότητα και ανοχή στα σφάλματα, ενώ ταυτόχρονα διευκολύνει τη συνεργασία της με άλλα εργαλεία της ίδιας οικογένειας όπως το MapReduce.

1.5.3 OpenStack

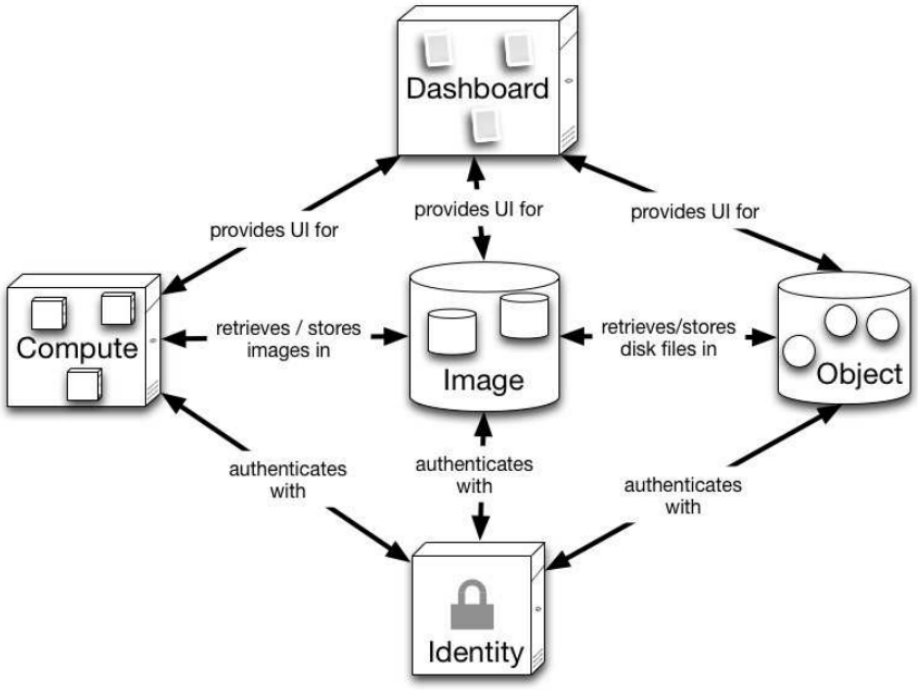


Figure 1.3: Τα τμήματα του OpenStack

Το OpenStack είναι μια πλατφόρμα ανοιχτού κώδικα η οποία παρέχει υπηρεσίες υπολογιστικού νέφους. Έχει τη δυνατότητα να φιλοξενήσει εξαιρετικά μεγάλους αριθμούς εικονικών μηχανών και δεδομένων, και να εκτελείται σε εκατομμύρια φυσικά μηχανήματα, υποστηρίζοντας μια μεγάλη γκάμα από τεχνολογίες εικονικοποίησης.

Το *OpenStack Compute*, επίσης γνωστό ως *Nova*, αναλαμβάνει τη διαχείριση της υποδομής του OpenStack, παρέχοντας ένα interface και ένα API για το χειρισμό μεγάλων δικτύων εικονικών μηχανών. Το *OpenStack Image* αναλαμβάνει την αποθήκευση των εικονικών μηχανών, και παρέχει ένα API μέσω του οποίου μπορούν να πραγματοποιηθούν ερωτήματα για τις εικονικές μηχανές που είναι αποθηκευμένες στα διάφορα συστήματα αποθήκευσης που

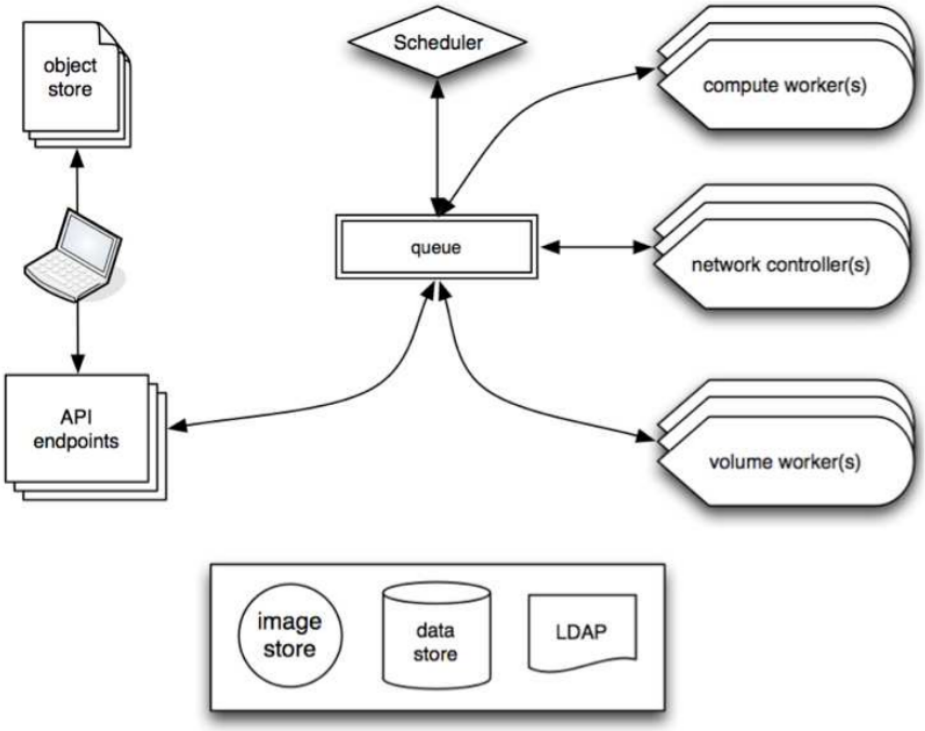


Figure 1.4: Η αρχιτεκτονική του συστήματος Nova

διαχειρίζεται. Τέλος, το *OpenStack Object* είναι ένας αποθηκευτικός χώρος με δυνατότητα φιλοξενίας πολλών Petabytes δεδομένων. Οι υπηρεσίες του OpenStack γίνονται διαθέσιμες μέσω του *OpenStack Dashboard*, που παρέχει ένα γραφικό interface για τους χρήστες και τους διαχειριστές του συστήματος.

1.5.4 Ganglia

Το Ganglia είναι ένα κατανεμημένο σύστημα επίβλεψης υπολογιστικών συστημάτων, το οποίο αναπτύχθηκε από το πανεπιστήμιο του Berkeley. Χρησιμοποιεί ένα κανάλι multi-cast για να διαμοιράζει δεδομένα που αφορούν την κατάσταση μιας συστάδας υπολογιστών, και συνδέει τις διαφορετικές συστάδες που έχει υπό την εποπτεία του μέσω ενός δέντρου συνδέσεων μεταξύ κόμβων-αντιπροσώπων της κάθε συστάδας. Τα δεδομένα αποθηκεύονται σε μορφή XML, ανταλλάσσονται χρησιμοποιώντας το πρωτόκολλο XDR και οπτικοποιούνται μέσω του εργαλείου RRDtool.

Ο Ganglia Monitoring Daemon (gmond) εγκαθίσταται σε κάθε κόμβο της συστοιχίας, συλλέγει μετρικές του συστήματος και τις ανακοινώνει στο κανάλι της συστοιχίας μέσω πακέτων UDP. Αποτελείται από μια σειρά από νήματα, τα περισσότερα εκ των οποίων αναλαμβάνουν να συλλέγουν τις τιμές μιας συγκεκριμένης μετρικής. Τα δεδομένα που μεταδίδονται στο κανάλι αποθηκεύονται από όλους τους δαίμονες gmond της συστοιχίας, ώστε να είναι δυνατή η ανάκτησή τους από οποιονδήποτε από αυτούς. Ο Ganglia Meta Daemon (gmetad) αναλαμβάνει τη διασύνδεση διαφορετικών συστοιχιών υπολογιστών που παρακολουθούνται από το Ganglia, και ταυτόχρονα συλλέγει, αποθηκεύει, και κάνει διαθέσιμες τις μετρικές που έχουν συλλεγεί. Η αποθήκευση και οπτικοποίηση των δεδομένων γίνεται από το εργαλείο RRDtool, το οποίο εξειδικεύεται στην αποθήκευση χρονοσειρών δεδομένων και

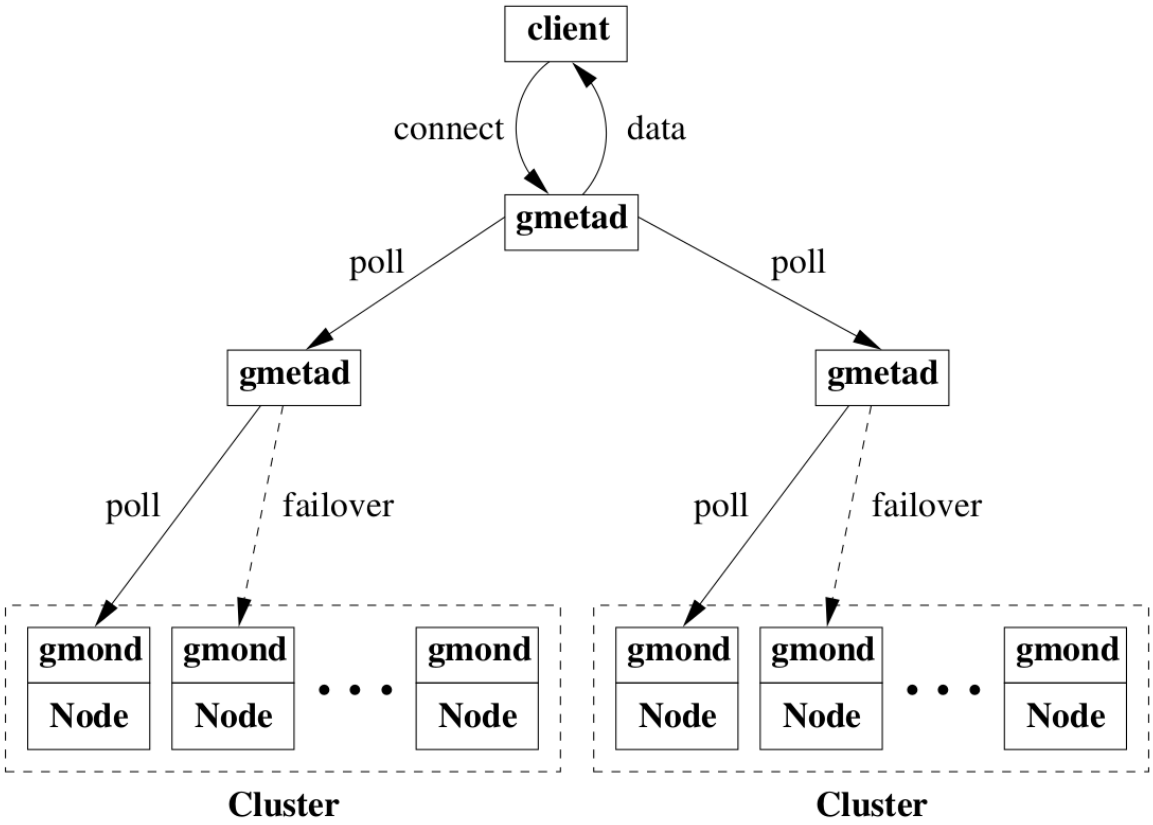


Figure 1.5: Η αρχιτεκτονική του Ganglia

την παρουσίασή τους σε μορφή γραφημάτων.

1.6 Ενισχυτική Εκμάθηση

Σε ένα τυπικό σενάριο ενισχυτικής εκμάθησης ένας δράστης έχει τη δυνατότητα να κινείται σε κάποιο περιβάλλον πραγματοποιώντας δράσεις. Μετά την εκτέλεση κάθε δράσης η κατάσταση στην οποία βρίσκεται ο κόσμος δυνητικά μεταβάλλεται, και το περιβάλλον επιστρέφει στο δράστη μια *ενίσχυση*, η οποία αποτελεί μια αριθμητική τιμή που του γνωστοποιεί το πόσο αποτελεσματική ήταν η δράση που πραγματοποιήθηκε. Ο σκοπός κατά συνέπεια του δράστη είναι να μεγιστοποιήσει κάποιο μακροπρόθεσμο μέτρο των ενισχύσεων που λαμβάνει.

Τυπικά, ένα μοντέλο ενισχυτικής εκμάθησης αποτελείται από τα παρακάτω:

i Ένα σύνολο από καταστάσεις $\mathcal{S}$

ii Ένα σύνολο από δράσεις $\mathcal{A}$

iii Ένα σύνολο από βαθμωτές ενισχύσεις

Το τυπικό μέτρο των ενισχύσεων που προσπαθεί να μεγιστοποιήσει ο δράστης σε ένα σενάριο ενισχυτικής εκμάθησης είναι το απομειωμένο άθροισμα των ενισχύσεων που θα λάβει καθ' όλη τη διάρκεια της ζωής του.

$$E\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] \tag{1.1}$$

Η εισαγωγή ενός συντελεστή εκθετικής απομείωσης στις ενισχύσεις κάνει το προκύπτον άθροισμα να είναι πεπερασμένο και ταυτόχρονα ωθεί το δράστη να συλλέξει τις διαθέσιμες ενισχύσεις στον ελάχιστο δυνατό χρόνο, πράγμα που είναι συνήθως επιθυμητό.

1.6.1 Διαδικασίες Αποφάσεων Markov

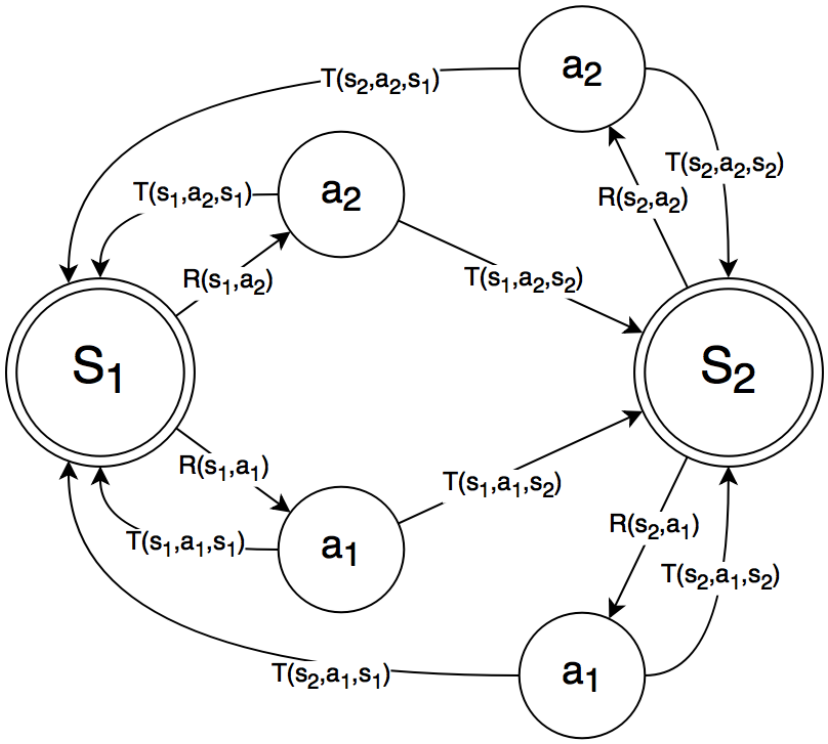


Figure 1.6: Γραφική αναπαράσταση μιας απλής Διαδικασίας Αποφάσεων Markov με δύο καταστάσεις και δύο δράσεις διαθέσιμες σε κάθε κατάσταση

Μια Διαδικασία Αποφάσεων Markov μοντελοποιεί ένα σενάριο ενισχυτικής εκμάθησης μέσω συναρτήσεων μετάβασης και ανταμοιβής για κάθε ζεύγος καταστάσεων και δράσεων. Η αξία μιας κατάστασης αντιπροσωπεύει το άθροισμα των απομειωμένων ενισχύσεων που θα λάβει ο δράστης αν παίξει βέλτιστα ξεκινώντας από τη συγκεκριμένη κατάσταση. Αντίστοιχα, η αξία ενός ζεύγους κατάστασης-δράσης είναι το άθροισμα των απομειωμένων ενισχύσεων που θα λάβει ο δράστης αν ξεκινήσει να παίζει από την συγκεκριμένη κατάσταση, εκτελέσει τη συγκεκριμένη δράση και παίξει βέλτιστα μετέπειτα.

$$V^*(s) = E\left[\sum_{t=0}^{\infty}\gamma^t R(s_t, \pi^*(s_t))\right] \tag{1.2}$$

$$Q^*(s,a) = R(s,a) + E\left[\sum_{t=1}^{\infty}\gamma^t R(s_t, \pi^*(s_t))\right] \tag{1.3}$$

Δεδομένων των συναρτήσεων μετάβασης και ανταμοιβής, οι αξίες των καταστάσεων και η βέλτιστη στρατηγική μπορούν να υπολογιστούν επιλύοντας το σύστημα εξισώσεων:

$$V^*(s) = \max_{a \in A(s)} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) \tag{1.4}$$

$$\pi^*(s) = arg\max_{a} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) \tag{1.5}$$

Σε περιβάλλοντα όπου οι συναρτήσεις μετάβασης και ανταμοιβής δεν είναι γνωστές μπορούν είτε να προσεγγισθούν με βάση τις εμπειρίες που αποκτούνται καθώς ο δράστης κινείται μέσα στο περιβάλλον, είτε ο δράστης μπορεί να επιχειρήσει να μάθει απευθείας τις αξίες των δράσεων από τις εμπειρίες που συλλέγει, χρησιμοποιώντας έναν αλγόριθμο γνωστό ως Q-learning. Κάθε φορά που εκτελείται μια δράση a από μια κατάσταση s, πραγματοποιείται μετάβαση στην κατάσταση s' και λαμβάνεται ενίσχυση r, η εκτίμηση για την αξία του ζεύγους s, a ενημερώνεται μέσω της σχέσης:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A(s')} Q(s',a') \right) \tag{1.6}$$

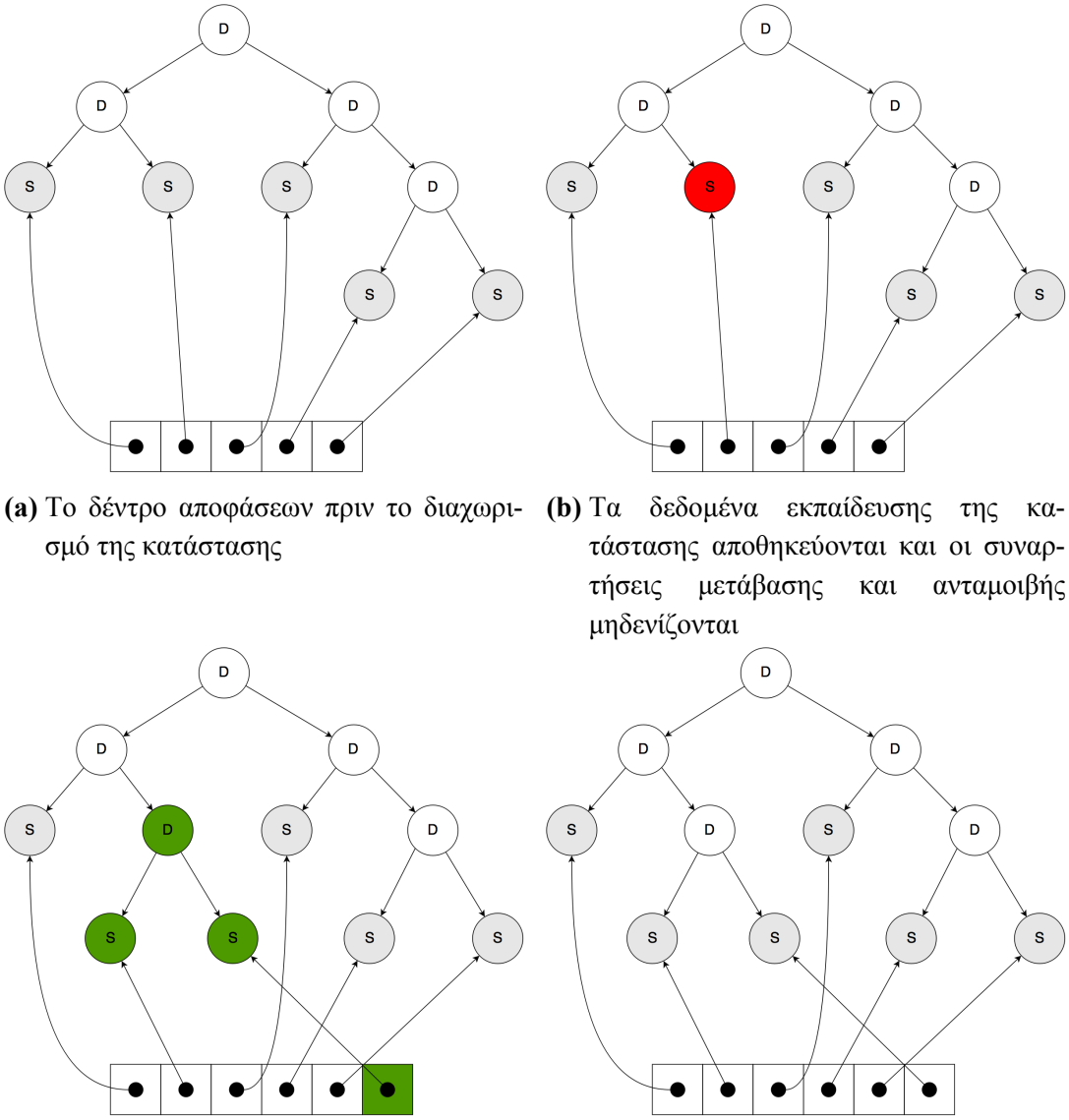
1.7 Περιγραφή της υλοποίησης

Όπως αναφέρθηκε προηγούμενα, η ύπαρξη ενός μεγάλου αριθμού παραμέτρων οι οποίες επηρεάζουν τη συμπεριφορά του συστήματος κάνει αδύνατη την αναπαράσταση του με ένα κλασικό μοντέλο Markov, δεδομένου ότι σε αυτή την περίπτωση ο αριθμός των καταστάσεων που θα χρειάζονταν για την περιγραφή της συμπεριφοράς του θα ήταν εκθετικά μεγάλος ως προς τον αριθμό των παραμέτρων. Για την αντιμετώπιση αυτής της δυσκολίας, προτείνουμε τη χρησιμοποίηση ενός αλγορίθμου ο οποίος χρησιμοποιεί ένα δέντρο αποφάσεων για τη διαμέριση του χώρου καταστάσεων σε περιοχές όπου η συμπεριφορά του συστήματος είναι ενιαία.

Ο αλγόριθμος ξεκινά έχοντας μία κατάσταση η οποία αναπαριστά ολόκληρο το χώρο καταστάσεων, η οποία αρχικοποιείται ως η ρίζα του δέντρου αποφάσεων. Ταυτόχρονα, ένας πίνακας από καταστάσεις διατηρείται, ο οποίος αρχικοποιείται ώστε να περιέχει τη μοναδική αρχική κατάσταση. Η τρέχουσα κατάσταση του κόσμου αναπαρίσταται από ένα σύνολο από μετρήσεις. Από αυτές τις μετρήσεις, η κατάσταση του μοντέλου που τους αντιστοιχεί μπορεί να βρεθεί μέσω του δέντρου αποφάσεων.

Μετά την εκτέλεση κάποιας δράσης, οι συναρτήσεις μετάβασης και ανταμοιβής ενημερώνονται κατάλληλα ώστε να αντικατοπτρίζουν τη μέχρι τώρα καταγραφείσα συμπεριφορά του συστήματος. Στη συνέχεια, η τετράδα του προηγούμενου και του τρέχοντος συνόλου μετρήσεων, σε συνδυασμό με τη δράση που πραγματοποιήθηκε και την ενίσχυση που αποκτήθηκε αποθηκεύονται στην αρχική κατάσταση της μετάβασης. Ακολούθως, ενημερώνονται οι εκτιμήσεις για τις αξίες των καταστάσεων και των δράσεων του μοντέλου.

Στη συνέχεια, πραγματοποιείται ένας έλεγχος για τον ενδεχόμενο διαχωρισμό της αρχικής κατάστασης της τελευταίας μετάβασης σε δύο νέες. Ο έλεγχος αυτός γίνεται με βάση



(a) Το δέντρο αποφάσεων πριν το διαχωρισμό της κατάστασης

(b) Τα δεδομένα εκπαίδευσης της κατάστασης αποθηκεύονται και οι συναρτήσεις μετάβασης και ανταμοιβής μηδενίζονται

(c) Η κατάσταση αντικαθίσταται από έναν κόμβο απόφασης και δύο νέες καταστάσεις

(d) Τα δεδομένα εκπαίδευσης αποκαθίστανται και οι συναρτήσεις μετάβασης και ανταμοιβής αναδημιουργούνται από τα δεδομένα

Figure 1.7: Ο διαχωρισμός μιας κατάστασης στο δέντρο αποφάσεων σε δύο νέες καταστάσεις. Η πρώτη κατάσταση αντικαθιστά την παλιά στον πίνακα καταστάσεων και η δεύτερη προσκολλάται στο τέλος του.

τις καταγεγραμμένες εμπειρίες κατά τις οποίες εκτελέστηκε η τρέχουσα βέλτιστη δράση. Για τον έλεγχο αυτό, συγκρίνονται με μια στατιστική μέθοδο οι τιμές κάθε παραμέτρου για τις ομάδες των σημείων που έχουν αξία μεγαλύτερη ή μικρότερη από την αξία της τρέχουσας κατάστασης. Οι αξίες των σημείων υπολογίζονται ως $q(m, a) = r + \gamma V(s')$ όπου r η ενίσχυση που αποκτήθηκε μετά την εκτέλεση της δράσης και s' η κατάσταση του μοντέλου στην οποία μετέβη ο δράστης. Ο διαχωρισμός θα γίνει χρησιμοποιώντας την τιμή της παραμέτρου που θα παράγει την ελάχιστη πιθανότητα σφάλματος, όπως αυτή θα προκύψει από το στατιστικό έλεγχο, στον μέσο όρο των μέσων όρων των δύο συνόλων τιμών.

Η στατιστική μέθοδος ελέγχου που χρησιμοποιήθηκε ήταν το Mann-Whitney U test, το οποίο βασίζεται στον υπολογισμό του αριθμού των συγκρίσεων μεταξύ των στοιχείων των δύο πληθυσμών που κερδίζονται από στοιχεία ενός από αυτούς. Ο διαχωρισμός της κατάστασης προχωρά αν η πιθανότητα σφάλματος που θα υπολογίσει το στατιστικό τεστ είναι μικρότερη από ένα προκαθορισμένο όριο.

Εφόσον αποφασιστεί ο διαχωρισμός μιας κατάστασης του μοντέλου σε δύο, μηδενίζονται οι συναρτήσεις μετάβασης και ανταμοιβής που αφορούν τη συγκεκριμένη κατάσταση σε όλες τις υπόλοιπες καταστάσεις του μοντέλου. Ταυτόχρονα, οι καταγεγραμμένες εμπειρίες που αφορούν μεταβάσεις από και προς την κατάσταση αυτή διατηρούνται ώστε να χρησιμοποιηθούν για την επαναφορά του μοντέλου σε συνεπή κατάσταση μετά το διαχωρισμό.

Στη συνέχεια η κατάσταση αντικαθίσταται στο δέντρο αποφάσεων από ένα νέο κόμβο απόφασης, ο οποίος έχει ως παιδιά του τις δύο νέες καταστάσεις που θα προστεθούν στο μοντέλο. Η πρώτη από αυτές αντικαθιστά την παλιά στον πίνακα των καταστάσεων, ενώ η δεύτερη προσκολλάται στο τέλος του. Τέλος, οι συναρτήσεις μετάβασης και ανταμοιβής σε ολόκληρο το μοντέλο επεκτείνονται ώστε να αντικατοπτρίζουν την εισαγωγή μιας επιπλέον κατάστασης, και εκπαιδεύονται χρησιμοποιώντας τα προσωρινά αποθηκευμένα δεδομένα εκπαίδευσης. Με αυτό τον τρόπο το μοντέλο επαναφέρεται στην κατάσταση που θα ήταν αν η εκπαίδευσή του είχε πραγματοποιηθεί εξ αρχής με αυτό το διαχωρισμό του χώρου καταστάσεων.

1.8 Αποτελέσματα Προσομοίωσης

Για την αξιολόγηση της λύσης μας πραγματοποιήσαμε μια σειρά από πειράματα σε σενάρια προσομοίωσης συστάδων εικονικών μηχανών, οι οποίες εκτελούν μια κατανεμημένη βάση δεδομένων που φορτίζεται με ερωτήματα διαφορετικών τύπων (εγγραφές και αναγνώσεις). Οι αλγόριθμοι καλούνται να πάρουν αποφάσεις για τη δυναμική αλλαγή του μεγέθους της συστάδας, η απόδοση της οποίας εξαρτάται όχι μόνο από το μέγεθός της, αλλά και από τον τύπο των εισερχόμενων ερωτημάτων. Η συνάρτηση ανταμοιβής επιβραβεύει τους αλγορίθμους όταν αυξάνουν το μέγεθος της συστάδας ώστε να μπορεί να εξυπηρετεί τα εισερχόμενα ερωτήματα, αλλά τους τιμωρεί όταν δίνουν περισσότερους από τους απαιτούμενους πόρους. Για να πραγματοποιήσουν επιτυχείς ενέργειες, θα πρέπει όχι μόνο να διακρίνουν τον τρόπο με τον οποίο αυτές αλλάζουν το μέγεθος της συστάδας και την επιρροή του εισερχόμενου φορτίου, αλλά και την επιρροή του τύπου των ερωτημάτων. Τα χαρακτηριστικά του σεναρίου προσομοίωσης συνοψίζονται παρακάτω.

- Το μέγεθος της συστάδας μπορεί να κυμαίνεται μεταξύ 1 και 20 εικονικών μηχανών.
- Οι δυνατές δράσεις σε κάθε βήμα είναι να προσθέσει μια εικονική μηχανή στη συστάδα, να αφαιρέσει μια εικονική μηχανή ή να μήν κάνει τίποτα.
- Το εισερχόμενο φορτίο είναι μια ημιτονοειδής συνάρτηση του χρόνου:
  $load(t) = 50 + 50sin\left(\frac{2\pi t}{250}\right)$
- Το ποσοστό των εισερχόμενων ερωτημάτων που είναι απλές αναγνώσεις είναι επίσης μια ημιτονοειδής συνάρτηση του χρόνου με διαφορετική περίοδο:
  $r(t) = 0.75 + 0.25sin\left(\frac{2\pi t}{340}\right)$

- Αν $vms(t)$ είναι ο αριθμός των εικονικών μηχανών, ο αριθμός των ερωτημάτων ανά δευτερόλεπτο που μπορεί να εξυπηρετήσει η συστάδα δίνεται από τη σχέση:
  $capacity(t) = 10 \cdot vms(t) \cdot r(t)$

- Η συνάρτηση ανταμοιβής εξαρτάται από την κατάσταση της συστάδας μετά την εκτέλεση μιας δράσης και δίνεται από τη σχέση:
  $R_t = min(capacity(t+1), load(t+1)) - 3 \cdot vms(t+1)$

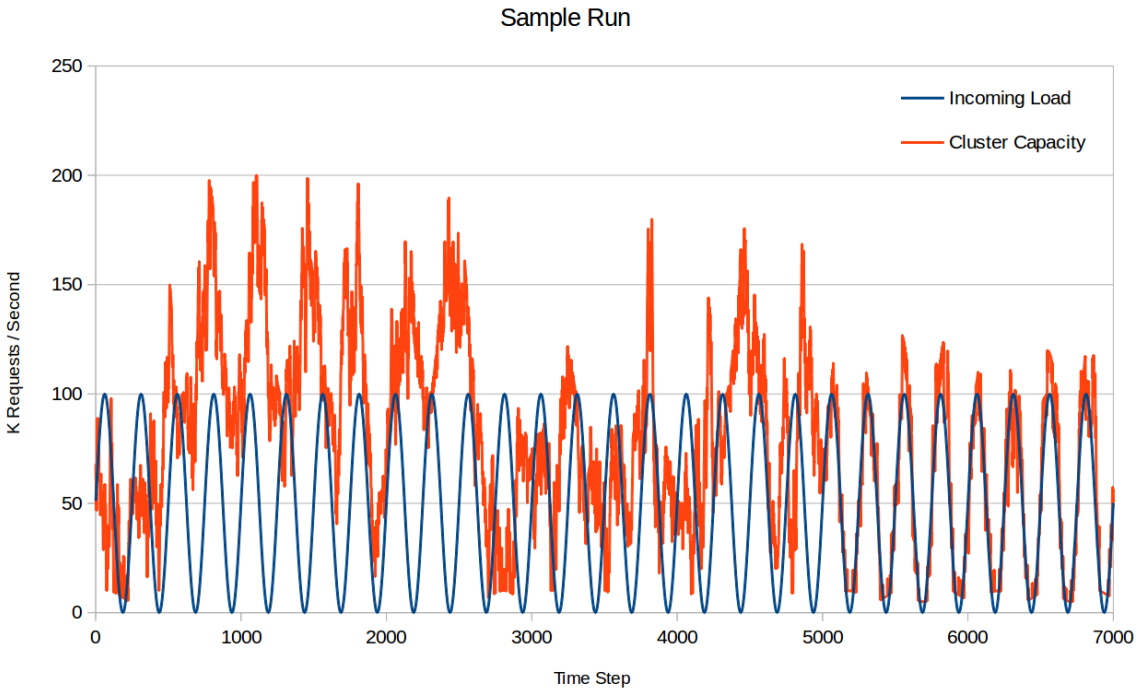


Figure 1.8: Εισερχόμενο φορτίο και διεκπεραιωτική δυνατότητα της συστάδας σε μια δοκιμαστική εκτέλεση με 5000 βήματα εκπαίδευσης, 2000 βήματα αξιολόγησης και συντελεστή εξερεύνησης $e = 1.0$

Για να ελέγξουμε τη δυνατότητα των αλγορίθμων να διαμερίσουν με αποδοτικό τρόπο το χώρο καταστάσεων, εκτός από τις τρεις παραμέτρους που επηρεάζουν την απόδοση του συστήματος (μέγεθος της συστάδας, εισερχόμενο φορτίο και ποσοστό των ερωτημάτων που είναι αναγνώσεις), η είσοδος περιείχε και 7 επιπλέον παραμέτρους οι τιμές των οποίων κυμαίνονταν τυχαία. Για να επιτύχουν στη λήψη αποφάσεων, οι αλγόριθμοι θα πρέπει να διαμοιράσουν το χώρο καταστάσεων με βάση τις τρεις σημαντικές μεταβλητές που αναφέραμε και να αγνοήσουν τις υπόλοιπες.

Οι προσομοιώσεις περιείχαν μια φάση εκπαίδευσης και μια φάση αξιολόγησης. Κατά τη διάρκεια της φάσης εκπαίδευσης, επιλεγόταν μια τυχαία δράση με πιθανότητα e, ή η βέλτιστη δράση που πρότεινε ο εκάστοτε αλγόριθμος με πιθανότητα $1-e$ (στρατηγική e-greedy). Κατά τη διάρκεια αξιολόγησης εκτελούνταν μόνο οι προτεινόμενες ενέργειες κάθε αλγορίθμου. Το μέτρο σύγκρισης των αλγορίθμων είναι το συνολικό ποσό των ανταμοιβών που κατάφεραν να συλλέξουν κατά τη διάρκεια της φάσης αξιολόγησης.

Οι αλγόριθμοι που χρησιμοποιήθηκαν σε αυτό το πείραμα είναι οι παρακάτω:

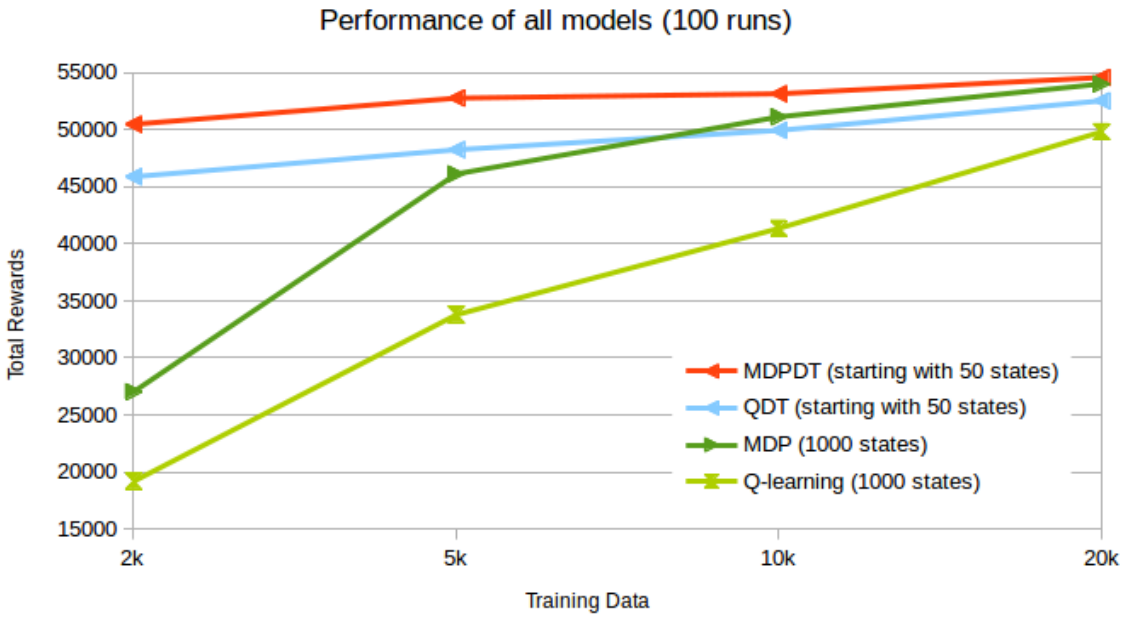


Figure 1.9: Σύγκριση της απόδοσης των τεσσάρων αλγορίθμων

- *MDP:* Διαδικασία Λήψης Αποφάσεων Markov πλήρους μοντέλου. Έχει προκαθορισμένο αριθμό καταστάσεων και διατηρεί αναλυτικά τις συναρτήσεις μετάβασης και ανταμοιβής για κάθε δράση. Περιγράφεται στην παράγραφο 4.3.6.

- *Q-Learning:* Ο δημοφιλέστερος τρόπος λήψης αποφάσεων χωρίς διατήρηση πλήρους μοντέλου της διαδικασίας. Επίσης έχει προκαθορισμένο αριθμό καταστάσεων αλλά δεν διατηρεί συναρτήσεις μετάβασης και ανταμοιβής. Περιγράφεται στην παράγραφο 4.3.5.

- *MDPDT:* Η βασιζόμενη σε δέντρα αποφάσεων υλοποίηση πλήρους μοντέλου που προτείνεται σε αυτή την εργασία. Η πλήρης υλοποίησή της περιγράφεται στην παράγραφο 5.4.

- *QDT:* Αλγόριθμος που χρησιμοποιεί δέντρα αποφάσεων αλλά βασίζεται σε Q-learning. Προτείνεται στο άρθρο [Pyea01] και περιγράφεται στην παράγραφο 5.2.

Η σύγκριση των αλγορίθμων σταθερού αριθμού καταστάσεων με τους αντίστοιχους που βασίζονται σε δέντρα αποφάσεων σε αυτό το πρόβλημα είναι άδικη, δεδομένου ότι οι δεύτεροι πρέπει να πραγματοποιήσουν το διαμοιρασμό του χώρου καταστάσεων σε ένα περιβάλλον με θόρυβο χωρίς να γνωρίζουν εκ των προτέρων την τοπολογία του, ενώ οι πρώτοι διέθεταν από την αρχή της εκτέλεσης έναν ομοιόμορφο διαμοιρασμό του χώρου καταστάσεων με βάση ακριβώς τις μεταβλητές που επηρέαζαν τη συμπεριφορά του συστήματος. Παρόλα αυτά, οι αλγόριθμοι που χρησιμοποιούν δέντρα αποφάσεων πέτυχαν ιδιαίτερα καλές επιδόσεις, και με την παροχή λίγης πληροφορίας υπό τη μορφή 50 αρχικών καταστάσεων κατάφεραν να τους ξεπεράσουν (εικόνα 1.9). Επιπλέον, το γεγονός ότι έχουν τη δυνατότητα να αυξάνουν δυναμικά το πλήθος των καταστάσεών τους, τους έδωσε τη δυνατότητα να εκπαιδευτούν πολύ γρήγορα όταν υπήρχαν πολύ λίγα δεδομένα εκπαίδευσης, αλλά όταν τα

δεδομένα έγιναν περισσότερα αύξησαν δυναμικά το μέγεθός τους και την ακρίβειά τους, και κατάφεραν να διατηρήσουν το προβάδισμα.

Ακόμη, συγκρίνοντας την απόδοση των αλγορίθμων πλήρους μοντέλου με τους αντίστοιχούς τους που βασίζονται στο *Q-learning*, παρατηρούμε μια σαφή υπεροχή των πρώτων έναντι των δεύτερων. Αυτό ήταν αναμενόμενο, δεδομένου ότι διατηρούν σημαντικά περισσότερη πληροφορία για το σύστημα και έχουν τη δυνατότητα να διαμοιράσουν την πληροφορία που προκύπτει από κάποια νέα εμπειρία στις υπόλοιπες καταστάσεις άμεσα, χάρη στον αλγόριθμο *Prioritized Sweeping*. Φυσικά, το πλεονέκτημα αυτό έρχεται με το τίμημα του σημαντικά μεγαλύτερου χρόνου εκτέλεσης, αλλά σε ένα περιβάλλον υπολογιστών νέφους ο απαιτούμενος χρόνος ανά δράση για την ενημέρωση όλων των μοντέλων είναι ούτως ή άλλως ασήμαντος.

1.9 Πειραματική Αξιολόγηση

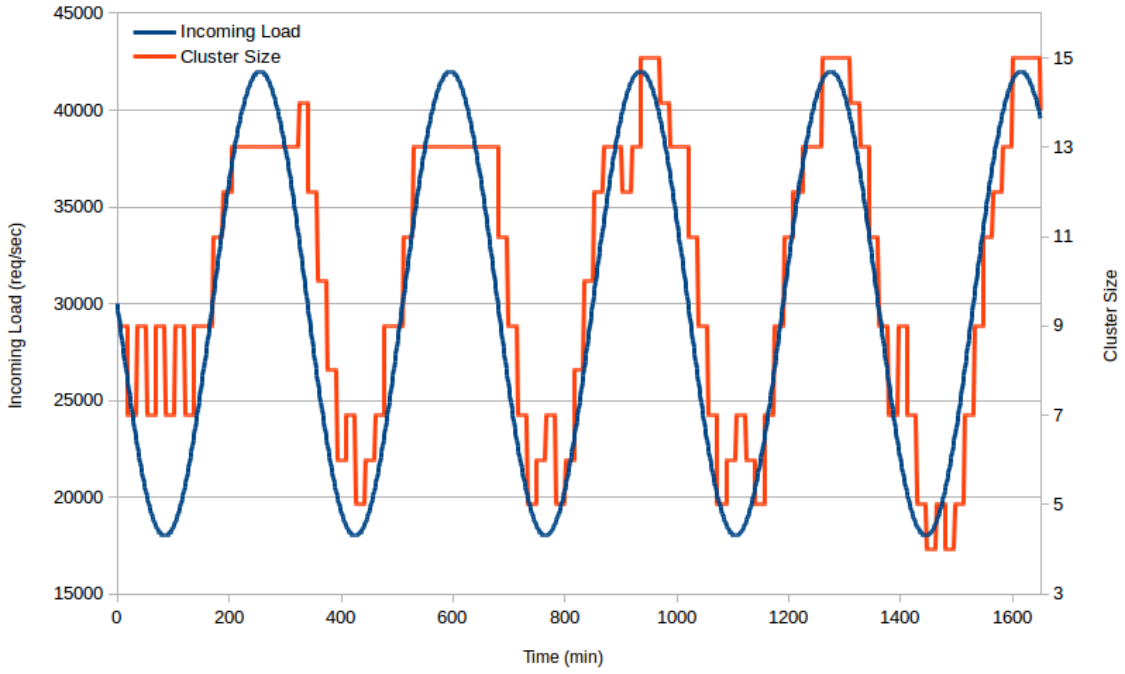


Figure 1.10: Συμπεριφορά του συστήματος υπό ένα ημιτονοειδές φορτίο, 500 βήματα εκπαίδευσης

Για την πειραματική αξιολόγηση της λύσης χρησιμοποιήθηκε μία συστάδα αποτελούμενη από 4 έως 15 εικονικές μηχανές. Κάθε εικονική μηχανή είχε 1GB μνήμη RAM, 10GB χώρο αποθήκευσης και μία εικονική CPU, ενώ ο Master Node είχε 4GB RAM, 10GB χώρο αποθήκευσης και 4 εικονικές CPUs. Για την εκπαίδευση των αλγορίθμων που βασίζονταν σε δέντρα αποφάσεων χρησιμοποιήθηκε ένα σύνολο από 12 μεταβλητές, οι οποίες περιελάμβαναν:

- Το μέγεθος της συστάδας υπολογιστών
- Την ποσότητα της μνήμης RAM ανά εικονική μηχανή

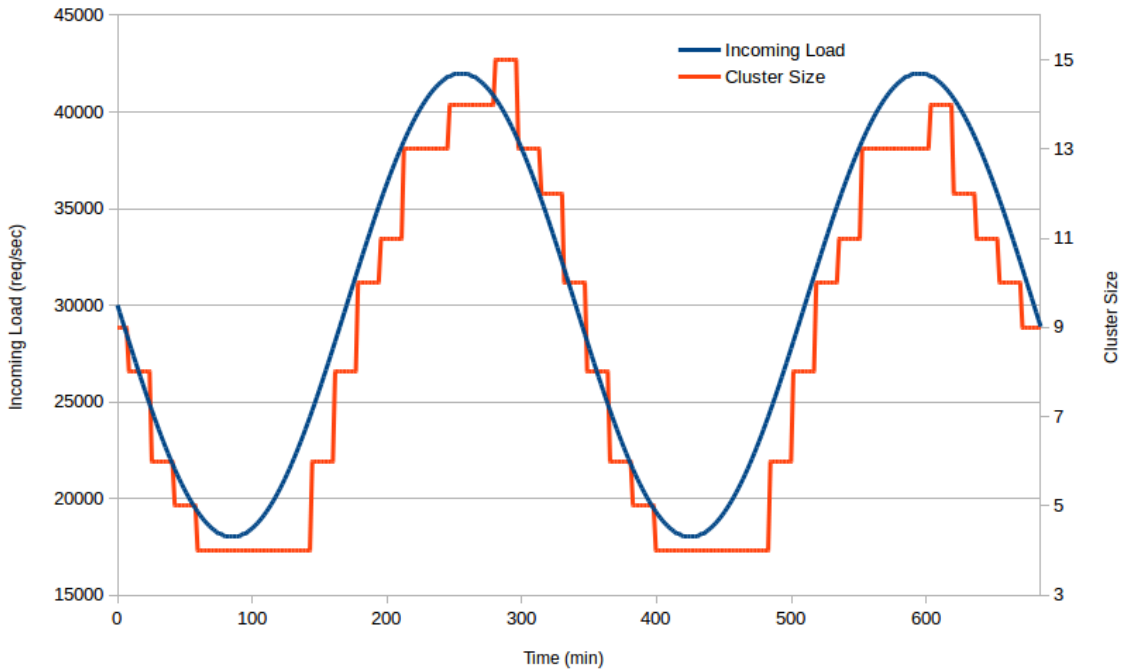


Figure 1.11: Συμπεριφορά του συστήματος υπό ένα ημιτονοειδές φορτίο, 1500 βήματα εκπαίδευσης

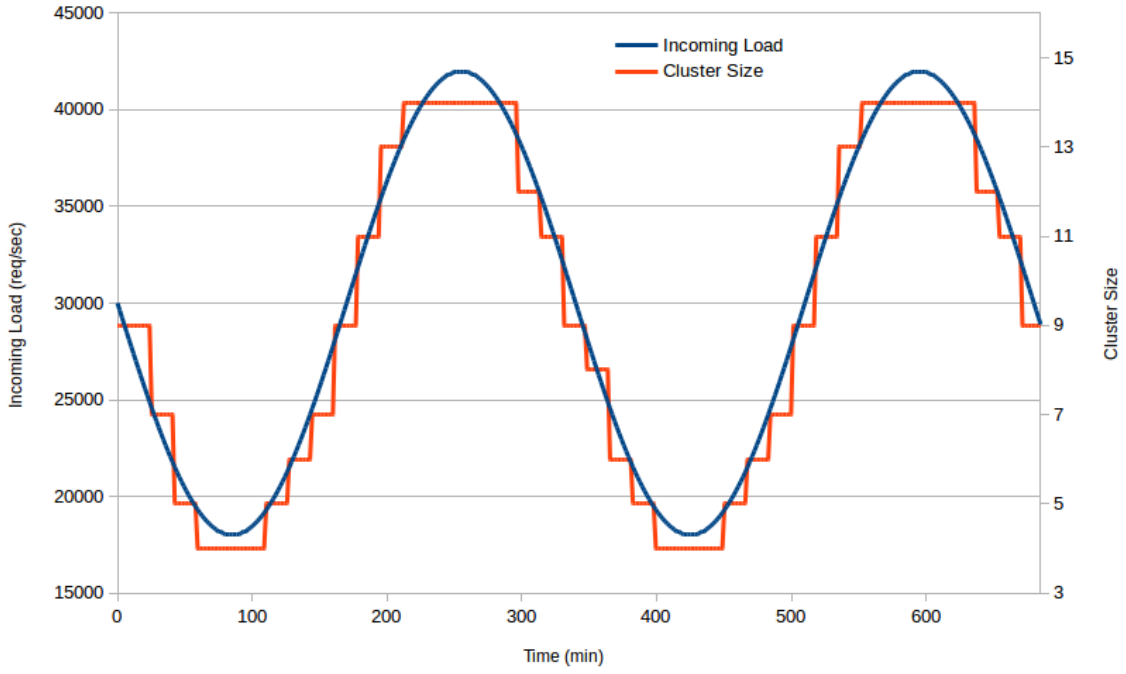


Figure 1.12: Συμπεριφορά του συστήματος υπό ένα ημιτονοειδές φορτίο, 20000 βήματα εκπαίδευσης

- Το ποσοστό της ελεύθερης μνήμης RAM
- Τον αριθμό των εικονικών CPU ανά εικονική μηχανή

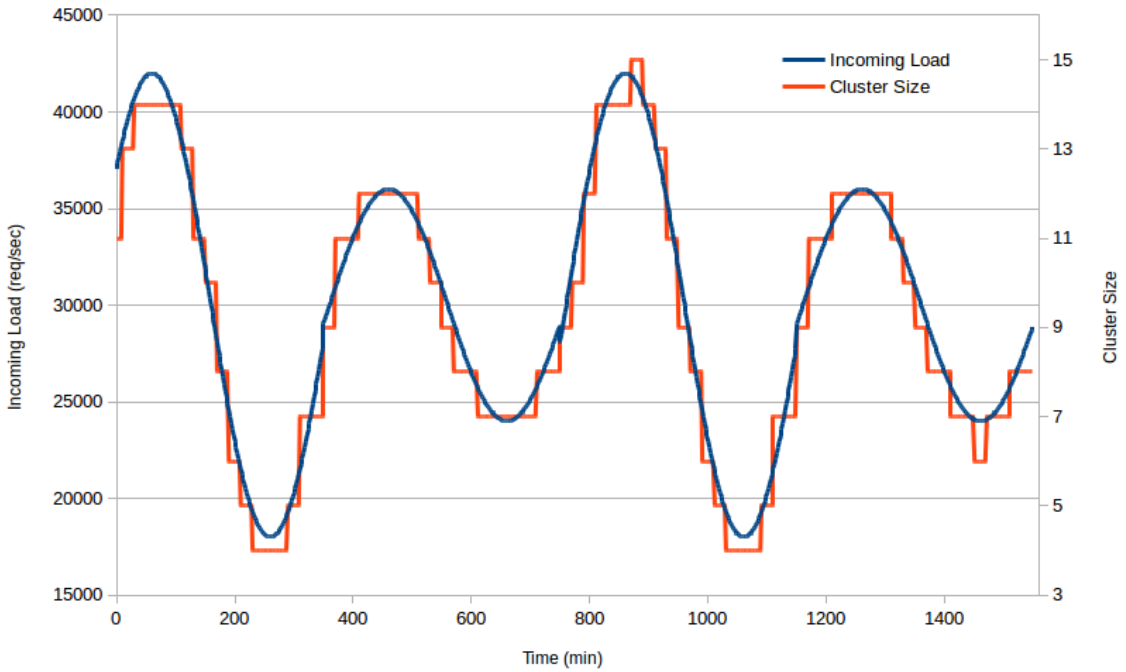


Figure 1.13: Συμπεριφορά του συστήματος υπό ένα ημιτονοειδές φορτίο εναλλασσόμενου ύψους

- Τη χρησιμοποίηση της CPU
- Την αποθηκευτική ικανότητα ανά εικονική μηχανή
- Τον αριθμό των αιτημάτων εισόδου/εξόδου ανά δευτερόλεπτο ανά εικονική μηχανή
- Το ποσοστό του χρόνου κατά τον οποίο η CPU ανέμενε λειτουργίες εισόδου-εξόδου
- Μια γραμμική πρόβλεψη για το ύψος του εισερχόμενου φορτίου
- Την αναλογία reads και updates στα αιτήματα που δεχόταν η βάση
- Το μέσο latency των αιτημάτων που εξυπηρετούνταν
- Τη μέση χρησιμοποίηση του δικτύου

Το μοντέλο αρχικοποιήθηκε με 6 καταστάσεις, και στη συνέχεια αφέθηκε να διαμοιράσει το χώρο καταστάσεων με βάση το κριτήριο διάσπασης κόμβων που περιγράφηκε στην ενότητα 1.7. Μια σειρά από 15 εικονικές μηχανές ανέλαβαν να πραγματοποιούν ερωτήματα προς τη βάση, και ο αλγόριθμος που περιγράψαμε ανέλαβε να ελέγχει το μέγεθος της και να το προσαρμόζει στο εισερχόμενο φορτίο μέσω 5 διαθέσιμων δράσεων.

Αρχικά εκπαιδεύσαμε το μοντέλο για 500 περίπου βήματα και στη συνέχεια ζητήσαμε από τον αλγόριθμο να εκτελέσει τη βέλτιστη στρατηγική που είχε καταφέρει να εξάγει μέχρι εκείνη τη στιγμή. Κατά τη διάρκεια της εκπαίδευσης είχαν πραγματοποιηθεί 17 διαχωρισμοί καταστάσεων (4 χρησιμοποιώντας το μέγεθος της συστοιχίας, και 13 χρησιμοποιώντας το εισερχόμενο φορτίο), αυξάνοντας το συνολικό αριθμό καταστάσεων του μοντέλου σε 22. Ο αριθμός αυτός των καταστάσεων ήταν αρκετός ώστε να επιτρέψει στον αλγόριθμο να

αρχίσει να ακολουθεί επαρκώς το εισερχόμενο φορτίο (εικόνα 1.10). Κατά τη διάρκεια της εκτέλεσης πραγματοποιήθηκαν 12 επιπλέον διαχωρισμοί (4 με το μέγεθος της συστοιχίας, 7 με το εισερχόμενο φορτίο και και 1 με το μέσο latency), με συνέπεια η ακρίβεια του μοντέλου σταδιακά να βελτιώνεται και να ακολουθεί το εισερχόμενο φορτίο με μεγαλύτερη αποτελεσματικότητα. Μετά από 1500 βήματα εκτέλεσης, η ακρίβεια του μοντέλου είχε αυξηθεί αισθητά, έχοντας πλέον 66 καταστάσεις και ακολουθώντας το εισερχόμενο φορτίο με μεγαλύτερη σταθερότητα (εικόνα 1.11). Τέλος, μετά από 20000 βήματα εκπαίδευσης, το σύστημα είχε σταθεροποιηθεί πλήρως και έχοντας πλέον 576 καταστάσεις κατάφερνε να παρουσιάσει μια πολύ ακριβή συμπεριφορά (εικόνες 1.12 και 1.13).

1.10 Συμπεράσματα

Κατά τη διάρκεια αυτής της εργασίας είχαμε τη δυνατότητα να πειραματιστούμε με αλγόριθμους ενισχυτικής εκμάθησης οι οποίοι βασίζονται σε δέντρα αποφάσεων για το δυναμικό διαμοιρασμό του χώρου καταστάσεων. Στο κεφάλαιο αυτό θα συνοψίσουμε τα συμπεράσματα που προέκυψαν από αυτή την εργασία.

- Η απόδοση των αλγορίθμων που βασίζονται σε δέντρα αποφάσεων ήταν ιδιαίτερα ανταγωνιστική και σε πολλές περιπτώσεις ξεπέρασε αυτή των παραδοσιακών μοντέλων. Αυτό οφείλεται όχι μόνο στη δυνατότητά τους να υλοποιήσουν έναν αποδοτικό διαμοιρασμό του χώρου καταστάσεων, αλλά και στο γεγονός ότι αυξάνουν δυναμικά το μέγεθός τους καθώς περισσότερα δεδομένα εκπαίδευσης γίνονται διαθέσιμα.

- Η χρήση ενός στατιστικού τεστ για το διαχωρισμό των καταστάσεων παρέχει πολύ καλή προστασία έναντι του θορύβου, εφόσον χρησιμοποιείται με επαρκώς αυστηρά κριτήρια. Απεναντίας, η χρήση κριτηρίων που βασίζονταν στην ελαχιστοποίηση της πληροφορίας έδειξε να αποδίδει λιγότερο, απαιτώντας πολύ αυστηρότερες προϋποθέσεις για την αποφυγή λαθών.

- Το Mann Whitney U test εκτελούμενο επί των τιμών των παραμέτρων σύμφωνα με το κριτήριο που περιγράφηκε απέδωσε καλύτερα από άλλα στατιστικά τεστ όπως το Student's t-test, το τεστ του Welch και το τεστ των Kolmogorov-Smirnov.

- Ο απαιτούμενος επιπλέον χρόνος υπολογισμού για τη διεξαγωγή των στατιστικών τεστ και το διαχωρισμό των καταστάσεων ήταν σχετικά μικρός, και ο συνολικός χρόνος εκτέλεσης του αλγορίθμου εξακολουθούσε να καθορίζεται από την επιλογή του αλγορίθμου ενημέρωσης των τιμών, όπως και στην περίπτωση των παραδοσιακών μοντέλων.

- Η βέλτιστη στρατηγική που προέκυψε με βάση τις προσομοιώσεις που έγιναν ήταν η έναρξη της εκτέλεσης του αλγορίθμου έχοντας ένα μικρό αριθμό από καταστάσεις (αντί μίας μοναδικής κατάστασης) και ο έλεγχος για διαχωρισμό της αρχικής κατάστασης μετά από κάθε παρατηρούμενη μετάβαση.

Chapter 2

Introduction

2.1 Motivation

The explosive growth of cloud computing over the last decade has radically changed the way applications and services are built. Data such as business documents, audio and video, social media content and much more are nowadays stored within the cloud and made accessible worldwide through web applications. The volume of this data is counted in trillions of gigabytes (or *Zetabytes*). The need to store and process this volume of information gave birth to a number of new technologies and paradigms.

Traditional SQL database systems were unable to scale up to this volume of data, and in order to fulfill the need for storage on this scale, NoSQL databases were introduced. These databases are designed to run on large scale distributed systems, managing not only the distribution of data and the coordination of the machines, but also tolerating hardware failures which are unavoidable on this scale.

Being able to easily scale to thousands of machines, these systems often run within a virtual environment, provided by an IaaS (Infrastructure as a Service) provider. Large scale IaaS providers have themselves the ability to host thousands of VMs, and often provide the ability to automatically scale up and down their services according to user requirements, a concept known as *elasticity*. However, in most cases, the methods used to implement this elasticity are threshold based, and require the user to manually select the conditions under which any elasticity action is performed.

These methods of decision making however, are often unable to perform well in such a complicated and dynamic environment, since their simplistic nature has no capability of performing strategic decisions. A more sophisticated approach to the problem is through the use of *Reinforcement Learning* algorithms such as *Markov Decision Processes* and *Q-Learning*. These algorithms are natural solutions in situations where decision making is necessary, and offer guarantees of optimality under reasonable conditions.

Even these more sophisticated methods though have their limitations. In the typical reinforcement learning setting, the world is assumed to be in one of a finite number of *states*, and from each state a number of *actions* are available. Upon the execution of an action, a scalar reinforcement is received and the world transitions to a new state. An algorithm is *optimal* in the sense that it chooses actions that maximize some predefined long-term measure of the reinforcements. However, this optimality is achieved under the assumption that the behavior of the system is the same each time it finds itself in any specific state. This means that the states need to be fine-grained enough to capture all the complexity of the system.

In the case of the management of a NoSQL cluster however, the number of parameters that affect the behavior of the system is exceedingly large, and many of them are continuous instead of discrete (size and characteristics of the cluster, live performance metrics, characteristics of the load etc). Even if we were to discretize their values, defining a different state for each of their different combinations would result in an exponential number of states. A reinforcement learning model of this scale is not only unrealistic to represent in memory, but even more so impossible to train, since the amount of experiences required to learn the behavior of the system would also be exponential. The subject of this work therefore, is to seek methods that can overcome this difficulty, while at the same time providing all the benefits that traditional reinforcement learning algorithms do.

2.2 Related Work

In [Chap91], the authors propose a modification of Q-learning that uses a decision tree to generalize over the input. The goal of the agent is to control a character in a two dimensional video game, where the state is a bit string representing the pixels of the on-screen representation of the game. Since the input consisted of a few hundreds of bits, the state space consisted of more than 2^{100} states and thus generalization was necessary. The proposed algorithm gradually partitioned the state space based on values of individual bits of the state. A t statistic was used to determine if and with what bit a state needs to be split.

In [Pyea01], a Q-learning algorithm that uses a decision tree to dynamically partition the state space is proposed. The motivation is to build reinforcement learning agents for two applications in the field of robotics where the state space is too large for classical lookup-table approaches. The algorithm builds a decision tree based on values of parameters of the input, and maintains a Q-learning model on the leaves of the tree. Different criteria are examined for the splitting of the nodes, and the performance of the algorithm is tested against lookup-table based approaches as well as neural networks.

In [Uthe98], a full-model, decision tree based algorithm is proposed, called *Continuous U Tree*. The algorithm is split into two phases. During the *Data Gathering* phase, the states of the MDP model remain unchanged, but experience tuples are stored for future use. During the *Processing Phase*, the stored experiences are used to determine the states of the model that need to be split into new states. Once the new states of the model have been decided, the stored experiences are used to calculate the transition and reward function for the new set of states, and the values of the states and Q-states are calculated. The algorithm continuously alternates between the two phases, periodically extending the decision tree and globally recalculating the current status of the MDP.

In [Gask99], the authors propose *Wire-fitted Neural Network Q-Learning*, a modification of Q-Learning that generalizes over continuous state and action spaces. The solution uses a neural network coupled with a novel interpolator to approximate the Q-function. The neural network calculates the Q-fuction given the current state as input. When a new estimate of the Q-value is acquired after executing an action, a new expected output of the neural network is calculated using the wire-fitter partial derivatives. Finally, the neural network is trained to output the new Q-function.

In [Tsou13] the authors present TIRAMOLA, a cloud-enabled open-source framework to perform automatic resizing of NoSQL clusters according to user-defined policies. The system decides on the most advantageous cluster size, and proceeds to automatically modify it by requesting/releasing VM resources from the provider and orchestrating them inside the cluster. The cluster is modeled as a Markov Decision Process, where the states represent different cluster sizes and the actions resizing decisions that modify that size. In order to isolate the most relevant experiences to the expected resulting state of each action, K-means clustering was used and the expected reward was calculated using the centroid of the cluster.

In [Kass14] the authors extend TIRAMOLA to identify different workload types. An analysis of how different query types are handled by modern NoSQL clusters of varying size is performed, and the resulting knowledge is utilized to fine tune TIRAMOLA's policies in order to take more accurate scaling decisions.

In [Nask], an approach to enforcing elasticity through the dynamic instantiation and online quantitative verification of Markov Decision Processes is proposed, using probabilistic model checking. Various concrete elasticity models and elasticity policies are studied and evaluated using traces from a real NoSQL database cluster under constantly evolving external load. The NoSQL cluster is modeled as a Markov Decision Process with multiple states per size of the cluster, and non-deterministic transitions are added among the states. The values of a metric determine the clustering, and the transition probabilities are proportional to the number of points in each state.

In [Maso15], the authors propose a method of consolidating Virtual Machines in clusters. The general goal of the method is to manage physical host nodes in order to avoid overloading and underloading, and to optimize the placement of VMs. Fuzzy Q-learning is used as a substitute to Q-learning, to consolidate the state space and accelerate the learning of the action values. A cooperative approach to learning is proposed, where multiple agents share their knowledge through a blackboard communication schema. The states are identified by $\langle CPU\ utilization, Number\ of\ VMs \rangle$ tuples, while the actions decide on the threshold values of a VM migration policy, as well as the selection criterion. The reward function is calculated using the energy consumption combined with the number of SLA violations.

2.3 Proposed Solution

The application this work is aimed at is building an agent that makes resizing decisions for an HBase cluster running on an IAAS provider. That problem provides two important challenges that need to be taken into consideration:

- There is a large number of variables that affect the behavior of the system, many of which are not discrete. In other words the solution must be able to generalize over a multi-dimensional continuous state space.

- The time interval between two decisions is in the order of minutes. This has two important consequences. First, collecting data to train the algorithm takes time. This means that the algorithm needs to make as good use of any data it has acquired as possible. Second, there is a lot of time to make decisions, which allows us to be more wasteful in terms of the computational power required by our solution.

To tackle these challenges, in section 5.4 we implement a solution with the following characteristics:

- We adopt a full-model based approach over a Q-learning approach. Having time in the order of minutes to make decisions, it is realistic to maintain a full MDP model of the system by storing reward and transition data, and running algorithms like prioritized sweeping or value iteration with each step to update it. The fact that experiences are acquired at a slow rate limits the size of the model and makes running expensive calculations in each step possible.

- We opt for a decision tree based algorithm in order to dynamically partition the state space. Decision tree based algorithms, unlike traditional approaches, are not limited by a fixed number of states that needs to be defined beforehand, but can dynamically create new states when needed, as instructed by the behavior of the system. This does not only allow them to work on a multi-dimensional continuous state space, but also to adjust their size based on the amount of training data available. Since these models start with a small number of states, it is easier to train them with a minimum amount of data. As more data are acquired, the number of states dynamically increases, and with it increases the accuracy of the model.

- It is essential in our approach that we never waste collected information. Therefore, when an old state is replaced with two new ones, the data that had been used to train that old state is used again to train the two new ones. This way, even though new states are introduced to the model, these new states are already trained and their values and Q-values already represent all the experiences acquired since the start of the model's life. In order to accomplish this, we implement an algorithm that can perform splits and retrain the new states in a fine-grained manner, without having to globally retrain the model from zero. This allows us to perform splits efficiently as each new experience comes in which is not only computationally more efficient, but also achieves better performance.

2.4 Thesis Structure

In Chapter 2 we define the problem this work is focusing on, we briefly reference related work on the subject and outline our solution.

In Chapter 3 we present the technologies used in our practical experiments. We used the *Hadoop Distributed File System* (HDFS), a distributed file system developed by Apache. On top the HDFS runs *HBase*, a distributed non-relational database. The infrastructure was provided by the *Openstack* IaaS provider, and the metrics needed to perform elasticity decisions were collected using *Ganglia*. Finally, all the above were controlled and orchestrated by *Tiramola*, a cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters.

In Chapter 4 we analyze Markov Decision Processes, which is the theoretical basis of our approach. We lay out the theory behind them, and describe the advantages and disadvantages of well known algorithms within that context.

In Chapter 5 we focus on Decision Trees, and describe how they have been combined with Markov Decision Processes in the past to perform dynamic partitioning of the state space. Here we present our proposal, a full-model Markov Decision Process based algorithm using a Decision Tree to generalize over its input.

In Chapter 6 we present results from a number of simulation experiments. The focus of the experiments is to provide some insight on the behavior of the algorithms discussed in Chapters 4 and 5, and evaluate the performance of our proposal, compared to traditional reinforcement learning solutions.

In Chapter 7 we present results from testing our solution in a real cluster running HBase on an OpenStack IaaS provider.

Finally, in Chapter 8 we summarize and evaluate our results, and point at subjects we did not have the opportunity to experiment with as future work.

Chapter 3

Elastic Resource Management

In this chapter, we provide an overview of the tools and technologies used throughout this work. In section 3.1 we describe the *HDFS*, a distributed file system designed to provide high throughput access to application data. In section 3.2, we focus on *HBase*, a distributed, non-relational database running on top of Hadoop and HDFS. In section 3.3, we review *OpenStack*, an open-source software platform providing Infrastructure as a Service, upon which HDFS and HBase can be utilized. In section 3.4, we describe *Ganglia*, a distributed monitoring system able to gather metrics from clusters and grids. Finally, in section 3.5, we describe *Tiramola*, a cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters.

3.1 HDFS

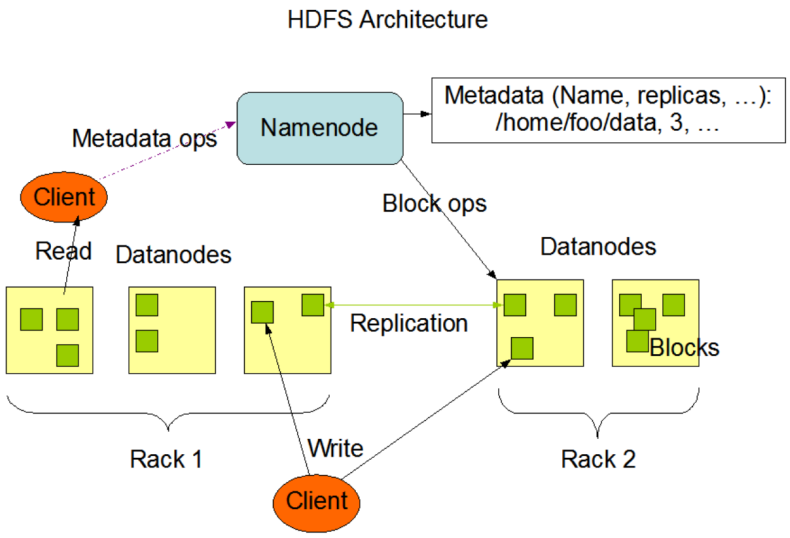


Figure 3.1: The HDFS Architecture

The Hadoop Distributed File System (HDFS) [Bort08] is a distributed file system designed to run on commodity hardware. It is essentially an open source implementation of the Google File System (GFS) [Ghem03], and it attempts to provide scalability and fault tolerance while deployed on low-cost hardware. It specializes in dealing with big amounts of data, and diverges from the POSIX standard to better suit applications that require streaming access to file system data.

HDFS uses a master/slave architecture. The *Namenode* takes on the role of the master, and is responsible for coordinating the filesystem and providing access to its files to the clients. Even though data in the HDFS are stored in multiple physical machines, the Namenode maintains a traditional hierarchical file organization. Clients can create files and directories, move and rename them in a manner similar to other existing file systems. Any change to the file system is recorded by the Namenode, who is responsible for maintaining the file system namespace. If the Namenode is not active, clients lose the ability to access the data stored in the HDFS, making it the single point of failure of the system. However, in order to increase reliability, a secondary Namenode is active at all times, and can recover the file system in case of a Master failure.

The slaves in HDFS are called *DataNodes*, and their responsibility is to store file data and serve read and write requests from the file system's clients. At the same time, they perform block creation, deletion and replication upon instruction of the NameNode. Each file in the file system is stored in multiple equally sized *blocks* (typically 64MB), and each of these blocks is hosted in multiple DataNodes in order to increase fault tolerance. It is possible for applications to specify or change the *replication factor* for each separate file.

In order for the NameNode to have an up-to-date knowledge of the active blocks in the system, *Heartbeat* messages are periodically sent to it from each of the DataNodes. If a DataNode fails to transmit a heartbeat message, the NameNode assumes that the DataNode is dead, stops forwarding new requests to it and attempts to quickly restore the replication factor of its blocks.

The placement of the blocks is decided by the NameNode. The criteria by which this is done is not only to increase fault tolerance, but also to improve performance. In the common case where the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy reduces the required communication between different racks during writes, while at the same time does not leave the system vulnerable to a single rack failure. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

Any change to the file system is recorded by the NameNode in a log called the *EditLog*. When the NameNode starts up, it loads the stored image of the file system and applies to it the changes recorded in the EditLog. The metadata required for the NameNode to have a full view of the file system are very compact, allowing it to maintain all this information in memory even when the cluster hosts a very large amount of data. After updating it, the image of the file system is flushed back to the disk, and the EditLog is truncated to a new *checkpoint*.

3.2 HBase

3.2.1 The HBase data model

HBase is an open source, distributed database for storing structured data. Its design is based on Google's *BigTable* [Chan08], and runs on top of the HDFS to enhance its storing capabilities. Its data model is different from traditional relational databases. It does not sup-

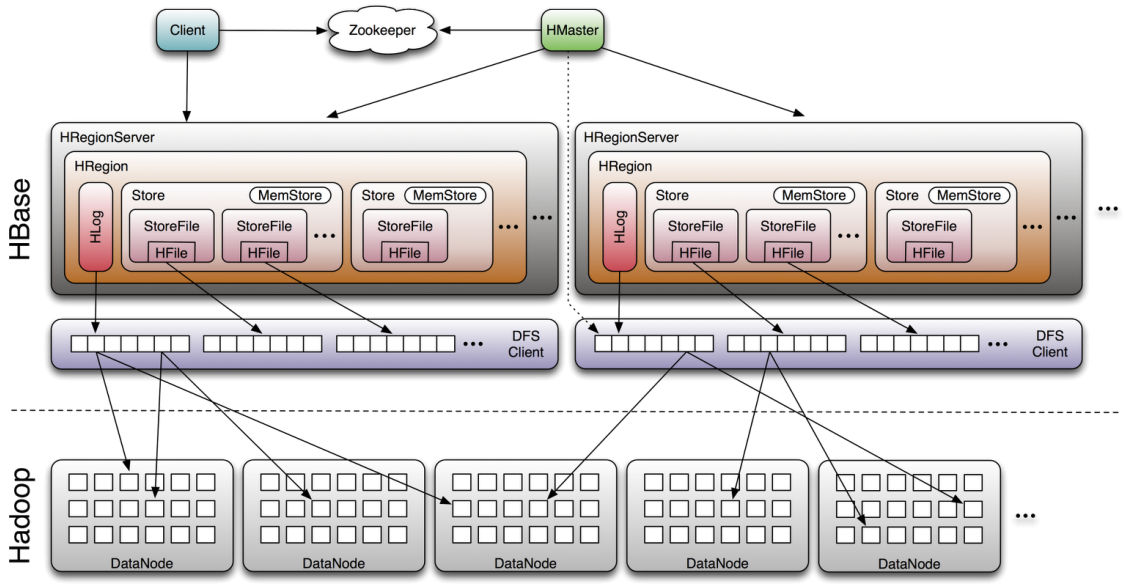


Figure 3.2: The HBase architecture

port a structured query language like SQL, but instead uses a key/value model where data are organized in columns. The building blocks of HBase's data model are the following:

- *Table:* The biggest building block in the database.
- *Row:* Each table consists of a number of rows. Each row possesses a unique key through which it can be identified, and all rows within a table are sorted based on that key. This enables the programmer to control the way data are stored and allows for easy and efficient access to ranges of rows.
- *Column Family:* Data within each row are split to separate column families that are the same for each row and need to be specified upon table creation (even though some rows may not contain data in all column families). Data stored within each column family are also physically stored in adjacent locations in order to more efficiently serve queries requesting data from them.
- *Column:* Each column family contains a number of columns. Unlike column families, columns are allowed to differ from row to row, and can change dynamically.
- *Cell:* A combination of a row key, a column family and a column uniquely identifies a cell. Each cell stores a byte array, which is its value.
- *Timestamp:* HBase has a built-in data versioning and recovery mechanism through the use of its *timestamps*. Instead of storing a single value in each cell, HBase stores a number of recent values. That number can be configured to be different for each column family, and is by default equal to three. If not specified, HBase will store data using the current timestamp and read the data with the latest timestamp, though the user is free to read and write the versions of the data he specifies.

3.2.2 The HBase architecture

HBase uses a Master/Slave architecture, and is made up of the following components:

- *Master Server:* The Master Server in HBase holds the metadata for all the tables stored in the database, and performs schema changes and table creation or deletion operations. At the same time, it controls the distribution of the regions among the Region Servers in order to evenly balance the workload.

- *Region Servers:* Each Region Server is responsible for serving and managing a number of regions. Even though data stored in the HDFS are spread across different physical locations, each region server stores the data that correspond to the regions it serves within the local HDFS DataNode in order to be able to serve requests locally.

- *ZooKeeper:* ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. HBase uses ZooKeeper to track the state of the servers in the cluster and handle communication between the master and the region servers.

HBase's architecture allows it to easily scale and store large amounts of sparse data. The fact that it runs on top of HDFS provides high availability and fault tolerance, and makes HBase easy to integrate with other tools within the Hadoop ecosystem, such as MapReduce. Finally, having only a single server responsible for each piece of data, allows it to guarantee strong consistency and perform atomic row operations.

3.3 OpenStack

OpenStack is an open source platform for cloud computing. It begun in 2010 as a joint project of Rackspace Hosting and NASA. As of 2015 it is managed by the OpenStack Foundation, a non-profit organization created specifically to promote OpenStack.

The main characteristics of OpenStack are:

- *Scalablility:* It is already deployed to host Petabytes of data, run on up to 1 million physical machines, host up to 60 million virtual machines and billions of stored objects [Sefr12].

- *Compatibility and Flexibility:* It supports a wide range of virtualization technologies, including ESX, Hyper-V, KVM, LXC, QEMU, UML, Xen and XenServer.

- *Open source:* The entire OpenStack code can be studied, and if needed modified and adapted.

3.3.1 The OpenStack Architecture

The OpenStack architecture consists of three main components:

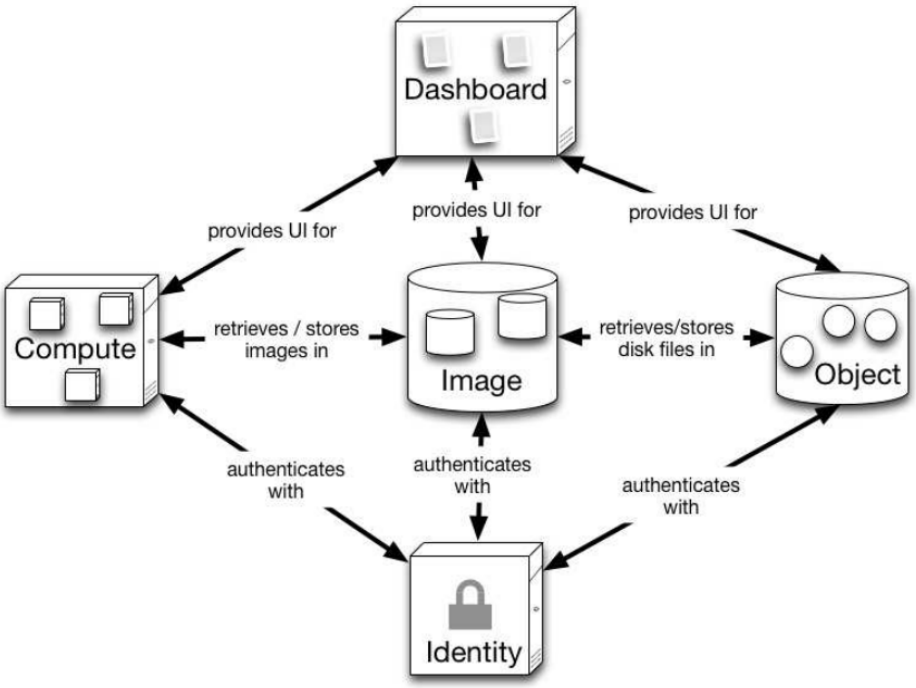


Figure 3.3: The fundamental building blocks of OpenStack

- *OpenStack Compute:* OpenStack Compute, also known as *Nova*, is a platform whose aim is to manage the OpenStack infrastructure. It provides an interface and an API that allows the management of large networks of virtual machines and redundant and scalable architectures. It is written in Python, and is designed to scale horizontally on standard hardware with no proprietary requirements.

- *Image:* Imaging Service manages the storage of the images of virtual machines, that can later be used as a template for new ones. It provides a RESTful API to perform queries for information about the images hosted on different storage systems.

- *Object:* Object Storage is a storage space that is designed for long term storage of large volumes, and can host up to multiple Petabytes of data. Objects and files are written to multiple disk drives spread throughout servers in the data center, while data replication is used to provide data integrity across the cluster.

The OpenStack services can be accessed through the OpenStack Dashboard (*Horizon*), which provides a graphical interface for users and administrators to access, provision, and automate cloud-based resources. Its design also accommodates third party products and services, such as billing, monitoring, and additional management tools. OpenStack Identity (*Keystone*) provides a mapping of users to the OpenStack services they can access. It acts as a common authentication system across the cloud operating system, and supports multiple forms of authentication including standard username and password credentials, token-based systems and AWS-style logins.

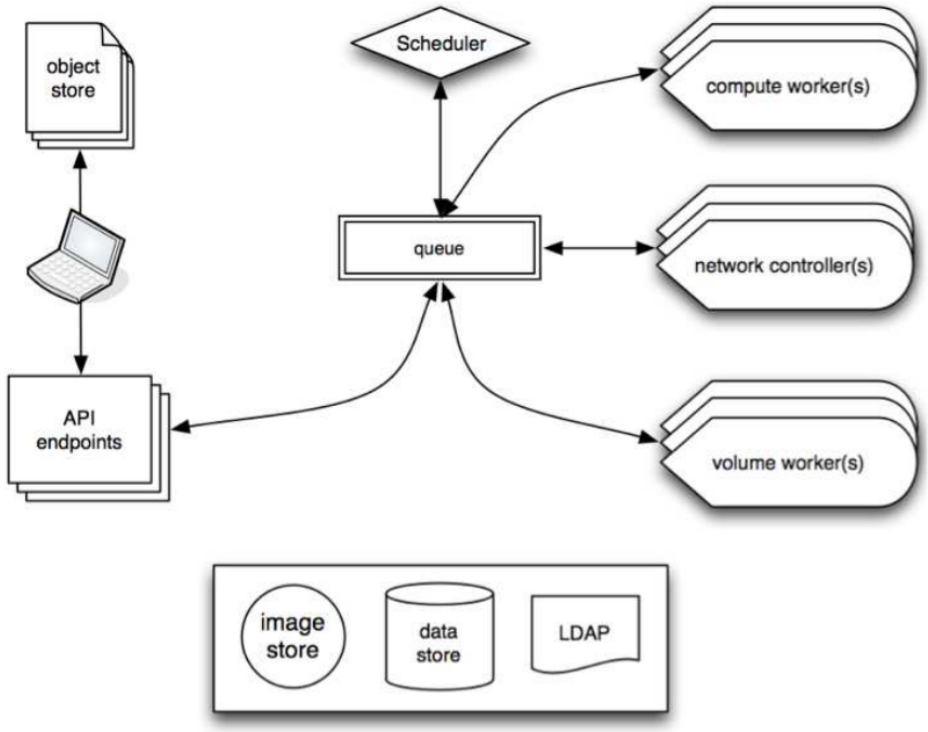


Figure 3.4: The Nova system architecture

3.4 Ganglia

Ganglia [Mass04] is a scalable distributed monitoring system for high performance computing systems such as clusters and Grids, developed by the University of California, Berkeley. It is based on a multicast, listen/announce protocol to monitor the state of the cluster, and uses a tree of point to point connections between representative cluster nodes to federate clusters and aggregate their state. Data are represented in XML format, exchanged using the XDR protocol and stored and visualized with the RRD tool. It manages to achieve very low per node overhead and high concurrency, and is available in a wide range of operating systems.

The Ganglia architecture consists of the following components:

- *gmond:* The Ganglia Monitoring Daemon (*gmond*) is installed in every node of the cluster from which metrics are to be collected. Its job is to collect the required metrics with the help of the operating system, as well as announce them to a multicast channel through UDP. It is organized as a collection of threads, most of which are assigned with the task of collecting data for a specific metric.

  The *collect and publish* thread takes on the responsibility of gathering the metrics collected by the local threads and publishing it on a well-known multicast channel in periodic messages called *heartbeats*. The *listening threads* are responsible for listening on the multicast channel for data transmitted by other nodes and storing it in a local hash table. This allows the data for the whole cluster to be available through any one of its nodes. Finally, a number of *XML export threads* accept and process client requests to provide access to that data.

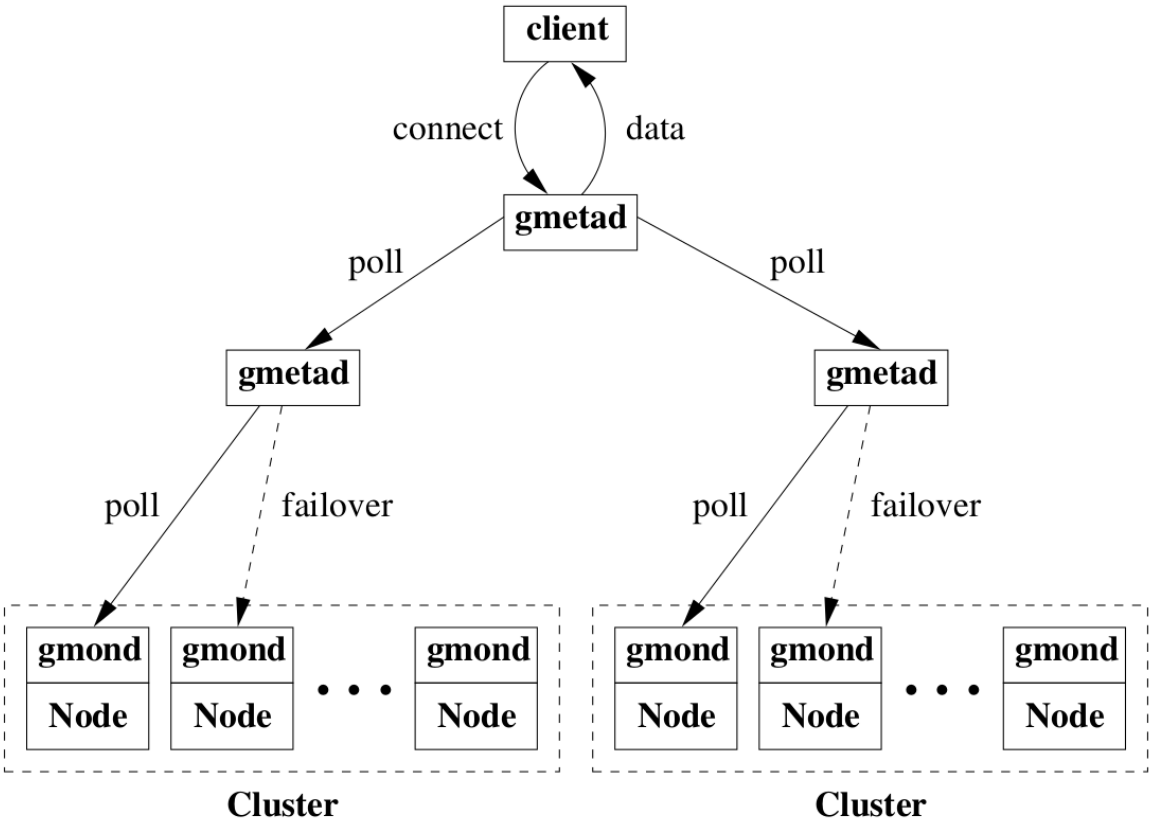


Figure 3.5: The Ganglia architecture [Mass04]

- *gmetad:* Federation in Ganglia is achieved using a tree of point-to-point connections amongst representative cluster nodes to aggregate the state of multiple clusters. At each node in the tree, a Ganglia Meta Daemon (*gmetad*) periodically polls a collection of child data sources, parses the collected XML, saves all numeric, volatile metrics to round-robin databases and exports the aggregated XML over TCP sockets to clients. Data sources may be either *gmond* daemons, representing specific clusters, or other *gmetad* daemons, representing sets of clusters.

  Data collection in *gmetad* is done by periodically polling a collection of child data sources which are specified in a configuration file, dedicating a unique data collection thread to each child source. Collected data is parsed in an efficient manner (using a SAX XML parser and a GNU gperf-generated hash table) to reduce CPU overhead and the memory footprint.

- *RRDtool:* Storage and visualization of the historical monitoring information for the grid is managed by *RRDtool* (Round Robin Database). *RRDtool* is specialized in storing time series data and is able to maintain different time granularities ranging from minutes to years in compact, constant size databases. Additionally, *RRDtool* is able to plot the historical trends of these metrics on graphs that are used by the *Ganglia PHP web front-end*, to be presented through a web interface.

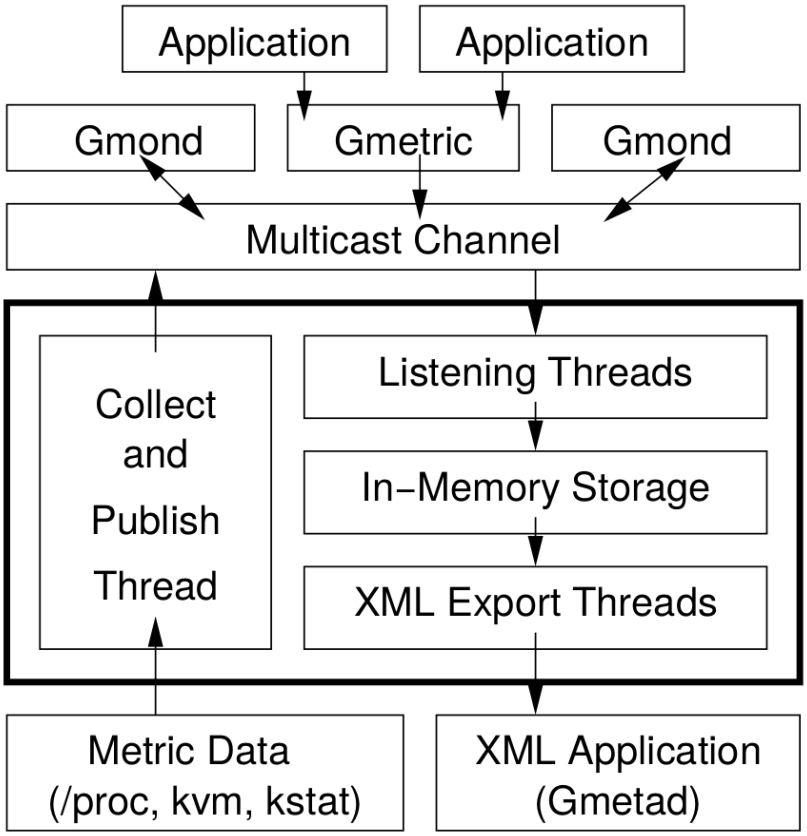


Figure 3.6: The Ganglia implementation [Mass04]

3.5 Tiramola

Tiramola [Tsou13] is a modular cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters. Its implementation is open-source, and contains modules that can control a number of different NoSQL databases, including Cassandra, HBase, Riak and Voldemort.

Tiramola's decision-making module is the unit that is responsible for materializing user defined policies into cluster-resizing actions. The user policies come in the form of reward functions that can evaluate the state of the cluster, and point Tiramola towards states that are in accordance to the user's needs. The state of the cluster is acquired by Tiramola's *Monitoring* module, which collects a number of metrics from both the cluster and the user, and makes them available to the decision-making module. Once a resizing action has been decided, the *Cloud Management* module communicates with the cloud provider as well as the virtual machines in order to modify and configure the cluster into its new state.

3.5.1 The Decision Making Module

Tiramola models the cluster as a Markov Decision Process (MDP). The states of the MDP correspond to the current size of the cluster.

$$S = \{s_{min}, s_{min+1}, ..., s_k, ..., s_{max}\} \tag{3.1}$$

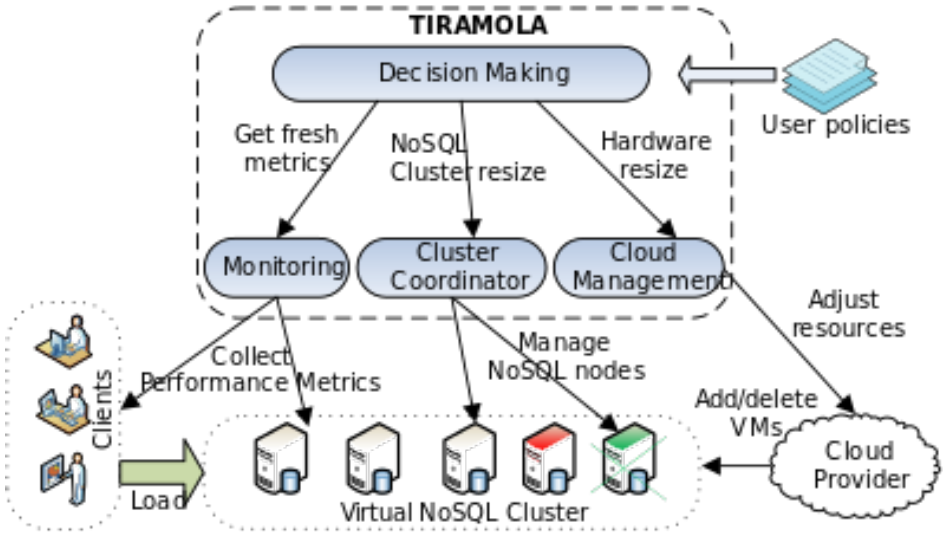


Figure 3.7: Tiramola's architecture

where k is the number of VMs currently in the cluster and min and max are the minimum and maximum cluster sizes. The available actions of the MDP are the resizing actions and include adding or removing pre-specified numbers of VMs, or simply leaving the cluster unmodified. If a certain resizing action would exceed the minimum or maximum cluster size if executed from a certain state, then that action is made unavailable at that state (for example if the minimum cluster size is four, an action that removes two VMs would not be available at state s_5).

In an MDP, the rewards are the feedback of the world towards the agent, that informs it how good or bad the outcome of an action was. In the case of Tiramola, the result of an action is the state of the cluster after executing that action. Therefore, the reward function was calculated using the resulting state after each transition. In order to achieve a balance between giving enough resources to satisfy the user's needs, but at the same time keeping the cost of the cluster as low as possible, the reward function generally included both positive and negative terms. For example, a reward function that aims to direct Tiramola towards performing actions that maximize the throughput and minimize the latency, while at the same time keeping the size of the cluster as low as possible, can be in the form:

$$r(s') = B \cdot throughput - C \cdot |VMs| - A \cdot latency \tag{3.2}$$

where A, B and C are appropriately chosen constants.

Since the actions that Tiramola performs add or remove VMs from the cluster, the transition function for each action is equal to 1 for a transition towards the state with the resulting number of VMs, and zero towards all other states. Whenever Tiramola needs to perform an action, it calculates $r(s')$ for all the states towards which a transition is possible. However, since the reward function generally depends on more parameters than the size of the cluster, those other parameters are identified beforehand using past experiences. Assuming that the system acts in a predictable manner, behaving similarly under similar conditions, only a subset of the past experiences is used. These experiences are selected by performing a k-clustering on all the past experiences for that specific number of VMs, and from those the centroid of the largest cluster is used as a representative point to calculate $r(s')$ (figure 3.8).

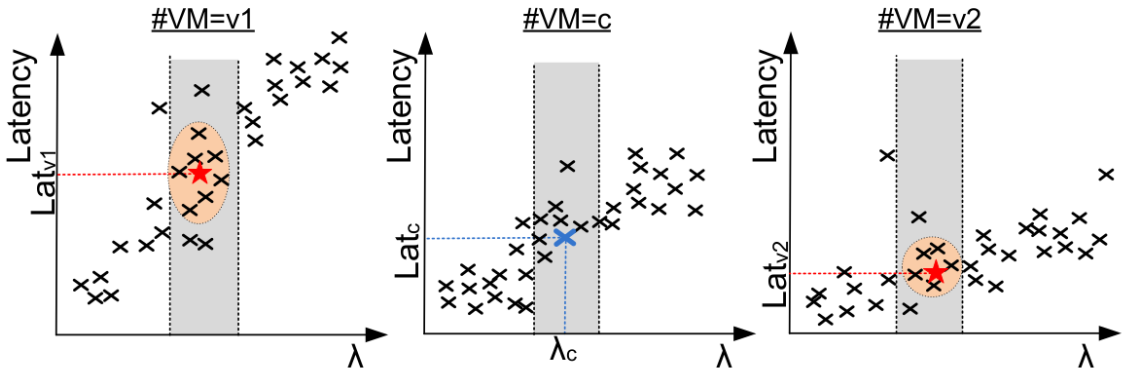


Figure 3.8: Example of Tiramola choosing the centroid of a clustering based on the value of the throughput λ [Tsou13]

Once a decision has been made and $r(s')$ has been calculated using the method described above, the Q-values of Tiramola's reinforcement learning model are updated using the standard Q-learning rule (explained in section 4.3.5):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r(s') + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{3.3}$$

where α is the *learning rate*, which controls how quickly the algorithm updates its knowledge according to new experiences, and γ is the *discount factor*, which controls how quickly future rewards are discounted.

Chapter 4

Reinforcement Learning

4.1 Introduction

4.1.1 The goal of Artificial Intelligence

The idea of a machine that can think far precedes the development of modern computers. However, thanks to the exponential growth of the capabilities of computers during the last few decades, this idea has turned from science fiction to an actual engineering goal, creating the field known as Artificial Intelligence.

Initially, it was believed that the way this goal would be reached was to create machines that mimic the function of the human brain. However, this proved to be much harder in practice than initially anticipated. Despite the significant progress in the field, the final goal of a machine that can think like a human kept moving further away, to the point that scientists today are less optimistic that the goal will be reached in the near future than they were a few decades ago.

Soon however, it was understood that an alternative path can exist. Instead of creating machines that think the same way humans do, it might be more realistic to create machines that can behave intelligently in a number of scenarios, despite "thinking" in a vastly different manner than humans. This approach allowed the development of algorithms that were much better suited for the capabilities of computers, and greatly expanded the applications of artificial intelligence to a number of practical problems.

4.1.2 Machine Learning

In that direction, Machine Learning emerged as the subfield of Computer Science and Artificial Intelligence that studies algorithms that can learn from data. The most widely used definition of learning was given by Tom M. Mitchell: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [Mitc97] . It is clear from that definition that Machine Learning focuses on the behavior of the system rather than the means through which that behavior is achieved.

There are three broad categories in Machine Learning, based on the way the algorithms learn. These are:

- *Supervised Learning:* In this category, the algorithm is provided with a set of labeled data, and attempts to infer the function that maps the data to the labels. The goal of the

algorithm is to generalize from the given data, and be able to correctly determine the labels of unseen instances.

- *Unsupervised Learning:* Here no labels are provided for the given data (nor any other type of reward signal), but instead the algorithm tries to find hidden structure in its input.

- *Reinforcement learning:* These are algorithms whose purpose is to take actions in an environment. After each action, the algorithm is provided with a reinforcement signal, as well as some indication of the new state of the environment. The goal here is to maximize the long term sum of values of the reinforcements.

4.2 Reinforcement Learning

4.2.1 Definition

In a standard reinforcement learning model, an agent is allowed to take actions within an environment. On each step, the agent is informed of the state of the environment and is requested to choose an action. As a result of that action, a scalar reinforcement signal is provided to the agent, and the state of the environment potentially changes. The agent attempts to choose actions that will maximize the reinforcements collected from the environment in the long run.

Formally, a reinforcement learning model consists of:

- A discrete set of states, $\mathcal{S}$

- A discrete set of actions, $\mathcal{A}$

- A set of scalar reinforcement signals

The agent is expected to find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes some long-run measure of reinforcement. In general, the environment can be non-deterministic, but is expected to be stationary.

4.2.2 The goal of Reinforcement Learning

In order to be able to develop algorithms that exhibit optimal behavior, we must first define what exactly optimality means. As mentioned above, the goal is to maximize the long term accumulation of rewards. The simplest interpretation of this concept is to attempt to maximize the sum of rewards the agent will receive throughout its life. The problem with this approach of course is that this sum is often infinite.

There are a number of ways in which we can avoid this problem. The simplest one is to attempt to maximize the rewards gained for only a finite number of steps.

$$E\left[\sum_{t=0}^{n} r_t\right] \tag{4.1}$$

Adopting this approach, we can choose to either consider a fixed or a receding horizon. A fixed horizon means that for each step the algorithms takes, the number of steps ahead that will be considered will be reduced by one. When this number reaches zero we assume that the algorithm will stop. This means that the policy that needs to be implemented is not a constant one, but it may vary depending on the number of steps left within the horizon. A receding horizon on the other hand means that at each step of the process the algorithm will consider the same constant amount of steps into the future, resulting into a constant policy throughout the agent's life. Unfortunately often times both these approaches are problematic. Not only is it not always possible to know the exact amount of steps for which the agent will be active, but in the case of the receding horizon it may even result in agents that never collect their rewards, thinking they will always be able to collect them at some point into the future.

Another way to define optimality is to attempt to maximize the average rewards gained per step.

$$\lim_{n\to\infty} E\left[\frac{1}{n}\sum_{t=0}^{n} r_t\right] \tag{4.2}$$

Again though, this approach suffers from not being able to distinguish between immediate rewards and rewards that can be obtained far into the future. In order for these problems to be avoided, the infinite sum of discounted rewards is often used as the quantity to maximize.

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{4.3}$$

By reducing the value of future rewards by a factor of γ^t (where $\gamma \in (0, 1)$ is known as the *discount factor*), we manage to bound the infinite sum to a finite value, while still considering rewards that lie far into the future. At the same time, we force the agent to collect any available rewards as soon as possible, which often leads to more reasonable behavior. Finally, following this approach gives us a simple and guaranteed to terminate algorithm to calculate the optimal policy.

4.2.3 Exploration vs Exploitation

An additional criterion by which an algorithm can be evaluated is the speed by which it converges to optimal. It is common in practical applications that there is a limit to the amount of time the algorithm can be given to learn. This makes the speed by which it learns critical to its performance.

Since the way through which the agent learns how to behave is by performing actions in the real world, there is often a choice to be made between exploring new options and exploiting known strategies. Exploiting the obtained knowledge will usually lead to higher immediate rewards, but exploring the world has a chance to reveal better options to exploit in the long run.

In order to find a balance between these two concepts, the idea of *regret* is often used. Regret is the expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally from the very beginning [Kael96]. Minimizing the regret results

in both attempting to adopt a near-optimal behavior from the beginning of the agent's life, as well as trying to converge to an optimal policy as soon as possible.

4.3 Markov Decision Processes

4.3.1 Markov Models

Markov Models are stochastic models used to describe non-deterministic processes. At any point in time, a Markov Model can be in one of a finite number of states, and with each time step the model has the ability to transition to a different state. The most characteristic property of a Markov Model is the fact that it is *memoryless* (also known as the Markov property). This means that the behavior of the system, as well as future transitions can depend only on the current state and not on the history of transitions.

Depending on the type of the system that is being modeled, Markov Models are divided into the following categories:

- *Markov Chains* are models used to describe autonomous observable systems. The purpose of the model is to extract information about the behavior of the system. They are used in a variety of scientific fields including but not limited to biology, chemistry, economics, information science and information technology.
- *Markov Decision Processes* are used to model observable systems where the behavior is partly random and partly affected by an action performed by an agent. They find uses in a number of fields including robotics, automated control, manufacturing and economics.
- *Hidden Markov Models* are statistical models used when the underlying process is assumed to be a Markov process whose states cannot be directly observed. They are used in speech, handwriting and gesture recognition, part-of-speech tagging and bioinformatics.
- *Partially observable Markov Decision Processes* are a generalization of Markov Decision Processes where the agent cannot directly observe the state of the system. Their uses include robot navigation, machine maintenance as well as a variety of other applications.

The most simple Markov Model is a Markov Chain. Mathematically, Markov Chains are defined as sequences of random variables that satisfy the *Markov Property*, which in this case is defined as:

$$\begin{aligned} &Pr\left(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, ..., X_n = x_n\right) = Pr\left(X_{n+1} = x \mid X_n = x_n\right) \\ &where\ Pr\left(X_1 = x_1, ..., X_n = X_n\right) > 0 \end{aligned} \tag{4.4}$$

Each such random variable can take values from a countable set, called the *state space* of the variable. Typically Markov Chains are represented using directed graphs, where the vertices are the possible values of a variable (its *states*), and the weighted edges hold the transition probabilities between those states (figure 4.1). *Markov Decision Processes* extend

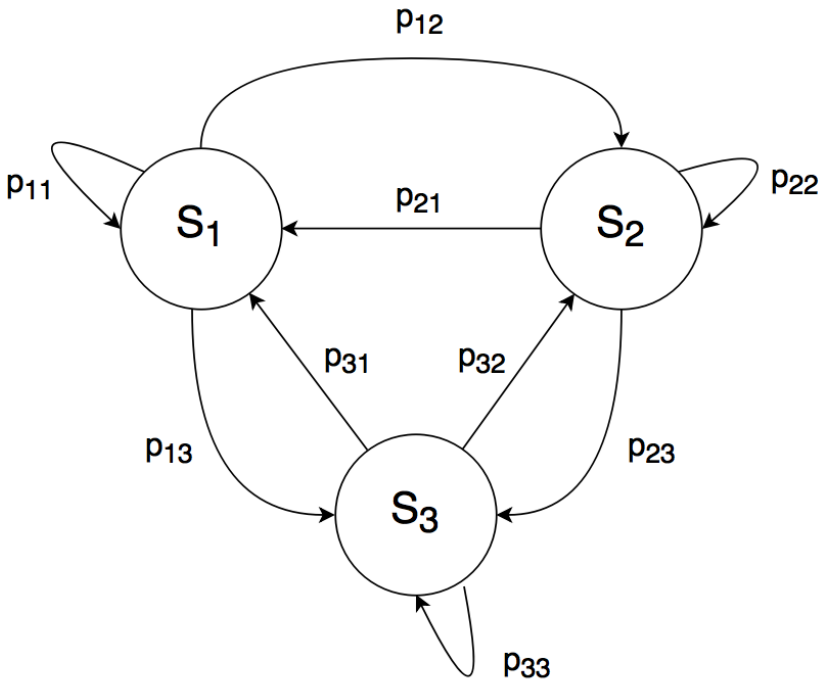


Figure 4.1: A simple Markov Chain with three states and nine transition probabilities between the states

Markov Chains by adding available *actions* to each state and *rewards* for executing those actions (figure 4.2). This allows Markov Decision Processes to be used to model environments where an actor is allowed to make decisions (actions), and the state of the world can be affected by those decisions. With these models it is then possible to calculate *optimal policies*, and thus build intelligent agents that can make strategic decisions within that context.

4.3.2 Optimal Policy Calculation

A Markov Decision Process (MDP) consists of the following:

- A set of states $\mathcal{S}$
- A set of actions $\mathcal{A}$
- A reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$
- A transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$

In order to be able to calculate an optimal policy for a given MDP, we assign optimal values to its states and actions. The optimal value of a state s, denoted by $V^*(s)$, is the expected sum of discounted rewards that an agent starting at state s would obtain under the optimal policy. The optimal value of an action a taken from state s, denoted by $Q^*(s, a)$, is the expected sum of discounted rewards that an agent would obtain after starting from state s, performing action a, and following the optimal policy thereafter.

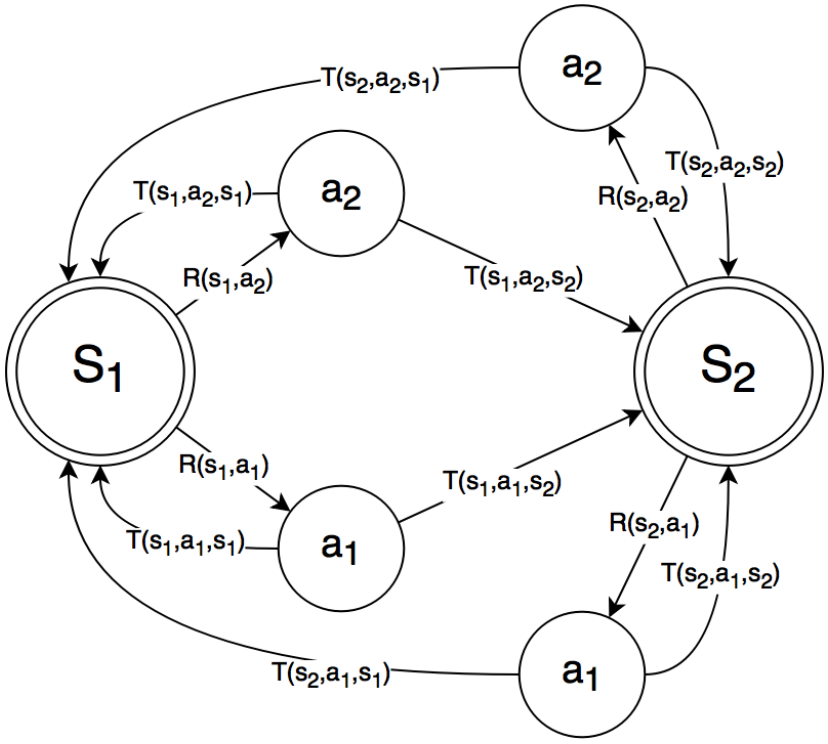


Figure 4.2: Graph representation of a simple Markov Decision Process with two states and two actions available in each state

$$V^*(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi^*(s_t))\right] \tag{4.5}$$

$$Q^*(s,a) = R(s,a) + E\left[\sum_{t=1}^{\infty} \gamma^t R(s_t, \pi^*(s_t))\right] \tag{4.6}$$

The expected sum of discounted rewards after executing action a from state s is equal to the immediate reward $R(s,a)$ gained by executing the action, plus the sum for every state s' of the probability of arriving at that state times the expected sum of discounted rewards we would obtain starting from that state, which is the optimal value $V^*(s')$ of the state, discounted by γ since the rewards would be obtained one step ahead in the future.

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') \tag{4.7}$$

The maximum expected sum of discounted rewards starting from state s are obviously obtained by executing the action with the highest optimal value.

$$V^*(s) = \max_{a \in A(s)} (Q^*(s,a)) \tag{4.8}$$

The optimal value therefore is the solution to the equation:

$$V^*(s) = \max_{a \in A(s)} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') \right) \tag{4.9}$$

The optimal policy, given the optimal values of the states, is:

$$\pi^*(s) = arg\max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) \tag{4.10}$$

We can calculate the optimal values of the states by a simple iterative algorithm based on equation (4.9), known as *value iteration*:

$$V_{k+1}(s) = \max_{a \in A(s)} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_k(s') \right) \tag{4.11}$$

In each step, a new estimate of the optimal value for the states is calculated based the the estimates of the previous step. The initial values can be chosen arbitrarily. This algorithm can be shown to converge to the correct optimal values [Bell57]. The algorithm stops when the value difference between two successive value functions in less than a value ϵ. The value of the greedy policy calculated at that point (the policy obtained by choosing, in every state, the action that maximizes the estimated discounted reward, using the current estimate of the value function) differs from the value function of the optimal policy by no more than $2\epsilon\gamma/(1-\gamma)$ [Will93a]. This means that we can use value iteration to calculate a policy that is arbitrarily close to the optimal, by choosing the appropriate value of ϵ.

Policy Iteration is an alternative strategy to value iteration that attempts to accelerate the computation of the optimal policy. Instead of calculating increasingly more accurate estimates of the optimal values of the states and Q-states, policy iteration breaks the computation into two phases. First, it calculates the state values under an arbitrary policy π. Since the policy is known this calculation does not require calculating a maximum over all actions, but instead uses the known action $\pi(s)$ in each state:

$$V_\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V_\pi(s') \tag{4.12}$$

The removal of the maximum operator simplifies the equations that connect the values of the states into linear equations that are solvable without an iterative process. After the values of the states under a given policy have been calculated, policy iteration attempts to improve that policy by choosing the optimal action in each state using those values:

$$\pi'(s) := argmax_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_\pi(s') \right) \tag{4.13}$$

This step is guaranteed to improve the policy, and when there are no possible improvements the policy is guaranteed to have converged to the optimal. However, there is no known tight worst-case bound in the number of iterations required for that convergence.

Algorithm 1 Value Iteration

initialize $V(s)$ *arbitrarily*
while *error* $>$ *max_error* **do**
    for $s \in S$ **do**
        for $a \in A$ **do**
            $Q(s,a) := R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$
        $V(s) := \max_a Q(s,a)$

Algorithm 2 Policy Iteration

choose an arbitrary policy π'
repeat
    $\pi := \pi'$
    compute the value function of policy π*:*
        solve the linear equations
            $V_\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s')V_\pi(s')$
    improve the policy at each state:
        $\pi'(s) := argmax_a(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_\pi(s'))$
until $\pi = \pi'$

4.3.3 Exploration Strategies

In a reinforcement learning context, since the only way the agent learns the environment is by performing actions, it is necessary that a strategy is implemented that forces the agent to perform all actions available. If that is not the case, the agent can get locked in attempting the first action that yielded a positive result again and again, missing out on opportunities to obtain better rewards. It is no surprise therefore that the greedy strategy of always choosing the most optimistic option is not very effective in practice.

To overcome this difficulty, a number of techniques have been proposed. One such useful heuristic is known as *optimism in the face of uncertainty*, in which actions are selected greedily, but strongly optimistic prior beliefs are put on their payoffs so that strong negative evidence is needed to eliminate an action from consideration. Of course, using this technique it is still possible to eliminate an optimal but unlucky action, but the risk can be made arbitrarily small.

Another simple approach to exploration strategies is to not always perform the best available action, but instead with a probability e perform an action at random. This probability can easily be adjusted throughout the life of the agent, so that at the start of the training where little is known about the system and a lot of exploration opportunities are available the probability of random actions is high, while as the agent learns the world and becomes more certain about its behavior the probability to perform random actions diminishes.

One alternative to the previous strategy is known as *Boltzmann exploration*, and it attempts to direct the choices of the agent towards more promising options instead of choosing actions at random. Under that strategy, the expected reward for taking action a, ER(a) is used to choose an action probabilistically, according to the distribution:

$$P(a) = \frac{e^{ER(a)/T}}{\sum_{a' \in A} e^{ER(a')/T}} \tag{4.14}$$

The parameter T is called the *temperature*, and it can be decreased over time to decrease exploration. This method however can perform poorly if the expected values of the actions are close, and the value of the temperature T needs to be carefully chosen in order to avoid converging unnecessarily slowly.

4.3.4 Learning from Experience

In section 4.3.2, we described how we can calculate the optimal strategy for a given MDP. For that purpose, we assumed that we know beforehand the behavior of the system, in the form of the given transition and reward functions. In many situations though, the behavior of the system is not known. In other words, we want to be able to build agents that figure out how the world works on their own, and not rely on the description of the world that was given to them. For that, we will see how an agent can still learn how to make optimal decisions even if the transition and reward functions are not known.

The only way for the agent to figure out an unknown environment is to perform actions in this environment and observe their rewards. We will assume that at any point in time, the agent is aware of the state of the world, and for every action he performs he is presented with a scalar reward as a feedback from the environment representing how good the outcome of the action was. As usual, the agent's goal will be to calculate a strategy that maximizes the infinite sum of discounted rewards.

There are two different approaches through which that strategy can be calculated.

- *Model-free*: In this approach the agent attempts to directly evaluate the effectiveness of the actions without learning the exact behavior of the world.
- *Model-based*: Here the agent tries to figure out the exact behavior of the world, in the form of the transition and reward functions, and then calculate the optimal strategy by using the methods we described previously.

4.3.5 Q-Learning

Perhaps the most common model free approach is an algorithm known as *Q-learning*. Q-learning maintains estimates of the Q-values (the values of the state-action pairs) and updates them with every new experience. As described earlier, the value of a state-action pair (s, a) is equal to the value of the immediate reward $R(s, a)$ obtained after performing action a from state s, plus the expected value of the resulting state discounted by a factor γ.

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s') = E\left[R(s, a) + \gamma V^*(s')\right] \tag{4.15}$$

After performing an action in the real world, the agent is presented with an immediate reward r, and observes the new state of the world s'. Based on that experience alone, he can calculate an estimate of the value of that action to be equal to:

$$q(s,a) = r + \gamma V(s') = r + \gamma \max_{a' \in A(s')} Q(s', a') \tag{4.16}$$

Using that estimate, we can update the value of the Q-state using both the previous estimate of the Q-value and the one deriving from the new experience.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha q(s,a) \tag{4.17}$$

The parameter α is known as the *learning rate* and controls how fast the agent learns from new experiences. Thus, the update rule for Q-learning is:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A(s')} Q(s', a') \right) \tag{4.18}$$

Having calculated the estimates of the Q-values, a policy can be calculated by simply choosing the action with the highest Q-value. If each action is executed in each state an infinite number of times on an infinite run and α is decayed appropriately, the Q-values will converge with probability 1 to Q^* [Watk89a] and so the policy calculated will be optimal. In practice, Q-learning manages to calculate an accurate estimate of the optimal strategy in a more reasonable amount of time.

Perhaps the strongest advantage of Q-learning is its computational efficiency. Since it does not construct a full model of the world, it only needs to store the values of the Q-states. Also, each update only involves calculating the maximum Q-value in the resulting state and performing a simple update on the value of the initial state. This not only allows Q-learning to run on systems with very limited computational power, but also allows it to train very fast from a large set of experiences if these are available.

On the other hand, Q-learning is limited to perform only local updates to the values. In each step, only the value of the performed action can be updated, and as a result the policy in only the initial state. The values and policies for all the other states will be unaffected, since no information concerning the probabilities of transitions is maintained, and they will only take into account the new experience once the agent finds itself again in one of those states and performs an action that causes a transition to the state whose value changed. As a result, Q-learning will require a large amount of experiences in order to converge to an optimal policy, especially in situations where the agent needs to perform a long sequence of actions in order to obtain a reward.

4.3.6 The Model-Based Approach

A different approach to the same problem is to attempt to construct a complete model of the real world and then calculate a policy using it. Again, the agent performs an action a from state s, receives a reward r and observes the new state s'. It then updates its estimates for $R(s,a)$ and $T(s,a,s_i)$. Typically $R(s,a)$ is set to the average reward obtained by taking action a from state s, and $T(s,a,s_i)$ is set to the number of times the system transitioned to

state s_i after taking action a from state s, divided by the total amount of times action a has been taken from state s.

After updating the estimates for R and T, the agent can use them to update his estimate of the optimal policy. Here there are a number of options depending on the amount of computation that is allowed in each step. The most straight forward approach is to simply run value iteration on the new model. This way the agent always has a fully updated estimate of the optimal policy and as a result converges to it after a significantly smaller amount of experiences. The problem of course is that in many situations this process is too computationally demanding. On the other hand, if the agent is required to act fast it can opt to update only $Q(s,a)$, $V(s)$ and $\pi(s)$. This strategy will result in very fast updates, but just as in the case of Q-learning, the updates will be local and will not affect the policy in any other state. As a compromise between the two above strategies, a number of algorithms have been proposed that partially update the model in a time much shorter than value iteration. Some of the most notable are *Dyna* and *Prioritized Sweeping / Queue-Dyna*.

In each step, Dyna [Sutt91] updates the Q-value for the state action pair that was performed using the rule:

$$Q(s,a) \leftarrow R(s,a) + \gamma \left(\sum_{s' \in S} T(s,a,s') \max_{a' \in A(s')} (Q(s',a')) \right) \tag{4.19}$$

It then chooses k additional state action pairs at random and performs the same update on them as well. As a result, the information about the results of recent actions is spread much faster to other states of the model, and the estimate of the optimal policy becomes more accurate. This translates into fewer required experiences for the agent to learn to behave optimally, at the cost of k times more operations per time step.

Prioritized Sweeping [Moor93] and Queue-Dyna [Peng93] were proposed as an improvement over Dyna. In these algorithms, instead of updating random state-action pairs in each step, the expected change in values is taken into account to direct the updates. Each state in the model remembers its *predecessors*, which are the states that have non-zero transition probability to that state under any action. Also, each state has a *priority*, which represents the importance of updating the value of that state, initially set to zero. When a new experience tuple $\langle s, a, s', r \rangle$ is received, the value of state s is stored as V_{old} and updated using the rule:

$$V(s) \leftarrow \max_{a \in A(s)} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V(s') \right) \tag{4.20}$$

The priority of state s is then set to zero and a value change $\Delta = |V_{old} - V(s)|$ is calculated. After that, the priorities of all the predecessors of s are set to $\Delta \cdot T(s_{pred}, a, s)$, unless their priority already exceeded that value. It then chooses to update the state with the highest priority, using a priority queue to efficiently select it. The process is repeated until k updates have been made, or the highest priority is below a threshold. As expected, this algorithms manages to maintain a better updated policy by directing the value updates where they are expected to have the most impact, thus converging faster to the optimal policy, at the cost of some extra computation and complexity.

Chapter 5

Decision Tree based Reinforcement Learning

5.1 Decision Trees

Decision Trees are a machine learning method used for the classification of previously unseen objects to one of a specified number of disjoint classes [Quin86]. Each object is assumed to possess a number of observable *attributes*, whose value belongs to a (usually small) set of mutually exclusive values. Each of these objects is also assumed to belong to one of a set of mutually exclusive *classes*. The algorithm is provided with a *training set* of objects whose classes are known. The goal is to develop a *classification rule* that is able to determine the class of any object from the values of its attributes.

The classification rule is typically expressed as a decision tree. Each node of a decision tree represents a decision made based on the value of a unique attribute. Each edge of the tree represents a unique value for the attribute of the parent node. The leaves represent classes that the object belongs to. In order for a new object to be classified, the algorithm starts from the root of the tree, and follows the edge that corresponds to the value of that specific attribute. The procedure is repeated until a leaf node is reached, at which point the class of the object is predicted to be the class of that node.

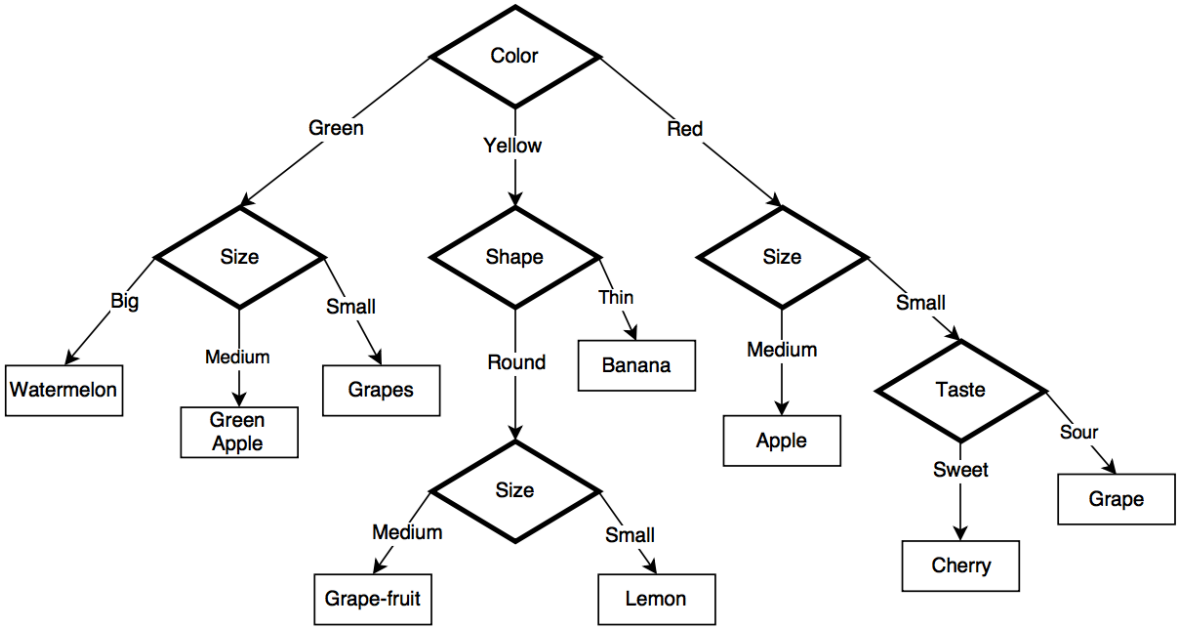


Figure 5.1: A simple decision tree

The purpose of an algorithm attempting to induce a decision tree, ideally, is to produce the simplest possible decision tree that accurately classifies the training set. That approach though is impractical, since the number of all possible correct decision trees is extremely large, and there is no known efficient algorithm that can identify the smallest one. To compensate for that, several algorithms have been proposed that can efficiently induce a reasonably simple (but not optimal) decision tree. One of the most widely known such algorithms is Quinlan's ID3 [Quin86]. ID3 iteratively builds a decision tree using the information gained from each split as the criterion to add additional decision nodes.

The expected average bits of information required to classify p objects belonging to class P and n objects belonging to class N is calculated as:

$$I(p,n) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n} \tag{5.1}$$

This definition is in line with Shannon's definition of entropy for a process with two outcomes with probability $\frac{p}{p+n}$ and $\frac{n}{p+n}$. Thus, the expected information for a tree A partitioning the state space into v categories can be calculated by:

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(p_i, n_i) \tag{5.2}$$

The *information gain* obtained by branching tree A into those subcategories is then given by:

$$gain(A) = I(p,n) - E(A) \tag{5.3}$$

ID3 examines all candidate attributes and chooses the attribute A that maximizes $gain(A)$. It then forms the tree, and repeats the process recursively to further grow the resulting subtrees. Quinlan later proposed an extension of ID3 known as C4.5, that can handle continuous features and incomplete data, and also provides a "pruning" technique to solve the problem of overfitting. In other works, Breiman proposed a criterion called the *Gini Criterion* [LBre84], later modified by Murthy [Murt94], that measures the probability of misclassifying a set of instances. Another criterion known as *Twoing Rule*, also proposed by Breiman, compares the number of examples per category on each side of the split.

5.2 Decision Tree based Q-learning

We described earlier how Markov Decision Processes can be used to perform decision making in classical reinforcement learning problems. One of the main characteristics of these models is that the system is always assumed to be in one of a finite number of states. There are a number of applications though, where the number of possible states the agent can find itself in is either extremely large, or even infinite. One such example is the problem of performing elasticity decisions on a database running in virtual machines in an IAAS cloud, where there

are numerous parameters affecting the behavior of the system (size and characteristics of the machines, type and magnitude of the incoming load, CPU utilization, RAM utilization etc).

In these types of problems, classical approaches that attempt to model the system as a Markov Decision Process cannot work since the number of required states grows exponentially with the number of parameters that define it. This does not only make the resulting models difficult to solve, but also greatly increases the amount of experiences needed in order to approximate an optimal behavior. This problem is also referred to as *input generalization*.

In order to tackle this problem, Chapman and Kaelbling proposed a modification of the classical Q-learning approach that uses a decision tree to partition the state space into meaningful regions, called the G algorithm [Chap91]. The problem they faced was writing an agent that could control a character in a video game called Amazon. The game involved moving the character in a two dimensional map and shooting projectiles at ghosts to kill them. The difficulty was that the input to the algorithm was the on screen representation of the game, in the form of an array of a few hundreds of black or white pixels. This meant that there were more than 2^{100} different states the system could be in. It was obvious that attempting to represent this problem as a classical Markov Decision Process was not realistic.

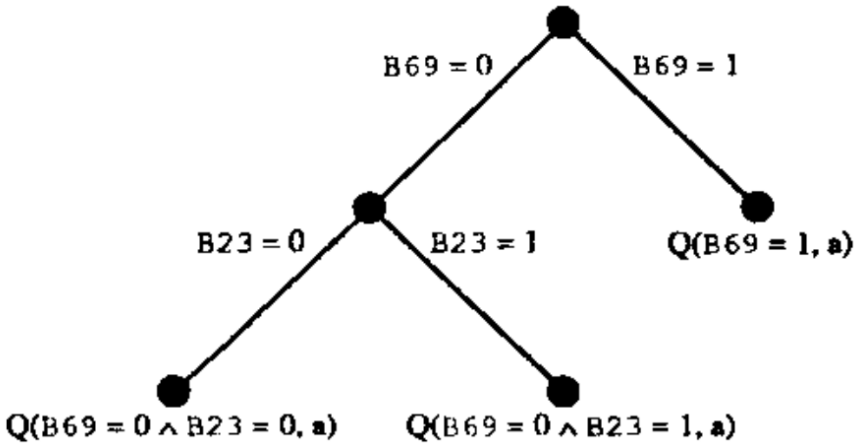


Figure 5.2: An example of the G algorithm partitioning the state space based on two bits of the input [Chap91]

The G algorithm partitions the state space by creating a decision tree based on the bits of the input. For each bit, the algorithm keeps separate track of the value of the state for the occurrences where the bit is 0 or 1. It then uses the *Student's t test* to determine when a bit is relevant. The *t* test determines the probability that two sets of data are samples of the same distribution. If that is the case, then the bit is irrelevant to the behavior of the system and there is no point splitting the state based on that bit. If the bit is found to be relevant, the algorithm splits the state into two different states, corresponding to the bit being 0 and 1. The information stored in the bits of the initial state is not transfered to the new states, since at that point it is not known on which of the two new states this information should go. This means that the algorithm throws away a lot of useful information as the tree grows. At the same time, the requirement for the input to be a string of bits can be restrictive in the types of problems this algorithm can be applied.

One more widely applicable approach was proposed by Larry D. Pyeatt and Adele E.

Howe in [Pyea01]. Their work was focused on developing agents for two simulated robotics environments: a Robot Automobile Racing Simulator and a desktop robot called Khepera. The state space for both these problems was too large for a classical table lookup method to be applied, and so a method to generalize over the input was necessary. For that purpose they proposed a decision tree based reinforcement learning algorithm that can work on continuous state spaces.

The core of the algorithm is based on Q-learning. For each state-action pair, a Q-value is maintained. When an action is performed and a new experience is acquired, the difference in Q-value is calculated as:

$$\Delta \leftarrow \alpha \left[r_{t+1} + \gamma \max_{\alpha} Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{5.4}$$

where α is the learning rate and γ is the discount factor. The states of the algorithm are organized in a binary decision tree in a way similar to the G algorithm. The states of the Q-learning model lie in the leaves of the tree, and the internal nodes of the tree are decision nodes. Each decision node represents a single decision about the value of one input variable. When presented with an input vector v, the state that v belongs to is found by starting from the root of the tree and descending towards the leaves, following each decision branch according to the values in v.

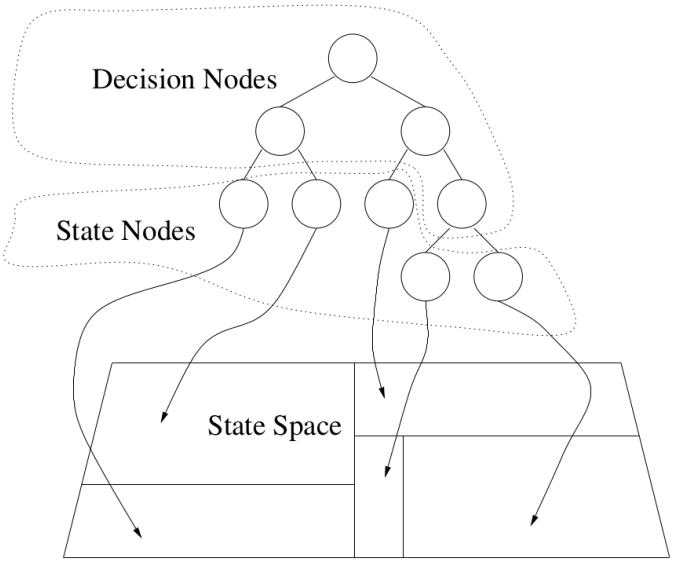


Figure 5.3: State space partition using a decision tree [Pyea01]

The algorithm starts with a single state representing the entire state space. Whenever an action is performed from a state, the difference in Q-values Δ is calculated and the Q-value is updated by $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \Delta$, according to the Q-learning rule. That Q-value difference Δ is then stored in a history list for that specific state-action pair, along with the exact values of the input vector v.

Following that, the algorithm decides if it should split the state. In order for a state to qualify for splitting, two tests are made. First, the length of the history list for that state needs to be at least equal to a *history_list_min_size* variable. Second, the standard deviation σ and

average μ are calculated for the Q-value differences Δ in the history list. The split can then proceed only if $|\mu| < 2\sigma$.

The split itself can happen based on a number of criteria. These include the Information Gain criterion proposed by Quinlan's ID3 [Quin86], the Gini Index metric [LBre84], the Twoing Rule [Murt94] and the *t* statistic. In the case of the *t* statistic, the history list l for that specific state-action pair is divided into two lists l_- and l_+ corresponding to the experiences where the Q-value difference Δ was negative or positive. For each input variable v_i, a *t* statistic is calculated on the values of the variable in l_- and l_+. The variable with the highest t-statistic is chosen for the split as long as the value of the statistic is higher that 0.1. The splitting point will be the midpoint between the average values of the variable in l_- and l_+.

The algorithm was tested on two classical reinforcement learning problems, a car trying to climb onto a mountain having to drive back and forth to gain the needed speed, and a pole balancing on a cart that moves in one direction, as well as a simulated race driver car that races against other cars. The tests supported the superiority of the t-test over the other splitting criteria. Also, they showed the algorithm to be superior to other approaches such as table-lookup and neural networks for these types of settings.

One downside of their approach however, is that when a node is split, the information stored in that node is thrown away. Considering that in each split the number of decision and state nodes increases by exactly one, at any point in time the number of decision nodes will be one less than the state nodes (since the tree starts from a single state node). All those decision nodes replaced state nodes that got split, discarding their training data. If we assume that each active state node contains on average half of the information that was discarded in each split, since a split has not been decided for it yet, we can conclude that during the training of the model it is possible to waste up to 2/3 of the training data on removed states. This is the reason that in our approach, explained in detail in section 5.4, we put great effort into reusing past experiences to retrain the new states that are produced when a state is split.

5.3 Continuous U Tree

The full-model based approach has certain advantages to offer over the model-free approach in the case of the decision tree based algorithms as well. Since more precise information is maintained about the exact properties of the model (its reward and transition functions), this information can also be used to improve the quality of the splits that the algorithm performs.

One such full-model, decision tree based algorithm was proposed by W. Uther and M. Veloso in [Uthe98], named *Continuous U Tree*. In a similar manner to the algorithms studied in section 5.2, a decision tree is used to partition the state space. The tree consists of decision nodes, that partition the state space based on the value of a parameter, and leaf nodes, that are the states of the Markov Decision Process.

Continuous U Tree goes through two distinct phases. The first phase is called *Data Gathering Phase*. In this phase the algorithm works like a traditional Markov Decision Process with a fixed number of states, with the only difference being that the mapping of input vectors to the state they belongs within the model is done using the decision tree. Additionally, all

experience tuples (I, a, I', r) acquired during this phase are recorded. Since during this phase the algorithm functions like a traditional MDP, updates to the state and Q-state values can be done using any update algorithm applicable to traditional MDPs.

The second phase is called a *Processing Phase*. During this phase the algorithm goes through the leaves of the decision tree one by one, and calculates the values for each of the datapoints in that leaf by the rule:

$$q(I, a) = r + \gamma V(s') \tag{5.5}$$

It then sorts the datapoints based on the value of each attribute and checks each point between two such consecutive values as a possible splitting point. Two splitting criteria were proposed:

- Using Kolmogorov-Smirnov test on the values of the datapoints on either side of the splits. The test will calculate the probability that the two sets of values follow the same distribution.
- Using the values of the states $Q(s, a)$ to approximate the values of the transitions $q(I, a)$, and calculating the resulting mean-squared error. The splitting criterion is then the weighted sum of the variances of the Q-values of the transitions on either side of the split.

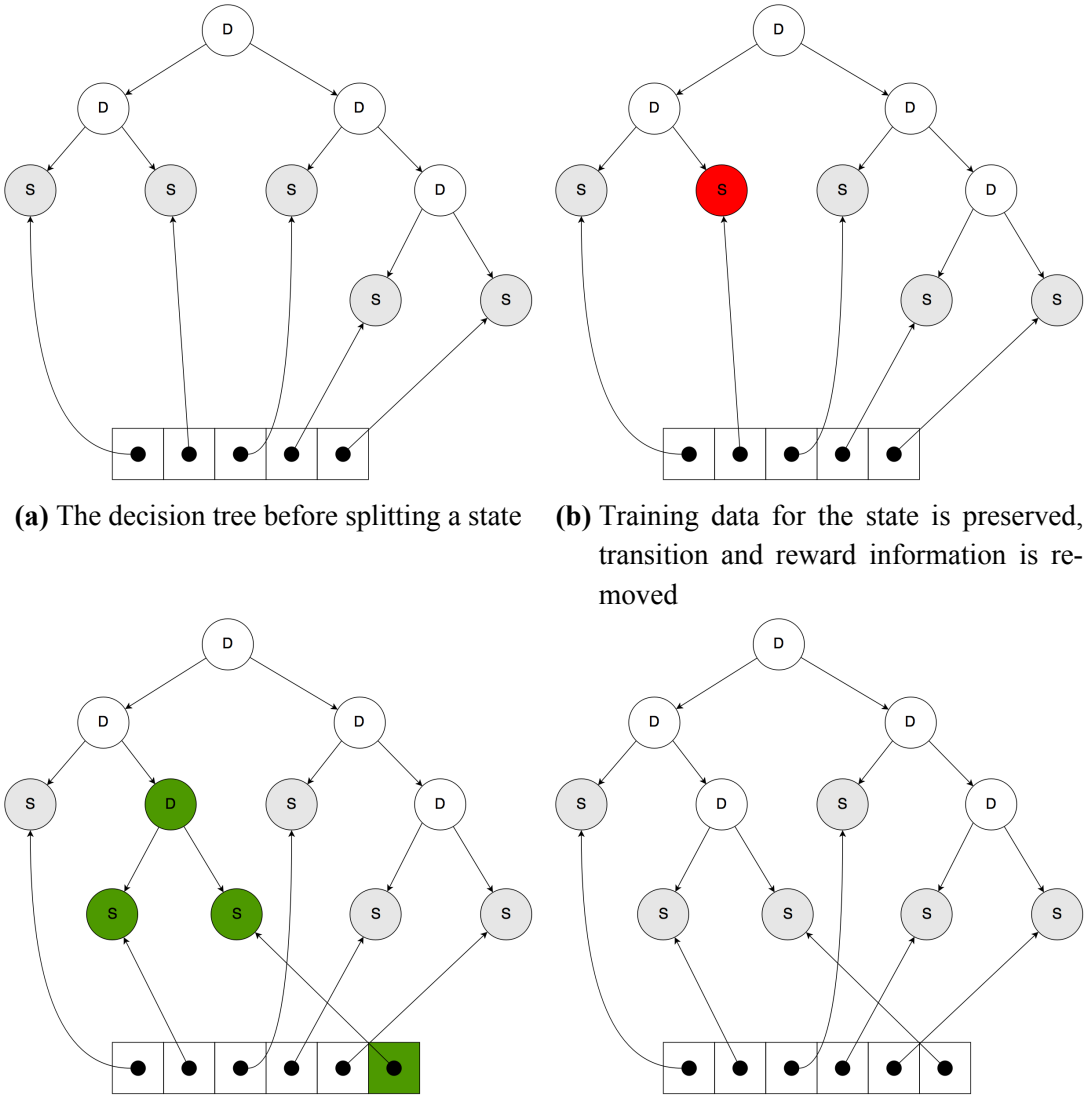
After the discretization has been decided, the transition and reward functions for all the states of the model have to be recalculated from the stored experience tuples. The new MDP is then solved using value iteration to calculate the state and Q-state values. The algorithm keeps alternating between the Gathering and the Processing phase, growing the decision tree and the number of states during the Processing phase, and acting as a standard MDP during the Gathering phase.

5.4 Description of our Implementation

5.4.1 Overview

In this section we will explain in detail the way our implementation works. The algorithm starts with a single leaf node, which is a state representing the entire state space. That node is initialized as the root of the decision tree. At the same time, a vector of all states is maintained. As required for a Markov Decision Process, a list of Q-states is stored in each state, with each Q-state holding the number of transitions and sum of rewards towards each state in the model, along with the total number of times the action has been taken. From this information we can easily calculate the transition and reward functions.

The state of the world is provided in the form of a set of measurements, containing the names and current values for all the parameters of the system. The last set of measurements m collected, is stored in the model along with the state s they correspond to. When an action is performed, the algorithm is presented with the name of the action a, the reward obtained r, and a new set of measurements m' representing the new state of the world. From m' we obtain the new state of the system s' using the decision tree. After updating the transition and reward



(a) The decision tree before splitting a state

(b) Training data for the state is preserved, transition and reward information is removed

(c) The state is replaced by a decision node and two new state nodes

(d) Training data is restored, transition and reward information is recreated from the data

Figure 5.4: Splitting a state in the decision tree into two new states. The first state replaces the old state in the states vector and the second one is appended at the end.

information for the (s, a) Q-state, we store the $\langle m, a, m', r \rangle$ experience tuple within s. Each state holds a separate history list for each of the states in the model, and so the $\langle m, a, m', r \rangle$ experience tuple is stored within s in the list that corresponds to s'.

Next, we update the state and Q-state values for our model. Since we maintain all the current states in the model in a single vector, all states are referenced to by their position in the vector. This means we can easily run any update algorithm used in classical Markov Decision Processes to update the values, including value iteration, policy iteration and prioritized sweeping. After updating the values we consider splitting state s into two new states, and finally update the current state s and current measurements m to s' and m'.

5.4.2 Splitting Criteria

We tested the following criteria for splitting a state:

i. *Parameter test:* From the experiences $\langle m, a, m', r \rangle$ stored in all the history lists of s, we isolate the experiences where the action a was the optimal action for that state (the one with the highest Q-value). For each of these experiences, we find the state s' in the current model that corresponds to m' using the decision tree, and calculate the value $q(m, a) = r + \gamma V(s')$. We then create two lists l_- and l_+, and if $q(m, a) < V(s)$ we append m to l_-, else we append it to l_+. If the length of either l_- or l_+ is less that a parameter $min\_num\_experiences$ we abort the split.

   For each parameter of the system, we collect the values of that parameter for the measurements in l_- and l_+ in two new lists p_- and p_+, and run a statistical test on p_- and p_+ to determine the probability that the two samples have arisen from the same population. We choose the parameter with the lowest such probability to perform the split, as long as it is lower that a $max\_type\_I\_error$ parameter, else we abort. If the split proceeds, the splitting point will be the average of the means of p_- and p_+.

   This splitting criterion is analogous to the one proposed in [Pyea01]. The two main differences are that the q-value derived from each experience is calculated using the current states of the system instead of the value of the resulting state at the time the action was performed, and the partitioning of the experiences is done by comparing them to the current value of the state s instead of partitioning them to experiences that increased or decreased the Q-value (which is again equal to comparing them to the Q-value at the time the experience was acquired).

ii. *Q-value test:* Again, from the experience tuples $\langle m, a, m', r \rangle$ stored in all the history lists of s, we isolate the experiences where the action a was the current optimal action for s. For each experience, we find the state s' that corresponds to m' using the current decision tree and calculate $q(m, a) = r + \gamma V(s')$.

   For each parameter p of the system, we calculate all the tuples $\langle m[p], q(m, a) \rangle$ and sort them based on the value of the parameter $m[p]$. For each two consecutive unequal values of the parameter $m_i[p]$ and $m_{i+1}[p]$ in that list, we consider splitting the state at the midpoint $\frac{1}{2}(m_i[p] + m_{i+1}[p])$. For that purpose, we run a statistical test on the sets of instantaneous Q-values $q_- = \{q(m_k, a) \mid k \leq i\}$ and $q_+ = \{q(m_k, a) \mid k > i\}$. In other words, we consider every midpoint between two consecutive measured values of a parameter, and run a statistical test on the instantaneous q-values below and above that threshold. If the split would leave less experiences on either side that $min\_num\_experiences$, we ignore it. We choose the splitting point that gives the lowest probability that the two sets of values are statistically indifferent, as long as that probability is less than $max\_type\_I\_error$.

   This criterion was used in [Uthe98], and resembles splitting criteria that are being used in traditional decision tree algorithms such as C4.5. It is also probably the most conceptually straight-forward criterion, comparing how good the action was on either side of the split.

Alternatively, instead of attempting to split the state between each two consecutive unequal measurements, we experimented with only attempting to split in the midpoint between the two unequal consecutive measurements that are closest to the median. This way we only consider a single splitting point per parameter, that splits the recorded experiences approximately equally in the two resulting states.

iii. *Information Gain:* This criterion is based on ID3's splitting criterion and was also tested in [Pyea01]. In our implementation of the criterion, we collect the experience tuples stored in the history lists of s where the action a was the optimal action, and calculate the values $q(m, a) = r + \gamma V(s')$ for each one.

Then, similarly to the *Q-value test* criterion, for each parameter of the system we sort the experiences based on the value of that parameter and consider as splitting points each midpoint between two unequal consecutive values of the parameter. We count the experiences where $q(m, a) < V(s)$ and $q(m, a) \geq V(s)$ on either side of the split, and calculate the expected classification information in the resulting subtrees using equations 5.1 and 5.2. We choose to split at the point that minimizes this expected information, as long as it is lower than the expected information for the initial state (again calculated using equation 5.1) minus a min_info_gain parameter.

Alternatively, similarly to the Q-value test criterion, we experimented with attempting to split the state only at the midpoint between the two unequal consecutive measurements that are closest to the median.

5.4.3 Performing the split

Once a split has been decided for a state s_i, all transition and reward information needs to be removed from the rest of the states in the model. Since we know the id number of s_i, this is done by just zeroing the i-th element on all the reward and transition vectors in all the Q-states in the model.

Before the node is to be discarded, the stored experiences need to be preserved. The history lists stored in s_i hold all recorded experiences where s_i is the starting state. These history lists are merged into a single one and stored temporarily in the model. All recorded experiences where s_i is the resulting state are stored in the i-th history list in some other state of the model (the starting state for that experience). All those lists are emptied and their contents are also stored in the temporary history list.

The next step is to replace s_i in the decision tree with a new decision node, holding references to two new children states. The first of these states will take the place of s_i in the states vector, and the second will be appended at the end. The reward and transition vectors in all the states of the model are extended with one zeroed element, and a new empty history list is appended in all states.

Finally the experiences temporarily stored are used to retrain the new states. For each $\langle m, a, m', r \rangle$ experience tuple, the corresponding states s and s' are found using the decision tree, the reward and transition vectors within s are updated, and the experience is stored again in the history list of s that corresponds to s', in a process similar to the one that happens when a new experience tuple is acquired.

One simple extension to the splitting mechanism, is allowing multiple splitting points. Even though all the criteria we tested split a state into exactly two new states, being able to perform splits at multiple splitting points allows for the easy and efficient construction of pre-defined decision trees. This is very useful in applications where some knowledge is available about the state space, and thus starting the model with a single node is unnecessarily pessimistic. When splitting a node in multiple points, the additional states are simply appended at the end of the states vector, and all other changes are performed accordingly.

5.4.4 Statistical Tests

Three out of four splitting criteria discussed in section 5.4.2 included a statistical test to determine whether the two groups of compared values are statistically different from each other. For that purpose, four different statistical tests were used.

- *Student's t-test:* The equal variance *t*-test, widely known as *Student's t-test*, estimates the probability that the two compared samples have a different mean, under the assumption that they share the same variance. The statistic for this test is calculated using the formula:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{s_{X_1X_2} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{5.6}$$

  where $s_{X_1X_2}$ is an estimator of the common standard deviation of the two samples given by:

$$s_{X_1X_2} = \sqrt{\frac{(n_1 - 1)s_{X_1} + (n_2 - 1)s_{X_2}}{n_1 + n_2 - 2}} \tag{5.7}$$

  The quantity $n_1 + n_2 - 2$ is the total number of degrees of freedom. This test was also used by [Pyea01], and is a very common way to test the similarity of two samples.

- *Welch's test:* The unequal variance *t*-test, also known as *Welch's test*, is an alternative to the Student's *t*-test that also tests whether the population means are different, but without assuming that they share the same variance. The statistic in this case is given by:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5.8}$$

  where s_1 and s_2 are the unbiased estimators of the variance of the two samples. The degrees of freedom for this test are given by the *Welch-Satterthwaite equation*:

$$v = \frac{\left(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}\right)^2}{\frac{s_1^4}{N_1^2 v_1} + \frac{s_2^4}{N_2^2 v_2}} \tag{5.9}$$

  This test has been proposed [Ruxt06] [Coom96] as the default way to test the statistical similarity of two samples when the equality of the variances is not known beforehand, over the *Student's t-test*.

- *Mann Whitney U test:* This test is also an alternative to the *t*-test that does not require the assumption that the two populations follow a normal distribution, and can be used on both ordinal and continuous data. It involves calculating a U statistic, whose distribution under the null hypothesis in known (for sample sizes above 20 a normal distribution can be assumed).

  If the compared samples are small, it can easily be calculated by making every possible comparison between the elements of the two groups, and counting the amount of times the elements of each group win (giving 0.5 to each group for ties). For larger samples the statistic can be calculated by ranking all the elements of the two groups in increasing order based on their value, adjusting the ranks in case of ties to the midpoint of unadjusted rankings, and summing up the ranks in the two groups (for example the ranks of (3,5,5,9) are (1,2.5,2.5,4) [Wiki15b]). The U statistic is then given by:

  $$U_1 = R_1 - \frac{n_1(n_1+1)}{2},\ U_2 = R_2 - \frac{n_2(n_2+1)}{2} \tag{5.10}$$

  where R_1 and R_2 are the sums of ranks for the samples 1 and 2. The minimum value among U_1 and U_2 is then used to consult the significance table.

- *Kolmogorov-Smirnov test:* The two sample Kolmogorov-Smirnov test can be used to test whether two underlying one-dimensional probability distributions differ, without assuming normality for the two distributions. It involves the calculation of the Kolmogorov Smirnov statistic:

  $$D_{n,n'} = sup_x |F_{1,n}(x) - F_{2,n'}(x)| \tag{5.11}$$

  where $F_{1,n}$ and $F_{2,n'}$ are the *empirical distribution functions* of the two samples, and sup is the *supremum function* [Wiki15a]. The *null hypothesis* is rejected at level α if:

  $$D_{n,n'} > c(\alpha)\sqrt{\frac{n+n'}{nn'}} \tag{5.12}$$

  where $c(\alpha)$ is the inverse Kolmogorov distribution at α. This statistic was used for performing splits in [Uthe98].

Chapter 6

Simulation Results

In this section we present results from a number of simulations. The goal of the simulations is to better understand the behavior of the reinforcement learning models described in Chapters 4 and 5, in the context of resource allocation problems in a cloud computing environment. Choosing the exact setup for each simulation was a challenging task, not only because the algorithms discussed allow for a great deal of parameterization, but also because a fine line has to be maintained between capturing the behavior of the algorithm in a realistic setting and maintaining some level of simplicity such that the results of the experiments can be interpreted.

In section 6.1 we run a number of simulations whose purpose is to understand the behavior of the algorithms under different possible settings. Since the amount and diversity of the available options was overwhelming, in most experiments we attempt to isolate one of those options and study its effect on the algorithm while using reasonable but constant values for everything else. In section 6.2, we attempt to compare the performance of the algorithms discussed in this work in two cloud management scenarios, and draw conclusions for the effectiveness of each solution in this type of application.

The following abbreviations will be used throughout this chapter:

- *MDP:* The full-model based Markov Decision Process approach, having a fixed number of states and maintaining transition and reward information in its Q-states, covered in section 4.3.6
- *Q-Learning:* The model-free reinforcement learning approach, also having a fixed number of states but not maintaining transition and reward information, covered in section 4.3.5
- *MDPDT:* Our full-model based decision tree implementation covered in section 5.4
- *QDT:* The Q-learning decision tree algorithm proposed in [Pyea01] and covered in section 5.2

All simulations were implemented in *Python*, and the *SciPy* library was used for the statistical tests.

6.1 Parameterization

Before attempting to evaluate the performance of our proposal, we experimented with a number of different options that affect that performance. For that purpose, we used a simu-

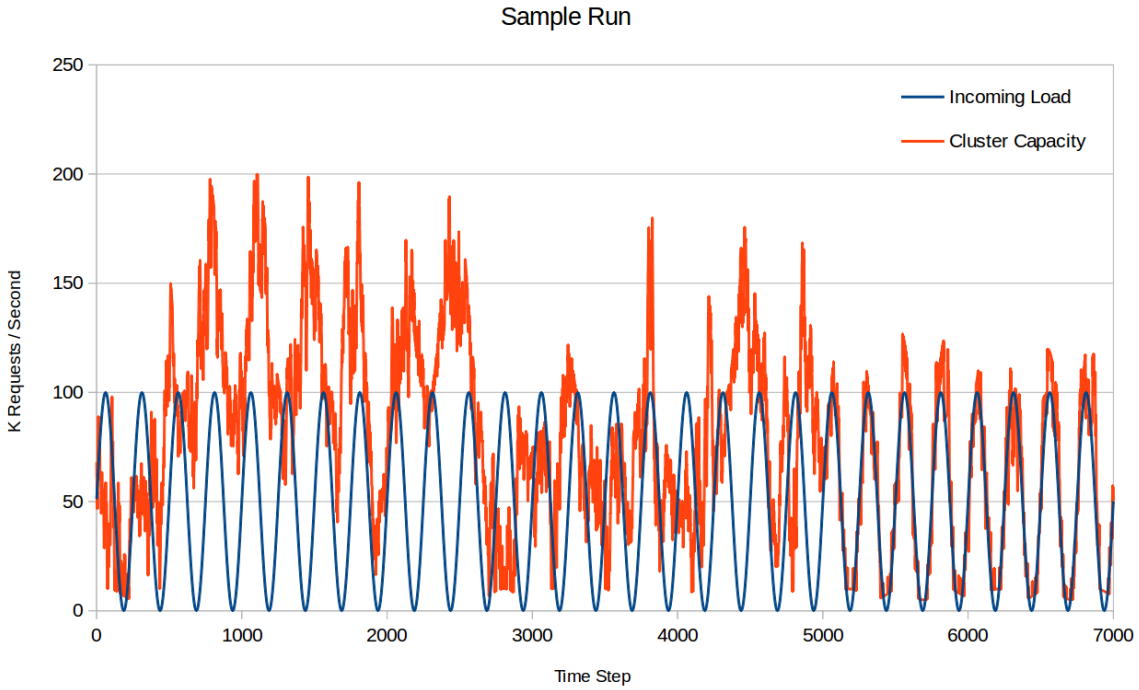


Figure 6.1: Incoming load and cluster capacity in a sample run with 5000 training steps, 2000 evaluation steps and $e = 1.0$

lation scenario from the field of cloud computing. In this scenario, the the agent is asked to make elasticity decisions that resize a cluster running a database under a varying incoming load. The load consists of read and write requests, and the capacity of the cluster depends on its size as well as the percentage of the incoming requests that are reads. Specifically:

- The cluster size can vary between 1 and 20 virtual machines.
- The available actions to the agent in each step are to increase the size of the cluster by one, decrease the size of the cluster by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) = 50 + 50sin\left(\frac{2\pi t}{250}\right)$
- The percentage of incoming requests that are reads is a sinusoidal function of time with a different period: $r(t) = 0.75 + 0.25sin\left(\frac{2\pi t}{340}\right)$.
- If $vms(t)$ is the number of virtual machines currently in the cluster, the capacity of the cluster is given by: $capacity(t) = 10 \cdot vms(t) \cdot r(t)$.
- The reward for each action depends on the state of the cluster after executing the action and is given by $R_t = min(capacity(t+1), load(t+1)) - 3 \cdot vms(t+1)$.

The reward function encourages the agent to increase the size of the cluster to the point where it can fully serve the incoming load, but punishes it for going further than that. In order for the agent to behave optimally, it needs to not only identify the way its actions affect the cluster's capacity and the dependence on the level of the incoming load, but also the dependence on the types of the incoming requests.

In order to test the algorithm's ability to partition the state space in a meaningful manner, apart from the three relevant parameters (size of the cluster, incoming load and percentage of reads) the input vector included 7 additional parameters, whose values varied randomly. Four of them followed a uniform distribution within $[0, 1]$, while the rest took integer values within $[0, 9]$ with equal probability. In order to be successful, the algorithm needs to partition the state space using the three relevant parameters and ignore the rest.

All tests included a training phase and an evaluation phase. During the training phase, the selected action in each step was a random action with probability e, or the optimal action with probability $1 - e$ (e-greedy strategy). During the evaluation phase only optimal actions were selected, as proposed by the algorithm. The metric through which different options are compared is the sum of rewards the agent managed to accumulate during the evaluation phase.

6.1.1 Statistical Significance

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT
- Splitting Criterion: Q-value test (Midpoint), Q-value test (Multiple Points), Parameter test
- Initial Decision Tree: Single State
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$
- Statistical test: Student's *t*-test, Welch's *t*-test, Mann-Whitney U test, Kolmogorov-Smirnov test
- Statistical test max error $\in \{0.05, 0.02, 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001, 0.00005, 0.00002, 0.00001\}$
- Minimum number of experiences to split: 4 per resulting state

All the statistical tests used attempt to estimate the probability that the two compared samples are statistically indifferent. If that is the case, performing a split at that point would be meaningless. Therefore, we only perform splits in points where the statistical test suggests a probability of error lower than $max\_type\_I\_error$. Higher values of that parameter translate to less strict testing, while lower values translate to stricter testing.

In this scenario, only 3 out of 10 parameters provided to the algorithm affected the system, and the rest varied randomly. The algorithm is expected to correctly distinguish the parameters that are relevant and only perform splits using them. Figures 6.2, 6.3 and 6.4 show the

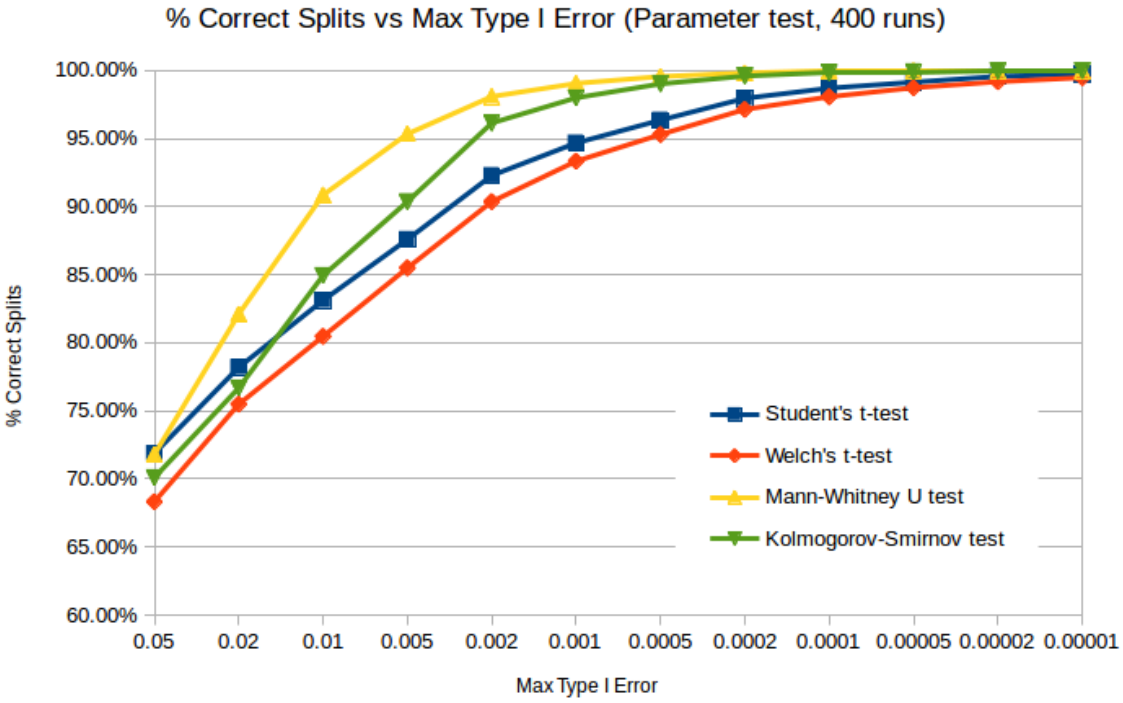


Figure 6.2: Accuracy of the four statistical criteria using the Parameter test, for different values of the maximum type I error

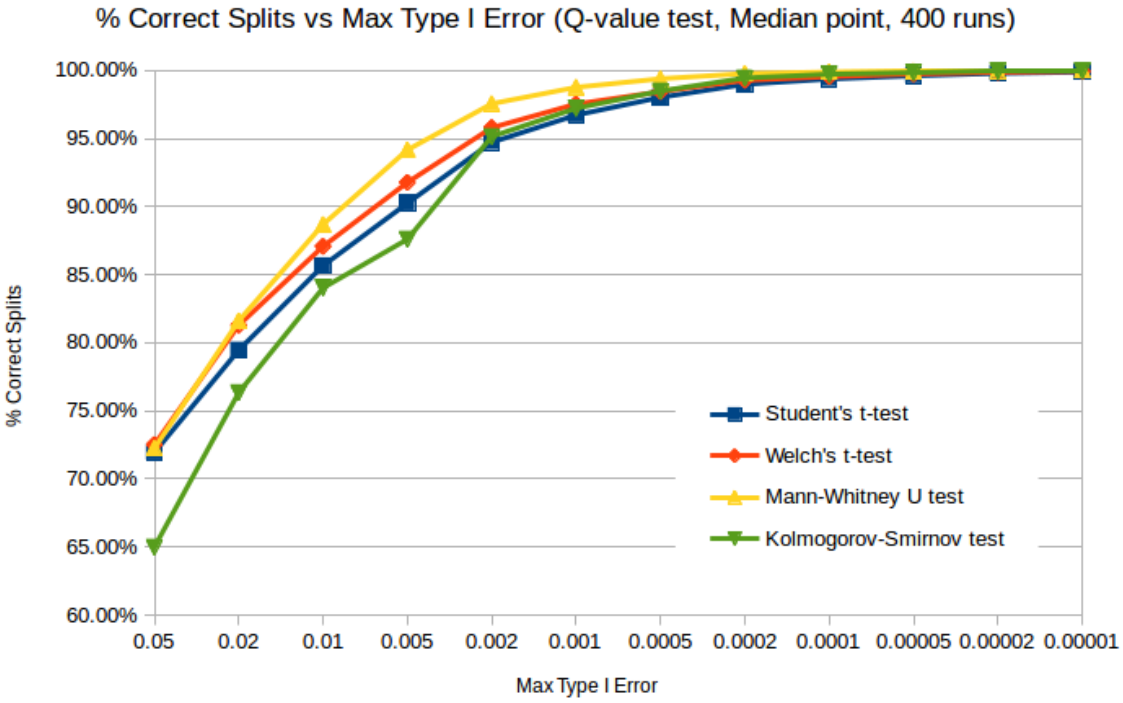


Figure 6.3: Accuracy of the four statistical criteria using the Q-value test at the median point, for different values of the maximum type I error

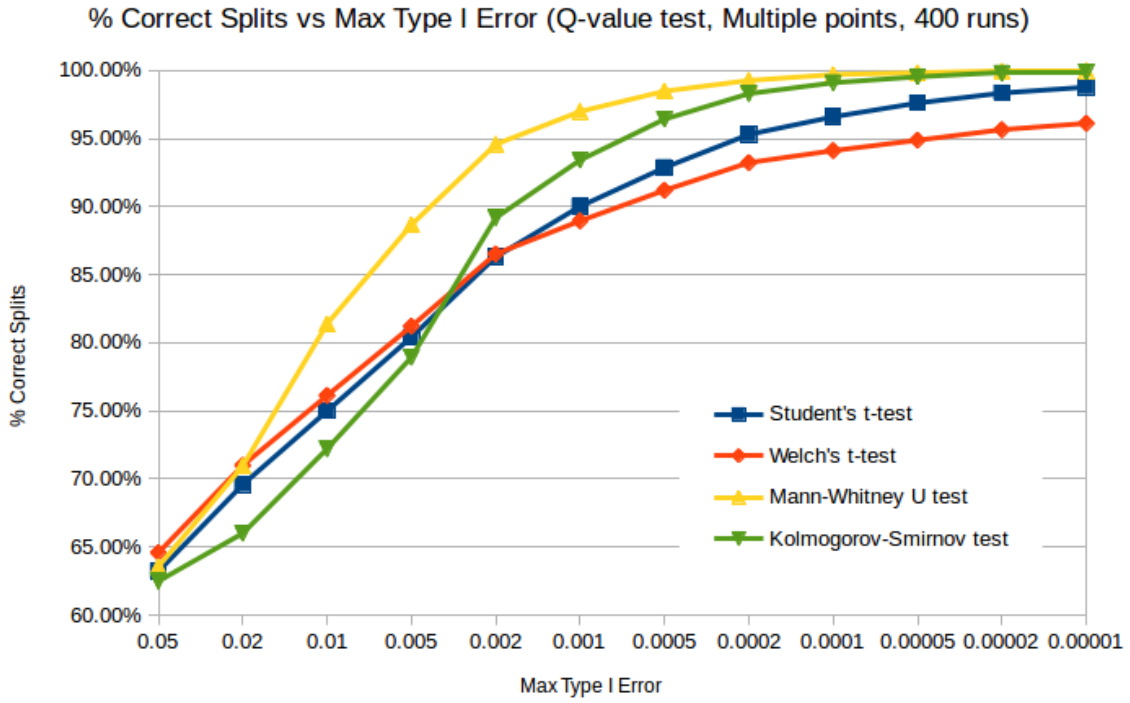


Figure 6.4: Accuracy of the four statistical criteria using the Q-value test at multiple points, for different values of the maximum type I error

percentage of splits performed on those relevant parameters as a function of the maximum statistical error. For all statistical tests and splitting criteria, the typically used margin of 0.05 (corresponding to 95% confidence) resulted in a very large number of incorrect decisions. In order to effectively restrict those incorrect decisions, the margin needs to be set significantly lower, in the area of 0.002 (or even lower for the Q-value test on multiple points). If set low enough though, in most cases the mistakes were completely avoided, especially when using splitting criteria that only consider a single splitting point per parameter (Parameter test and Q-value test at the median point). Perhaps the only exception to this is Welch's test in the case of the Q-value test with multiple splitting points, where even with a very strict margin of 0.00001 it only managed to achieve an approximately 95% accuracy.

All the criteria achieved their lowest accuracy when using the Q-value test with multiple splitting points. This is not a surprise, since the other two criteria only consider a single splitting point per parameter, which approximately equally divides the available experiences, ending up comparing sets of approximately equal sizes. On the other hand this criterion considers splitting points that leave only a handful of experiences in each set, making the decision significantly more difficult. The test that was affected the most by this fact was Welch's test, since it assumes that the two populations have different standard deviations, and as a result is very easily mislead when one of the two groups has very few elements.

Comparing the accuracy of the tests between the two criteria that consider a single splitting point per parameter (figures 6.2 and 6.3), the two tests that do not assume a normal distribution of the values (Mann-Whitney U test and Kolmogorov-Smirnov test) both achieved a better accuracy when run on the values of the parameters, which were generally less "noisy"

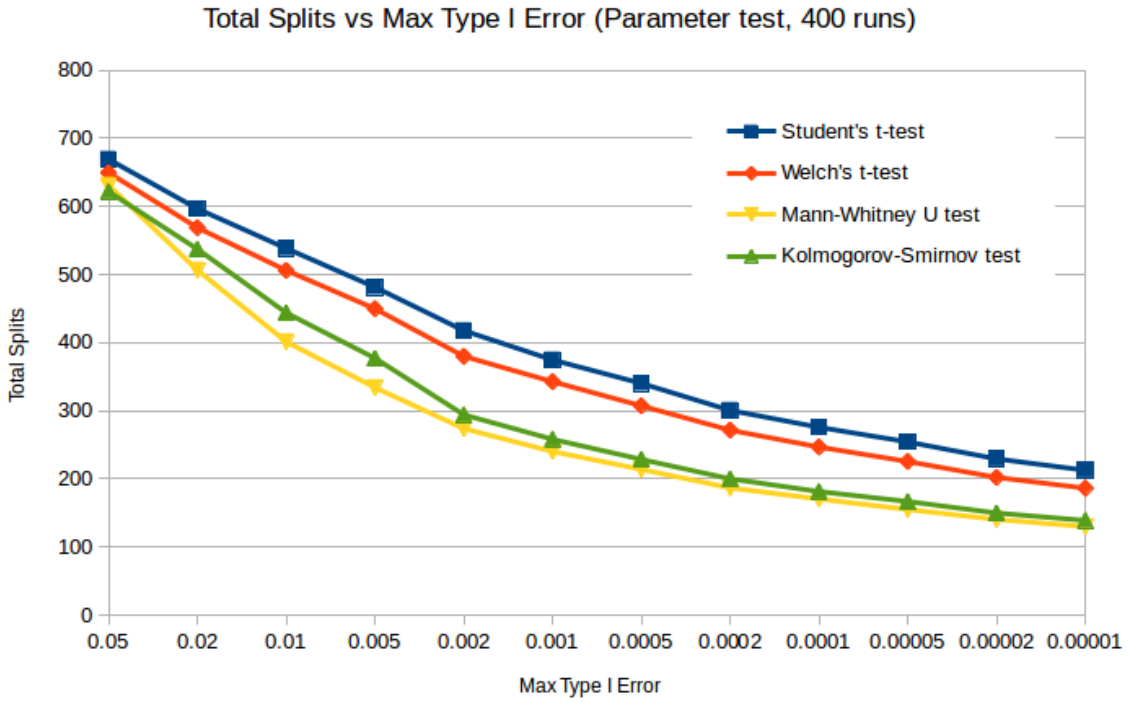


Figure 6.5: Number of splits for the four statistical criteria using the Parameter test, for different values of the maximum type I error

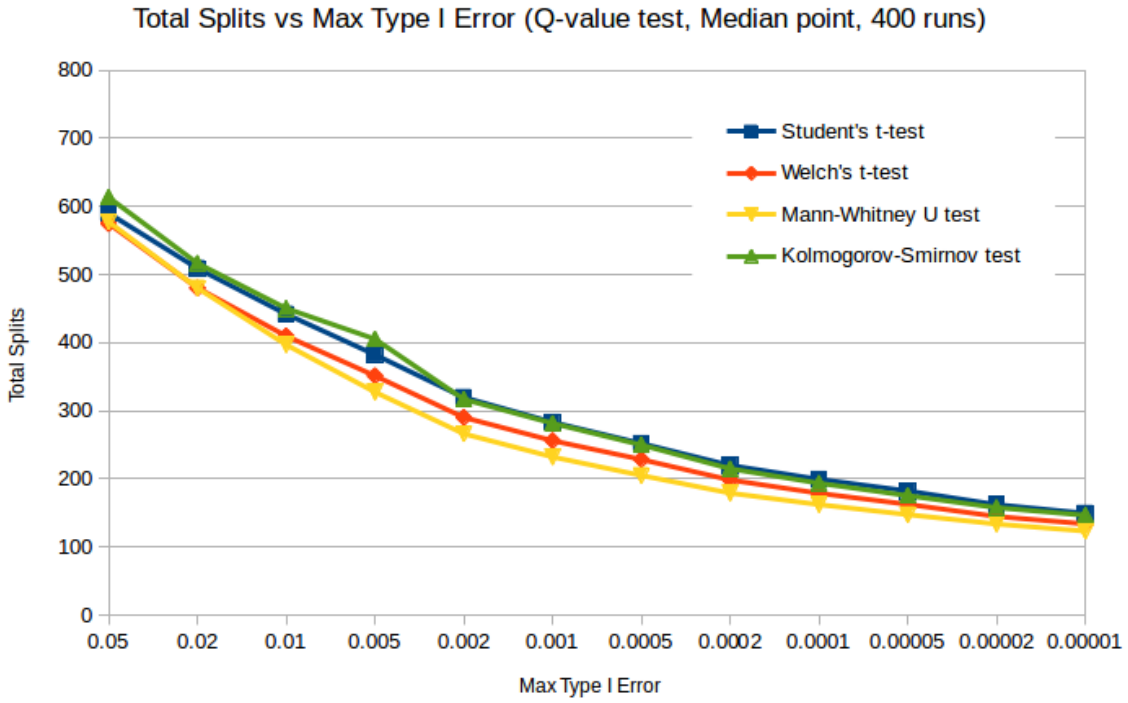


Figure 6.6: Number of splits for the four statistical criteria using the Q-value test at the median point, for different values of the maximum type I error

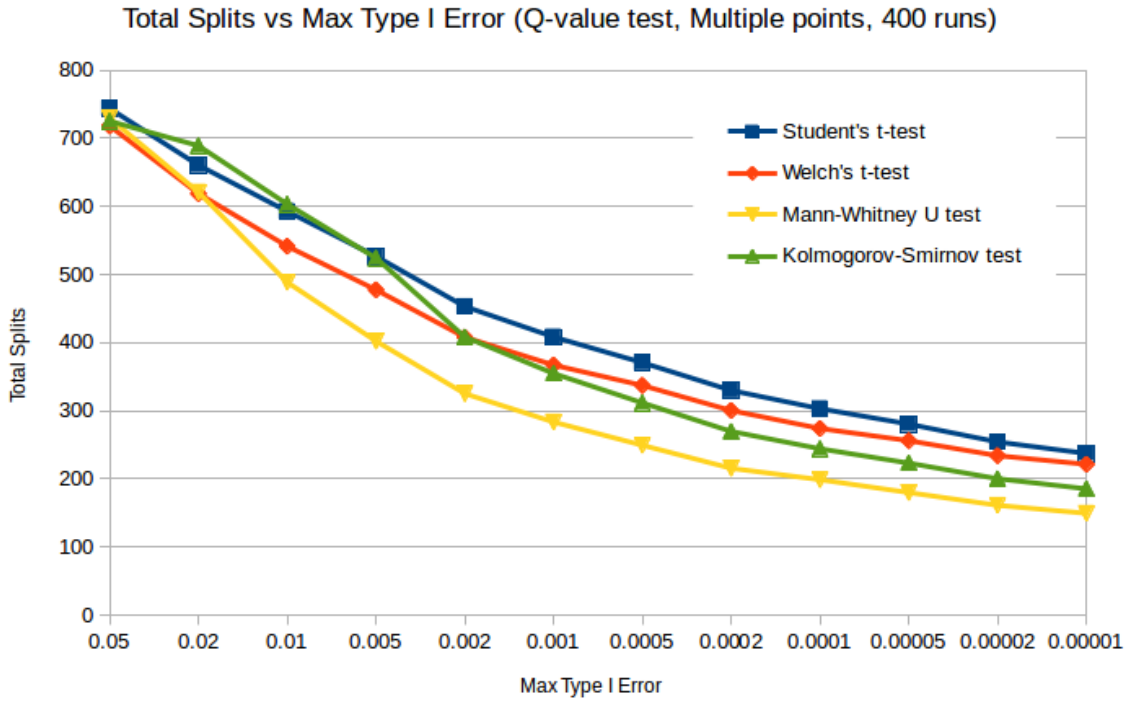


Figure 6.7: Number of splits for the four statistical criteria using the Q-value test at multiple points, for different values of the maximum type I error

that the values of the actions. However, the two tests that do assume a normal distribution (Student's t-test and Welch's test) had their accuracy very noticeably reduced when run on the values of the parameters. This can of course be explained by the fact that the values of some of the parameters were discrete. In this case, these tests can come up with a very misleading estimation of the standard deviation of the populations. For example, if the size of the cluster in a set takes the values $\{4, 4, 4\}$, these tests will estimate the standard deviation to be zero, which can be far from true, resulting in completely misleading type I error probabilities.

Overall, the Mann-Whitney U test achieved the highest accuracy with all the criteria. The Kolmogorov-Smirnov test was second in all the criteria, as long as the error margin was strict enough, while on the contrary, it performed very poorly when the margin was lenient (maximum type I error values of 0.002 or higher). Finally, the two tests assuming normal distributions, Student's t-test and Welch's test, only performed well when running on the Q-values (which were not discrete), and considering a single splitting point.

In figures 6.5, 6.6 and 6.7 we can see how the total number of splits performed is affected by the maximum statistical error. Even though selecting a low value is effective in preventing incorrect splits, it also greatly lowers the total amount of splits performed by all the tests. Among the tests, the Mann-Whitney U test performed the least amount of splits using all splitting criteria, which was the downside of being the most accurate. When using the Q-value test on the median point (figure 6.6), the amount of splits for all the tests was very close. However, when using the Parameter test, the two tests that assume normal distributions (which were also the least accurate on this criterion) performed significantly more splits, thus ending up with significantly bigger decision trees. When using the Q-value test with multiple

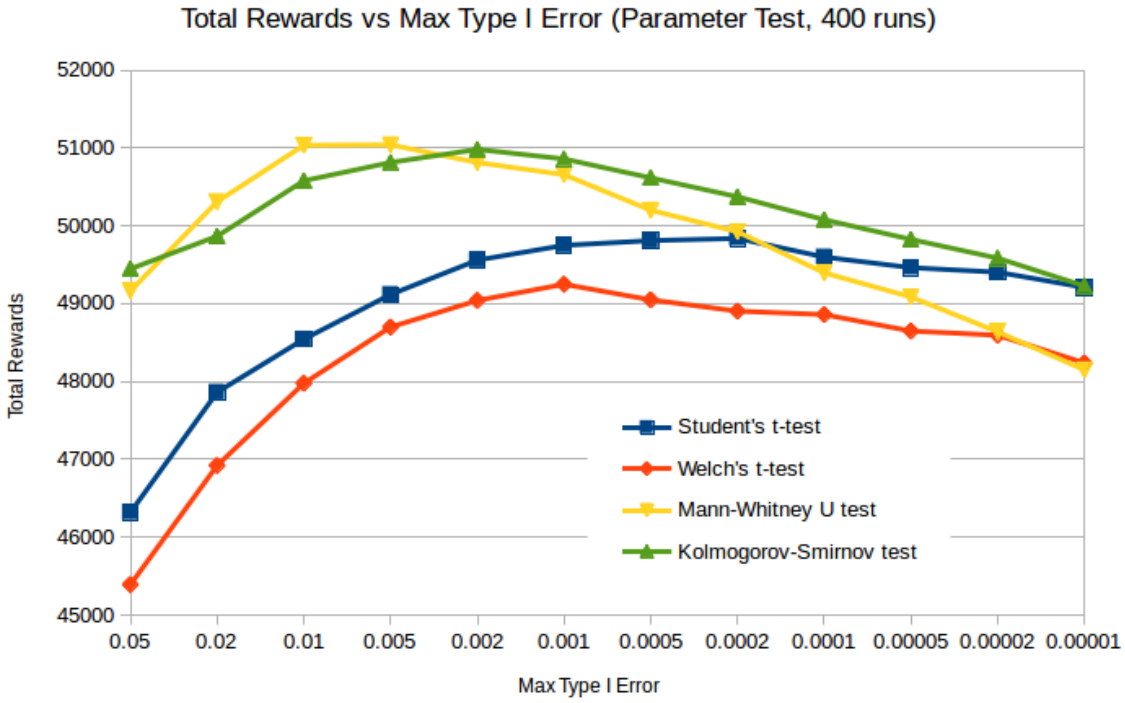


Figure 6.8: Performance of the four statistical criteria using the Parameter test, for different values of the maximum type I error

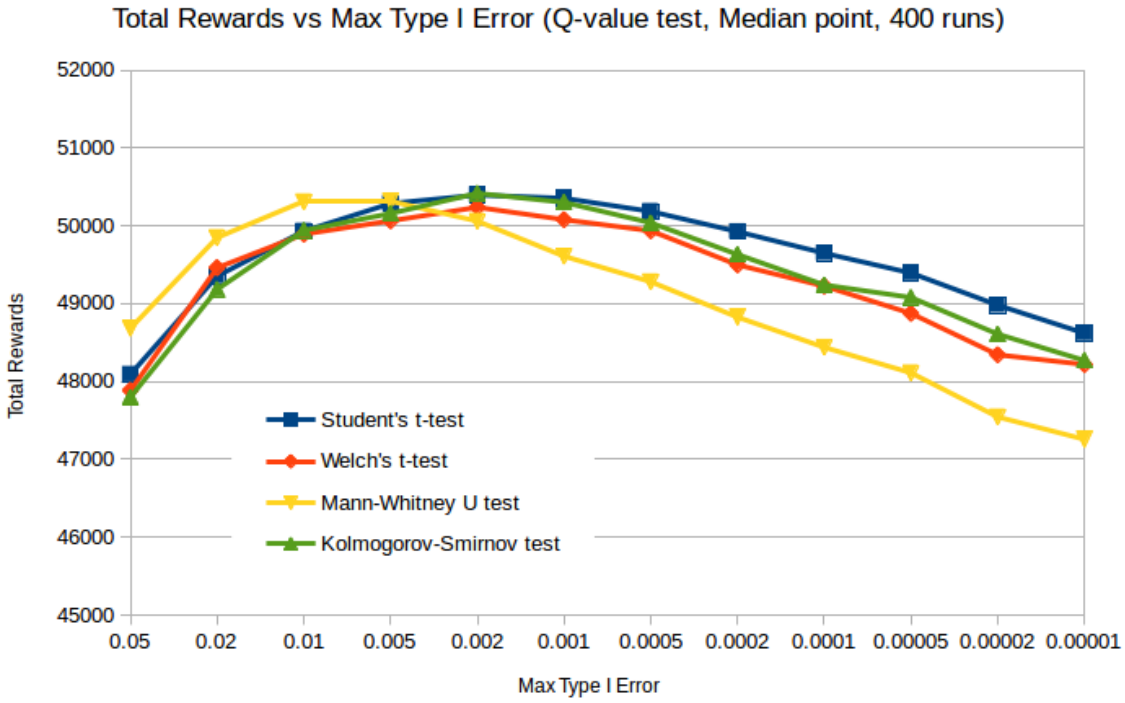


Figure 6.9: Performance of the four statistical criteria using the Q-value test at the median point, for different values of the maximum type I error

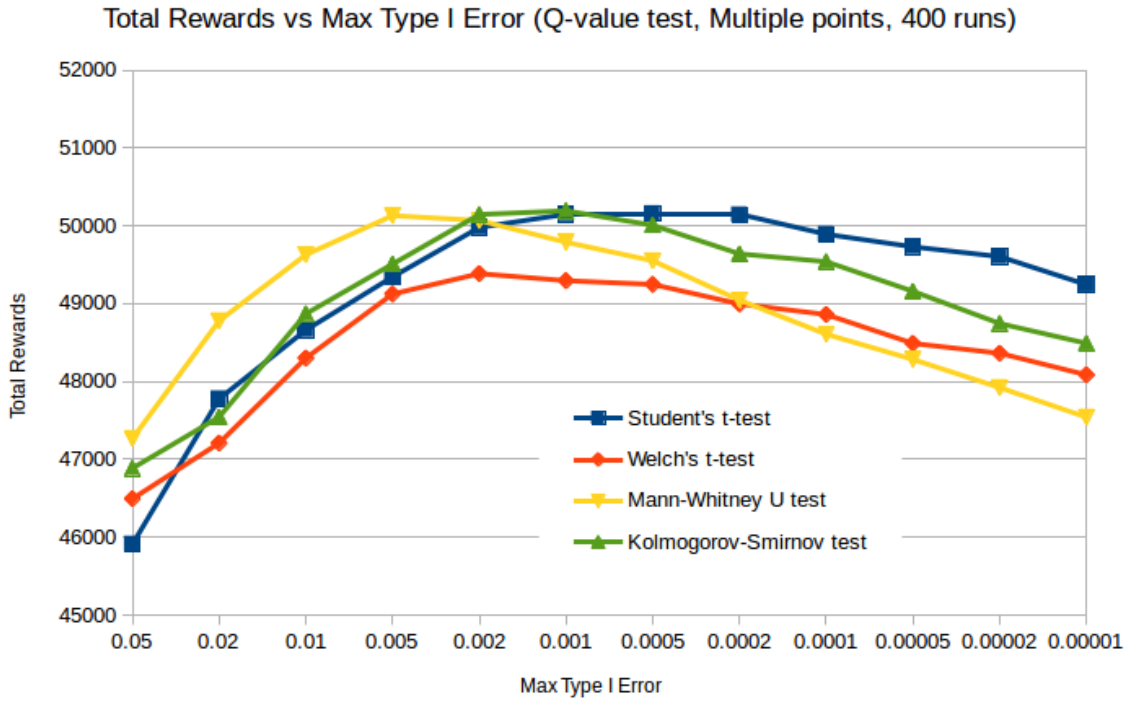


Figure 6.10: Performance of the four statistical criteria using the Q-value test at multiple points, for different values of the maximum type I error

splitting points, as expected, all the tests increased the number of splits performed. However, the Mann-Whitney U test was least affected.

How all this translates to the final performance of the algorithm can be seen in figures 6.8, 6.9 and 6.10. The typical value of 0.05 for the maximum type I error achieved significantly suboptimal performance with all criteria and statistical tests. The Mann-Whitney U test, being the most accurate one, managed to perform very well with all the criteria. However, performing the least amount of splits, in achieved that performance for higher (less strict) values of the error margin, in the area of 0.005. The Kolmogorov-Smirnov test also did well with all the criteria, achieving its best performance for a slightly more strict value of 0.002.

As mentioned before, the difference between the equal variance t-test (Student's t-test) and its unequal variance counterpart (Welch's t-test) is that the Student's t-test makes the assumption that the variances of the two populations are equal, while Welch's t-test is expected to be more accurate when this does not necessarily hold [Ruxt06] [Coom96]. On a first approach, we can imagine situations where the two compared variances are unequal. For example, if certain values of a parameter cause instability of the system, then the variance of the q-values in that part of the state space would be expected to have a higher variance. This fact points towards Welch's t-test being more widely applicable. However, the assumption of equal variances increases the strength of the Student's t-test, and in our experiments we expect the variances of the population to be approximately equal more often than not. This fact resulted in the Student's t-test clearly outperforming Welch's t-test, often with by a significant margin.

Compared to the other tests, the Student's t-test did very well with the Q-value criterion, where the values tested were continuous. When using the parameter test, where the distribution of the values is greatly different from a normal distribution, its performance dropped significantly below that of the Mann-Whitney and Kolmogorov-Smirnov test.

6.1.2 Minimum Information Gain

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT with Information Gain criterion
- Initial Decision Tree: Single State
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$
- Minimum Information Gain $\in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$ bits
- Minimum number of experiences to split: 8 per resulting state

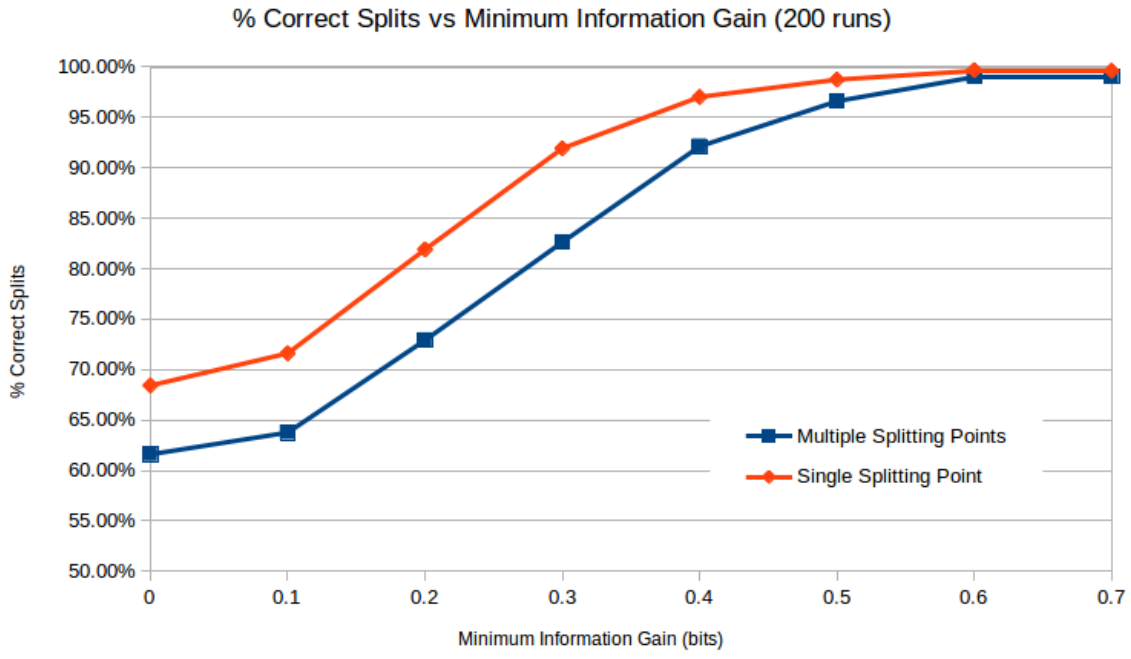


Figure 6.11: Percentage of splits performed on parameters that affected the behavior of the system for different values of the minimum information gain

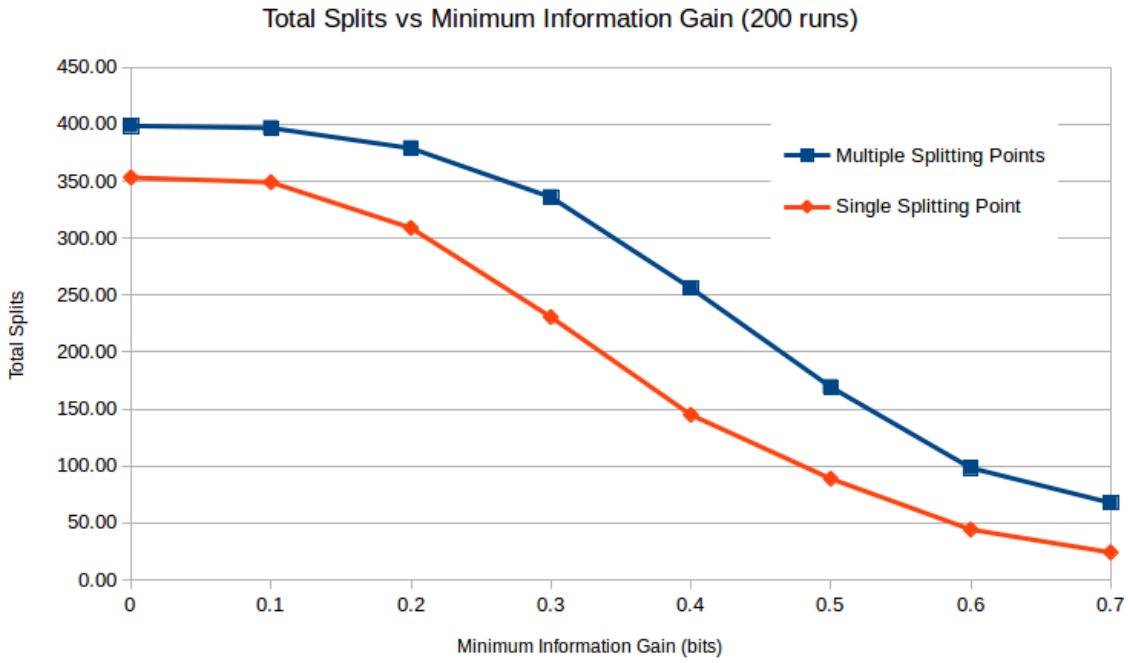


Figure 6.12: The total number of splits for different values of the minimum information gain

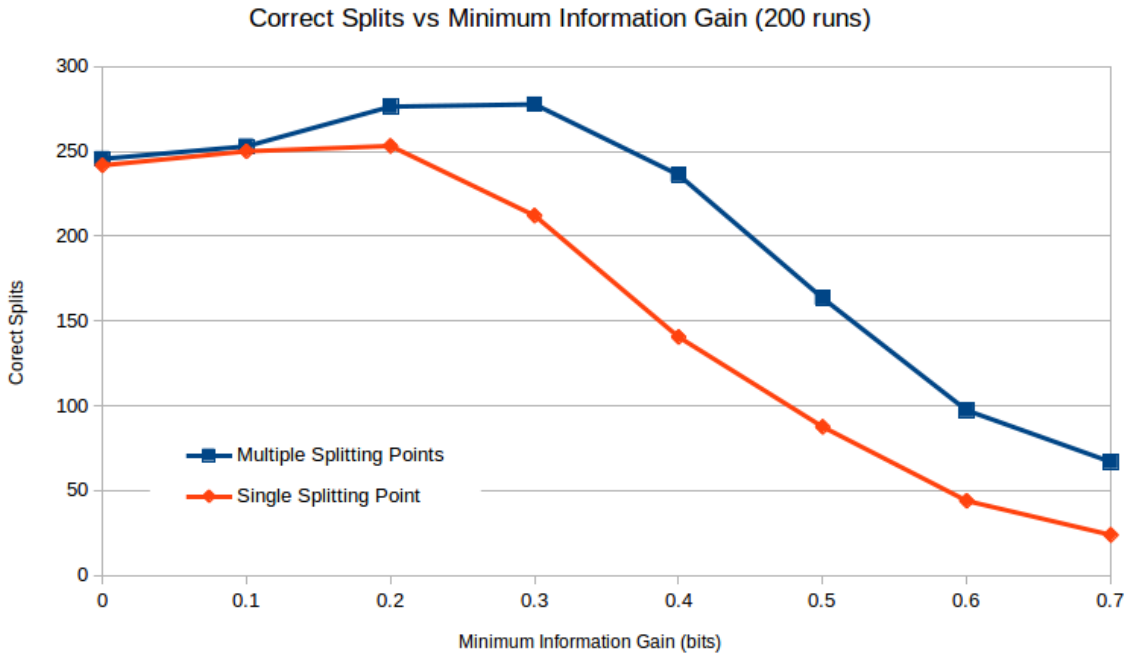


Figure 6.13: The total number of splits performed on parameters that affected the behavior of the system for different values of the minimum information gain

In order to strengthen the *Information Gain* criterion, to perform a split we require that the classification information in the two resulting states is at least min_info_gain lower that that of the initial state. In this experiment we observe how different values of this parameter affect the performance of the algorithm in the two cases studied, namely when considering

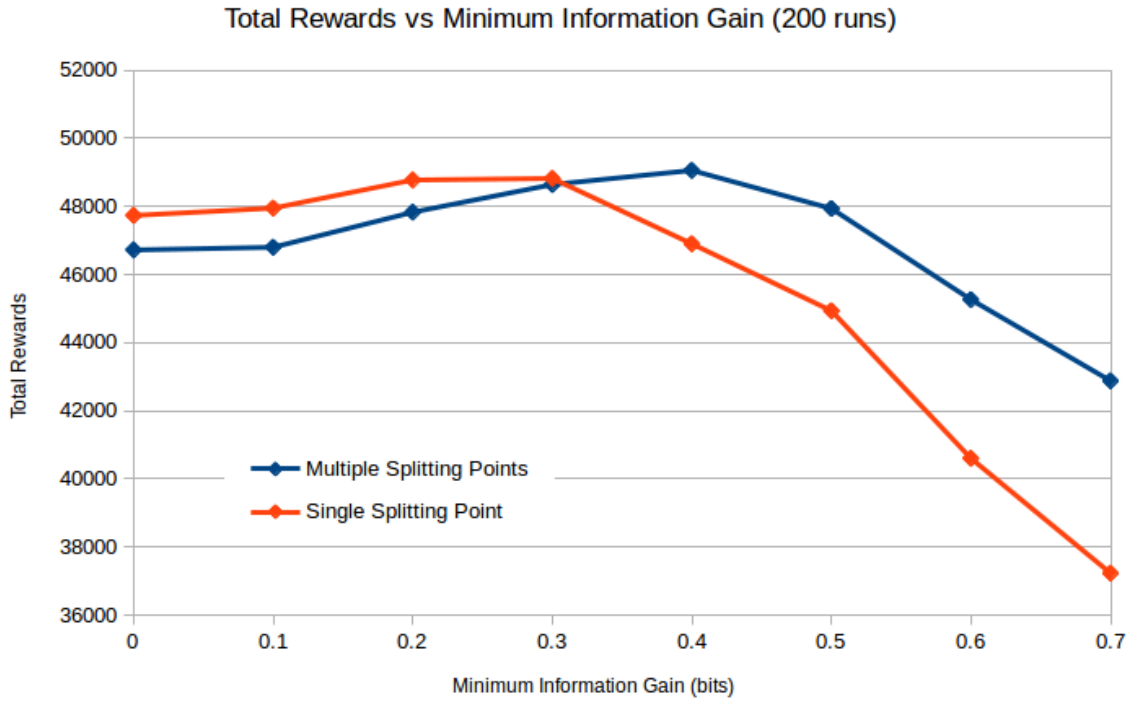


Figure 6.14: The sum of rewards obtained for different values of the minimum information gain

multiple splitting points per parameter, or only considering splitting on the point nearest to the median.

In figure 6.11 we can see that for high values of this parameter, this criterion can distinguish the correct correlations from the noise, and achieve accuracy that is competitive to the statistical criteria. However, especially when considering only a single splitting point per parameter, requiring such a high information gain in order to split greatly reduces the total splits performed, dropping them to below 50 on average for a value of 0.7 (in this case in some of the executions the algorithm failed to perform any splits during the whole run). This fact causes a great drop in performance for values greater than 0.5 (figure 6.14) in both cases. On the contrary, values below 0.3 have a very negative effect on the accuracy of the splits and also noticeably reduce performance.

When considering multiple splitting points per parameter, the accuracy of the splits was significantly lower, but that was balanced by the much larger amount of splits achieved. The result was that both strategies achieved very similar performances, with the difference that the single splitting point strategy reached its peak at a lower margin of 0.2 to 0.3 bits instead of the 0.4 bits for the multiple splitting points case.

6.1.3 Minimum Number of Experiences to Perform a Split

Setup:

- Training steps: 5000

- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT
- Splitting Criterion: Q-value test
- Statistical test: Student's *t*-test, Welch's *t*-test, Mann-Whitney U test, Kolmogorov-Smirnov test
- Initial Decision Tree: Single State
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$
- Minimum number of experiences $\in \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$

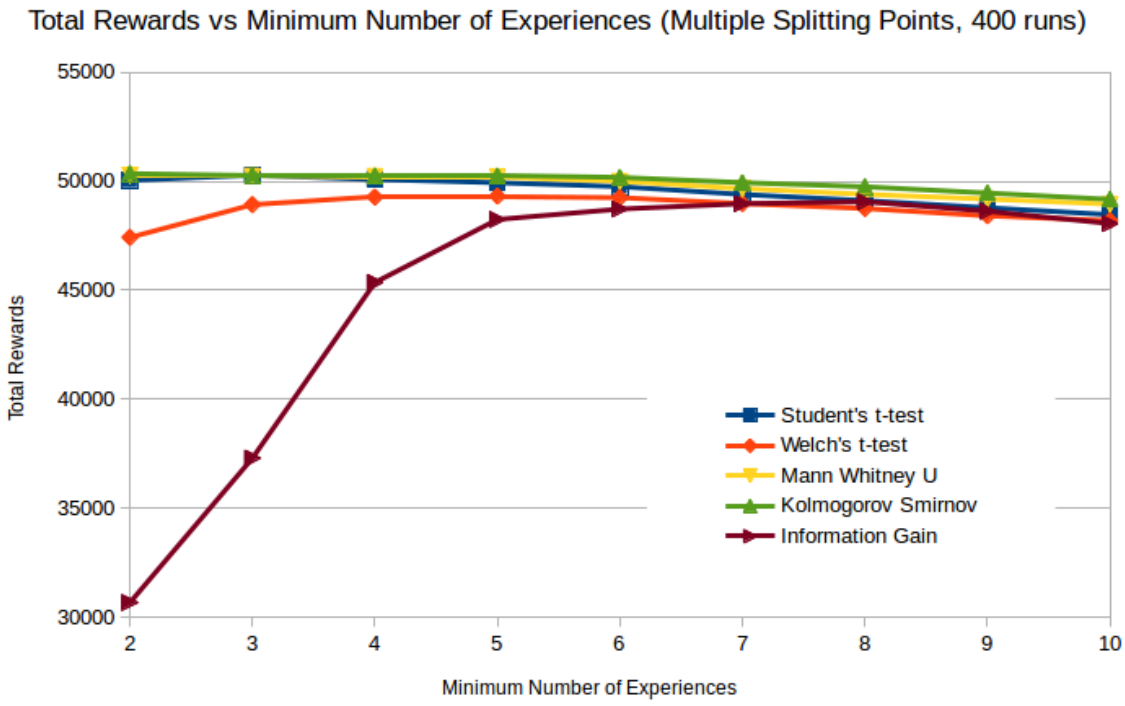


Figure 6.15: The performance of the algorithm as a function of the minimum number of experiences required in either side of a split when allowing multiple splitting points

One more parameter that controls the accuracy of the splits is the minimum number of experiences required on either side of a split. If splits are allowed to happen with fewer available data, we generally expect a drop in their accuracy. On the other hand, a very conservative requirement would reduce the options of the algorithm and thus also hurt performance. To observe this behavior, we tested the performance of the four different statistical tests, along

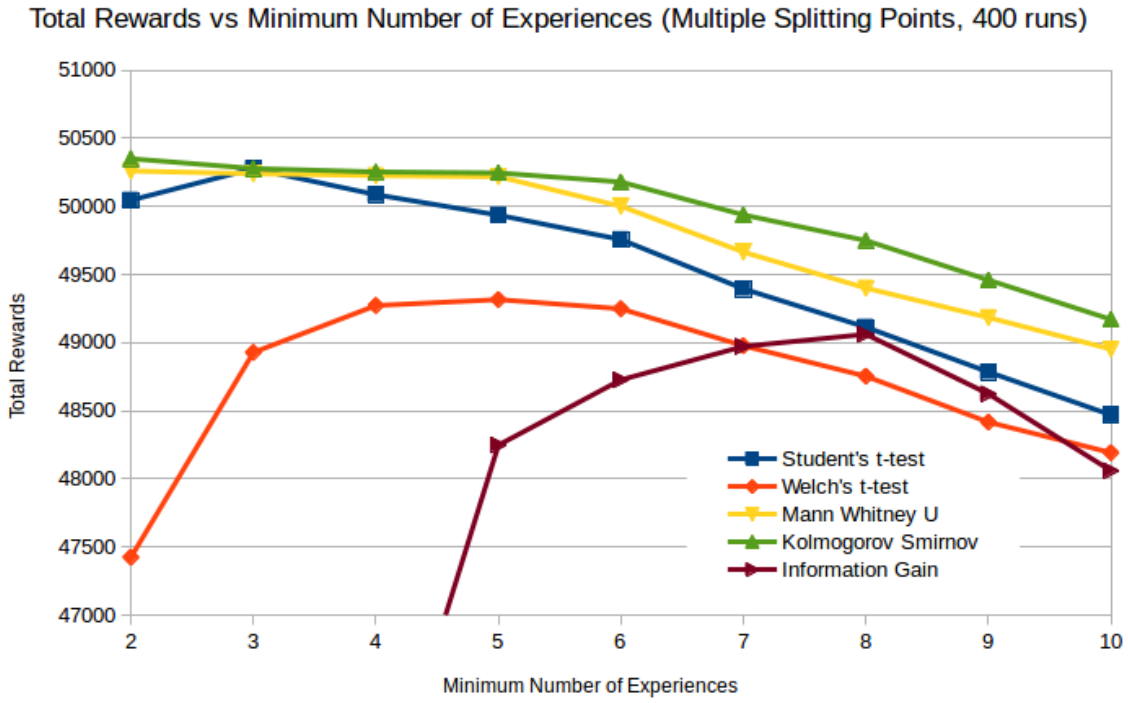


Figure 6.16: Zoom in on figure 6.15

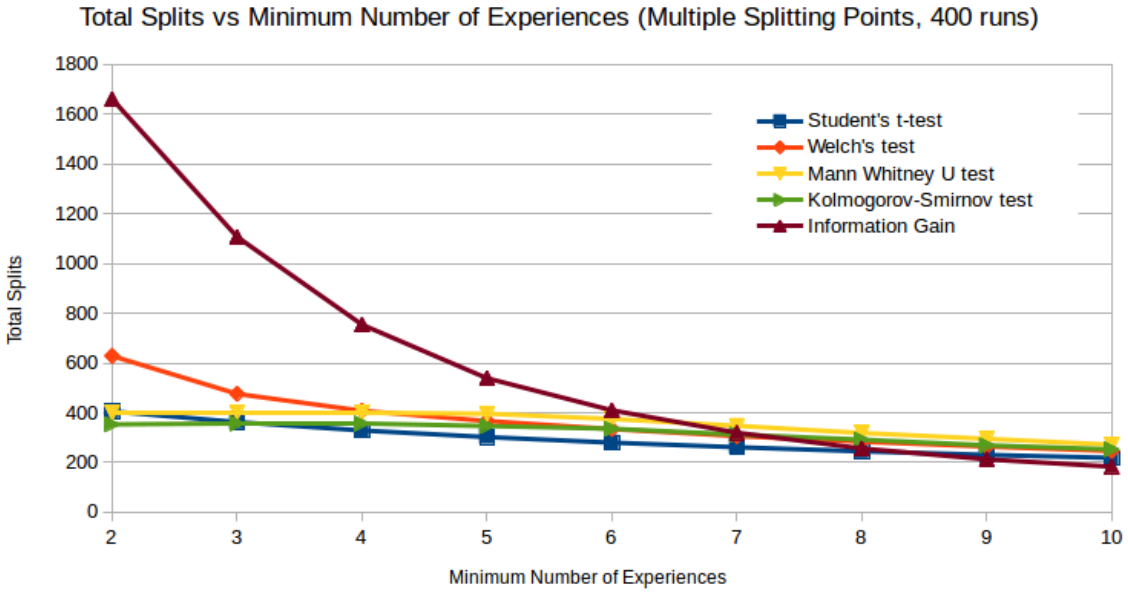


Figure 6.17: The total number of splits performed by all tests as a function of the minimum number of experiences required in either side of the split when allowing multiple splitting points

with information gain, for different values of this parameter. The most suitable splitting criterion for this test is the Q-value test with multiple splitting points. Since this criterion considers splitting a state between any two consecutive points, it will always test cases where at least

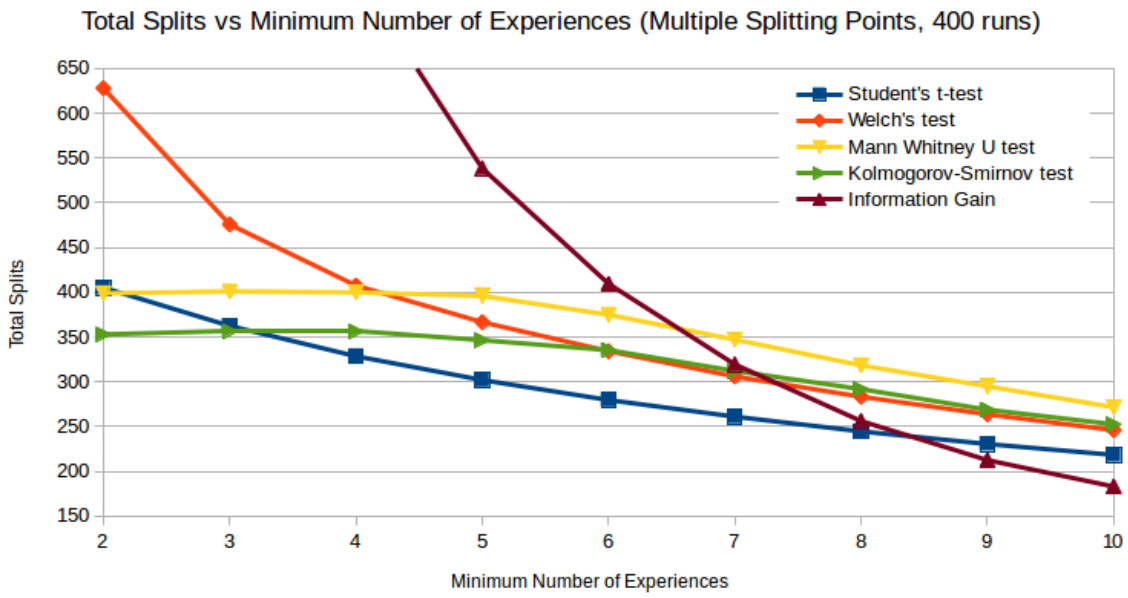


Figure 6.18: Zoom in on figure 6.17

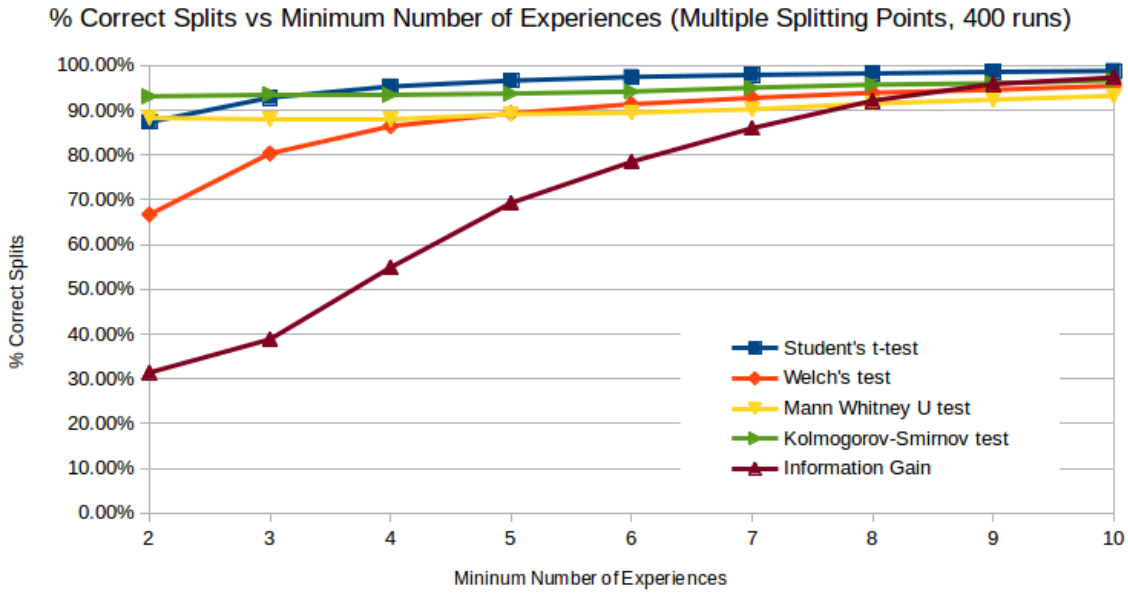


Figure 6.19: The percentage of splits performed on parameters that affect the behavior of the cluster as a function of the minimum number of experiences in either side of the split when allowing multiple splitting points

one of the two tested sets contains a very small amount of elements.

With the exception of Welch's test, the other three statistical tests did not need a restriction in the number of available points in order to be effective (figures 6.15 and 6.16). Student's t-test reached its maximum when the limit was set to 3, while the Kolmogorov-Smirnov and the Mann Whitney tests achieved their best performance for the minimum value of 2 points per resulting state.

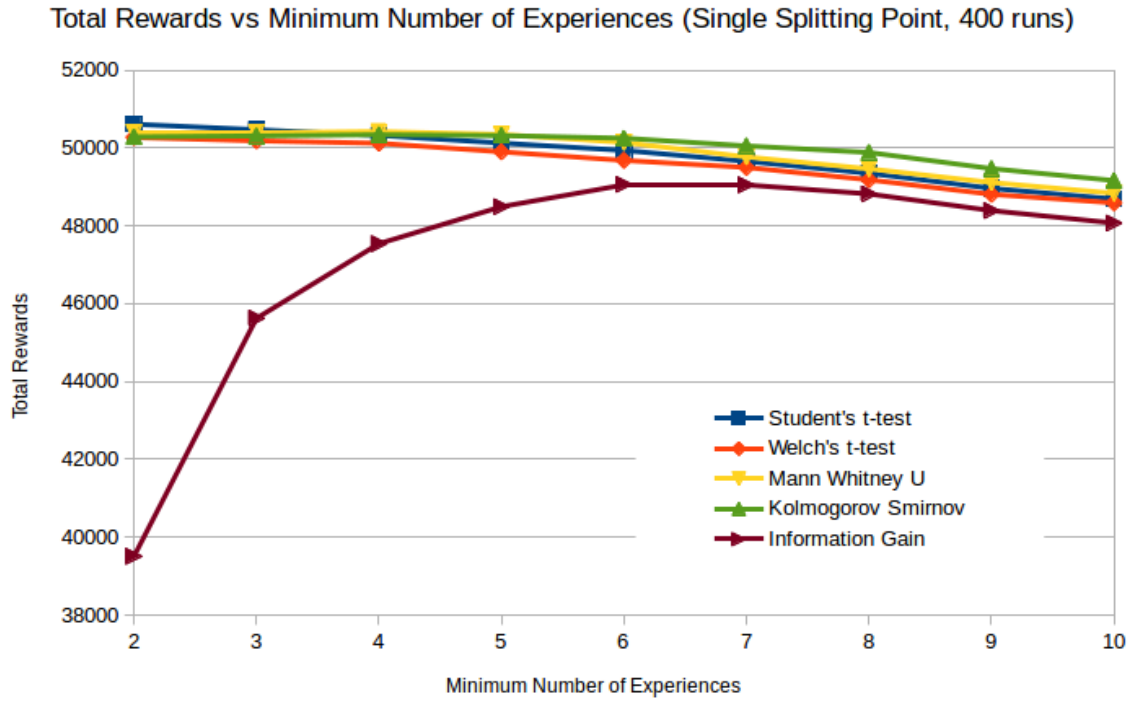


Figure 6.20: The performance of the algorithm as a function of the minimum number of experiences required in either side of a split when allowing a single splitting point

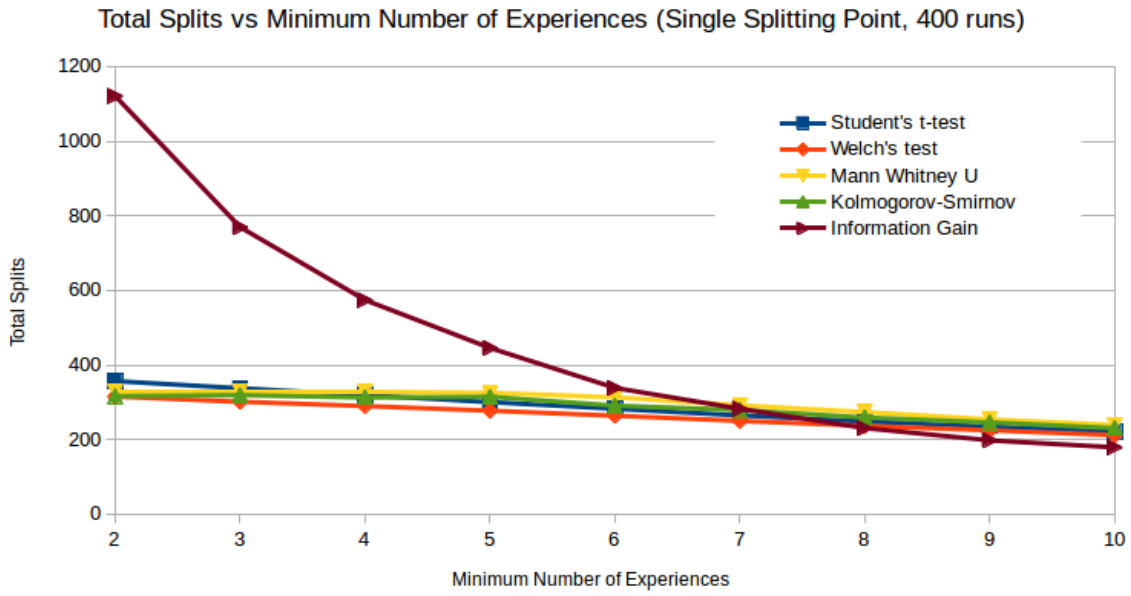


Figure 6.21: The total number of splits performed by all tests as a function of the minimum number of experiences required in either side of the split when allowing a single splitting point

Welch's test on the other hand had its performance drop very significantly when allowed to consider splitting points with very few points on either side. This can be understood by

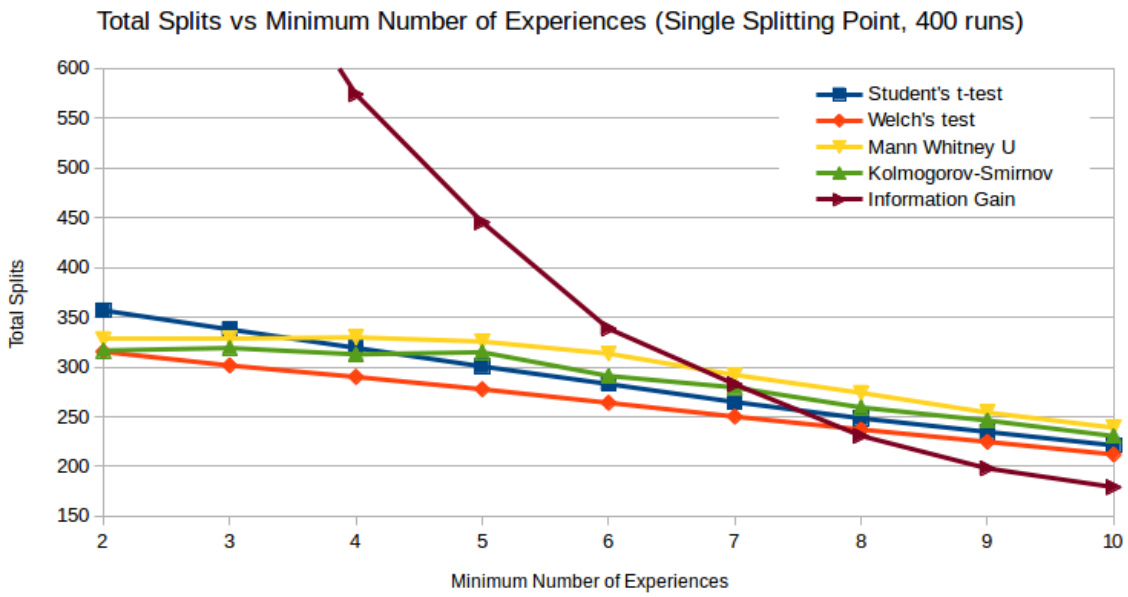


Figure 6.22: Zoom in on figure 6.21

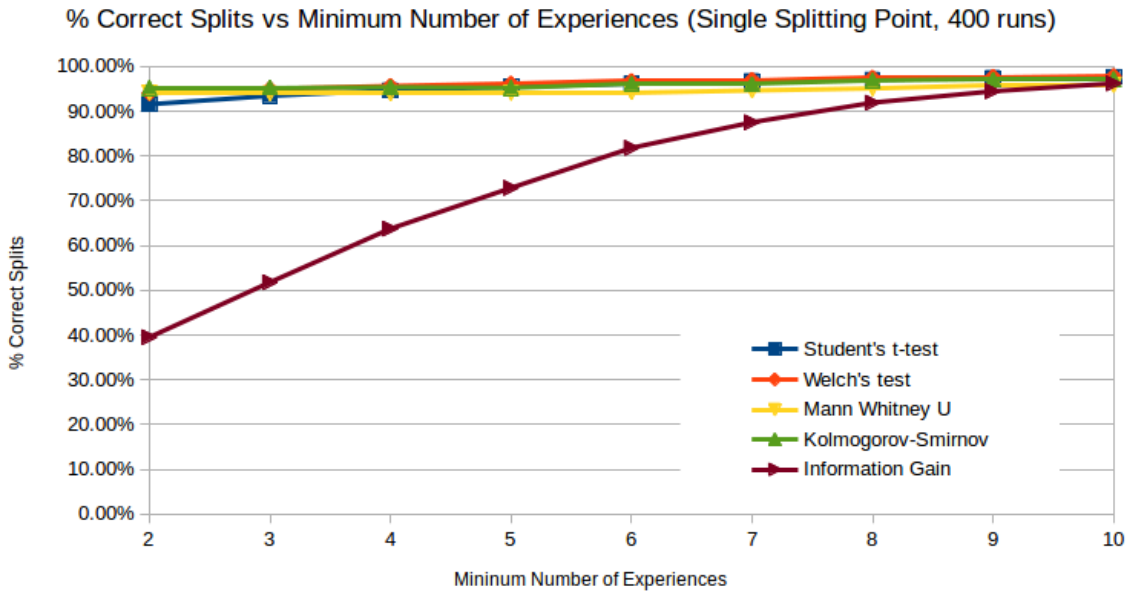


Figure 6.23: The percentage of splits performed on parameters that affect the behavior of the cluster as a function of the minimum number of experiences in either side of the split when allowing a single splitting point

the fact that this test assumes the two populations to have unequal variances. This means that it calculates its estimation of the variance of each sample by using only the elements of that sample. This way, if the elements within each sample are almost equal but the two samples have different means, it will assume that both variances are almost zero and because of the non-zero difference in the means it will produce an unrealistically low error probability. This weakness is to a large extent avoided by the Student's t-test, since it assumes that the

two populations have equal variances, and calculates a total variance for both samples. As a result, if there is a non-trivial difference between the means of the two samples it will inevitably calculate a non-trivial variance for the total population, and thus avoid producing a much lower than expected error rate. This fact is also evident in the total number of splits performed, where Welch's test ends up a significantly larger amount for lower values of the parameter, Student's t-test increases its splits by a significantly smaller amount, and the other two tests remain unaffected (figures 6.17 and 6.18).

Information gain exhibited a very significant dependence on the minimum number of points available. When allowed to split with a minimum of 2 points, it performed more than 1600 splits (with obviously very low accuracy), despite the existing requirement for 0.4 bits of gained information. Therefore, for this criterion limiting the amount of points per side of the split is crucial, and only achieved its best performance for a very high value of 8 points.

When splits are only allowed at the median point of the values of the parameters (figure 6.20), the requirement for a limit in the number of points goes away for both the Student's t-test and Welch's test. Even though the number of splits still slightly goes up even when the requirement is lifted (figures 6.21 and 6.22), the accuracy is not affected (figure 6.23), and the performance for both the tests ends up improving. As expected, the two statistical tests that did not require a limit in the multiple splitting points case do not require a limit in this case either. However, information gain is still very vulnerable (even though slightly less so), and requires a minimum of 6 to 7 points per side to reach its potential.

6.1.4 Splitting Criteria Overview

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT
- Splitting Criteria: Q-value test, Parameter test, Information Gain
- Statistical test: Student's *t*-test, Welch's *t*-test, Mann-Whitney U test, Kolmogorov-Smirnov test
- Initial Decision Tree: Single State
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$

In this experiment we compare the performance of all the implemented splitting criteria under the optimal settings derived from the previous experiments (figure 6.24). The two tests that did not assume a normal distribution of the values, the Kolmogorov-Smirnov test and the Mann Whitney U test, performed better using the Parameter test criterion, while the two tests

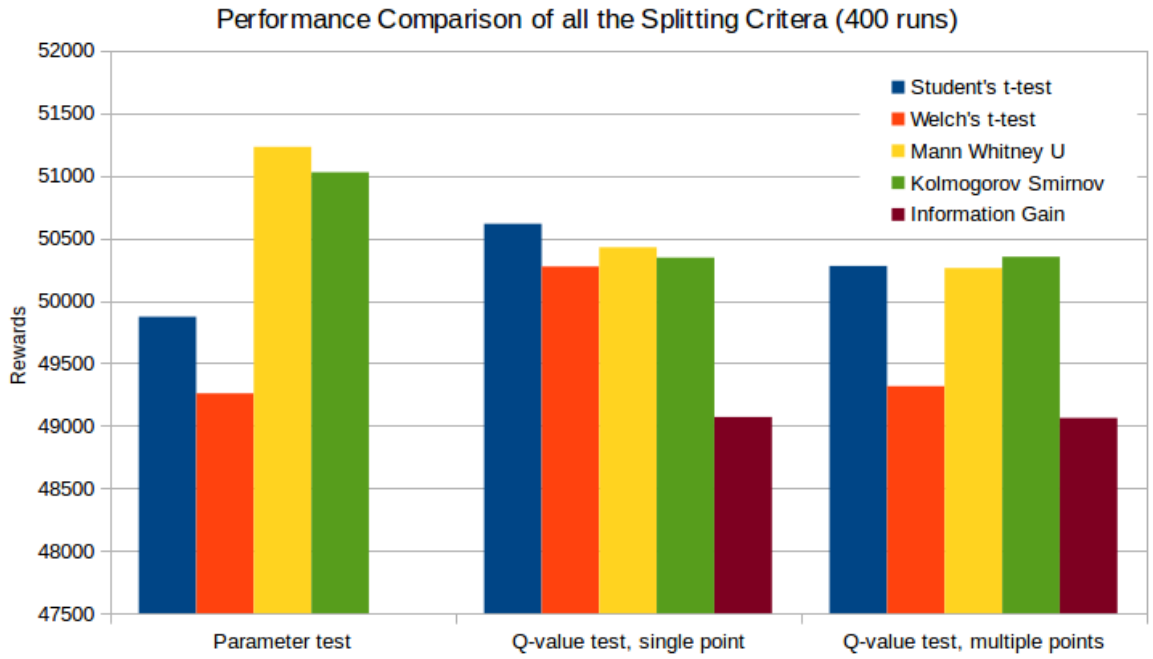


Figure 6.24: Performance comparison of all the splitting criteria using their optimal settings

that did assume normal distributions performed better when using the Q-value test. This is not surprising, considering the fact that some of the parameters in this scenario were discrete. This means that their distribution differs very significantly from a normal distribution, making then inaccurate when the values of these parameters were provided for comparison. On the other hand, since the Q-values were generally not discrete, when performing a test on them the performance improved.

Among all the options, the Mann Whitney U test achieved the best results under the Parameter test criterion, capitalizing on the fact that it was the most accurate in terms of the number of incorrect splits, as exhibited in the experiment in section 6.1.1. The Information Gain criterion on the other hand seemed to be the least effective among all the criteria, requiring the most restrictions in order to perform, and achieving the overall lowest performance. Finally, between the two available options for the Q-value test, namely considering a single or multiple splitting points, for most criteria the consideration of a single splitting point achieved better results, while at the same time producing smaller decision trees.

6.1.5 Splitting Strategy

Setup:

- Training steps: 5000
- Evaluation steps: 5000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT

- Splitting Criterion: Parameter test
- Statistical test: Mann-Whitney U test
- Update algorithm: Prioritized Sweeping
- Initial Decision Tree: Single State
- Discount Factor: $\gamma = 0.5$
- *t*-test max error: 0.005
- Minimum number of experiences to split: 2 per resulting state

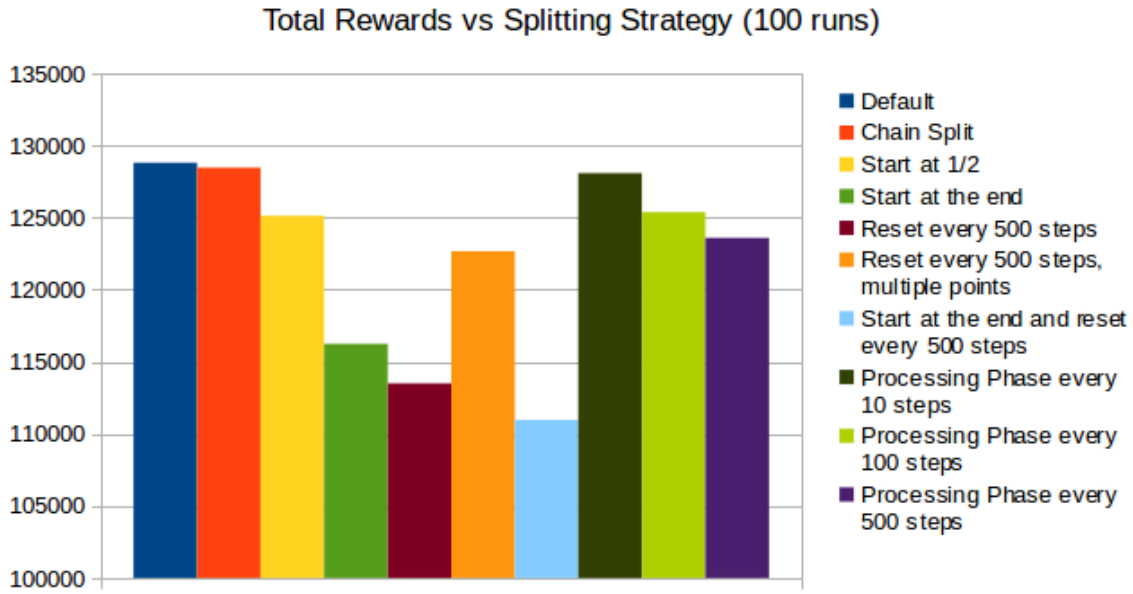


Figure 6.25: Performance comparison for ten different splitting strategies

By default, MDPDT attempts to perform a split on the starting state of an experience after the experience has been acquired. In this experiment we test the performance of different approaches on this decision.

One such approach, whose aim is to accelerate the growth of the tree, is to attempt to split every node in the model, regardless of whether it was involved in an experience. This makes sense because the splitting criteria take into account the values of other states, and so it is possible that a change in the value of one state triggers a split in another. If a split is indeed performed on any state, we repeat the process. We call this procedure a *Chain Split*.

One other strategy is to delay the splitting until a significant amount of data have been acquired. The reasoning behind this is that once more data are available the splitting criteria may be able to make better decisions and build a better decision tree compared to the one built with less data. Additionally, we experimented with periodically resetting the decision tree and performing a *Chain Split* to rebuild it.

Finally, we tested the splitting strategy used in [Uthe98]. This included splitting the algorithm in two phases, a *Data Gathering* phase where data are collected but no splits are

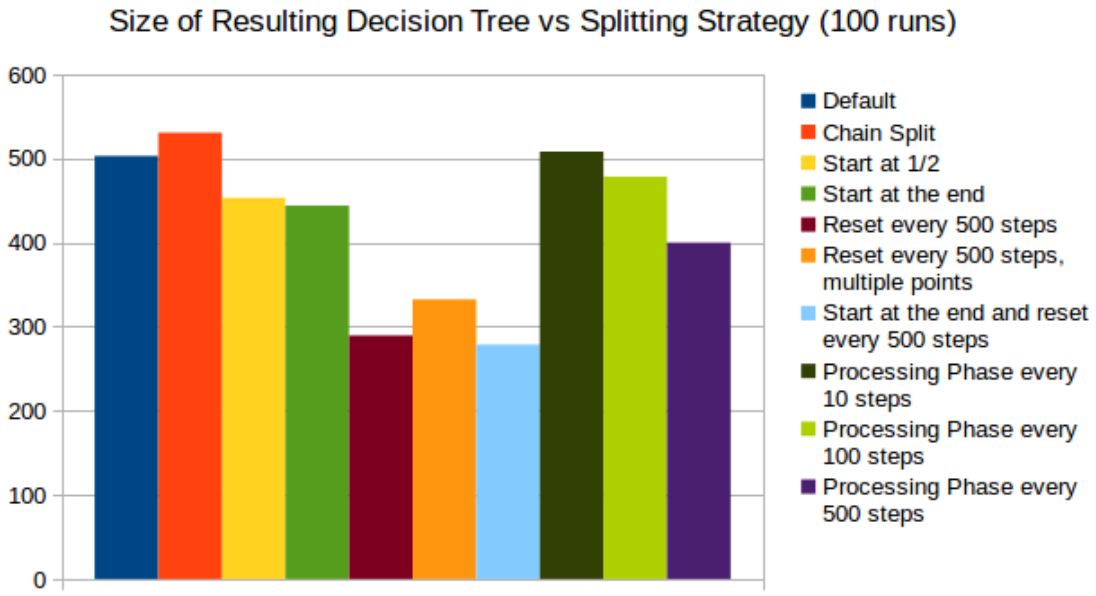


Figure 6.26: The size of the decision tree at the end of the evaluation phase for all splitting strategies

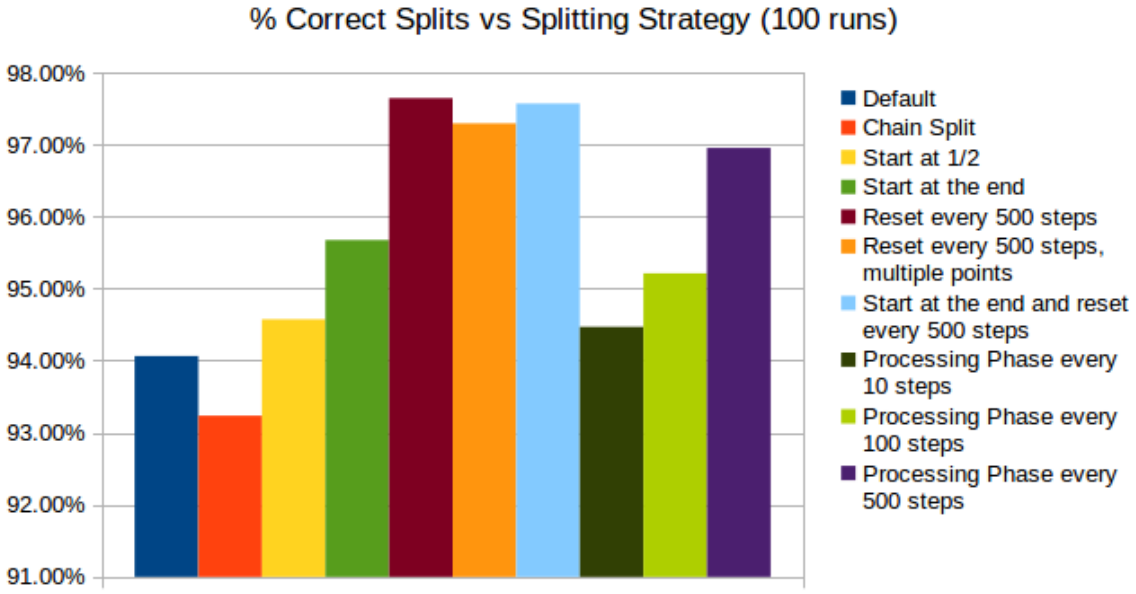


Figure 6.27: The percentage of decision nodes of the final decision tree that partition the state space using parameters that affect the performance of the system

performed, and a *Processing Phase*, where all the nodes of the model are tested one by one to check if a split is needed, and if so, perform the splits. We tested performing this Processing Phase every 10, 100 and 500 steps.

To evaluate these approaches, we tested the following strategies:

i. *Default:* Only attempt to split the starting state for each new experience.

ii. *Chain Split:* Perform a *Chain Split* with every new experience.

iii. *Start at 1/2:* Allow splitting to begin 1/2 into the training.

iv. *Start at the end:* Allow splitting to begin at the end of the training, and also perform one chain split at that time.

v. *Reset every 500 steps:* Reset the decision tree and perform a *Chain Split* every 500 steps.

vi. *Reset every 500 steps, multiple points:* Same as above, but using the multiple points Q-value test criterion, attempting to split each state at multiple points per parameter.

vii. *Start at the end and reset every 500 steps:* Allow splitting to begin at the end of the training, and after that reset the decision tree and perform a *Chain Split* every 500 steps.

viii. *Processing Phase every 10 steps:* Do not allow splits, but run a *Processing Phase* (see above) every 10 steps.

ix. *Processing Phase every 100 steps:* Do not allow splits, but run a *Processing Phase* every 100 steps.

x. *Processing Phase every 500 steps:* Do not allow splits, but run a *Processing Phase* every 500 steps.

Even though *Chain Split* adopted a much more aggressive (and computationally intensive) strategy in attempting to grow the decision tree, the results were somewhat underwhelming. Comparing strategy (i) with strategy (ii) shows that even though Chain Split managed to perform 30 additional splits on average, the quality of the splits went down and as a result the total performance slightly deteriorated. This can be explained by the fact that the more aggressive strategy also creates more opportunities for errors. Also, the relatively low amount of additional splits reveals that the default strategy already depletes most of the opportunities to create new states.

Waiting for more data to be available in order to start splitting performed even worse. Despite offering a slight increase in the accuracy of the splits, (in the order of 1-2%), it caused a 10% reduction in their number and in the case of strategy (iv) a very significant drop in performance. We believe that the magnitude of this impact also reveals the importance of maintaining the decision tree throughout the training phase. As discussed in section 6.1.8, MDPDT favors lower values of the exploration constant because repeating the optimal action multiple times makes more data available to perform splits. This is only true though if the decision tree is already in place. If the decision tree is still a single state, that opportunity is missed and only the optimal action of the single global state is repeated.

Periodically resetting the decision tree in order to rebuild it provided the most accurate splits on the final tree, which was expected since the splits were performed with the maximum amount of data. However, the resulting size of the tree was significantly smaller in this case, limiting the performance of this strategy. Additionally, in order to test our hypothesis on the impact of not having the decision tree in place during the training, we also experimented with preventing any splits during the training phase and then resetting the decision tree every 500 steps and chain splitting during the evaluation phase. Note that since the decision tree is reset right at the start of the evaluation phase, any previous splits performed should not

directly affect the state of the tree during the evaluation. However, the fact that the tree was not active during the training resulted in a smaller tree, and very clear consequences on the performance.

Finally, using a *Processing Phase* periodically instead of regularly splitting performed better the smaller that period was. If performed every 10 steps it nearly reached the performance of the default strategy (though having a significantly larger running time), but for periods larger than that it quickly fell behind, making it a less favorable option.

Overall, the results of this experiment make us believe that the default method of attempting to split the initial state of each experience is both efficient and effective.

6.1.6 Initial Size of the Decision Tree

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT with Parameter test
- Statistical test: Mann Whitney U test
- Update algorithm: Prioritized Sweeping
- Initial Decision Tree: {single state, 1-dimensional grid (10 states), 2-dimensional grid (50 states), 3-dimensional grid (150 states)}
- Discount Factor: $\gamma = 0.5$
- *t*-test max error: 0.005
- Minimum number of experiences to split: 2 per resulting state

Even though the decision tree algorithms are designed to work on large and unknown state spaces, it is easy to imagine applications where some partial knowledge about the state space is known beforehand. For example, in a cloud computing environment where a reinforcement learning agent performs elasticity decisions on a cluster, it can be expected that the size of the cluster will always be an important parameter in the decisions.

With that in mind, we experimented with the state of the decision tree at the start of the training process. Instead of starting with a single state, we created decision trees that are equivalent to 1-dimensional, 2-dimensional and 3-dimensional grids. One such decision tree can be seen in figure 6.28. We then followed the training and evaluation process as normal, allowing the algorithm to perform additional splits on their own.

As expected, this pre-partitioning of the state space helped MDPDT improve its performance (figure 6.29). Moreover, in the case of the 1-dimensional grid, it even allowed more splits to be performed during the run (figure 6.30). This is not unexpected, since the initial

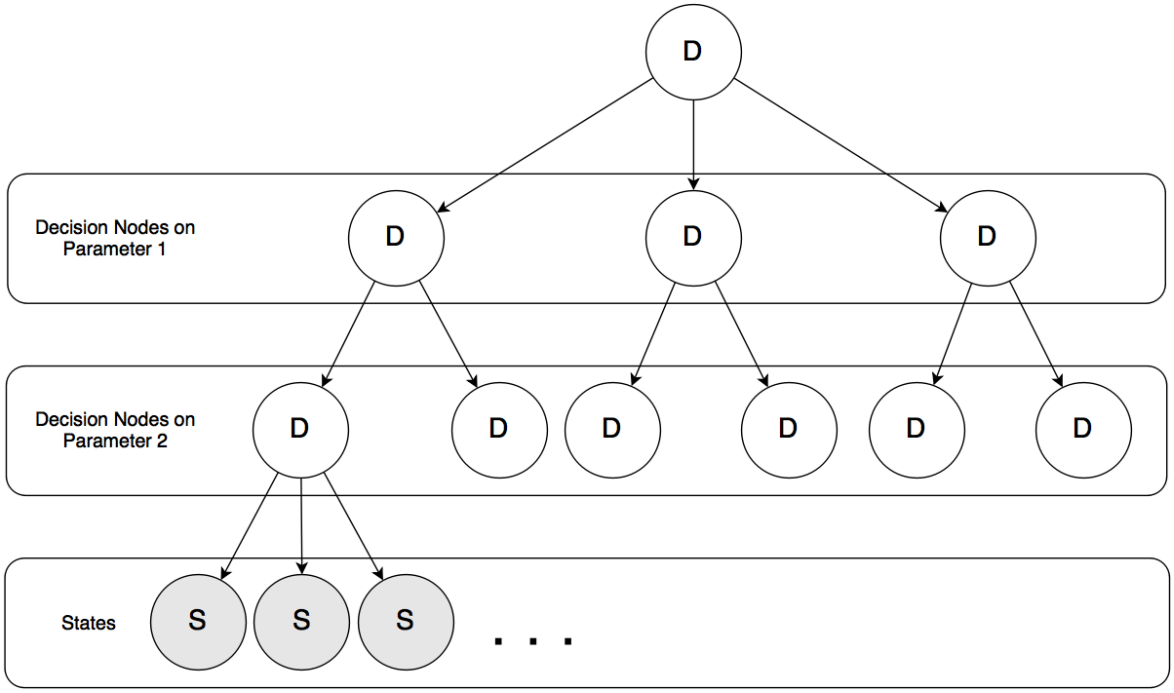


Figure 6.28: A decision tree implementing a 2-dimensional grid on the values of two parameters

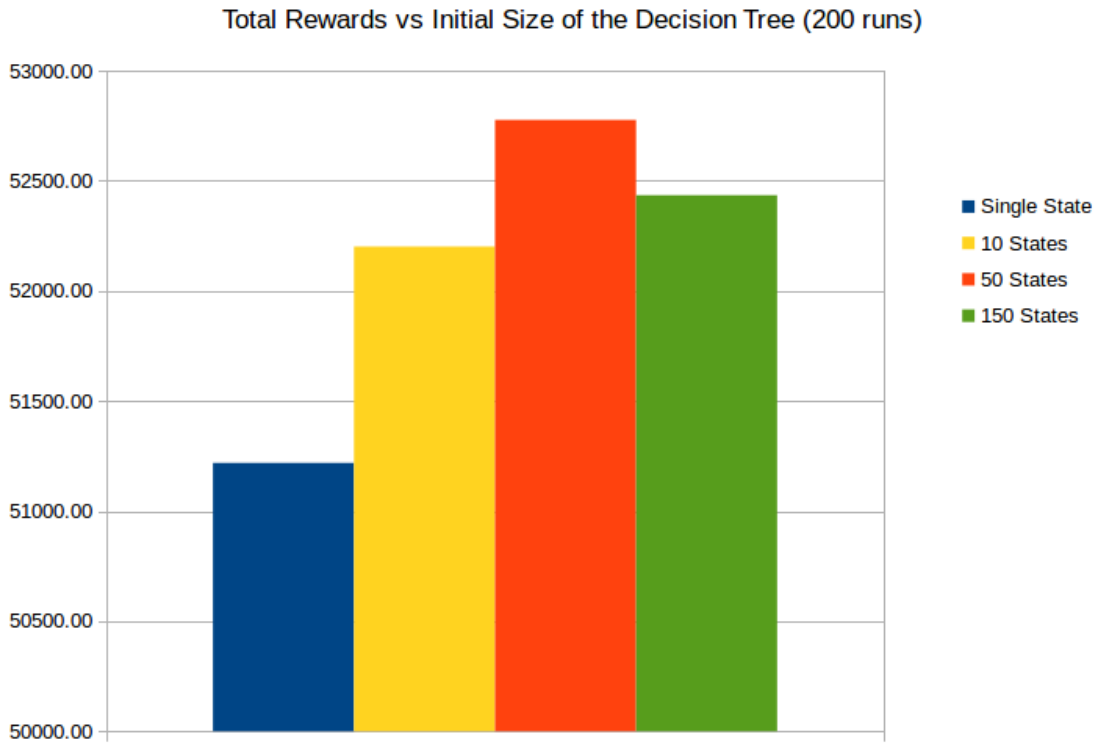


Figure 6.29: The effect of starting with an existing decision tree on the performance

tree only contained 10 states (leaving still a lot of room for additional splits), and the resulting subspaces were easier to handle. On the contrary, adding a third dimension hurt the performance of the algorithm, even though there still was room to perform a large number of splits

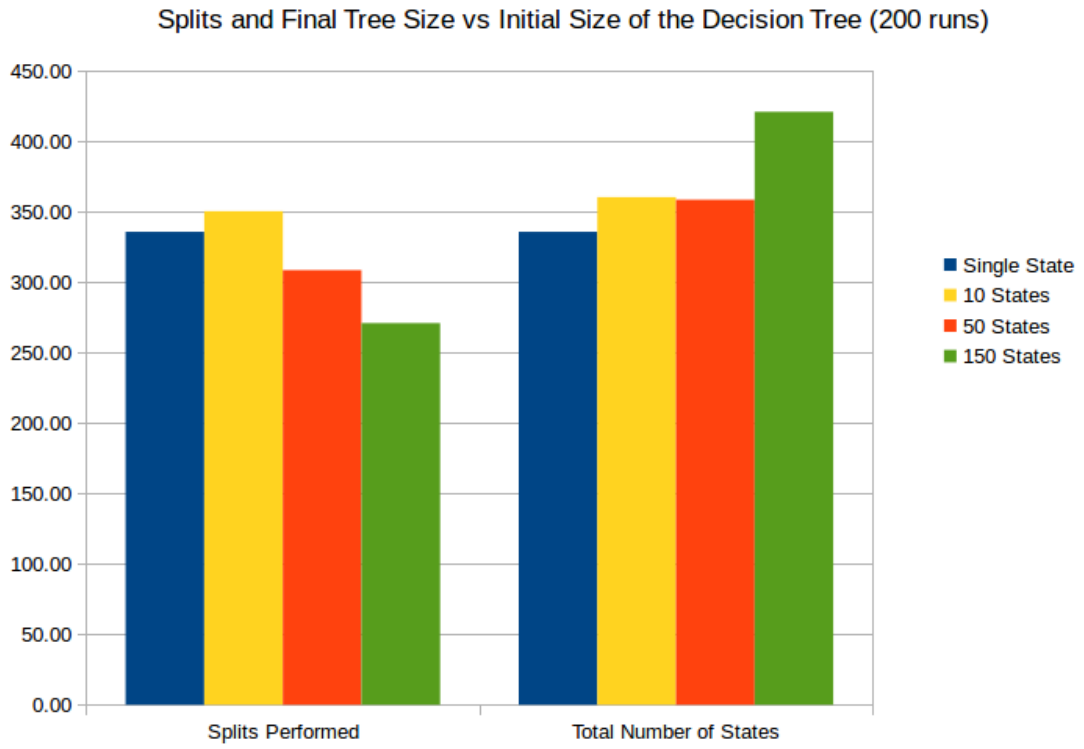


Figure 6.30: The number of splits performed and the final number of states as a function of the initial size of the decision tree

during the run. This is an indication that the state space in this problem can be partitioned more efficiently that an orthogonal grid.

On a more general note, we believe that the ability to begin with a predefined partitioning of the state space and still having the ability to dynamically increase the resolution during the run can be very useful, and can allow decision tree-based models to be used in a much wider range of applications.

6.1.7 Discount Factor

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT with Q-value test, QDT, MDP, Q-learning
- Statistical test for MDPDT: Mann Whitney U test
- Statistical test for QDT: Student's t-test
- Initial Decision Tree: Single State

- MDP states: 3-dimensional grid (500 states)
- Q-learning states: 3-dimensional grid (500 states)
- Update strategy for MDPDT and MDP: Prioritized Sweeping
- Discount Factor: $\gamma \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$
- Statistical test max error for MDPDT: 0.005
- Statistical test max error for QDT: 0.002
- Minimum number of experiences to split: 2 per resulting state for MDPDT, 10 total for QDT

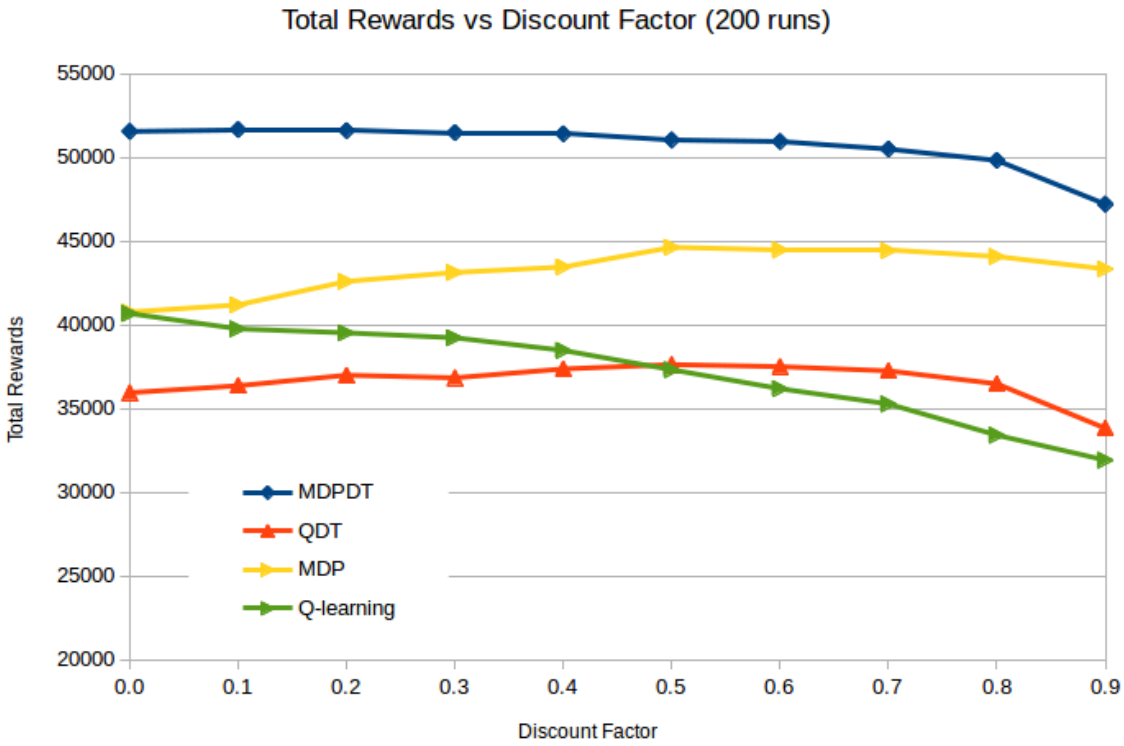


Figure 6.31: Performance for four different models as a function of the discount factor

The discount factor γ is the parameter that controls how much the algorithms consider future rewards in their decisions. On a first approach, one would expect that increasing the discount factor would improve performance, since it would allow the agent to make more strategic decisions that are better in the long run. However, there are two important characteristics of the scenario we are using that cause a high discount factor to negatively influence performance.

First, the transitions in this scenario are noisy. Out of the three parameters that influence the performance of the cluster, two vary independently of the actions of the agent (the incoming load and the types of the requests). This means that judging the effect of the actions by

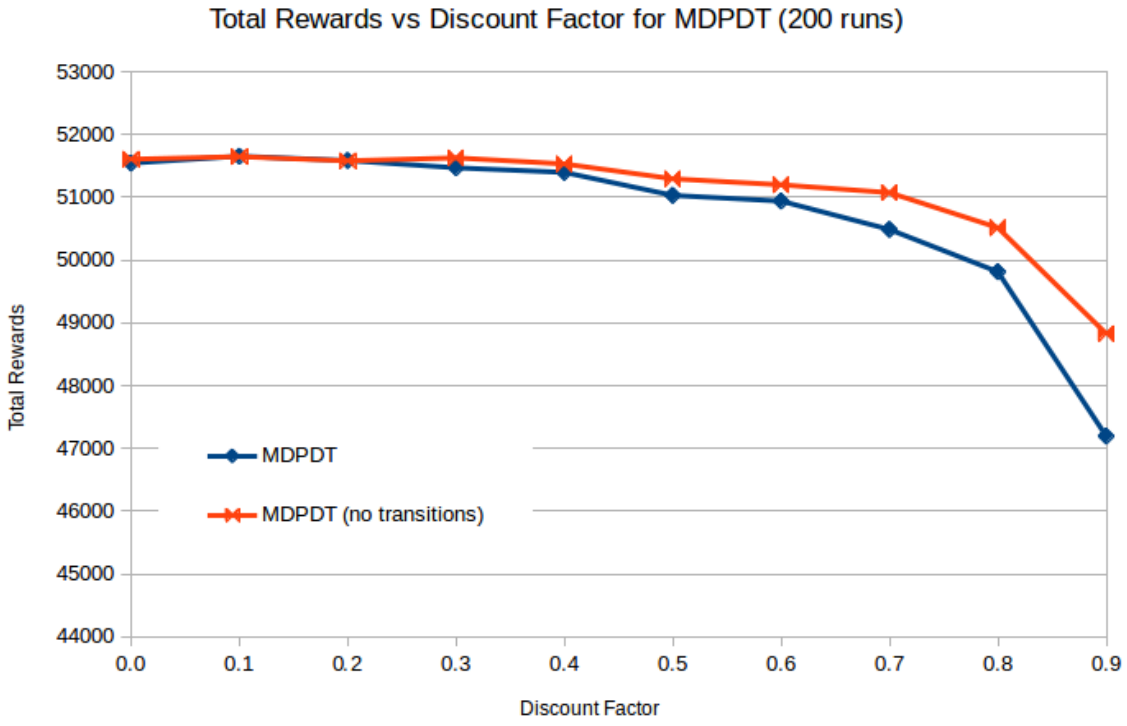


Figure 6.32: The effect of ignoring transitions on MDPDT splits for different values of the discount factor

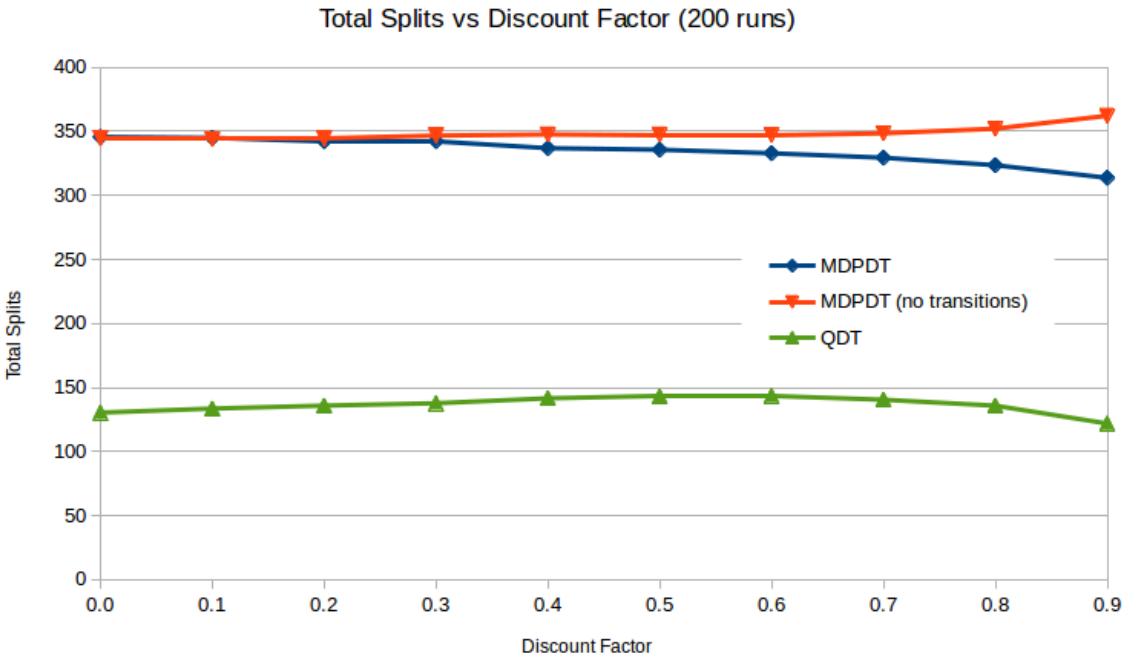


Figure 6.33: The effect of ignoring transitions on the total amount of splits performed by MDPDT as a function of the discount factor

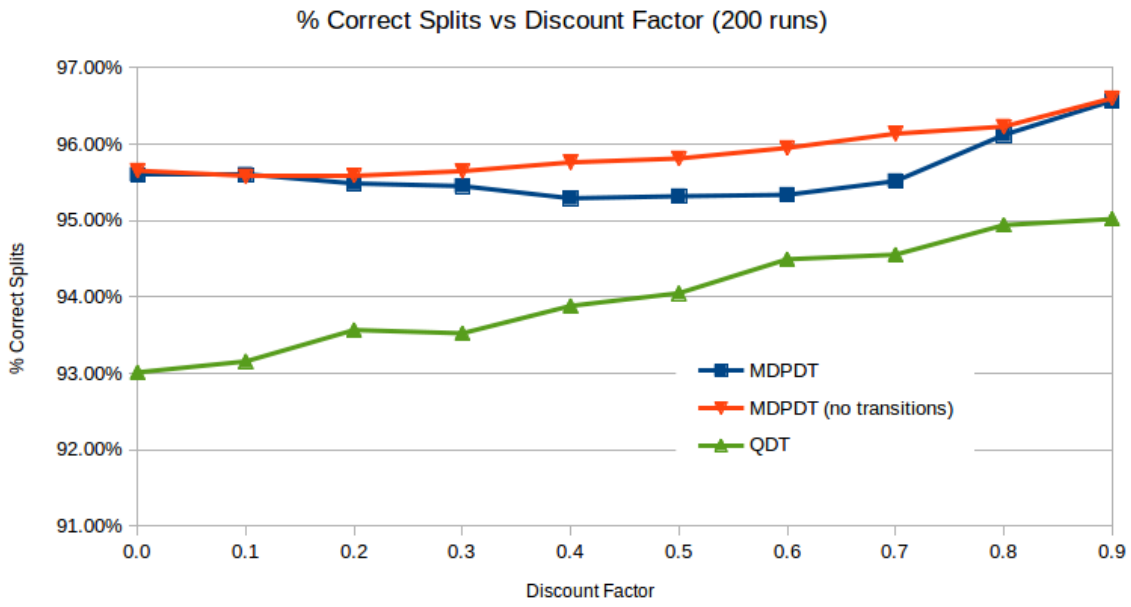


Figure 6.34: The effect of ignoring transitions on the accuracy of the splits for both models as a function of the discount factor

the value of the resulting state can lead to false judgments and suboptimal actions. Second, in this type of scenario, good actions generally offer greater immediate rewards, and so greedy choices are not punished.

In the case of the MDP model, this noise is overcome by the fact that the values of the actions are calculated as the averages of the instantaneous values deriving from each experience. Thus, in MDP's case, the performance increases with higher values of the discount factor. In the case of Q-learning however, since the algorithm does not calculate averages but instead values recent experiences more that past ones, this noisiness significantly hurts its performance.

When it comes to the decision tree models, the transitions affect not only the evaluation of the actions, but also the splitting of the states. In order to identify this influence, we tested the following modification to the *Q-value test* criterion in MDPDT: instead of calculating the instantaneous q-values for each experience (which are equal to the immediate reward plus the value of the resulting state multiplied by the discount factor) we ignored the value of the resulting state and instead only used the value of the immediate reward. This is equivalent to making splitting decisions with a zero discount factor. This modification allowed MDPDT to maintain the same number of splits for higher values of the discount factor as it did with a value of zero (figure 6.33). On the contrary, the default version of *Q-value test* which takes transitions into account had its total number of splits suffer for high values of the discount factor.

6.1.8 Exploration Strategy

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- Algorithm: MDPDT, QDT, MDP, Q-learning
- Splitting Criterion for MDPDT: Parameter test
- Statistical test for MDPDT: Mann Whitney U test
- Statistical test max error: 0.005
- Initial Decision Tree for MDPDT: Single State
- MDP states: 3-dimensional grid (500 states)
- Q-learning states: 3-dimensional grid (500 states)
- Discount Factor: $\gamma = 0.5$
- Learning Rate for Q-learning and QDT: $\alpha = 0.5$
- Minimum number of experiences to split: 2 per resulting state for MDPDT, 10 total for QDT

In a reinforcement learning context, where the agent is expected to learn the behavior of the world from the outcomes of its actions, it is important that there is some mechanism that will allow the exploration of all options before converging to a single policy. One simple way to accomplish that is through the use of an e-greedy exploration strategy. In this strategy e is simply the probability that a random action will be selected during the training phase. If set to zero, the agent may never explore different alternatives and can get stuck in repeating a suboptimal action forever. If set to one, the agent behaves completely randomly during the training.

At first glance we would expect that opting for a high value will yield better results since the agent will have a better chance of exploring all available options. The only downside to this is that setting it to a slightly lower value will direct the agent to spend more time in more fruitful regions of the state space, thus understanding these regions better. This is exactly the behavior we see in figures 6.35 and 6.36 for MDP and Q-learning. A value of zero completely prevents the agent from exploring any options while higher values achieve better results (figures 6.39 - 6.41).

In the case of the decision tree algorithms though the behavior is different. Increasing the exploration constant too much seems to hurt performance. The reason behind that is simple. Splits only take into account the optimal action. This means that the more the agent performs the best available option, the more data are available to grow the decision tree. This is clear

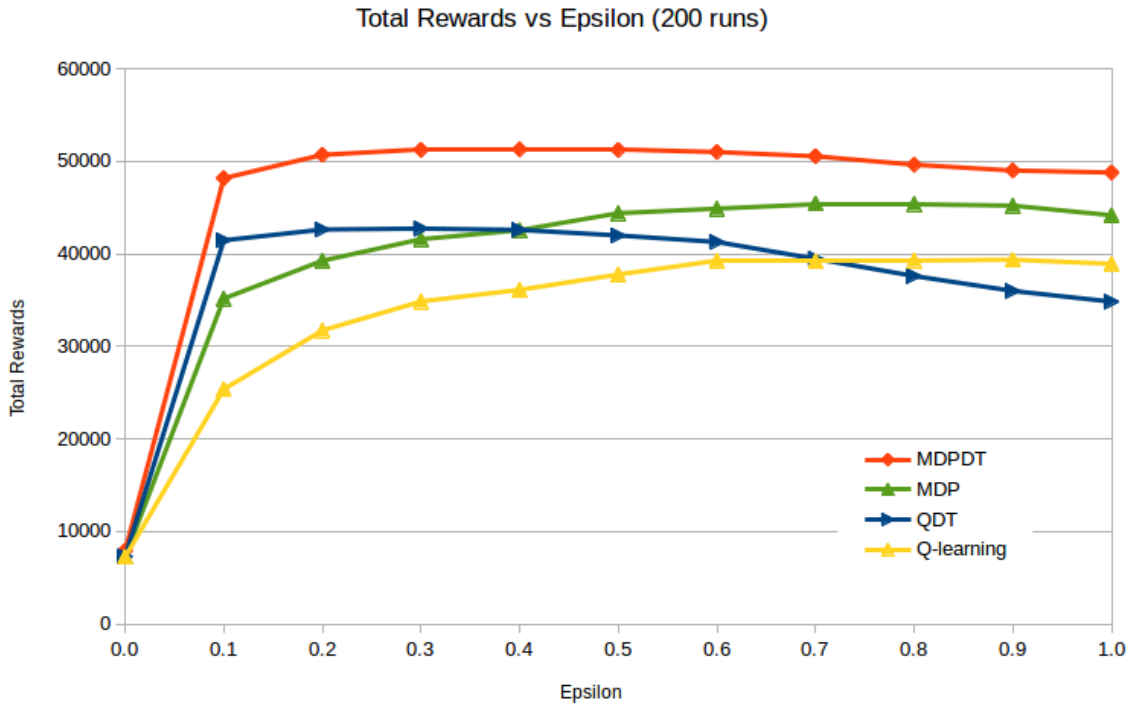


Figure 6.35: The performance of four algorithms for different levels of the exploration constant ϵ

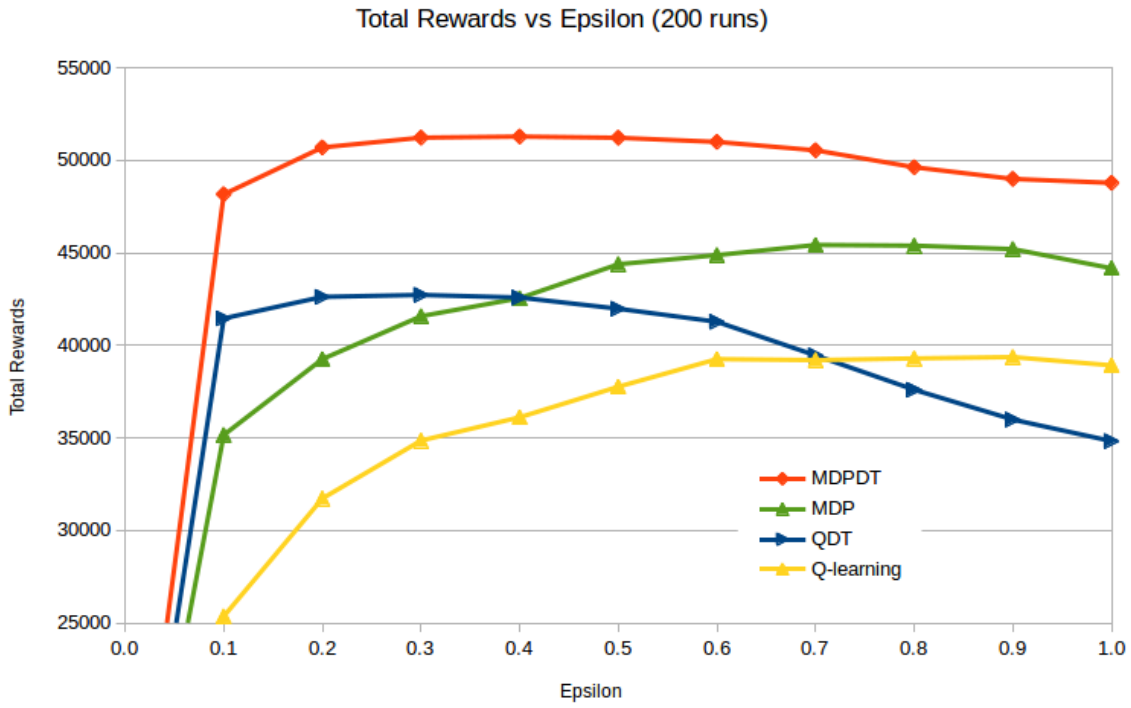


Figure 6.36: Zoom in on figure 6.35

in figures 6.37 and 6.38. Even though the accuracy of the splits does not drop (and even to a certain extent improves) at high values of ϵ, reflecting a better understanding of the world,

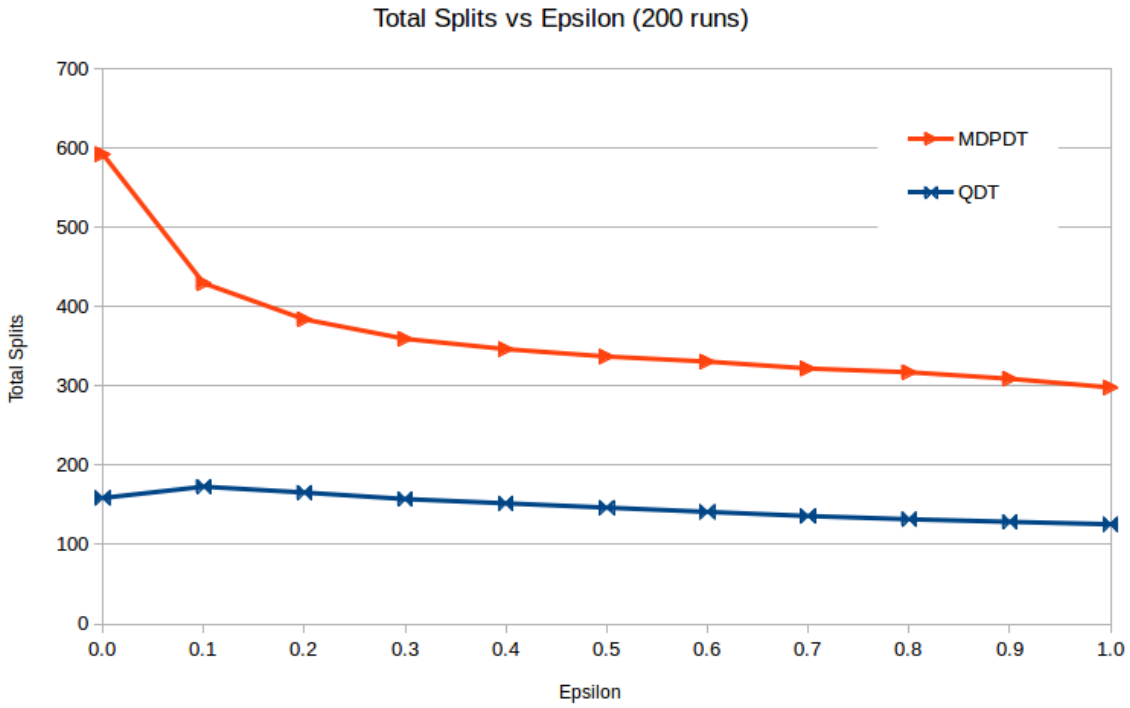


Figure 6.37: Total number of splits as a function of the exploration constant

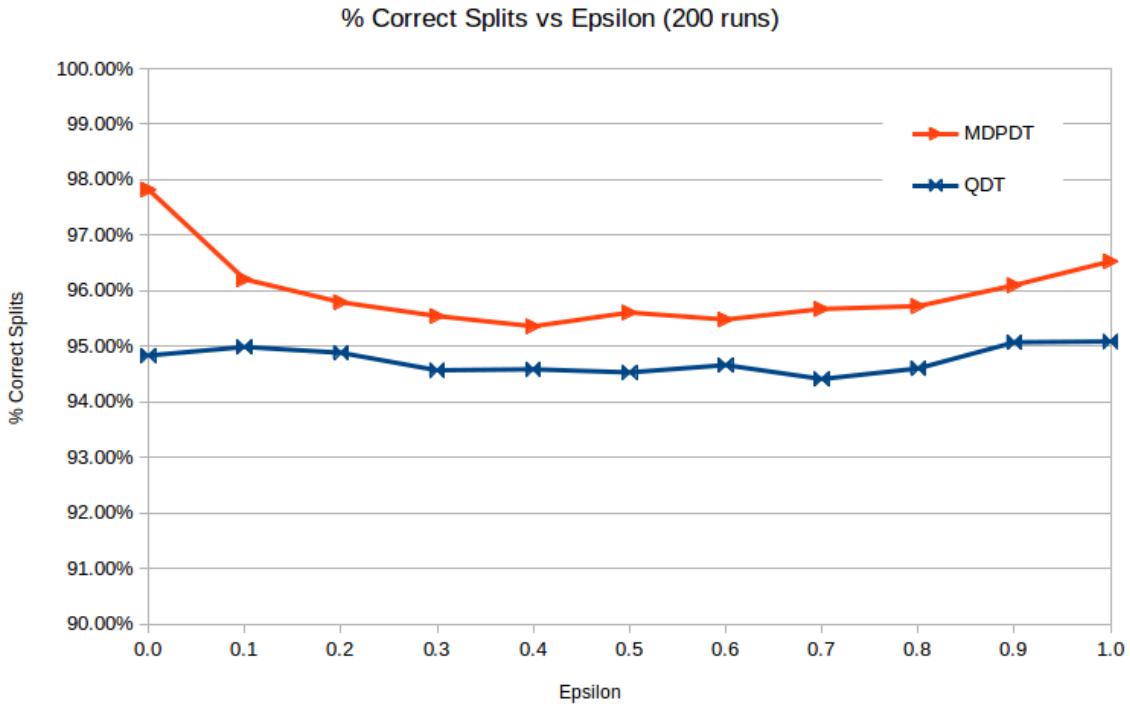


Figure 6.38: Percentage of splits on parameters that affect the behavior of the system as a function of the exploration constant

their total number drops, overall hurting the performance. Most characteristically, by far the largest number of splits is achieved by MDPDT when $\epsilon = 0$, since the agent is stuck in

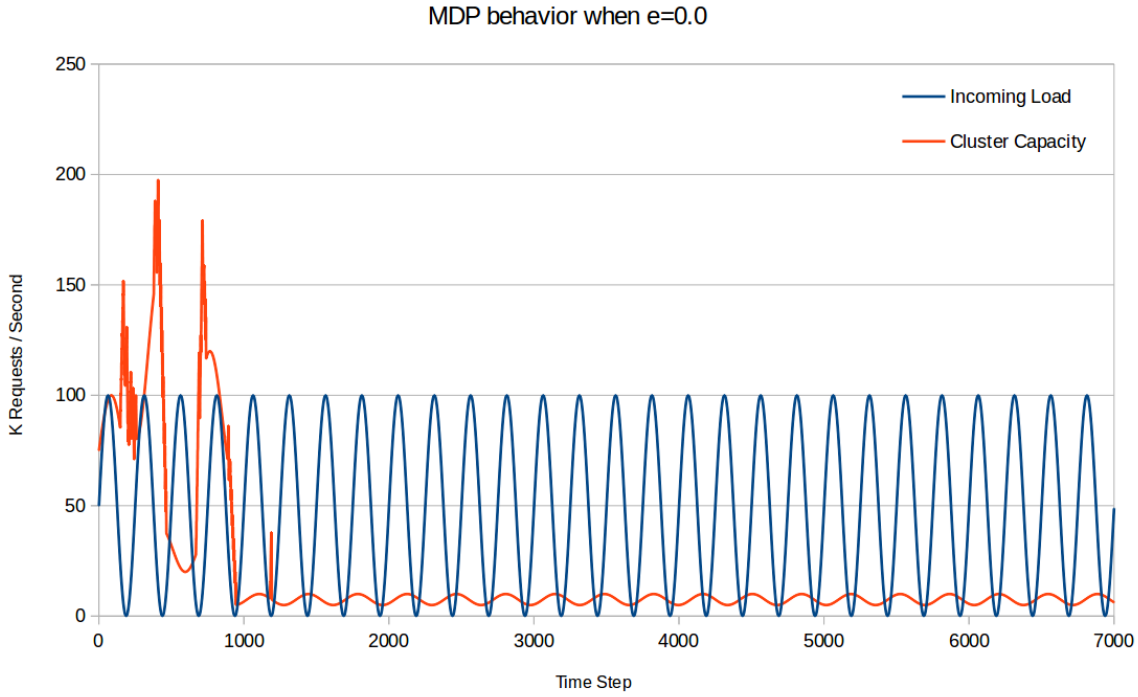


Figure 6.39: MDP behavior when $\epsilon = 0.0$. The fluctuations in the capacity of the cluster are caused by the types of the requests. The size of the cluster remains constant.

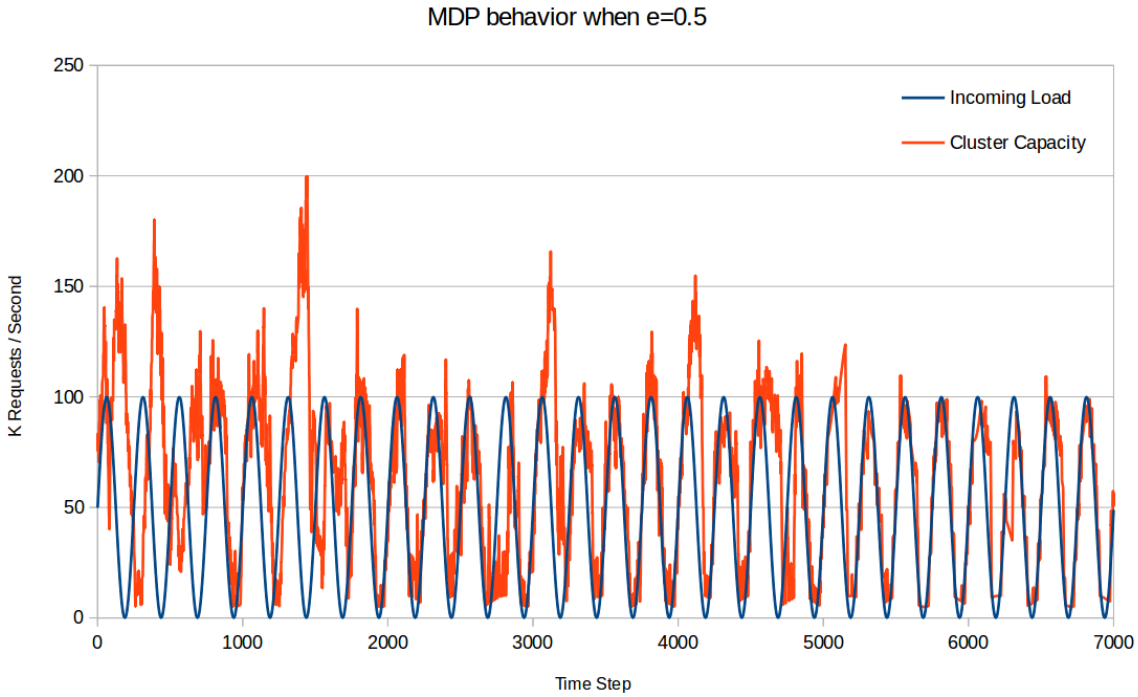


Figure 6.40: MDP behavior when $\epsilon = 0.5$. The exploration is focused around the better regions of the state space.

repeating the same action.

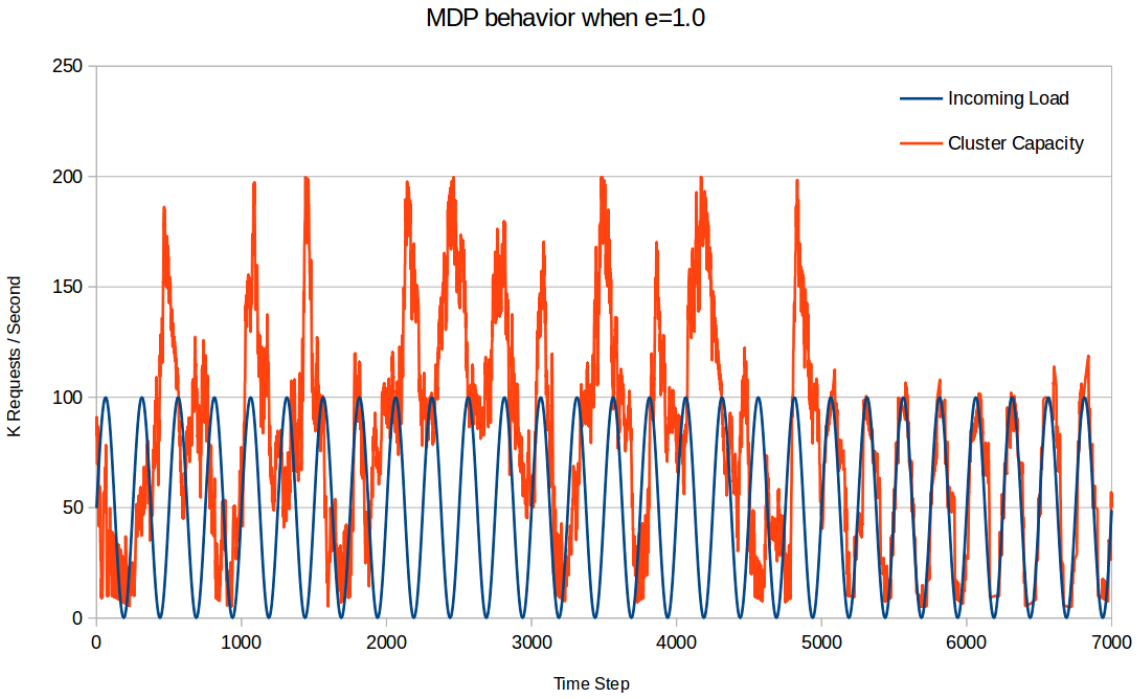


Figure 6.41: MDP behavior when $\epsilon = 1.0$. The exploration is completely random.

6.1.9 Update Algorithm

Setup:

- Training steps: 5000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: MDPDT with Parameter test, MDP
- Statistical test: Mann Whitney U test
- Statistical test max error: 0.005
- Initial Decision Tree for MDPDT: Single State
- MDP states: 3-dimensional grid (500 states)
- Update strategy: {Single Update, Prioritized Sweeping, Value Iteration}
- Discount Factor: $\gamma = 0.5$
- Minimum number of experiences to split for MDPDT: 2 per resulting state

One of the advantages of the full-model based algorithms is the ability to perform global updates to the model when new experiences are acquired. In this experiment, we compare three common update algorithms in terms of performance and execution time.

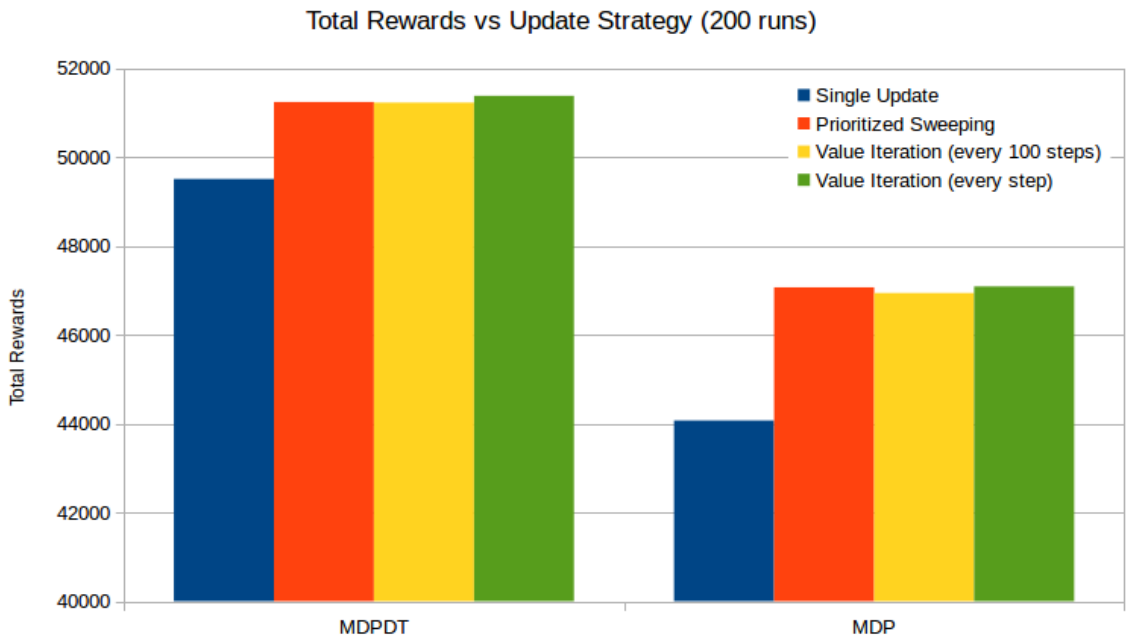


Figure 6.42: Performance of MDPDT and MDP for different update algorithms

As expected, the most simple update algorithm, which is to perform a single update on the value of the initial state with each new experience, turned out to be the fastest but also the worst performing one (figure 6.42). This algorithm is similar in logic to the Q-learning update rule, and thus suffers from the disadvantages of Q-learning discussed on Chapter 4.

On the other hand, Value Iteration is guaranteed to fully update the values of all states, and is expected to achieve the maximum possible performance, which was indeed verified. This is indicative of the importance of global updates in the model, since as discussed before, this is a problem where good actions are generally rewarded immediately. This means that one would only expect the effect of global updates to be much greater in scenarios where a long series of actions is needed before acquiring a reward. However, when it comes to execution time, Value Iteration was 3 orders of magnitude slower than performing a single update at each step (table 6.1). This is of course expected, since the time complexity of Value Iteration is $O(S^2A)$ per iteration, and in this experiment often more than 10 iterations were needed before converging.

Prioritized Sweeping, discussed in paragraph 4.3.6, is an algorithm that attempts to combine the advantages of both previous methods, and in this experiment manages to do exactly that. By performing only targeted updates, it was 2 orders of magnitude faster than Value Iteration while being only slightly less effective. Finally, we tested the performance of running single updates but also periodically updating the model through Value Iteration, in this case every 100 steps. The results of this approach were also very competitive. However, even though this approach has a total running time similar to Prioritized Sweeping, the worst-case performance per update is at least as bad as Value Iteration, making it a weaker option in comparison.

Algorithm	Times slower than Single Update
Single Update	1
Prioritized Sweeping	11
Value Iteration	2270

Table 6.1: Execution time for MDP in comparison to performing a single update per experience

6.1.10 Learning Rate

Setup:

- Training steps: 20000
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithm: QDT, Q-learning
- Statistical test: Student's *t*-test
- Q-learning states: 3-dimensional and 2-dimensional grid
- Discount Factor: $\gamma = 0.5$
- Learning Rate: $\alpha = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- *t*-test max error: 10^{-3}
- Minimum number of experiences: 20

The learning rate is the parameter that controls how fast Q-learning learns from new experiences. The higher its value, the more the algorithm values new experiences over previous ones. If set equal to 1, the algorithm completely forgets its history and only memorizes the value of an action the last time it was taken. On the contrary, if set equal to 0, it learns nothing from new experiences. In order to get a better understanding of the affect the learning rate has on the algorithm's behavior, we tested the performance of Q-learning, as well as QDT which is based on Q-learning, on our scenario using different values for the parameter.

The behavior of the system in this scenario depends on the values of three parameters: the size of the cluster, the incoming load and the types of the queries. In order to fully model this behavior we would need a 3-dimensional grid, partitioning the state space based on the values of these three parameters. In that case, since within each cube of the state space the behavior of the system is nearly constant, we would expect that a higher value of the learning rate would be favorable, since the algorithm would be able to immediately learn the correct value of an action. Since that value is not expected to change, forgetting past experiences and only remembering the latest one has no downside. In figure 6.43 we can see that exact

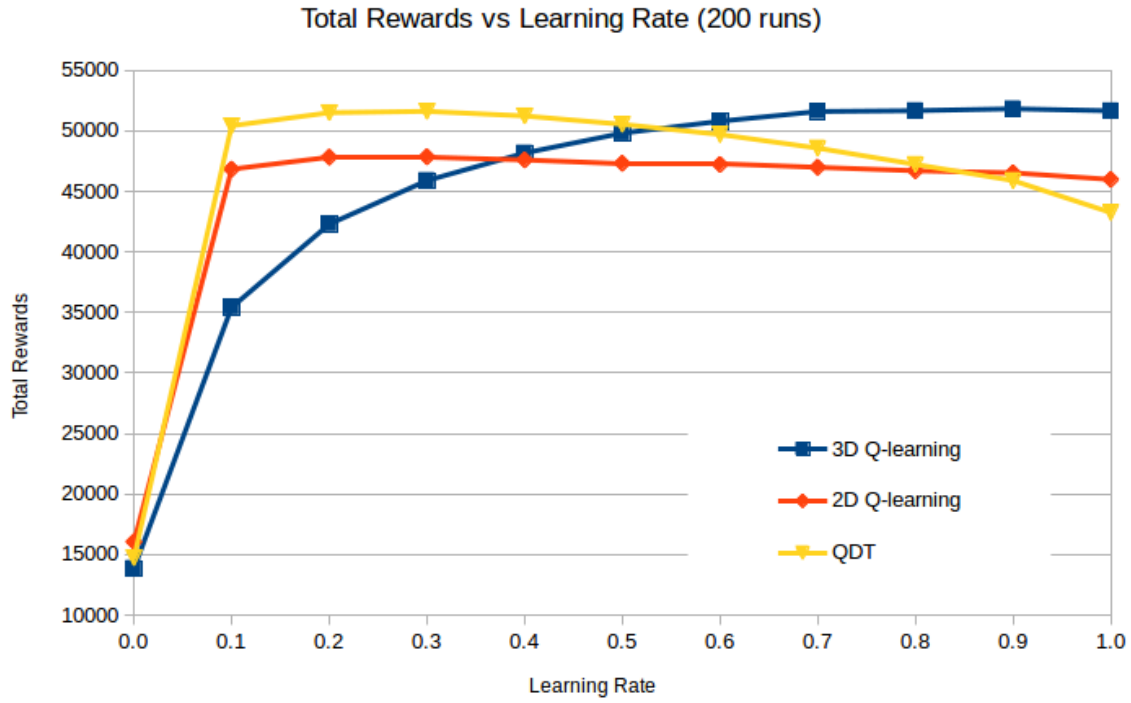


Figure 6.43: The performance of Q-learning and QDT for different values of the learning rate

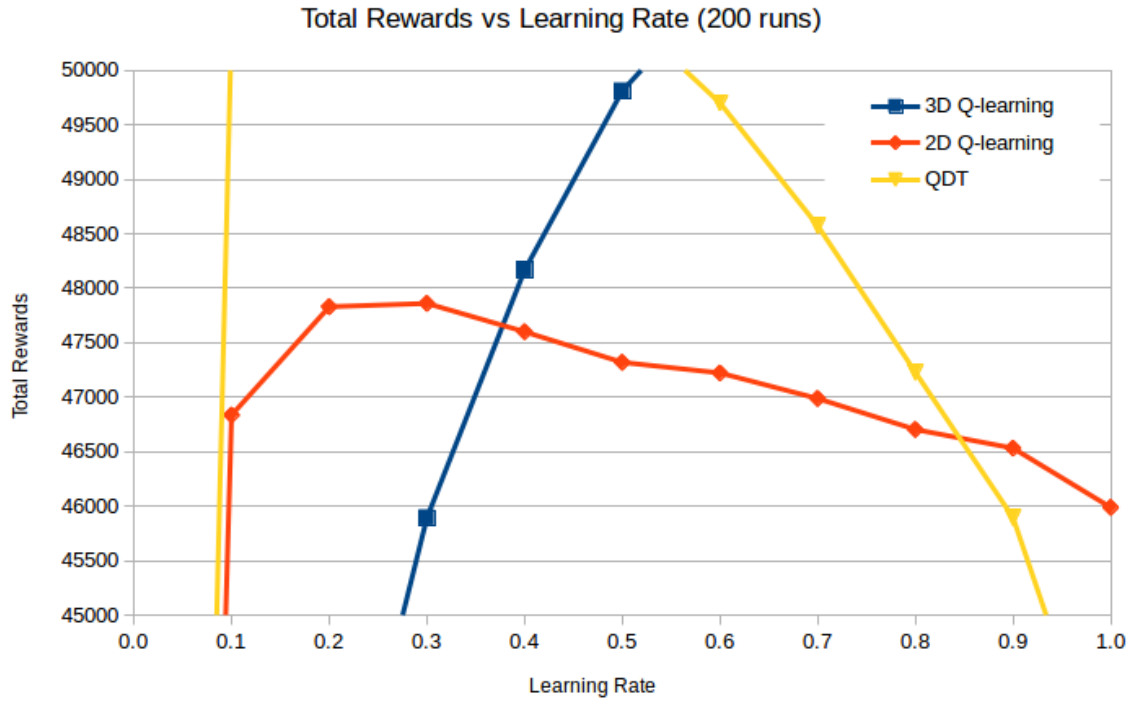


Figure 6.44: Zoom in on the performance of the 2-dimensional Q-learning model

behavior: for the model that partitions the state space using a 3-dimensional grid increasing the learning rate monotonically increases performance.

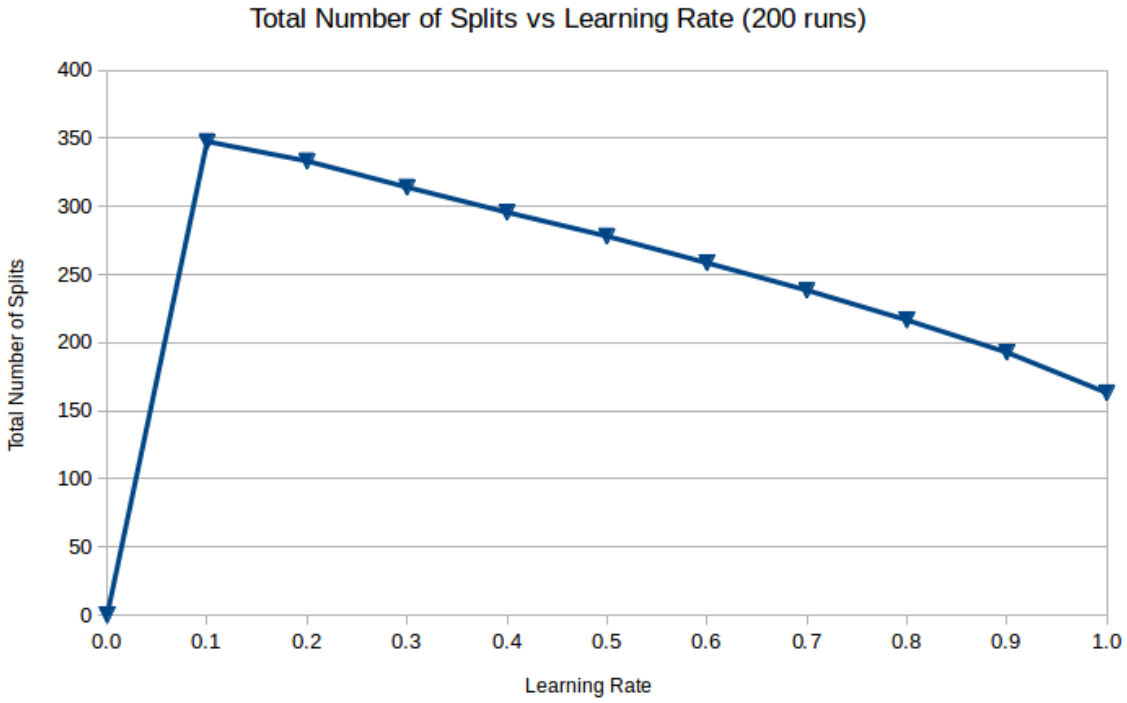


Figure 6.45: The total number of splits performed by QDT for different values of the learning rate α

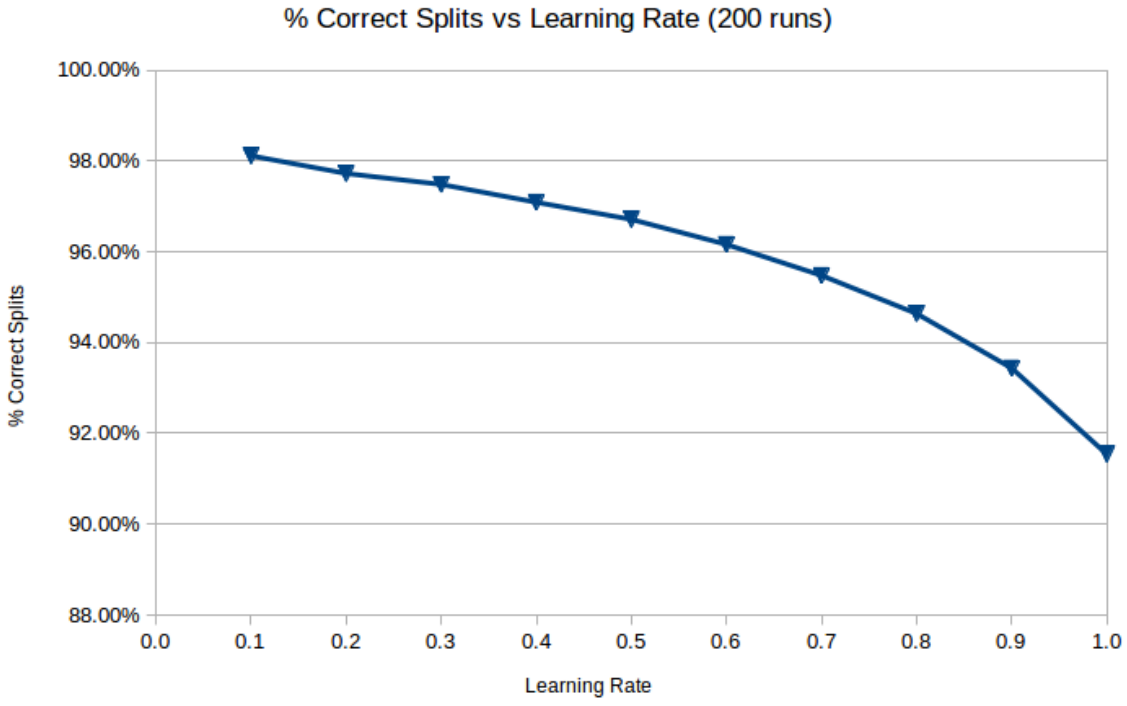


Figure 6.46: The percentage of splits performed on parameters that affect the behavior of the system as a function of the learning rate

In the same setting, we repeated the experiment using a Q-learning model that partitioned the state space using only a 2-dimensional grid, disregarding the effect of the types of the queries. In this case, the behavior of the system within each state differed significantly, as the value of the third unmodeled parameter changed. Here, using a higher learning rate instead hurt the performance, since the algorithm never got to learn an average behavior within each state. This correlation can be seen more clearly in figure 6.44.

Finally, in the case of QDT, there was a strong preference in a lower learning rate. This behavior is explained by figures 6.45 and 6.46. First of all, as expected, zero splits were made when the learning rate was zero since the Q-values never increased or decreased. However, for values higher than 0.1, the higher the learning rate was, the less splits the algorithm performed, and the less accurate they were, due to the noise in the measured Q-values.

6.2 Performance

In this section we will attempt to compare and evaluate the overall performance of the four algorithms discussed in this work, namely MDP-DT, QDT, MDP and Q-learning. For that purpose we will use two simulation scenarios from the field of cloud computing. The first one is the simple cluster used for the parameterization of the algorithms in chapter 6.1. The second scenario is a more challenging iteration of the first one, whose behavior depends on more parameters in a more complicated manner.

In both scenarios we attempt to evaluate the performance under relatively limited training data, since we believe that this allows us to better understand the efficiency of each solution. When ample training data was available, all solutions converged towards a near optimal behavior, and therefore we find the performance under limited training data more interesting.

6.2.1 Simple Cluster

Setup:

- Cluster size: 1-20 VMs
- Actions: Add 1 VM, Remove 1 VM, do nothing
- Incoming load: $load(t) = 50 + 50sin\left(\frac{2\pi t}{250}\right)$
- Percentage of reads: $r(t) = 0.75 + 0.25sin\left(\frac{2\pi t}{340}\right)$
- If $vms(t)$ is the number of virtual machines in the cluster at time t, the capacity of the cluster is given by: $capacity(t) = 10 \cdot vms(t) \cdot r(t)$
- Reward function: $R_t = min(capacity(t+1), load(t+1)) - 3 \cdot vms(t+1)$
- Training steps $\in \{2000,\ 5000,\ 10000,\ 20000\}$
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$

- Algorithms: MDPDT, QDT, MDP, Q-Learning
- Statistical test for MDPDT: Mann Whitney U test
- Statistical test max error for MDPDT: 0.005
- Initial decision tree size $\in \{1\ state,\ 50\ states\}$
- MDP and Q-Learning number of states $\in \{200,\ 500,\ 1000,\ 2000\}$
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$

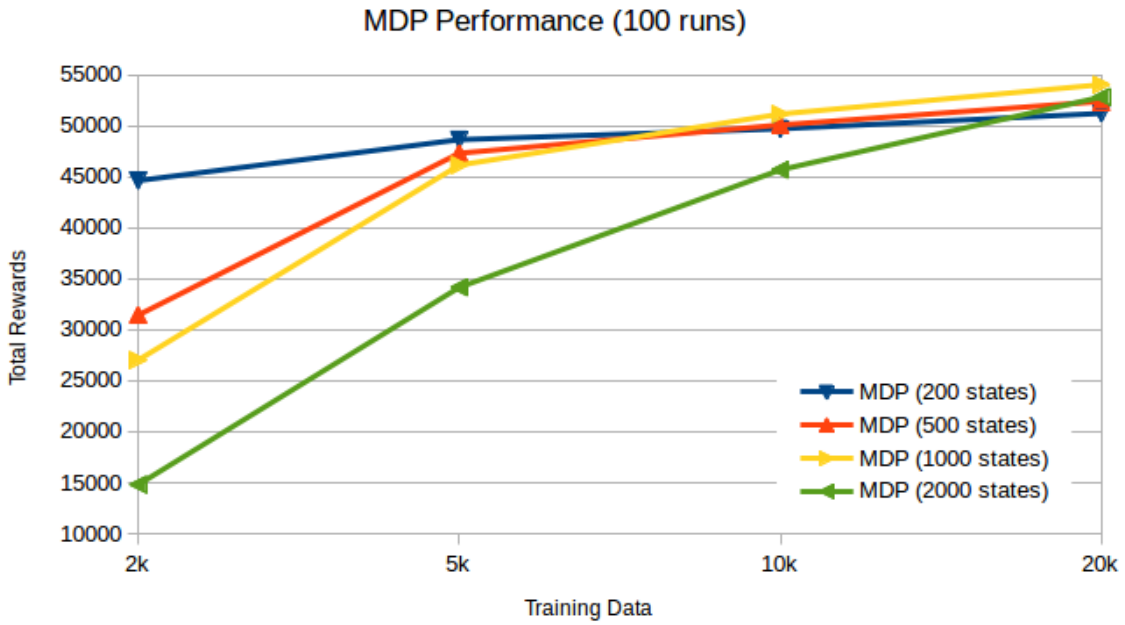


Figure 6.47: The performance of four different MDP models in the simple cluster scenario

Similarly to the experiments in section 6.1, in addition to the three parameters described above (cluster size, incoming load, percentage of reads), the input vector included 7 additional parameters, whose values varied randomly. Four of them followed a uniform distribution within $[0, 1]$, while the rest took integer values within $[0, 9]$ with equal probability.

We tested the performance of the algorithms under varying amounts of training data to observe how it evolves over time. In the case of the MDP and Q-Learning models, which have a fixed number of states, we tested four different state configurations. In the case of the decision tree based models we tested their performance when their initial decision tree is a single state, or a small tree with 50 states (a small grid over the number of VMs and the incoming load).

In both the cases of the MDP and the Q-learning model, in figures 6.47 and 6.48, the smaller models achieved a better performance when trained with the smaller dataset, but got outperformed by the larger models when more data was available. This was expected since

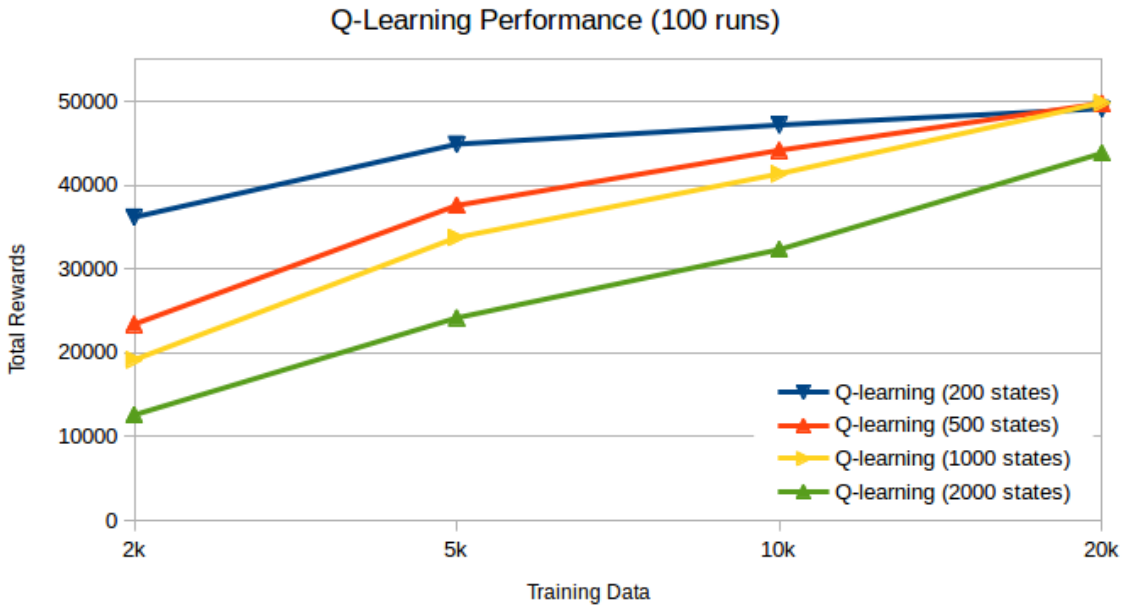


Figure 6.48: The performance of four different Q-Learning models in the simple cluster scenario

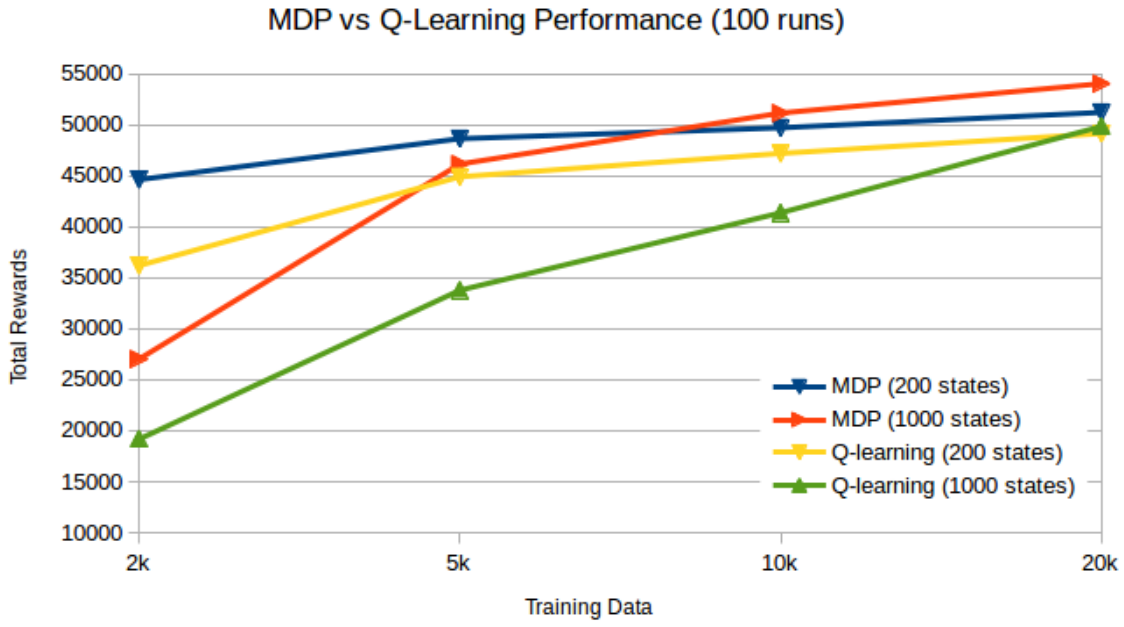


Figure 6.49: Performance comparison between MDP and Q-Learning in the simple cluster scenario

the more states a model has, the more data it requires to be trained, but once trained having more states allows the larger models to be more accurate. In the case of the 2000-state models, even the 20k timesteps dataset was not enough to train them adequately, and so even though they covered most of the distance towards the smaller models, they still did not manage to get trained in time. This indicates that the performance we see is roughly what can be expected

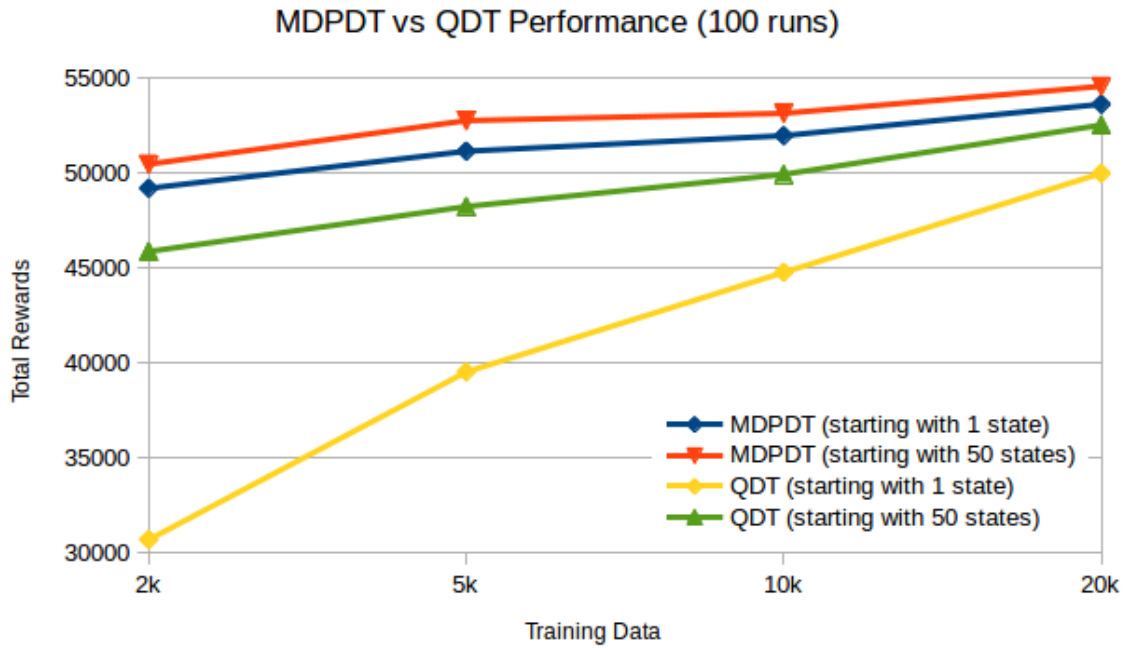


Figure 6.50: Performance comparison of the decision tree based models in the simple cluster scenario

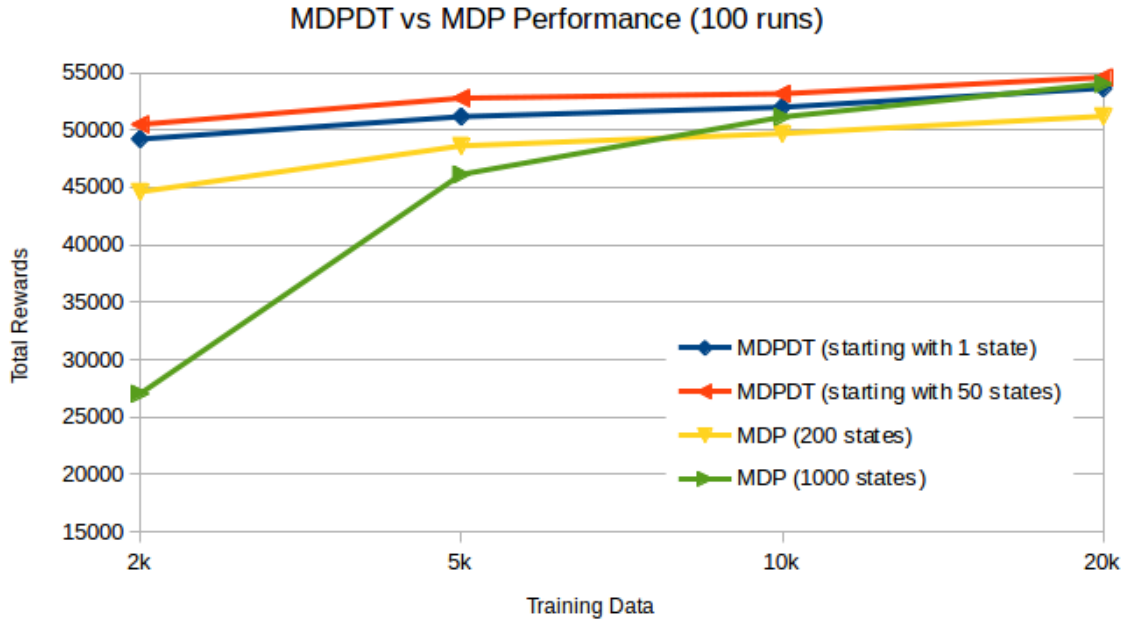


Figure 6.51: Performance comparison of the full-model decision tree based model with its fixed size counterpart in the simple cluster scenario

from these models with this size of a training set.

Comparing the performance of MDP and Q-Learning in figure 6.49, we see a clear win for the full-model based approach. This is expected, since the algorithm maintains much more information about the world, and immediately spreads any information gained from new

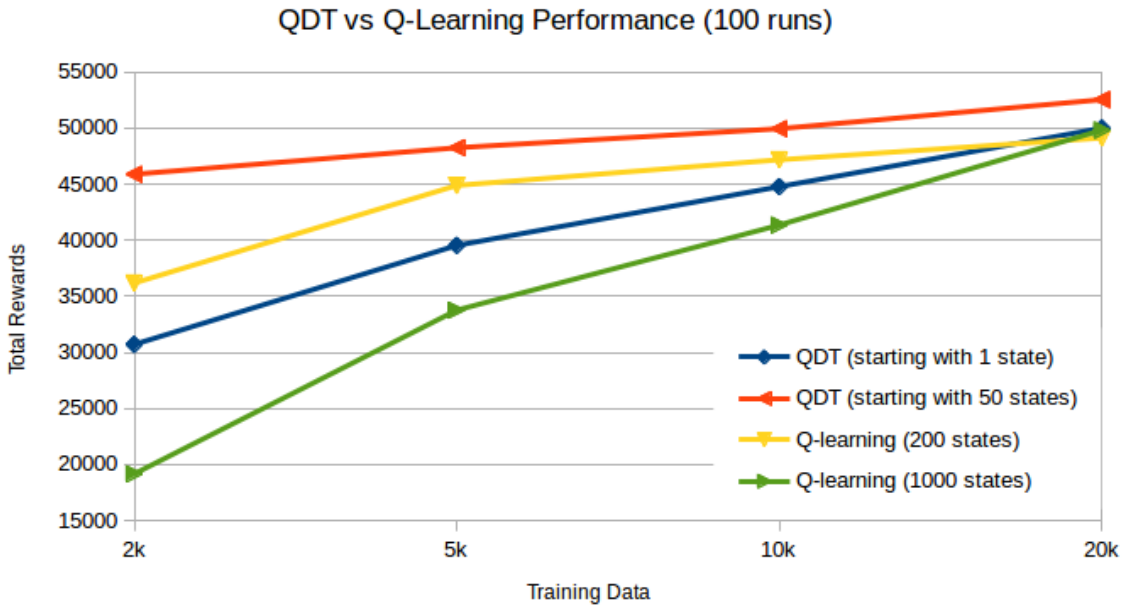


Figure 6.52: Performance comparison of the Q-Learning decision tree based model with its fixed size counterpart in the simple cluster scenario

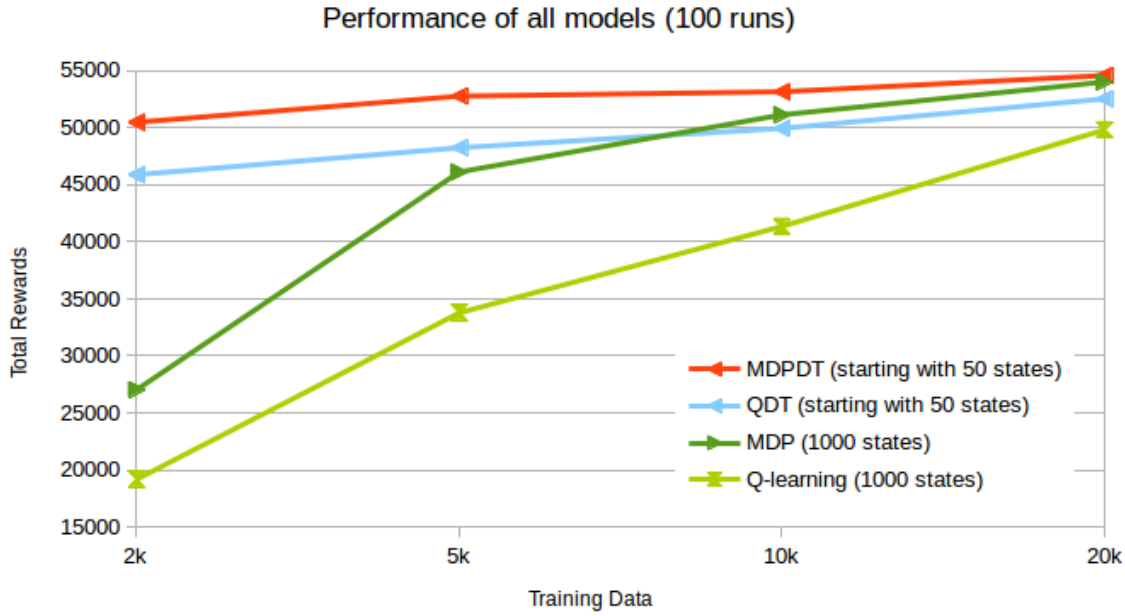


Figure 6.53: Performance for all models in the simple cluster scenario

experiences to the whole model thanks to Prioritized Sweeping. Of course, this performance advantage comes at the cost of a significantly increased computation time, but in the setting of cloud computing applications the required time per update for all models is trivial.

This performance advantage of the full-model based approach is also clear in the case of the decision tree based models (figure 6.50). This gap is of course also widened by the fact that the Q-Learning based approach did not reuse its training data after node splits, essentially

wasting useful information. Additionally, we also see a noticeable boost in performance by adding a small amount of states in the initial tree in all situations.

The comparison of the fixed-size models with the decision tree based models can be seen in figures 6.51 and 6.52. This comparison is unfair for the decision tree based models, since they had to figure out how to partition the state space in a noisy environment, while the fixed size counterparts were provided with an evenly partitioned state space on exactly the parameters that mattered. However, they still managed to perform exceptionally well, and when provided with a little information about the state space in the form of a small set of 50 starting states, they clearly outperformed them. Additionally, one more advantage of the decision tree based approaches is apparent. Fixed size models have to make a choice between a small model that learns quickly but is inaccurate, and a bigger model that is more accurate but learns slowly. Decision tree models on the other hand, by increasing their number of states dynamically, can work well in both situations using the same settings.

6.2.2 Complex Cluster

Setup:

- Cluster size: 1-20 VMs
- Actions: Add 1 VM, Remove 1 VM, do nothing
- Incoming load: $load(t) = 50 + 50sin\left(\frac{2\pi t}{250}\right)$
- Percentage of reads: $r(t) = 0.7 + 0.3sin\left(\frac{2\pi t}{340}\right)$
- I/O operations per second: $IO(t) = 0.6 + 0.4sin\left(\frac{2\pi t}{195}\right)$
- I/O penalty: $IO_{pen}(t) = \begin{cases} 0 & \text{if } 0.7 > IO(t) \\ IO(t) - 0.7 & \text{if } 0.7 \leq IO(t) < 0.9 \\ 0.2 & \text{if } 0.9 \leq IO(t) \end{cases}$
- If $vms(t)$ is the number of virtual machines in the cluster at time t, the capacity of the cluster is given by: $capacity(t) = 10(r(t) - IO_{pen}(t))vms(t)$
- Reward function: $R_t = min(capacity(t+1), load(t+1)) - 2vms(t+1)$
- Training steps $\in \{2000,\ 5000,\ 10000,\ 20000,\ 50000\}$
- Evaluation steps: 2000
- Exploration strategy: e-greedy with $e = 0.5$
- Algorithms: MDPDT, QDT, MDP, Q-Learning
- Statistical test: Mann Whitney U test
- Statistical test max error: 0.005
- Initial decision tree size $\in \{1\ state,\ 50\ states\}$

- MDP and Q-Learning number of states $\in \{300,\ 800,\ 1440,\ 2400\}$
- Update strategy: Prioritized Sweeping
- Discount Factor: $\gamma = 0.5$
- Minimum number of experiences to split: 2 per resulting state

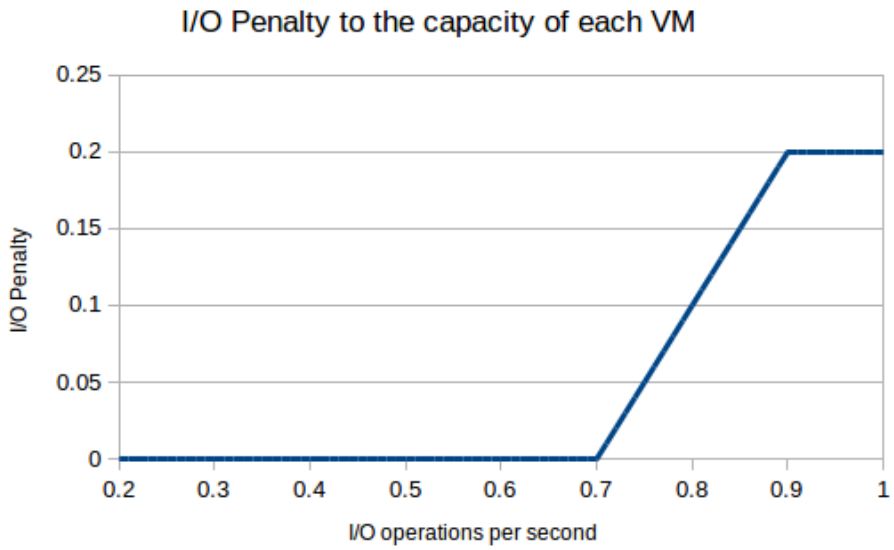


Figure 6.54: The penalty to the capacity of each VM in the complex cluster scenario

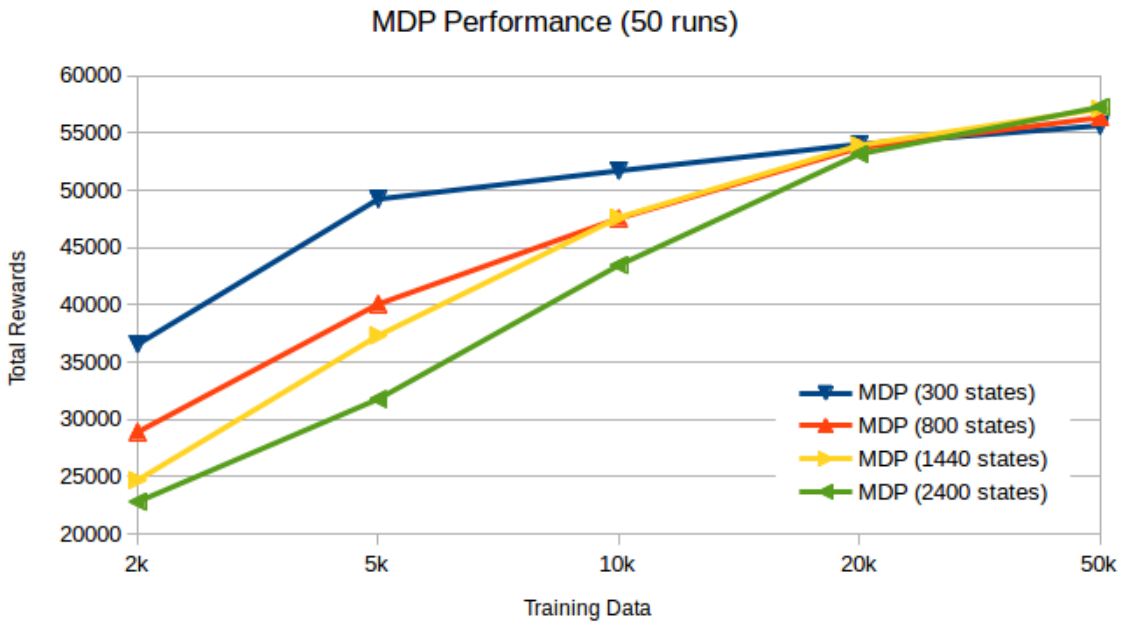


Figure 6.55: The performance of four different MDP models in the complex cluster scenario

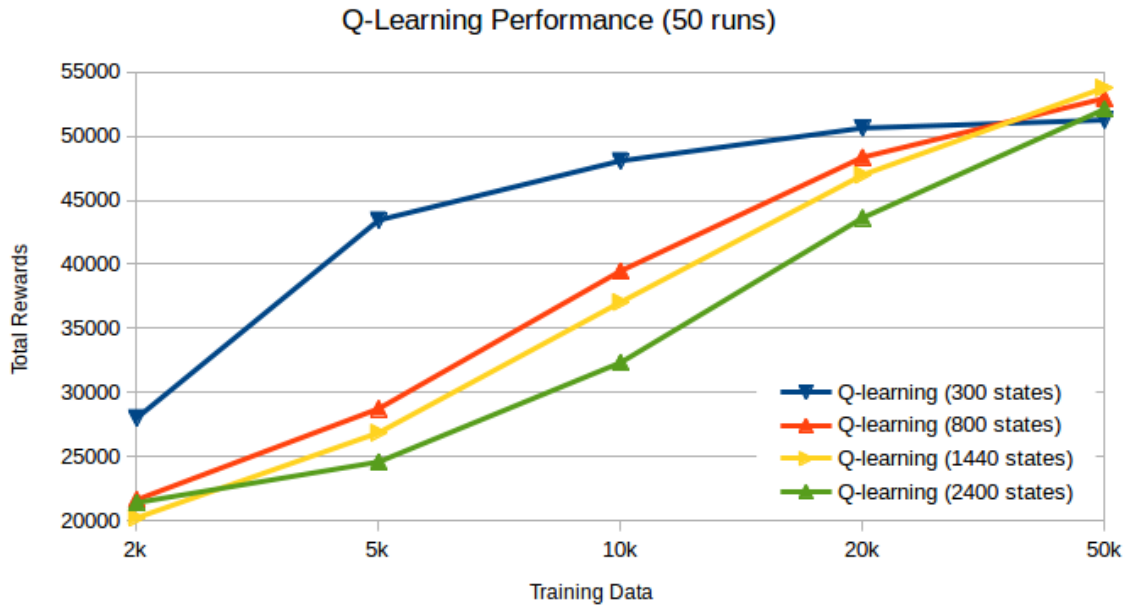


Figure 6.56: The performance of four different Q-Learning models in the complex cluster scenario

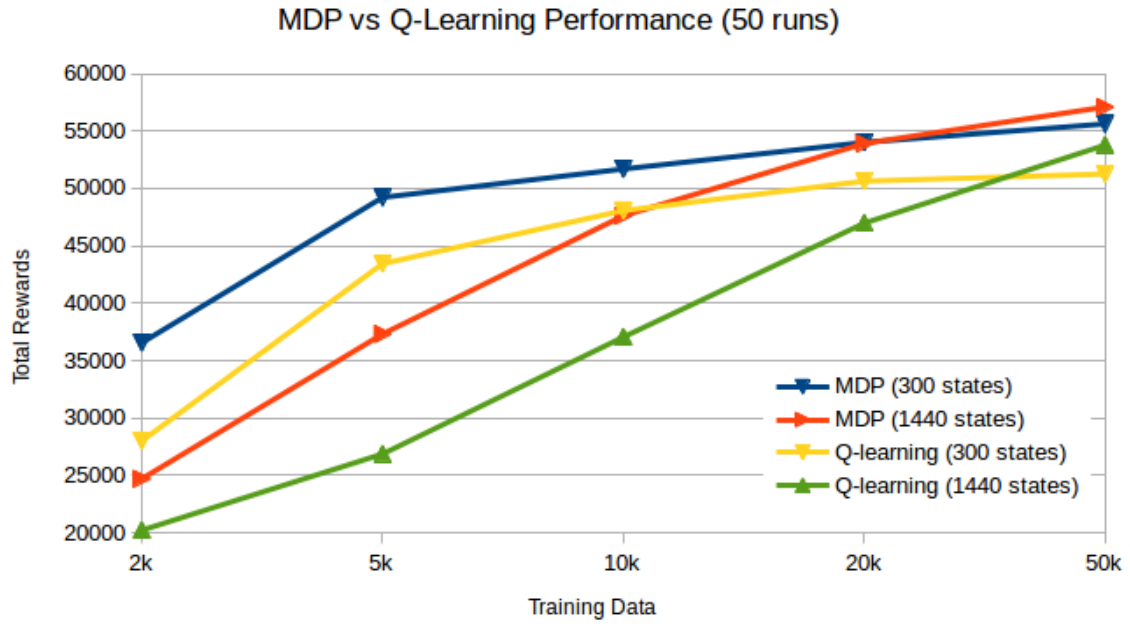


Figure 6.57: Performance comparison between MDP and Q-Learning in the complex cluster scenario

To increase the difficulty of this scenario compared to the previous one, we have increased the effect of the types of the queries to the capacity of the cluster, and added one more parameter that affects the behavior of the system in a non-linear manner, namely the I/O operations per second. This parameter takes values between 0.2 and 1.0, but only affects the performance of the cluster if its value is higher that 0.7 by adding a penalty to the per-

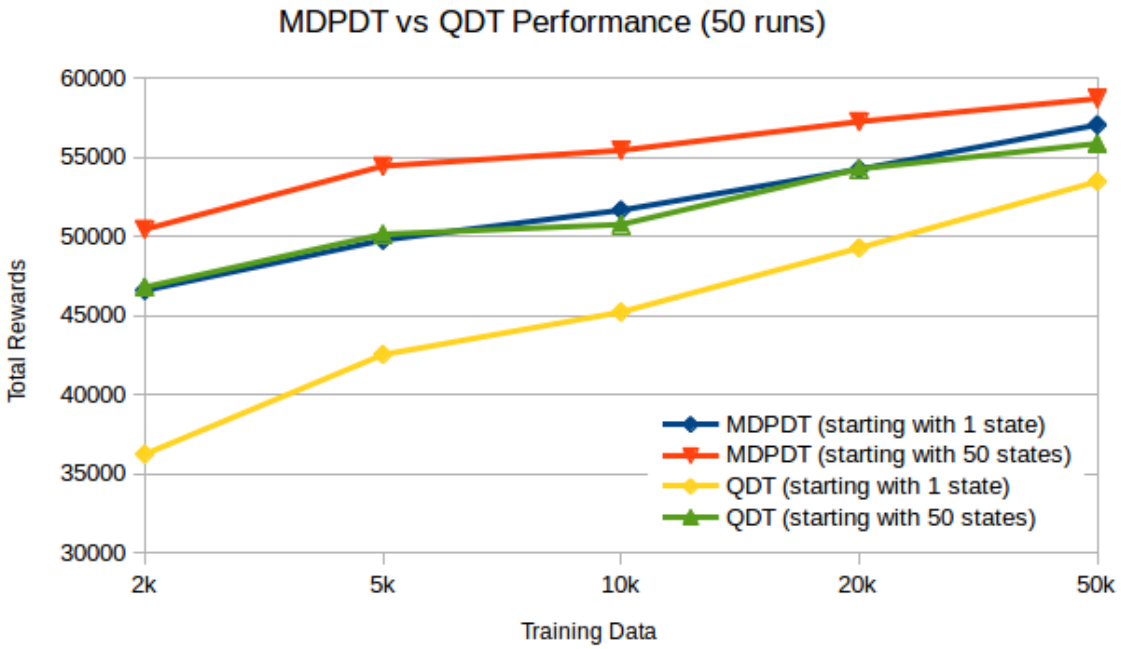


Figure 6.58: Performance comparison of the decision tree based models in the complex cluster scenario

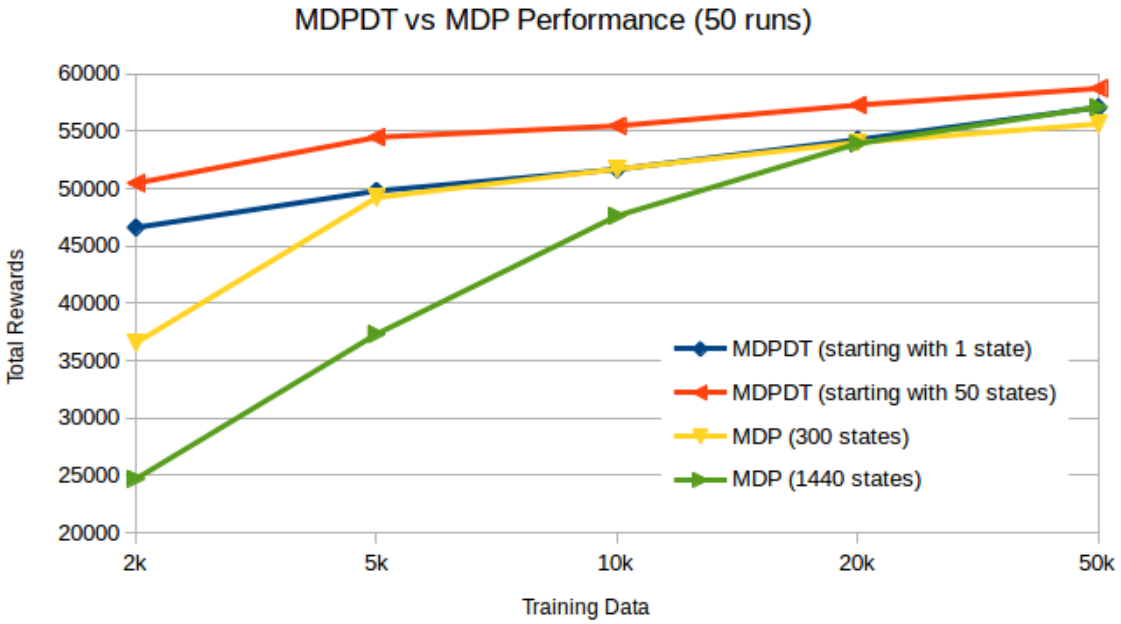


Figure 6.59: Performance comparison of the full-model decision tree based model with its fixed size counterpart in the complex cluster scenario

formance of each VM. The value of the penalty can be seen in figure 6.54. Similarly to the simple scenario in section 6.2.1, in addition to the four relevant parameters, the input vector included 6 additional random ones. Three of them followed a uniform distribution within $[0, 1]$, and another three took integer values within $[0, 9]$ with equal probability.

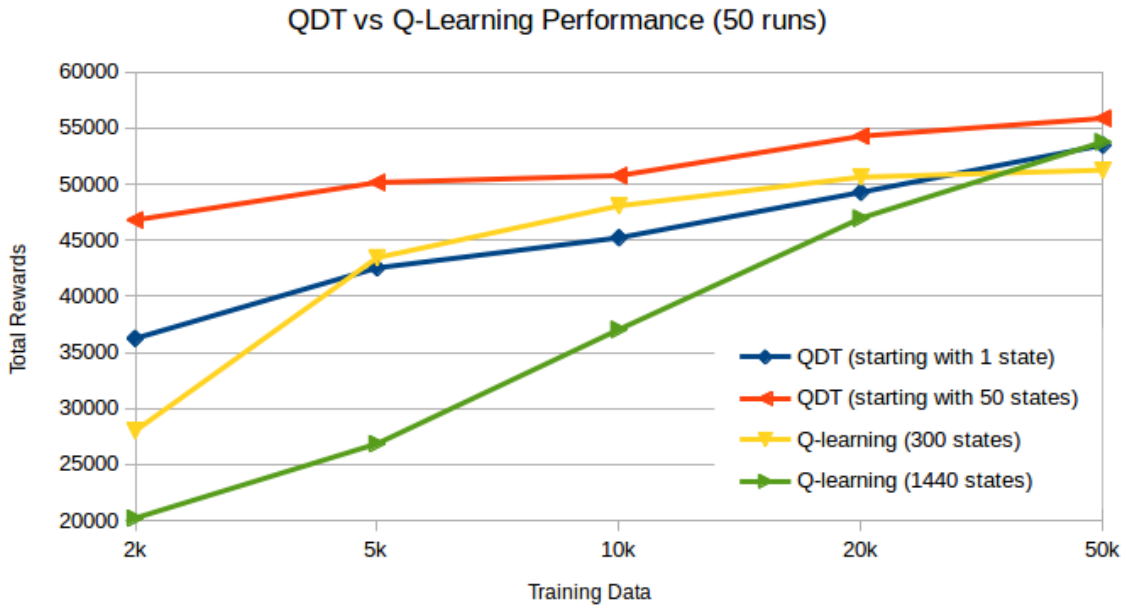


Figure 6.60: Performance comparison of the Q-Learning decision tree based model with its fixed size counterpart in the complex cluster scenario

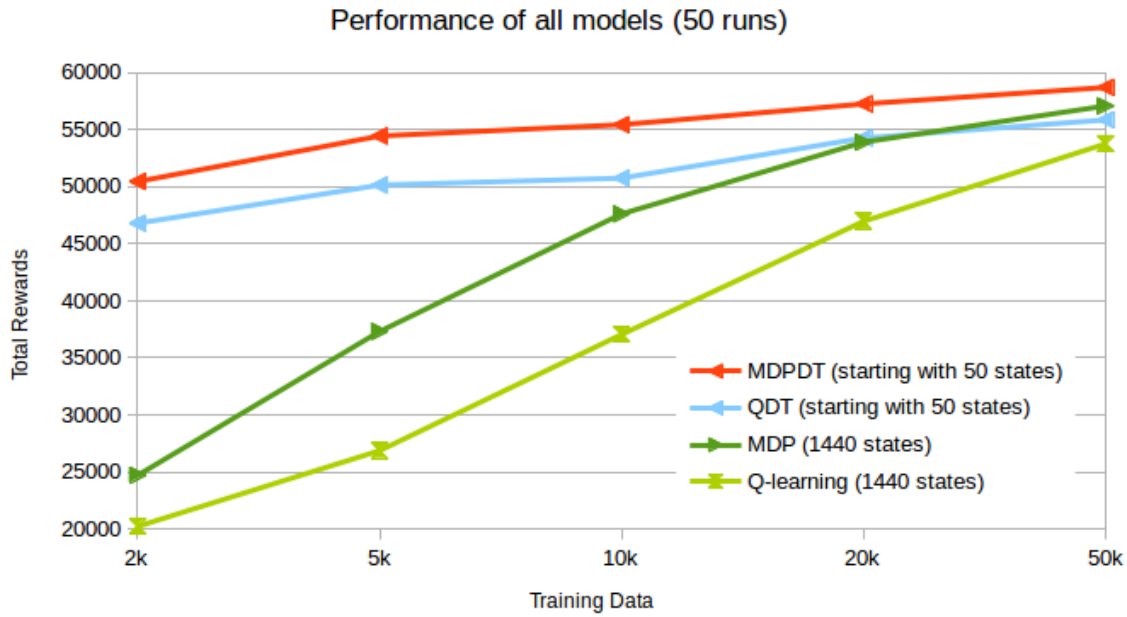


Figure 6.61: Performance for all models in the complex cluster scenario

Again, in the case of the fixed-size models (figures 6.55 and 6.56), smaller models achieved a better performance with a smaller training set, but got outperformed by the larger models when more data was available. Since the models were larger in this case (due to the larger number of relevant parameters in the scenario), in both the MDP and the Q-learning cases they required more data to catch up to the small model compared to the simple scenario. As expected, the full-model based MDP model outperformed Q-Learning (figure 6.57), but of

course at the cost of significantly more computation.

A similar performance difference is also apparent in the case of the decision tree based models (figure 6.58). In this case, starting the training with a small tree of 50 states offered a very significant boost in performance. This boost allowed both decision tree based models to clearly outperform their fixed size counterparts (figures 6.59 and 6.60).

Let us note here that the starting tree for the decision tree models implemented a small 10x5 grid on the number of VMs and the incoming load, and it was the same in both the simple and complex scenarios. In other words, the decision tree based models run with the exact same configuration on both scenarios and still managed to outperform the traditional models even though the latter required changing their configuration of states to adopt to the new scenario.

Chapter 7

Experimental Results

7.1 Experimental Setup

In this section we will briefly describe the way the different components used to perform our experiments were coordinated. We used *HBase* as our distributed database, which runs on top of the *HDFS*. The role of the client was played by *YCSB*, while *Ganglia* was used for the collection of cluster metrics. The cluster was running on top of an *OpenStack* IaaS provider.

7.1.1 Cloud Management

The management of the IaaS provider was performed by TIRAMOLA's Cloud Management module. This module offered a number of useful services that enabled TIRAMOLA to dynamically control the VMs that were active in the cluster. The communication was done using Python's *novaclient* module, which offers an API towards OpenStack's services. The services offered by TIRAMOLA's Cloud Management module are:

- Querying OpenStack for the details of all current active instances. Information about all VMs in the account are acquired using *novaclient*, and then filtered to isolate the ones belonging to the cluster.
- Querying OpenStack for the list of all current *images* or *flavors*. The image used for the creation of a VM determines the contents of its disk, while the flavor determines the virtual hardware it runs on (number of virtual CPU's, amount of physical memory etc).
- Creating new instances. Since the VMs created were required to immediately join a running HBase cluster, the image used was a snapshot of a VM already having Hadoop 2.5.2, HBase 1.1.2 and Ganglia already installed.
- Destroying instances. When a VM was decommissioned from the cluster, it was also removed from OpenStack.
- Resizing instances. Resizing an instance in OpenStack includes two steps. First, a resizing command has to be issued. This can be done by directly calling the *resize* method on a *server* item returned by *novaclient*, passing it the required new flavor. A new instance is then created that replaces the previous one. However, the previous instance is not deleted, but instead kept alive until the resizing is confirmed through the *confirm_resize*

method. Alternatively, if something goes wrong, the resizing can be canceled, restoring the old instance.

- Waiting until a number of instances were running. When a new instance is created, TIRAMOLA had to wait until it had properly booted before attempting to copy files or issue commands to it. This was performed by testing that the instance responds to *ping* and *nc* commands.

7.1.2 Cluster Management

In order for an HBase cluster to function properly, a number of configuration files need to be properly distributed and updated in all the nodes of the cluster, so that the nodes are aware of the correct configuration of the cluster at any point in time. Since the characteristics of the cluster were dynamic, these configuration files needed to also be dynamically updated. For that purpose, the up-to-date configuration files were locally created by TIRAMOLA's *Cluster Coordinator* module by using stored template files and replacing place-holder keywords. When a new node was introduced to the cluster, new configuration files were created and distributed to the cluster to inform the rest of the nodes of its presence.

Additionally, the Cluster Coordinator module offered additional functionality that allowed controlling the NoSQL database running within in the cluster. This included:

- Formatting HDFS's Namenode. When a new cluster was created, Hadoop's distributed file system needed to be formatted before HBase could use it to store its regions within it.

- Starting or stopping the cluster. This includes starting or stopping Hadoop's Namenode on the master and Datanodes on the slaves, as well as HBase's Master and Region servers.

- Creating tables within HBase. In order to apply load to HBase, a table needed to be created upon which to perform the queries. This was done automatically upon a new cluster's creation.

- Adding and removing nodes. When a new node was added or removed from the cluster, the configuration files in all the nodes were updated to reflect that change.

  In the case of node removal, the regions stored in that node were transfered away from it and the Region Server was shut down with the use of HBase's *graceful_stop.sh* script. Hadoop's Datanode was then shut down by adding the IP of the node to the master's *datanode-excludes* configuration file, and refreshing HDFS's nodes. When the process was complete, the node was registered as *decommissioned* in HDFS's *dfsadmin*, at which point the node was free to be physically removed from OpenStack.

  In the case of node addition, the Datanode and Region Server were started on that node, and the HBase balancer was triggered in order to transfer regions to the new Region Server. Since it was often the case that OpenStack gave new instances the IP's of old instances that had been removed from the cluster, when a new instance was created its

IP had to be preemptively cleared from the *datanode-excludes* configuration file on the Namenode.

7.1.3 Generating the Workload

The role of the client performing the queries in our experiments was carried out by a framework called *YCSB*. YCSB is a benchmarking tool written in Java that can generate traffic for a number of database systems such as HBase, Cassandra and MongoDB. It offers many configuration options, such as allowing the specification of certain target loads (in requests per second), the range of the queries, the percentage of reads, writes and updates, the distribution of the queries within the range and many more.

On top of the nodes making up our HBase cluster, an additional 15 VMs were created in order to generate the incoming load towards the database, each with a YCSB client installed. When the cluster was initialized, the *hosts* file containing the IP of the Master Server was transfered to these machines, along with the configuration file containing the characteristics of the workload. When a load needed to be executed, the appropriate command was sent by TIRAMOLA to all 15 YCSB clients, in order to generate traffic towards the database. Each of the clients executed its part of the workload, and stored the output containing the results in a file that was later parsed by TIRAMOLA. This way, information about the throughput and the latency of the queries was acquired and included to the rest of the metrics used for decision making.

7.1.4 Collecting Metrics

The majority of the metrics needed by the algorithm to determine the current state were collected using the monitoring system called Ganglia. For that purpose, two different Ganglia networks were used, one running on the host machines along with OpenStack, and another within the VMs of the cluster. In both cases, the data were acquired from the machine running the *gmetad* daemon which in the case of the cluster was the Master node, while a *gmond* daemon was running on each of the other machines.

The metrics were collected in XML format and parsed using Python's *xmltodict* library. After being parsed, the metrics were grouped based on the host they corresponded to. If data from any of the VMs in the cluster was missing, the process was repeated. The duration over which we collected measurements in our experiments lasted 3 minutes, during which data were collected every 10 seconds. After that duration ended, the metrics were averaged over all nodes of the cluster and over all measurements over time, and returned to TIRAMOLA's Coordinator module to be fed along with the rest of the metrics collected from YCSB to the Decision Making module.

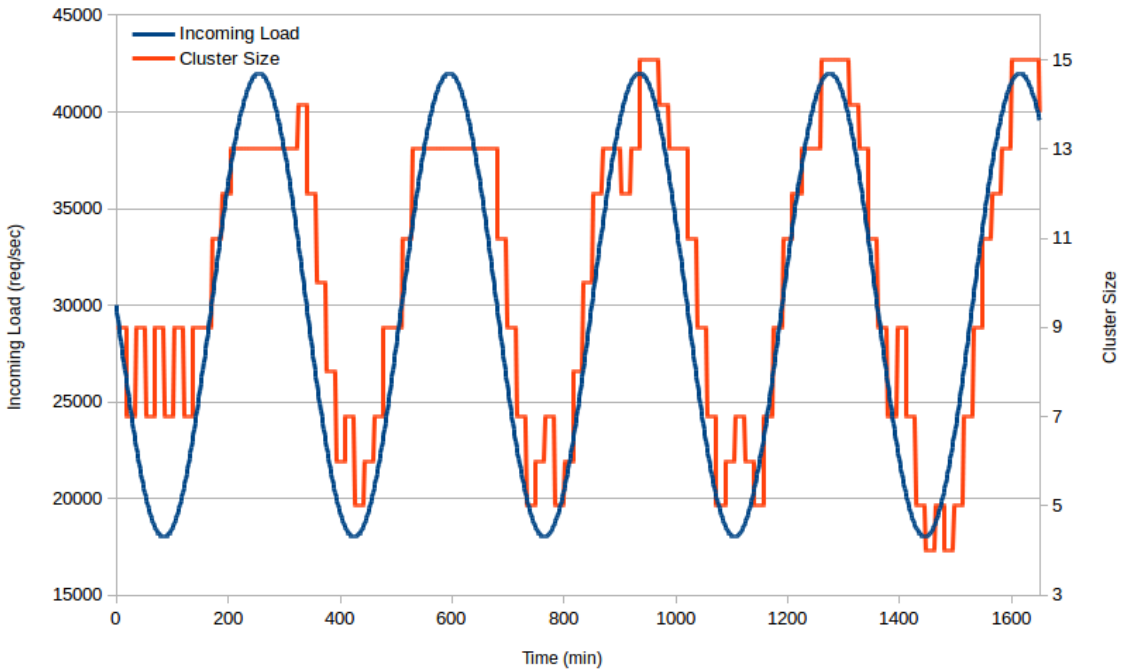


Figure 7.1: System behavior under a sinusoidal load (minimal dataset)

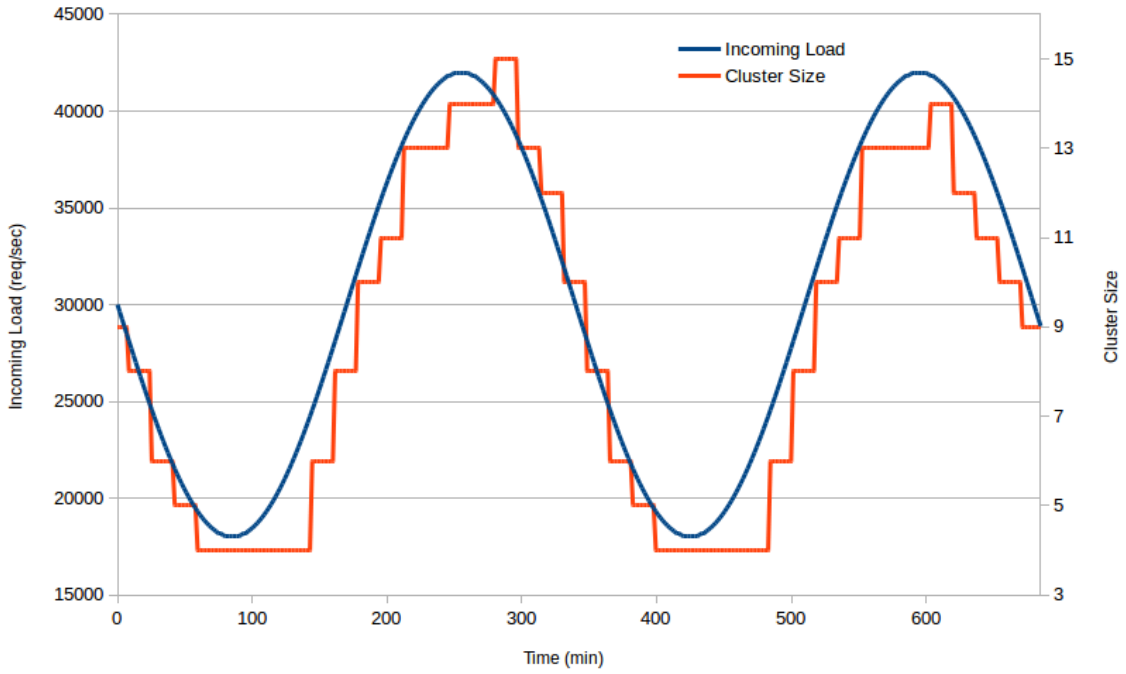


Figure 7.2: System behavior under a sinusoidal load (small dataset)

7.2 Results

7.2.1 System Behavior

The size of the cluster used in our experiments ranged between 4 and 15 VMs. Each VM in the HBase cluster had 1GB of RAM, 10GB of storage space and 1 virtual CPU, while the

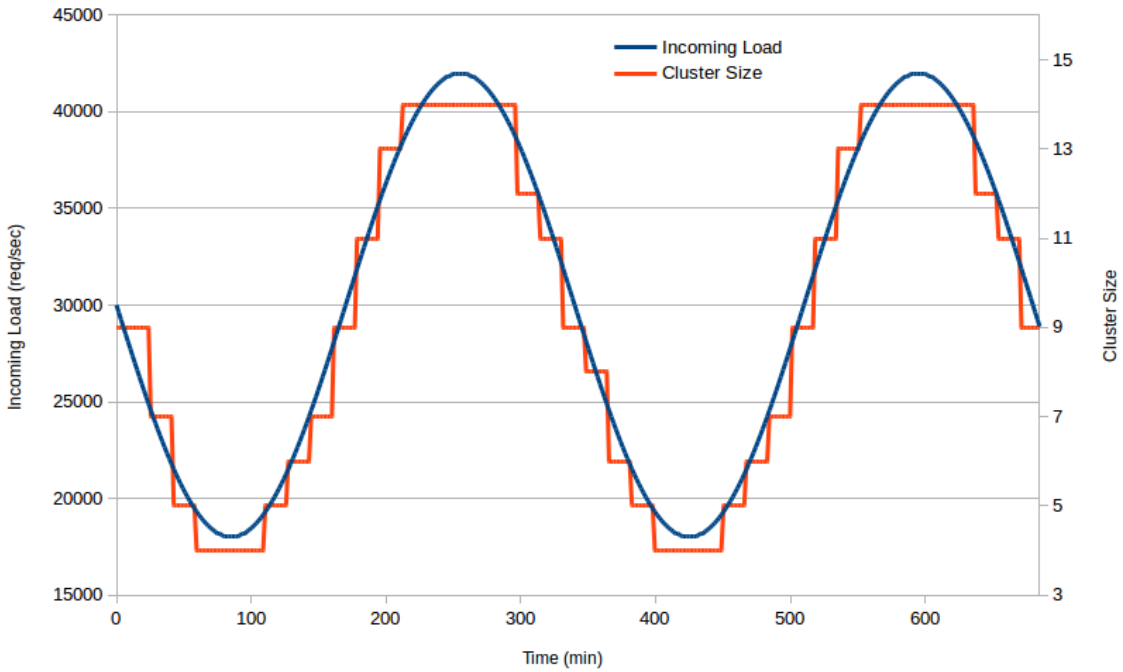


Figure 7.3: System behavior under a sinusoidal load (large dataset)

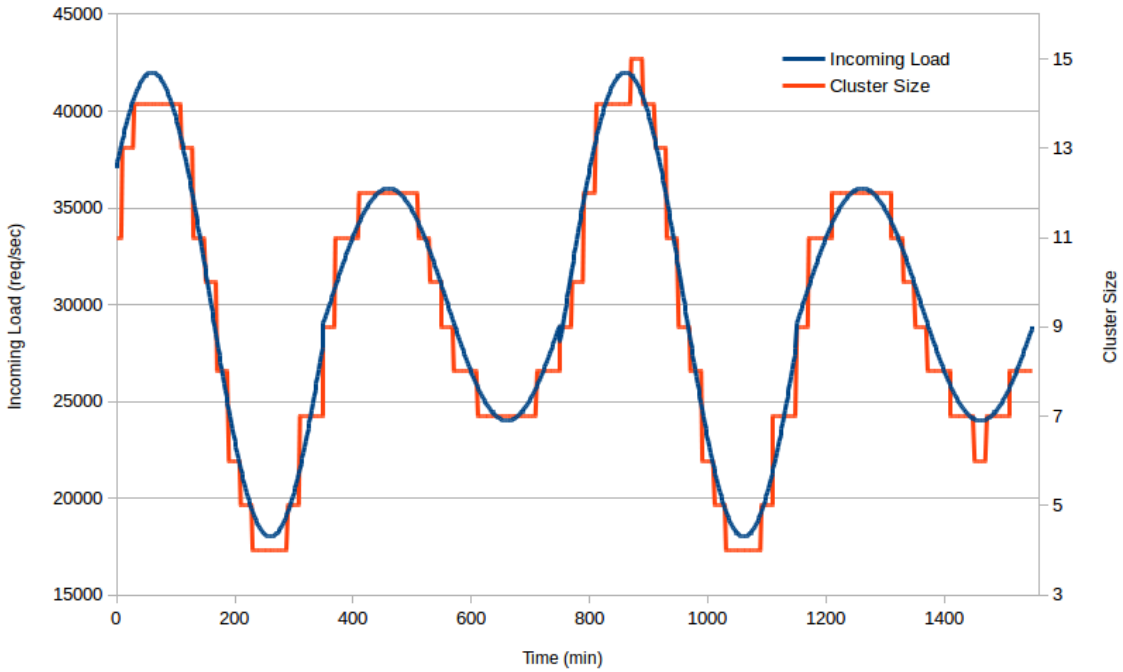


Figure 7.4: System behavior under a sinusoidal load with alternating amplitude

master node had 4GB of RAM, 10GB of storage and 4 virtual CPU's.

For the training of the decision tree based models we used a set of 12 parameters including:

- The size of the cluster
- The amount of RAM per VM

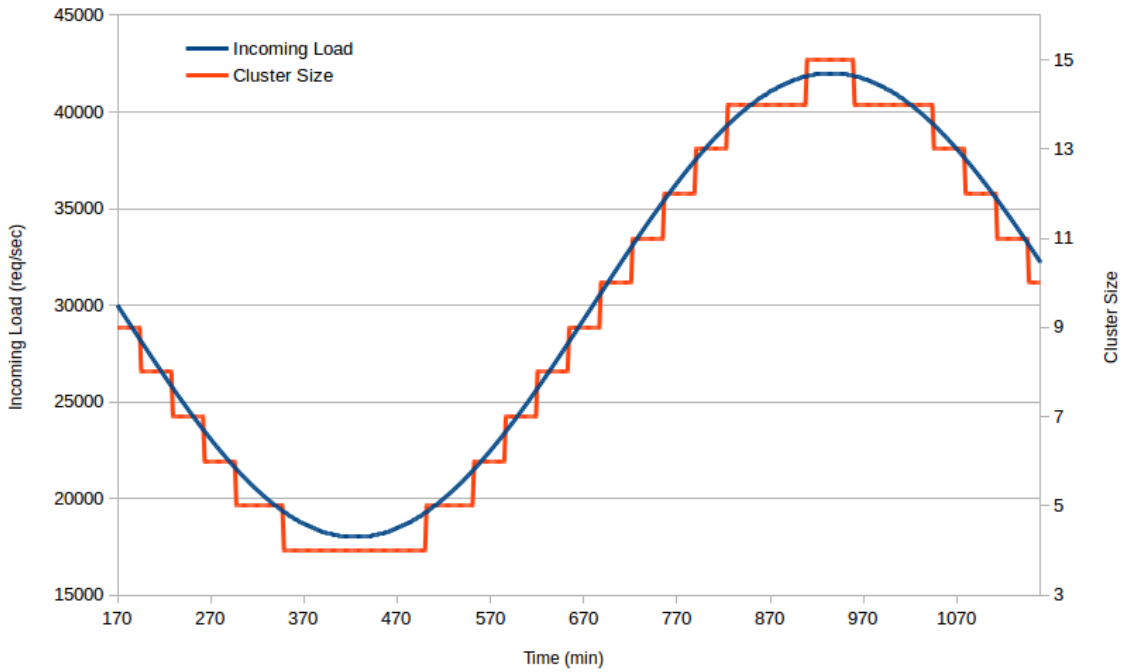


Figure 7.5: System behavior under a slow sinusoidal load

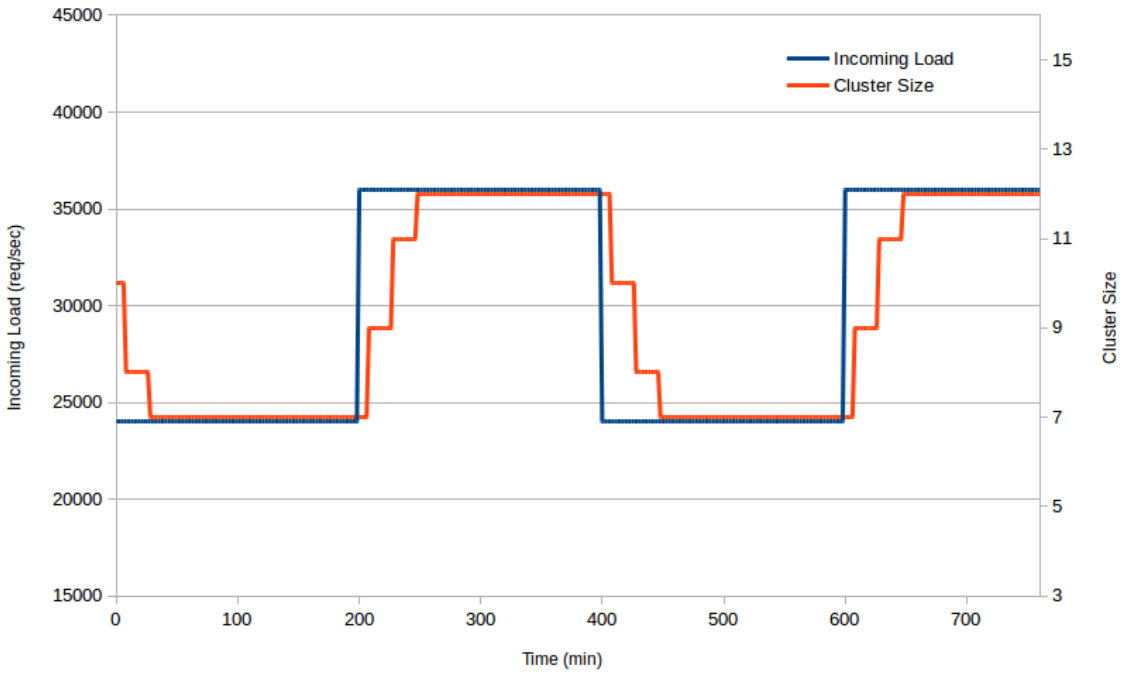


Figure 7.6: System behavior under a square pulse load

- The percentage of free RAM
- The number of virtual CPU's per VM
- The CPU utilization
- The storage capacity per VM

- The number of I/O requests per second
- The CPU time spent waiting for I/O operations
- A linear prediction of the next incoming load (equal to two times the current load minus the last recorded load)
- The percentage of read requests in the queries
- The average latency of the queries
- The network utilization

We initialized the MDPDT decision tree with 6 states and let it partition the state space on its own from that point on. We tested the behavior of TIRAMOLA after training it with datasets of different sizes. The training load was a sinusoidal load of varying amplitude. We allowed for 5 different actions, which included adding or removing 1 or 2 VMs from the cluster, or doing nothing. First, we run TIRAMOLA with a minimal dataset of 500 experiences. When trained with this dataset, only 17 splits were performed during the training (4 using the size of the cluster and 13 using the incoming load), increasing the total number of states to 22. During this run 12 additional splits were performed (4 using the size of the cluster and 7 using the incoming load and 1 using the latency), allowing TIRAMOLA to continuously adapt and follow the incoming load (figure 7.1). When provided with bigger datasets of 1500 and 20000 experiences, the performance improved and very closely converged to the incoming load, ending up with 66 and 576 states respectively (figures 7.2 and 7.3).

7.2.2 Effect of the initial number of states

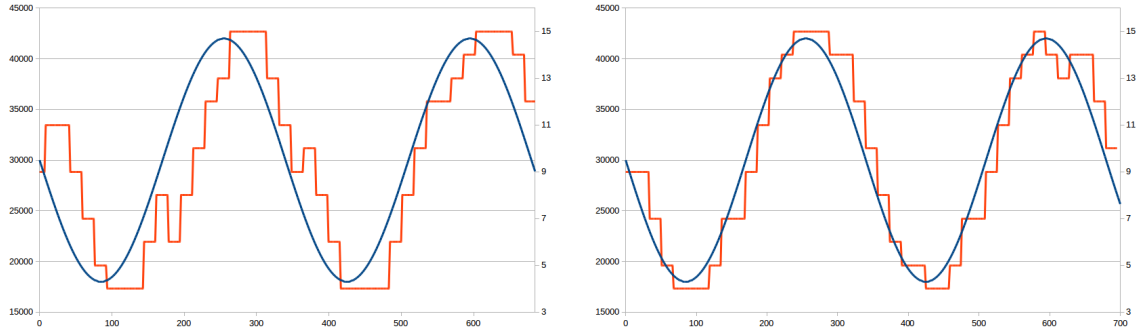
As the simulations presented in chapter 6 suggested, MDPDT's performance improves when instead of starting from a single state, the model starts from a small number of states, thus having to solve a small number of easier problems instead of a harder one. This was also the case when controlling a real cluster, as seen in figure 7.7. Starting the model with 6 states instead of 1 made a very noticable difference in the performance of the algorithm when trained with a small dataset, allowing TIRAMOLA to very closely follow the incoming load with little training.

Further increasing the number of initial states to 50 though had a negative effect on the algorithm's performance. This is an indication that even this relatively simple state space can be partitioned by a decision tree in a more efficient manner than the orthogonal grid partitioning that was provided as the initial state configuration.

Of course, when a larger dataset was provided, MDPDT significantly improved its performance in all three cases, even though the behavior when starting with 6 initial states still seemed to be the most stable of the three.

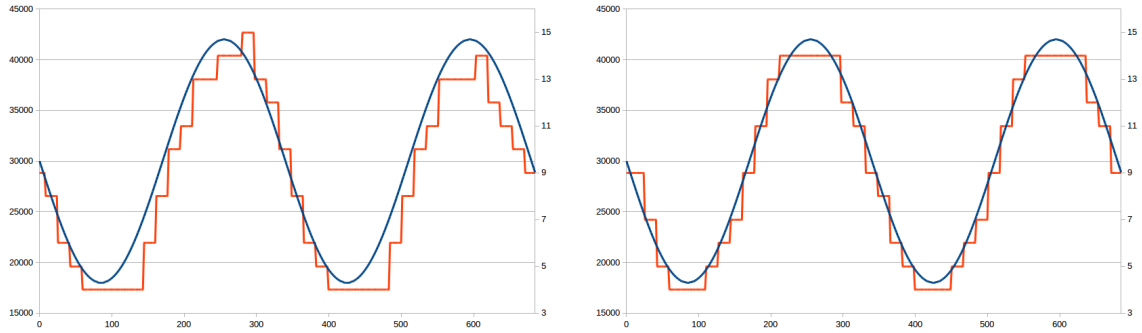
7.2.3 Using Different Models

In this experiment, we had the opportunity to test the behavior of TIRAMOLA when using the other three models participating in the simulation experiments in chapter 6. The



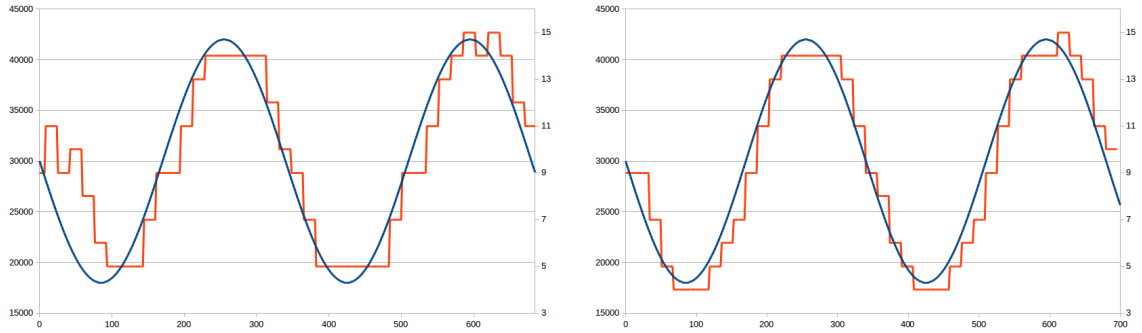
(a) 1 state, small dataset

(b) 1 state, large dataset



(c) 6 states, small dataset

(d) 6 states, large dataset



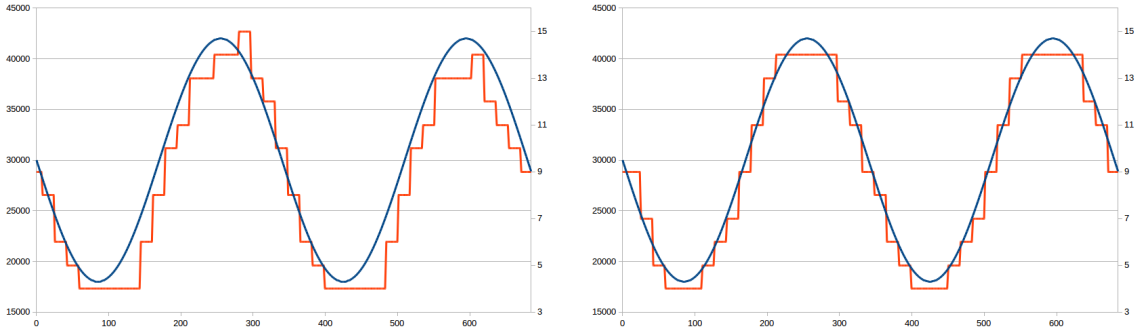
(e) 50 states, small dataset

(f) 50 states, large dataset

Figure 7.7: The effect of the initial number of states in the behavior of MDPDT

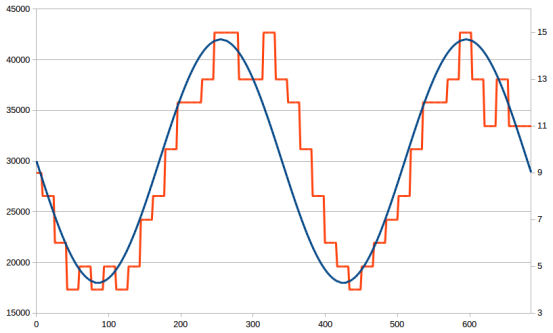
full-model based MDP model also performed reasonably well in this setting. In the case of the small dataset it did seem to still require more training, but when trained with a larger dataset it followed the incoming load very cleanly. Let us note that this is a problem where this approach was expected to do very well, since the state space is relatively simple, and a partitioning using only the size of the cluster and the incoming load was sufficient to capture the behavior in this experiment. At the same time, since it maintains a full MDP model of the system, it was able to very accurately make use of the collected information.

The Q-learning based models though both required a large amount of data to follow the incoming load effectively. In this experiment the decision tree based Q-learning model (QDT) achieved the weakest performance with the small dataset. With this few data this is not totally unexpected, since at the start of the training that model uses the first data it acquires to perform splits, but then discards it after the splits have been performed, leaving it with very little available information to make decisions. When more training data was provided though, it did

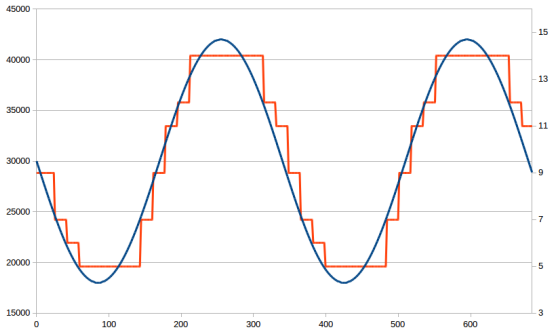


(a) MDPDT, small dataset

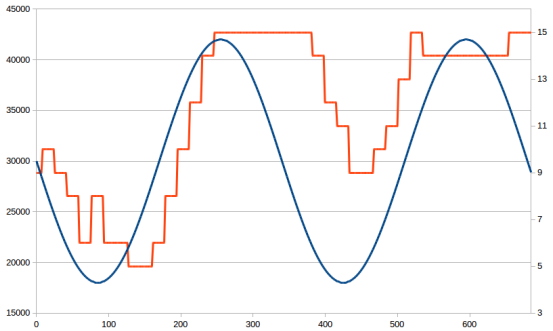
(b) MDPDT, large dataset



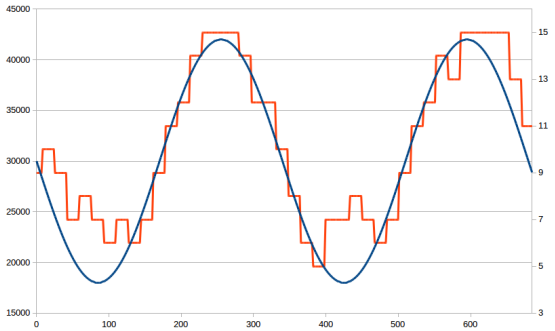
(c) MDP, small dataset



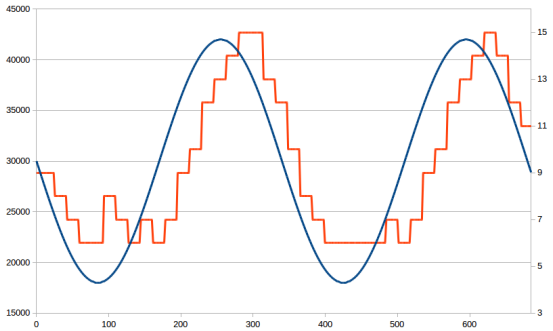
(d) MDP, large dataset



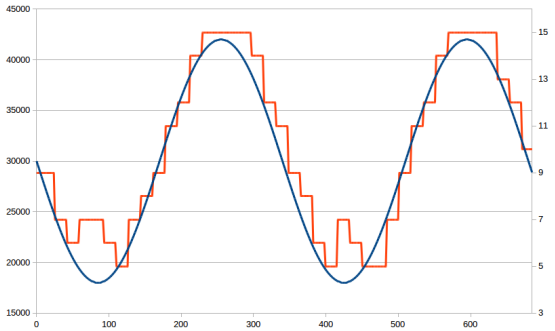
(e) QDT, small dataset



(f) QDT, large dataset



(g) Q-learning, small dataset

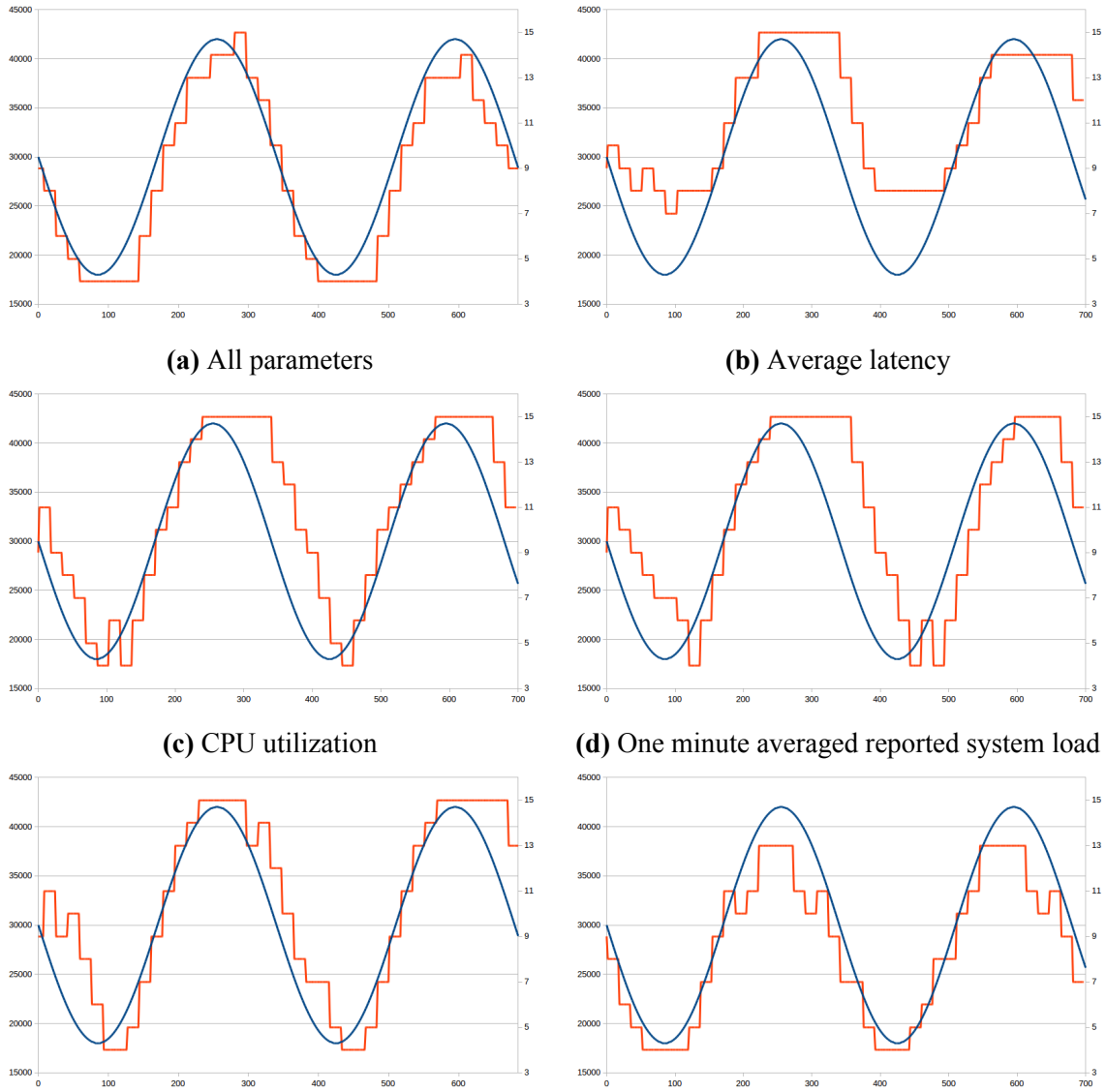


(h) Q-learning, large dataset

Figure 7.8: Comparison of the behavior of all four models

manage to catch up to the traditional Q-learning model. However, they both were noticably less stable compared to the full model approaches, to a large extent verifying the results deriving from the simulation experiments.

7.2.4 Restricting the Splitting Parameters



(a) All parameters

(b) Average latency

(c) CPU utilization

(d) One minute averaged reported system load

(e) Network Usage

(f) Incoming load prediction

Figure 7.9: System behavior when allowing splits with only the cluster size plus one additional parameter

In order to test the algorithm's ability to partition the state space using different parameters, as well as to test the reliability of some of the parameters in predicting the incoming load, we experimented with restricting the parameters with which the algorithm is allowed to partition the state space. For that purpose, we experimented with training the algorithm from a small dataset of 1500 experiences, but restricting the parameters with which the algorithm is allowed to partition the state space to only the size of the cluster plus one additional parameter each time. The parameters used were the CPU utilization, the one minute averaged reported system load, the prediction of the incoming load, the network usage and the average latency.

For all the parameters, the system seems to be able to find a correlation between the

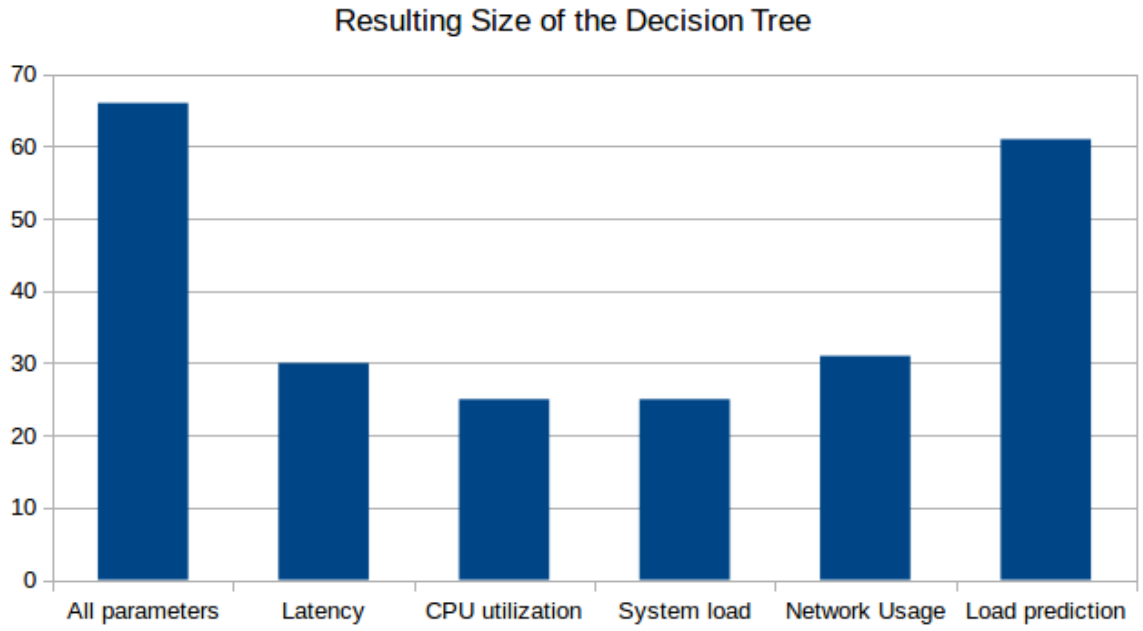


Figure 7.10: Resulting size of the decision tree when allowing splits with only the cluster size plus one additional parameter

given parameter and the rewards obtained, and starts following the incoming load. Of course, the performance is significantly worse compared to the default case where all the available information is provided, and thus the training of the model is noticeably slower. The resulting size of the decision tree for each individual case reflects this fact, and in most cases the model ended up having only 20 to 30 states compared to the 66 of the default case. However, the fact that these correlations exist and can be detected even from a small dataset of only 1500 points, reveals the fact that it is possible, using techniques like the ones described in this work, to exploit these correlations in order to implement policies in systems with complicated and not very well understood behavior.

Chapter 8

Epilogue

8.1 Conclusions

During this work, we had the opportunity to experiment with decision tree based reinforcement learning algorithms, and test their performance on the challenging problem of performing dynamic resource allocation for non-relational databases. Here, we will summarize the conclusions derived from this work.

- The performance of the decision tree based models was surprisingly good compared to their traditional reinforcement learning counterparts. This fact is not only due to the decision tree's ability to create an efficient partitioning of the state space, but also because of the fact that these models adaptively increase their model size as more data become available. This allowed them to train quickly at the start of the process, but still gradually increase their size to keep up with larger models as more data were gathered. Additionally, since they do not require a predefined state space configuration, they can be used in different types of scenarios with the same settings.

- The splitting criteria that were based on statistical tests were very efficient in distinguishing real correlations from random noise. However, in order to achieve this, the error margin needs to be set much lower than the typical value of 0.05, depending on the statistical test and splitting criterion used. The information-based criterion, that is common in traditional decision tree algorithms, also performed reasonably well, when restricted appropriately to minimize mistakes. However, even in that case, it did not manage to reach the effectiveness of the statistical criteria. Of course, since all the criteria detect correlation between the efficiency of actions and the values of certain parameters, one still needs to be aware of situations where there are temporary correlations between certain parameters of the system and its performance. As long as these correlations hold, partitioning the state space based on those parameters may not cause a problem, but if these correlations suddenly break, the model may stop behaving optimally. For this reason, the parameters with which the model is allowed to partition the state space is an important decision that needs to be made very carefully.

- The combination of the Mann Whitney U test with the Parameter test splitting criterion achieved the best performance among all the tested statistical criteria. For the Q-value test in particular, where it is possible to consider multiple splitting points per parameter, attempting to split on only the median achieved better results than allowing multiple options, and at the same time produced smaller decision trees.

- The fine grained splitting and retraining mechanism we implemented during this work performed better than the one used in [Uthe98], while at the same time being more computationally efficient. More complicated splitting strategies, like delaying the beginning of the splits or periodically resetting the decision tree did not manage to improve performance. However, when allowing multiple splitting points per parameter, the latter did not fall too far behind the default strategy, and thus could potentially be used to correct mistakes caused by misleading data at the start of the training.

- In terms of computational efficiency, maintaining the training data to retrain the new states and performing tests on them to decide splits does have a considerable effect on the running time, when comparing models with approximately equal numbers of states. However, often the decision tree based models managed to achieve better performance using a significantly smaller number of states. Moreover, in the case of the full-model based approaches, if an update algorithm such as prioritized sweeping or value iteration is used to update the values of the states and Q-states, the running time is dominated by the performance of the update algorithm. As a result, since the running time of these algorithms depends on the number of states of the model, decision tree models ended up running faster.

  In the context of cloud computing, where there is generally a lot of computational power available and a lot of time between decisions to perform calculations, the running time of the algorithms was completely trivial, and the only concern was the time needed to perform the initial training from a very large dataset. However, if needed, implementing the algorithms in a statically typed, compiled programming language (like C) and using prioritized sweeping as the update algorithm would make the training time trivial even in those cases. Of course, in scenarios where the computational and energy efficiency is critical (for example when controlling mobile robots), Q-Learning provides by far the fastest running time and lowest memory requirement, at the cost of being the least accurate of the approaches.

- Even though decision tree based algorithms have the potential to model the state space with zero knowledge of its topology by starting from a single state, providing a little information in the form of a small number of initial states significantly improved performance. This is not surprising, since mistakes in the structure of the decision tree are much more expensive the closer they are to the root. Despite the fact that in most experiments the decision trees used begun as a single state, we believe that if applied in practical problems a small configuration of starting states should always be used, whenever that kind of information is available. In the case of performing elasticity decisions for distributed databases, this kind of information is almost always available, since the current size of the cluster and the incoming load are always expected to be a deciding factor in the decisions. Therefore, these two parameters should be used to create a small initial partitioning of the state space. From that point on, the decision tree algorithm can be used to further partition the state space and capture more complicated behaviors of the system that are not obvious beforehand.

- The fact that the decision tree algorithms expect as a state-input a vector of continu-

ous values of parameters suited the way metrics were collected in practice, since they could be fed directly into the algorithm. Additionally, the fact that all the reinforcement learning algorithms can learn from past experiences without having to have selected the actions themselves, allows for offline training using existing data, or even sharing training data among different databases running on a common infrastructure provider.

8.2 Future Work

In this section, we will discuss briefly a number of topics that we did not have the opportunity to tackle during this work, but we believe are worth investigating in the future.

Splitting criteria using multiple parameters

The decision tree algorithms discussed in this work attempt to partition the state space by examining the correlation between the values of the provided parameters and the effectiveness of the available actions. However, all the criteria we examined attempt to do that by only considering the values of a single parameter. In practice, it is possible to imagine situations where the behavior of a system depends on values of parameters in such a complicated manner that figuring out the correlation by examining a single parameter can be very challenging. For example, if a certain behavior is exhibited under a combination of two specific values of two parameters, that behavior will not be easily detected by looking at each one of them separately. To overcome this problem, splitting criteria could be developed that take into account values of multiple parameters instead of a single one.

Determining which parameters are reliable for decision making

In resource allocation problems in particular, the most common metric that is used today in practice to perform elasticity decisions by threshold-based decision makers is the CPU utilization. However, a large number of other parameters seem to be relevant to the behavior of the system (load characteristics, I/O operations per second, network utilization etc). Even though the decision-tree based algorithms discussed in this work have the ability to handle such a large number of parameters, studying which of these parameters are most reliable to be used in decision making and filtering out the less reliable ones could only benefit their performance, as well as the performance of more simplistic threshold based solutions like the ones used in practice.

Mechanism to prune the decision tree

In this work we have adopted a somewhat more aggressive approach in splitting states, opting for options that to some extent sacrifice accuracy to grow the decision tree faster. The reason behind this is the fact that under limiting training data this approach simply provides better results. However, when more data are available, more strict splitting strategies could become more beneficial and efficient. Additionally, in order to correct possible poor decisions resulting from such aggressive splitting strategies, it might be worth investigating

mechanisms that can cancel already existing splits. Such mechanisms that can *prune* an existing decision tree are being used in classification algorithms such as C4.5, and could be used to improve the long-term efficiency of decision tree based reinforcement learning algorithms as well.

Context detection as an alternative to dynamic state space partitioning

An alternative approach to dynamically creating new states in order to capture the behavior of a complex system such as a NoSQL cluster, could be to instead maintain a large number of traditional reinforcement learning models, each with a different state configuration, and developing a mechanism to evaluate their accuracy as the system runs. Having an estimate of how accurately each of the different models behaves at any point in time, it could then be possible to choose actions that take into account that fact. This could either be done through a polling mechanism, where each model's opinion about the optimal action is weighted by its accuracy, or by simply executing the action of the most accurate model. Such solutions have been proposed in the past [Doya02] [DaSi06a], and use each model's ability to predict the rewards and transitions of the system as an indication of its accuracy.

Bibliography

[Ange12] Evangelos Angelou, Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris, "Automatic scaling of selective SPARQL joins using the TIRAMOLA system", in *Proceedings of the 4th International Workshop on Semantic Web Information Management*, p. 1, ACM, 2012.

[Bell57] R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ., 1957.

[Bort08] Dhruba Borthakur, "HDFS architecture guide", *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[Chan08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E Gruber, "Bigtable: A distributed storage system for structured data", *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[Chap91] David Chapman and Leslie Pack Kaelbling, "Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons.", in *IJCAI*, vol. 91, pp. 726–731, 1991.

[Choi01] Samuel PM Choi, Dit-Yan Yeung and Nevin L Zhang, "Hidden-mode markov decision processes for nonstationary sequential decision making", in *Sequence Learning*, pp. 264–287, Springer, 2001.

[Coom96] William T Coombs, James Algina and Debra Olson Oltman, "Univariate and multivariate omnibus hypothesis tests selected to control Type I error rates when population variances are not necessarily equal", *Review of Educational Research*, vol. 66, no. 2, pp. 137–179, 1996.

[DaSi06a] Bruno C Da Silva, Eduardo W Basso, Ana LC Bazzan and Paulo M Engel, "Dealing with non-stationary environments using context detection", in *Proceedings of the 23rd international conference on Machine learning*, pp. 217–224, ACM, 2006.

[DaSi06b] Bruno C Da Silva, Eduardo W Basso, Filipo S Perotto, Ana L C Bazzan and Paulo M Engel, "Improving reinforcement learning with context detection", in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 810–812, ACM, 2006.

[Doya02] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri and Mitsuo Kawato, "Multiple model-based reinforcement learning", *Neural computation*, vol. 14, no. 6, pp. 1347–1369, 2002.

[dWin13] J.C.F. de Winter, "Using the Student's t-test with extremely small sample sizes. Practical Assessment, Research and Evaluation", vol. 18, no. 10, 2013.

[Even04] Eyal Even-Dar and Yishay Mansour, "Learning rates for Q-learning", *The Journal of Machine Learning Research*, vol. 5, pp. 1–25, 2004.

[Gask99] Chris Gaskett, David Wettergreen and Alexander Zelinsky, "Q-learning in continuous state and action spaces", in *Australian Joint Conference on Artificial Intelligence*, pp. 417–428, Springer, 1999.

[Ghem03] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, "The Google file system", in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[Huan15] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda and Frederick R Reiss, "Resource Elasticity for Large-Scale Machine Learning", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 137–152, ACM, 2015.

[Jaco91] Robert A Jacobs, Michael I Jordan, Steven J Nowlan and Geoffrey E Hinton, "Adaptive mixtures of local experts", *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[Jefr08] D Jefrey and S Ghemawat, "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, 2008.

[Kael96] Leslie Pack Kaelbling, Michael L Littman and Andrew W Moore, "Reinforcement learning: A survey", *Journal of artificial intelligence research*, pp. 237–285, 1996.

[Kass14] Evie Kassela, Christina Boumpouka, Ioannis Konstantinou and Nectarios Koziris, "Automated workload-aware elasticity of NoSQL clusters in the cloud", in *Big Data (Big Data), 2014 IEEE International Conference on*, pp. 195–200, IEEE, 2014.

[Kobe13] Jens Kober, J Andrew Bagnell and Jan Peters, "Reinforcement learning in robotics: A survey", *The International Journal of Robotics Research*, p. 0278364913495721, 2013.

[Kons11] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos and Nectarios Koziris, "On the elasticity of nosql databases over cloud management platforms", in *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 2385–2388, ACM, 2011.

[Kons12] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, Christina Boumpouka, Nectarios Koziris and Spyros Sioutas, "Tiramola: elastic nosql provisioning through a cloud management platform", in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 725–728, ACM, 2012.

[Kots07] Sotiris B Kotsiantis, I Zaharakis and P Pintelas, "Supervised machine learning: A review of classification techniques", 2007.

[Lagu13] Ignacio Laguna, Subhasish Mitra, Fahad Arshad, Nawanol Theera-Ampornpunt, Zongyang Zhu, Saurabh Bagchi, Samuel P Midkiff, Mike Kistler, Ahmed Gheith et al., "Automatic Problem Localization via Multi-dimensional Metric Profiling", in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pp. 121–132, IEEE, 2013.

[LBre84] R. A. Olshen L. Breiman, J. H. Friedman and C. J. Stone, *Classification and regression trees*, Wadsworth International, Monterey, CA, 1984.

[Litt95] Michael L Littman, Thomas L Dean and Leslie Pack Kaelbling, "On the complexity of solving Markov decision problems", in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pp. 394–402, Morgan Kaufmann Publishers Inc., 1995.

[Maso15] Seyed Saeid Masoumzadeh and Helmut Hlavacs, "Dynamic Virtual Machine Consolidation: A Multi Agent Learning Approach", in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pp. 161–162, IEEE, 2015.

[Mass04] Matthew L Massie, Brent N Chun and David E Culler, "The ganglia distributed monitoring system: design, implementation, and experience", *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[McCa96] Andrew Kachites McCallum, *Reinforcement learning with selective perception and hidden state*, Ph.D. thesis, University of Rochester, 1996.

[Mitc97] Tom M. Mitchell, *Machine Learning p2*, McGraw-Hill, 1997.

[Moor93] Andrew W Moore and Christopher G Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time", *Machine Learning*, vol. 13, no. 1, pp. 103–130, 1993.

[Murt94] Sreerama K. Murthy, Simon Kasif and Steven Salzberg, "A system for induction of oblique decision trees", *Journal of artificial intelligence research*, 1994.

[Murt95] Kolluru Venkata Sreerama Murthy and Steven L Salzberg, *On growing better decision trees from data*, Ph.D. thesis, Citeseer, 1995.

[Nask] Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou and Spyros Sioutas, "Dependable Horizontal Scaling Based On Probabilistic Model Checking".

[Peng93] Jing Peng and Ronald J Williams, "Efficient learning and planning within the Dyna framework", *Adaptive Behavior*, vol. 1, no. 4, pp. 437–454, 1993.

[Pute14] Martin L Puterman, *Markov decision processes: discrete stochastic dynamic programming*, John Wiley & Sons, 2014.

[Pyea01] Larry D Pyeatt, Adele E Howe et al., "Decision tree function approximation in reinforcement learning", in *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, vol. 1, p. 2, 2001.

[Pyea03] Larry D Pyeatt, "Reinforcement Learning with Decision Trees.", in *Applied Informatics*, pp. 26–31, 2003.

[Quin86] J. Ross Quinlan, "Induction of decision trees", *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[Ruxt06] Graeme D Ruxton, "The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test", *Behavioral Ecology*, vol. 17, no. 4, pp. 688–690, 2006.

[Sefr12] Omar Sefraoui, Mohammed Aissaoui and Mohsine Eleuldj, "OpenStack: toward an open-source solution for cloud computing", *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[Shan01] Claude Elwood Shannon, "A mathematical theory of communication", *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[Sing92] Satinder Pal Singh, "Transfer of learning by composing solutions of elemental sequential tasks", *Machine Learning*, vol. 8, no. 3-4, pp. 323–339, 1992.

[Stre06] Alexander L Strehl, Lihong Li, Eric Wiewiora, John Langford and Michael L Littman, "PAC model-free reinforcement learning", in *Proceedings of the 23rd international conference on Machine learning*, pp. 881–888, ACM, 2006.

[Sutt91] Richard S. Sutton, "Dyna, an Integrated Architecture for Learning, Planning, and Reacting", *Working Notes of the 1991 AAAI Spring Symposium*, pp. 151–155, 1991.

[Sutt98] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*, vol. 1, MIT press Cambridge, 1998.

[Tsou13] Dimitrios Tsoumakos, Ioannis Konstantinou, Christina Boumpouka, Spyros Sioutas and Nectarios Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola", in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 34–41, IEEE, 2013.

[Uthe98] William TB Uther and Manuela M Veloso, "Tree based discretization for continuous state space reinforcement learning", in *Aaai/iaai*, pp. 769–774, 1998.

[Watk89a] Christopher J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. thesis, King's College, Cambridge, UK, 1989.

[Watk89b] Christopher John Cornish Hellaby Watkins, *Learning from delayed rewards*, Ph.D. thesis, University of Cambridge England, 1989.

[Watk92] Christopher J. C. H. Watkins, "Q-Learning", *Machine Learning*, vol. 8, pp. 279–292, 1992.

[Whit10] Martha White and Adam White, "Interval estimation for reinforcement-learning algorithms in continuous-state domains", in *Advances in Neural Information Processing Systems*, pp. 2433–2441, 2010.

[Wier98] MA Wiering and Jürgen Schmidhuber, "Learning exploration policies with models", 1998.

[Wiki15a] Wikipedia, "Kolmogorov–Smirnov test", `https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test`, accessed 2-Dec-2015.

[Wiki15b] Wikipedia, "Mann–Whitney U test", `https://en.wikipedia.org/wiki/Mann-Whitney_U_test`, accessed 2-Dec-2015.

[Will93a] R. J. Williams and L. C. Baird, III, "Tight performance bounds on greedy policies based on imperfect value functions", *Tech. rep. NU-CCS-93-14*, 1993.

[Will93b] Ronald J Williams and Leemon C Baird, "Tight performance bounds on greedy policies based on imperfect value functions", Technical report, Citeseer, 1993.