ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη Αποδοτικών Αλγορίθμων για Markov Decision Processes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παπαγεωργίου Ιωάννης

Επιβλέπων: Βασιλική Καντερέ
Επίκουρη Καθηγήτρια Ε.Μ.Π.

Αθήνα, Μάρτιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη Αποδοτικών Αλγορίθμων για Markov Decision Processes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παπαγεωργίου Ιωάννης

Επιβλέπων: Βασιλική Καντερέ
Επίκουρη Καθηγήτρια Ε.Μ.Π.

.....	.....	.....
Βασιλική Καντερέ Επίκουρη Καθηγήτρια Ε.Μ.Π.	Παναγιώτης Καρράς Αναπληρωτής Καθηγητής Aarhus University	Συμεών Παπαβασιλείου Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2022

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Efficient Markov Decision Process Algorithms

DIPLOMA THESIS

Ioannis Papageorgiou

Supervisor: Vassiliki Kantere
Assistant Professor at the Electrical and Computer Engineering Department of NTUA

Athens, March 2022

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

Περίληψη

Οι Διαδικασίες Αποφάσεων Markov (Markov Decision Processes ή MDP για συντομία) αποτελούν ένα από τα σημαντικότερα εργαλεία επίλυσης προβλημάτων αποφάσεων υπό αβεβαιότητα την σήμερον ημέρα. Χρησιμοποιούνται κατά κόρον σε σύγχρονες εφαρμογές, ιδιαίτερα σε αυτές που αφορούν Ενισχυτική Μάθηση. Σε αυτά, ένας παίκτης καλείται να λάβει αποφάσεις οι οποίες προκαλούν μεταβολές στο περιβάλλον του ενώ επιπλέον του αποδίδουν μια ανταμοιβή-κίνητρο, προκειμένου να μεγιστοποιήσει αυτήν την ανταμοιβή. Αναλόγως με το αν ο αριθμός των αποφάσεων που καλείται να κάνει ο πράκτορας είναι πεπερασμένος ή άπειρος, το MDP χαρακτηρίζεται ως Πεπερασμένου ή Απείρου Ορίζοντα.

Τα MDP Πεπερασμένου Ορίζοντα επιλέγονται σε αρκετές εφαρμογές έναντι του Απείρου, εφόσον αντικατοπτρίζουν καλύτερα προβλήματα του πραγματικού κόσμου, τα οποία εξ ορισμού κάποτε θα ολοκληρωθούν, όπως σε προβλήματα διαχείρισης πόρων. Εντούτοις, ένα σημαντικό πρόβλημα που εμφανίζουν έγκειται στην μνήμη που καταλαμβάνει η λύση που υπολογίζεται, ιδιαίτερα σε περιπτώσεις όπου ο αλγόριθμος εκτελείται σε συσκευές περιορισμένων δυνατοτήτων υλικού, όπως κινητά ή tablets.

Στην παρούσα εργασία προτείνονται δυο νέες μέθοδοι που αντιμετωπίζουν το πρόβλημα μνήμης των MDP Πεπερασμένου Ορίζοντα. Οι μέθοδοι αυτές επιλέγουν να αποθηκεύουν στη μνήμη ένα μέρος της λύσης και στη συνέχεια να χρησιμοποιούν αυτό για επανυπολογισμό της υπόλοιπης, ανάλογα με τις εκάστοτε ανάγκες. Η πρώτη, που ονομάζεται Λύση Ρίζας, απαιτεί σημαντικά λιγότερη μνήμη και σχεδόν ίδιο χρόνο εκτέλεσης με την επικρατέστερη μέθοδο που χρησιμοποιείται ως τώρα για επίλυση MDP. Η δεύτερη λύση (Λογαριθμική Λύση) αποθηκεύει ακόμη μικρότερο μέρος της λύσης στη μνήμη (σχεδόν μηδαμινό), με μια μικρή επιβάρυνση χρόνου.

Τα παραπάνω συμπεράσματα, αφού θεμελιώθηκαν πρωτίστως θεωρητικά, επιβεβαιώθηκαν και πειραματικά, σε ήδη υπάρχοντα, προσαρμοσμένα στις ανάγκες, τεχνητά δεδομένα που αφορούν διαχείριση πόρων συστάδων υπολογιστών, τόσο για τον χρόνο εκτέλεσης των αλγορίθμων όσο και για την μνήμη που καταλαμβάνουν. Επιπλέον, συγκρίθηκαν τόσο με ήδη υπάρχοντες μεθόδους καθώς και με προσεγγιστικές μεθόδους επίλυσης.

Η συνεισφορά μας μέσω αυτής της εργασίας έγκειται στο γεγονός πως, με την πρόταση αυτών των νέων αλγορίθμων, ο χρήστης που αξιοποιεί MDP στην εκάστοτε εφαρμογή έχει την δυνατότητα να επιλέξει την λύση-αλγόριθμο που εξυπηρετεί όσο το δυνατόν καλύτερα τις ανάγκες του, αναλόγως με το σύστημα που διαθέτει.

Τέλος, έγινε προσπάθεια βελτιστοποίησης του χρόνου εκτέλεσης του βασικού επαναληπτικού αλγορίθμου επίλυσης MDP Απείρου Ορίζοντα (Value Iteration) με χρήση φραγμάτων ώστε να μειωθεί ο χρόνος σύγκλισης. Εντούτοις, οι προσπάθειες ήταν ανεπιτυχείς, πιθανώς εξαιτίας της εφαρμογής που επιλέχθηκε για αξιολόγηση.

Λέξεις-Κλειδία: Μαρκοβιανές Διαδικασίες Αποφάσεων, Ενισχυτική Μάθηση, Διαχείριση Πόρων, Πεπερασμένος Ορίζοντας, Άπειρος Ορίζοντας, Χωρική Πολυπλοκότητα, Χρονική Πολυπλοκότητα, Δέντρο Δυαδικής Αναζήτησης

Abstract

Markov Decision Processes (MDPs) are one of the most important statistical tools utilized towards solving decision problems under uncertain conditions. They are widely used in modern application, especially those involving the Reinforcement Learning framework. In such problems, an agent is required to make decisions which incur changes to its environment while also granting them a reward, acting as a motivation. The decisions the agent makes must be chosen such that the total reward they receive is maximized. Depending on whether the number of choices the agent makes is finite or infinite, the MDP can be characterized as having a Finite or Infinite Horizon.

Finite Horizon MDPs are preferred in a variety of application over the Infinite Horizon ones, as they better simulate real world problems, which must eventually terminate, such as resource management problems. Nevertheless, an important issue they present involves the memory the solution occupies on the system it runs, especially when the algorithm is executed in machines with limited hardware abilities, such as mobile phones or tablets.

In this work two new MDP-solving methods are introduced dealing with the memory problems Finite-Horizon MDPs face. Those methods opt for storing in memory a part of the solution and utilizing it to recalculate every other part of the solution as needed. The first, known as Root Solution, requires much less memory and almost the same execution time as the most predominant method used to solve Finite-Horizon MDPs. The second method, known as Logarithmic Solution, stores an even smaller, almost non-existent, part of the solution in memory with a small toll on execution time.

The above results were first and foremost discussed and proven in theory and validated afterwards using experiments on pre-existing, fit-to-need, simulated data regarding elastic resource management in cloud computing clusters. Those experiments involved execution time, memory needs as well as comparisons of the newly introduced methods with pre-existing ones and approximations. Our contribution through this thesis lies on the fact that, having suggested those new algorithms, any user desiring to utilize a FHMDP in their application is now able to opt for the algorithm that meets their needs as best as possible, depending on the system the possess.

Finally, an attempt towards optimizing the execution time of the fundamental iterative Infinite-Horizon MDP solving algorithm (Value Iteration) was made. This attempt involved calculating upper and lower bounds of the function the algorithm is iterating over to greatly reduce the execution time. Despite the attempt, our efforts did not turn out to be fruitful, probably because of the application used for evaluation.

Keywords: Markov Decision Processes, Reinforcement Learning, Resource Management, Infinite Horizon, Finite Horizon, Space Complexity, Time Complexity, Binary Search Tree

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον κ. Παναγιώτη Καρρά, αναπληρωτή καθηγητή στο Πανεπιστήμιο του Aarhus, που χωρίς αυτόν δεν θα μπορούσε να έχει εκπονηθεί το παρόν έργο. Η αδιάκοπη ενασχόλησή του με το θέμα αποτέλεσε πηγή έμπνευσης για εμένα, η καθοδήγηση και οι ιδέες του ήταν καταλυτικές για την υλοποίηση της διπλωματικής μου εργασίας, ενώ η συνεργασία μας ήταν άψογη και συνεχής.

Επίσης, αποδίδω ευχαριστίες και στην κ. Βασιλική Καντερέ, επίκουρη καθηγήτρια Ε.Μ.Π και επιβλέπουσα της εργασίας, τόσο για την παροχή πόρων και δεδομένων όποτε αυτό κρινόταν απαραίτητο όσο και για την άρτια οργάνωση της εργασίας και τις καίριες παρεμβάσεις της σε αυτήν.

Πολύτιμη ήταν και η συνεισφορά του επιστημονικού συνεργάτη στο πανεπιστήμιο του Aarhus, κ. Κωνσταντίνου Σκίτσα, τόσο στην πειραματική μελέτη των αποτελεσμάτων όσο και στην βιβλιογραφική έρευνα και τον ευχαριστώ για την άψογη συνεργασία μας.

Ευχαριστώ πολύ τον κ. Μιχαηλ Κατεχάκη, καθηγητή στο Πανεπιστήμιο του Rutgers, που με την τεράστια γνωστική του εμπειρία συνετέλεσε στην αρτιότερη εκπόνηση της εργασίας.

Τέλος ευχαριστώ την οικογένειά μου για την ανιδιοτελή στήριξη και εμπιστοσύνη τους σε εμένα.

Contents

0 Εκτεταμένη Ελληνική Περίληψη

0.1 Εισαγωγή

Η παρούσα διπλωματική εργασία ασχολείται με την βελτίωση αλγόριθμων επίλυσης Διαδικασιών Αποφάσεων Markov (MDPs) ως προς την χωρική πολυπλοκότητα. Τα MDPs χρησιμοποιούνται εκτεταμένα σε μια πληθώρα σύγχρονων εφαρμογών, ιδιαίτερα εφόσον είναι το βασικό εργαλείο που χρησιμοποιείται στα πλαίσια της Ενισχυτικής Μάθησης, ενός κλάδου της Μηχανικής Μάθησης με τεράστια απήχηση. Ένα MDP αποτελεί ένα στατιστικό μοντέλο που προσπαθεί να μοντελοποιήσει ένα πρόβλημα του πραγματικού κόσμου. Σε αυτό, ένας πράκτορας καλείται να λάβει αποφάσεις οι οποίες του αποδίδουν ανταμοιβές ενώ επηρεάζουν και την κατάσταση του συστήματος, με σκοπό να μεγιστοποιήσει την συνολική ανταμοιβή που θα συλλέξει μέσω των αποφάσεων, είτε μέχρι να πραγματοποιήσει όσες αποφάσεις έχουν οριστεί εξ αρχής είτε μέχρι να φτάσει σε μια τελική κατάσταση-στόχο. Στην πρώτη περίπτωση, εφόσον ο αριθμός των βημάτων είναι πεπερασμένος και ορισμένος εκ προοιμίου, το MDP μπορεί να χαρακτηριστεί ως Πεπερασμένου Ορίζοντα. Σε αντίθετη περίπτωση, όταν δηλαδή ο πράκτορας είτε δεν γνωρίζει εκ των προτέρων τον αριθμό των αποφάσεων που θα κληθεί να λάβει, είτε ο αριθμός αυτός είναι (φαινομενικά) άπειρος, το MDP διαθέτει Άπειρο Ορίζοντα. Η μοντελοποίηση ενός σύγχρονου προβλήματος οδηγεί συχνά στην δημιουργία πολύπλοκων MDP, τα οποία επιλύονται δύσκολα και απαιτούν αρκετούς υπολογιστικούς πόρους. Τα πράγματα γίνονται ακόμη πιο δυσμενή αν αναλογιστεί κανείς ότι όλο και περισσότερες εφαρμογές τρέχουν σε φορητές συσκευές, όπως κινητά ή tablets, τα οποία διαθέτουν περιορισμένες ικανότητες από πλευράς υλικού, όπως η μνήμη RAM. Ως εκ τούτου, η ανάγκη βελτιστοποίησης των ήδη υπαρχόντων αλγορίθμων για την επίλυση των MDP είναι πιο σημαντική από ποτέ. Η παρούσα εργασία προτείνει δυο νέους αλγορίθμους επίλυσης MDP Πεπερασμένου Ορίζοντα, με ραγδαίες βελτιώσεις στην απαιτούμενη κατά την εκτέλεση μνήμη. Ο πρώτος εξ αυτών απαιτεί τον ίδιο χρόνο εκτέλεσης με την ήδη υπάρχουσα βέλτιστη λύση, ενώ ο δεύτερος διαθέτει ελάχιστες απαιτήσεις σε μνήμη, με μια μικρή επιβάρυνση στον χρόνο εκτέλεσης. Ακόμη, έγινε προσπάθεια για βελτίωση του χρόνου εκτέλεσης του αλγορίθμου Value Iteration που χρησιμοποιείται κατά κόρον σε εφαρμογές MDP Απείρου Ορίζοντα, χωρίς όμως τα αποτελέσματα να είναι τα αναμενόμενα.

0.2 Θεωρητικό Υπόβαθρο

Για καλύτερη κατανόηση των βελτιώσεων των αλγορίθμων που προτείνονται στην Ενότητα 0.4, απαιτείται θεωρητική θεμελίωση των εννοιών των Markov Decision Processes, καθώς και των αλγορίθμων που χρησιμοποιούνται για επίλυση αυτών.

0.2.1 Markov Decision Processes

Όπως ήδη αναφέρθηκε στην Εισαγωγή, μια Διαδικασία Απόφασεων Markov (Markov Decision Process ή MDP) είναι ένα στατιστικό μοντέλο που χρησιμοποιείται για μοντελοποιήση ενός προβλήματος του πραγματικού κόσμου στο οποίο ένας πράκτορας (agent) αλληλεπιδρά με το περιβάλλον του, η συμπεριφορά του οποίου είναι μη ντετερμινιστική. Η αλληλεπίδραση αυτή συμβαίνει με την μορφή αποφάσεων του πράκτορα και εκτέλεση δράσεων, οι οποίες επηρέαζουν την κατάσταση του συστήματος. Κάθε τέτοια απόφαση λαμβάνει χώρα σε μια συγκεκριμένη χρονική στιγμή, η οποία ονομάζεται εποχή απόφασης (decision epoch). Αυτές οι χρονικές στιγμές μπορούν να είναι διακριτές ή συνεχείς. Στα πλαίσια αυτής της εργασίας θα εστιάσουμε σε διακριτές εποχές απόφασης. Το σύνολο που περιέχει όλες τις εποχές απόφασης συμβολίζεται ως H. Αναλόγως με το αν το πλήθος των στοιχείων αυτού του συνόλου είναι πεπερασμένο ή άπειρο, το MDP χαρακτηρίζεται ως Πεπερασμένου Ορίζοντα (Finite-Horizon) ή Άπειρου Ορίζοντα (Infinite-Horizon) αντίστοιχα.

Σε κάθε εποχή απόφασης, ο πράκτορας βρίσκεται σε μια κατάσταση του συστήματος, έστω s. Το σύνολο το οποίο περιέχει όλες τις καταστάσεις του συστήματος συμβολίζεται με S. Σε κάθε μια εξ αυτών, ο πράκτορας μπορεί να επιλέξει και να εκτελέσει μια δράση από ένα συγκεκριμένο σύνολο δράσεων που είναι διαθέσιμες στην εκάστοτε κατάσταση. Το σύνολο δυνατών δράσεων μιας κατάστασης s συμβολίζεται με $A(s)$. Επίσης, με A συμβολίζεται το σύνολο το οποίο περιέχει όλες τις πιθανές δράσεις του πράκτορα σε κάθε κατάσταση, δηλαδή $A = \bigcup_{s \in S} A(s)$.

Επιλέγοντας και εκτελώντας μια δράση, ο πράκτορας λαμβάνει μια ανταμοιβή, η οποία είναι ένας βαθμωτός αριθμός (θετικός, αρνητικός ή μηδέν), που προκύπτει από την πραγματική συνάρτηση $R(s, a)$ που χαρακτηρίζει το MDP. Εν συνεχεία, ο πράκτορας μεταβαίνει σε μια νέα κατάσταση με μη ντετερμινιστικό τρόπο. Αυτό συμβαίνει διότι σε κάθε κατάσταση s και για κάθε επιλεγμένη δράση a, ο πράκτορας έχει πιθανότητα $p(j|s, a)$ να μεταφερθεί στην κατάσταση j. Ορίζουμε την συνάρτηση $T(s, a, j)$ η οποία ισούται με $p(j|s, a)$. Για μια συγκεκριμένη κατάσταση s και δράση a, όλες οι πιθανότητες μετάβασης αθροίζουν στη μονάδα. Η συνάρτηση μετάβασης T φαίνεται ότι δεν έχει μνήμη από τον τρόπο που ορίστηκε, δηλαδή η επόμενη κατάσταση στην οποία θα μεταβεί ο πράκτορας εξαρτάται μόνο από την τρέχουσα κατάσταση στην οποία βρισκέται καθώς και την επιλεγμένη δράση.

Έχοντας ορίσει τα παραπάνω μεγέθη, μπορούμε να ορίσουμε φορμαλιστικά μια Διαδικασία Αποφάσεων Markov ως την συλλογή των αντικειμένων $\{H, S, A, T, R\}$.

0.2.2 Επιλύοντας το MDP

Βασικό κίνητρο για τον πράκτορα ενός MDP όπως αυτό περιγράφτηκε παραπάνω είναι η μεγιστοποίηση της συνολικής ανταμοιβής που θα λάβει μέσα από όλες τις αποφάσεις του. Αν συμβολίσουμε με T την τελευταία εποχή απόφασης (η οποία στην περίπτωση του Άπειρου Ορίζοντα ισούται με άπειρο), τότε η συνολική ανταμοιβή που θα λάβει ο πράκτορας μετά την εποχή απόφασης t είναι: $G_t = \sum_{i=t+1}^{T} R_i$.

Σε περιπτώσεις Άπειρου Ορίζοντα, όπου το άνω όριο της άθροισης απειρίζεται, η σειρά αποκλίνει, πράγμα το οποίο όπως θα δούμε στην επόμενη υποενότητα καθιστά την επίλυση αδύνατη. Προκειμένου να αντιμετωπιστεί αυτό το πρόβλημα, εισάγεται ένας παράγοντας έκπτωσης γ στο άθροισμα, με σκοπό να εξασφαλίσει την σύγκλιση. Η νέα συνολική ανταμοιβή με έκπτωση ισούται με: $G_t = \sum_{i=0}^{T} \gamma^i R_{i+t+1}$ Η σύγκλιση είναι εξασφαλισμένη για τιμές του γ μεταξύ 0 και 1.

Η τιμή του γ επηρεάζει την βαρύτητα που αποδίδει ο πράκτορας στις ανταμοιβές. Αν ο παράγοντας έκτπωσης είναι μηδέν ή κοντά στο μηδέν, ο πράκτορας θεωρεί σημαντικότερες τις άμεσες ανταμοιβές, αφού ο παράγοντας που πολλαπλασιάζεται με κάθε μελλοντική ανταμοιβή είναι όλο και πιο κοντά στο μηδέν. Αντίθετα, αν η τιμή του γ τείνει στην μονάδα, ο πράκτορας δίνει τεράστια βαρύτητα στις μελλοντικές ανταμοιβές.

Στόχος του πράκτορα είναι να ανακαλύψει τη βέλτιστη δράση για κάθε κατάσταση στην οποία μπορεί να βρεθεί, με σκοπό την μεγιστοποίηση των ανταμοιβών του. Ορίζοντας ως πολιτική μια αντιστοίχιση μεταξύ των καταστάσεων και πιθανοτήτων επιλογής μια δράσης a από την κάθε κατάσταση, μπορούμε να αναδιατυπώσουμε τον στόχο του πράκτορα λέγοντας ότι προσπαθεί να ανακαλύψει την βέλτιστη πολιτική. Η τιμή της πολιτικής π για μια δράση a σε μια κατάσταση s συμβολίζεται ως $\pi(a|s)$.

Προκειμένου να συνδράμουμε τον πράκτορα στην διαδικασία των αποφάσεών του, ορίζουμε ένα κριτήριο για αυτόν με βάση το οποίο θα μπορεί να κατανοεί πόσο ωφέλιμο ή όχι είναι να βρεθεί σε κάθε κατάσταση. Συγκεκριμένα, ορίζουμε ως συνάρτηση αξίας κάθε κατάστασης δεδομένης πολιτικής π ως:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{i+t+1} \middle| S_t = s\right]$$

Η συγκεκριμένη συνάρτηση κατ΄ ουσίαν εκφράζει την εκτιμώμενη συνολική ανταμοιβή που θα λάβει ο πράκτορας ξεκινώντας από την κατάσταση s και ακολουθώντας την πολιτική π. Ορίζουμε επίσης και την συνάρτηση δράσης-αξίας για μια συγκεκριμένη κατάσταση, δράση και πολιτικη ως:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{i+t+1} \middle| S_t = s, A_t = a\right]$$

Οι παραπάνω συναρτήσεις επιτρέπουν την σύγκριση μεταξύ πολιτικών ως εξής: μια πολιτική π είναι βελιστότερη ή το ίδιο βέλτιστη μιας πολιτικής π' αν η συνάρτηση αξίας της π είναι μεγαλύτερη ή ίση από την αντίστοιχη της π' σε κάθε κατάσταση

s. Ένα MDP μπορεί να έχει παραπάνω της μιας βέλτιστες πολιτικές. Η συνάρτηση αξίας αυτών είναι: $V^*(s) = \max_{\pi \in P(m)} V_\pi(s)$, όπου $P(m)$ το σύνολο που περιέχει όλες τις πολιτικές για ένα συγκεκριμένο MDP m.

Μπορεί να αποδειχθεί ότι η συνάρτηση αξίας που ορίστηκε παραπάνω ικανοποιεί την εξής αναδρομική σχέση, η οποία και ονομάζεται Εξίσωση Bellman:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)(r + \gamma V_\pi(s'))$$

Κατ'ουσίαν, η συνάρτηση αξίας μιας κατάστασης ως προς μια συγκεκριμένη πολιτική π προκύπτει ως η εκτιμώμενη τιμή της άμεσης ανταμοιβής που θα λάβει ο πράκτορας στο επόμενο βήμα συν την εκτιμώμενη ανταμοιβή που θα λάβει ο πράκτορας ξεκινώντας από την επόμενη κατάσταση στην οποία θα μεταβει, ακολουθώντας την πολιτική π. Η βέλτιστη συνάρτηση αξίας (δηλαδή αυτή που αντιστοιχεί στην βέλτιστη πολιτική) είναι εκείνη που μεγιστοποιείται σε κάθε κατάσταση, δηλαδή:

$$V_\pi(s) = \max_{a \in A(s)} \sum_{r,s'} p(s', r|s, a)(r + \gamma V_\pi(s'))$$

Αυτή η εξίσωση είναι γνωστή ως Εξίσωση Βελτιστότητας Bellman. Αποδεικνύεται ότι για πεπερασμένα MDP, δηλαδή MDP με πεπερασμένο αριθμό καταστάσεων και δράσεων (και οχι απαραίτητα πεπερασμένο ορίζοντα), η Εξίσωση Βελτιστότητας Bellman έχει μοναδική λύση, εφόσον κρύβει μέσα της ένα σύστημα τόσων εξισώσεων όσες και οι καταστάσεις του συστήματος. Εντούτοις, η λύση αυτού του συστήματος δεν είναι πάντοτε απλή υπόθεση, αφού πρέπει να ισχύουν τρεις θεμελιώδεις προϋποθέσεις:

- Να γνωρίζουμε όλες τις παραμέτρους του συστήματος εκ των προτέρων.
- Να διαθέτουμε τις απαραίτητες προδιαγραφές υπολογιστικών πόρων και μνήμης για να επιλυθεί το σύστημα σε ρεαλιστικό χρόνο και με ρεαλιστική μνήμη.
- Να ισχύει η Μαρκοβιανή ιδιότητα στο σύστημα (ισχύει εξ ορισμού αν μπορούμε να κατασκευάσουμε MDP)

Η πρώτη και η τρίτη προϋπόθεση αρκετά συχνά ισχύουν, τα πράγματα όμως δεν είναι έτσι και για την δεύτερη, αφού είναι ουκ ολίγες οι φορές που ένα MDP (ακόμα και σχετικά μικρού μεγέθους) χρειάζεται μη ρεαλιστικό χρόνο ή μνήμη για να επιλυθεί.

Προκειμένου να λυθεί η Εξίσωση Βελτιστότητας Bellman ακόμη και όταν δεν ισχύει η δεύτερη προϋπόθεση, επιστρατεύονται προσεγγιστικές μέθοδοι, με κύρια τον Δυναμικό Προγραμματισμό. Αν ισχύει η πρώτη και η τρίτη προϋπόθεση, μπορεί κανείς να υπολογίσει την συνάρτηση αξίας για μια συγκεκριμένη πολιτική π χρησιμοποιώντας την Εξίσωση Bellman ως αναδρομική εξίσωση ως:

$$V_k(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)(r + \gamma V_{k-1}(s'))$$

Η συγκεκριμένη αναδρομική εξίσωση, αφού συγκλίνει, θα έχει υπολογίσει την συνάρτηση αξίας για μια συγκεκριμένη πολιτική, χωρίς όμως να υπάρχει εγγύηση ότι αυτή ειναι βέλτιστη. Θα μπορούσε κανείς να επαναλάβει αυτή τη διαδικασία για μια άλλη πολιτική και στην συνέχεια να την συγκρίνει με αυτήν που υπολόγισε νωρίτερα, κρατώντας τελικά την πιο βέλτιστη. Έπειτα, η διαδικασία μπορεί να επαναληφθεί για όλες τις πολιτικές του MDP. Αυτή η λύση ονομάζεται Εκτίμηση Πόλιτικής (ή Πρόβλεψη Πολιτικής) και όπως γίνεται σαφές δεν είναι ιδιαίτερα αποδοτική αφού είναι εξαντλητική. Παρόλα αυτά, η Εκτίμηση Πολιτικής είναι ένα σημαντικό εργαλείο για τον υπολογισμό της συνάρτησης αξίας μιας πολιτικής. Σημειώνεται ότι για έλεγχο σύγκλισης χρησιμοποείται η μέγιστη διαφορά μεταξύ παλαιάς και νέας τιμής συνάρτησης αξίας για μια κατάσταση, με τον αλγόριθμο να τερματίζει αν αυτή είναι μικρότερη από ένα επιλεγμένο κατώφλι.

Μια ακόμη λύση περιλαμβάνει την ανταλλαγή δράσεων μιας πολιτικής αν αυτό επιφέρει βελτιστοποίηση αυτής. Με άλλα λόγια, κοιτάζοντας κάθε κατάσταση, προσπαθούμε να διαπιστώσουμε αν αλλάζοντας την ήδη επιλεγμένη δράση με μια άλλη η συνάρτηση αξίας βελτιώνεται και αν ναι εκτελούμε την ανταλλαγή. Αυτό μπορεί να εκφραστεί ως άπληστο κριτήριο μέσω της εξίσωσης υπολογισμού της νέας, βελτιστότερης πολιτικής ως:

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, Q_\pi(s,a) = \underset{a}{\operatorname{argmax}} \sum_{r,s'} p(s',r|s,a)(r + \gamma V_\pi(s'))$$

. Η μέθοδος με την οποία μια πολιτική μετατρέπεται σε μια βελτιστότερη ονομάζεται Βελτίωση Πολιτικής. Αυτή είναι και η βάση για τον πρώτο προσεγγιστικό αλγόριθμο που ονομάζεται Επανάληψη ως προς την Πολιτική (Policy Iteration). Ξεκινώντας από μια τυχαία, αρχική πολιτική, εφαρμόζουμε την Εκτίμηση Πολιτικής για υπολογισμό της συνάρτησης αξίας. Έπειτα, επιδιώκουμε Βελτίωση Πολιτικής. Αν η πολιτική πράγματι αλλάξει, επαναλαμβάνουμε το προηγούμενο βήμα ξανά, αλλιώς ο αλγόριθμος τερματίζει και επιστρέφεται η βέλτιστη, πλέον, πολιτική.

Η παραπάνω μέθοδος, αν και λειτουργική, δεν είναι ιδιαίτερα αποδοτική στην πράξη, εξαιτίας της εκτεταμένης χρήσης της Εκτίμησης Πολιτικής. Η δεύτερη και πιο αποδοτική προσεγγιστική λύση, προκύπτει αν επιλέξουμε να εκτελέσουμε μόνο μια επανάληψη της Εκτίμησης Πολιτικής τη φορά, ενώ αποδεικνύεται ότι δεν παραβιάζονται οι συνθήκες σύγκλισης. Κατ΄αυτήν την λογική, η αναδρομική συνάρτηση Bellman μετατρέπεται σε:

$$V_k(s) = \max_a \sum_{r,s'} p(s',r|s,a)(r + \gamma V_{k-1}(s'))$$

Ως συνθήκη τερματισμού της αναδρομής χρησιμοποείται και πάλι η μέγιστη διαφορά συνάρτησης αξίας των καταστάσεων. Ο αλγόριθμος αυτός, που ονομάζεται Επανάληψη ως προς την Αξία (Value Iteration), παρουσιάζεται αναλυτικά σε ψευδοκώδικα στον Αλγόριθμο 1

Algorithm 1 Επανάληψη ως προς την Αξία

```
function valueIteration(θ)
    Initialize V_tmp(s), V_aux(s), Q(s,a) and π(s) (arbitrarily)
    δ ← ∞
    while δ ≥ θ do
        δ ← 0
        for s ∈ S do
            for a ∈ A(s) do
                Q(s,a) ← 0
                for s' ∈ S(s) do
                    Q(s,a) ← Q(s,a) + T(s,a,s')(R(s,a,s') + V_tmp(s'))
                end for
            end for
            V_aux(s) ← max_a Q(s,a)
            π(s) = argmax_a Q(s,a)
        end for
        V_tmp ← V_aux
    end while
    return V and π
end function
```

Με $S(s)$ συμβολίζεται το υποσύνολο των καταστάσεων του συστήματος που είναι προσβάσιμα από την κατάσταση s. Στην περίπτωση που ο ορίζοντας του MDP είναι γνωστός και πεπερασμένος, η διαδικασία επαναλαμβάνεται για τόσα βήματα όσο και το μέγεθος του ορίζοντα, δεν χρησιμοποιείται (απαραίτητα) ο παράγοντας έκπτωσης γ, ενώ είναι απαραίτητο να αποθηκευτεί στη μνήμη η πολιτική που υπολογίζεται σε κάθε βήμα του αλγορίθμου (μη στάσιμη πολιτική).

0.2.3 Θέωρημα Turnpike

Συχνά η επίλυση MDP Πεπερασμένου Ορίζοντα μπορεί να αποδειχθεί πολύπλοκη ακόμη και με τις παραπάνω βελτιστοποιήσεις, εφόσον εξαρτάται από το μέγεθος του ορίζοντα N. Αντίθετα, σε περιπτώσεις Απείρου Ορίζοντα οι συνθήκες είναι σαφώς πιο ομαλές, αφού η σύγκλιση μπορεί να επιτευχθεί πολύ γρηγορότερα από ό,τι σε ένα αντίστοιχο MDP Πεπερασμένου Ορίζοντα.

Η "μετατροπή" ενός MDP Πεπερασμένου Ορίζοντα σε Απείρου είναι επιτεύξιμη, χρησιμοποιώντας ένα θεώρημα που ονομάζεται Θεώρημα Turnpike. Σύμφωνα με αυτό, σε κάθε MDP Πεπερασμένου Ορίζοντα και για κάθε τιμή του παράγοντα έκπτωσης γ, υπάρχει ένας αριθμός εναπομεινάντων βημάτων $N^*(\gamma)$, γνωστός ως Ακέραιος Turnpike, έως και τον οποίον η πολιτική που υπολογίζεται είναι στάσιμη και προκύπτει εκτελώντας τον αλγόριθμο επίλυσης Άπειρου Ορίζοντα. Χρησιμοποιώντας αυτό το γεγονός, ενα MDP Πεπερασμένου Ορίζοντα θα μπορούσε να επιλυθεί ως εξής: υπολογίζεται η πολιτική σαν ο ορίζοντας να ήταν άπειρος και χρησιμοποιείται για να ληφθούν αποφάσεις από αριθμό βημάτων N (μέγεθος ορίζοντα) έως και $N^*(\gamma)$. Έπειτα, το MDP αντιμετωπίζεται ως MDP Πεπερασμένου

Ορίζοντα, με μέγεθος ορίζοντα $N^*(\gamma)-1$ οπότε και χρησιμοποιείται μια κλασσική λύση. Η δυσκολία αυτού του αλγορίθμου έγκειται στο γεγονός πως δεν υπάρχει σαφής τρόπος έυρεσης του Ακεραίου Turnpike.

0.3 Μια Σημείωση για την Βελτιστοποίηση Άπειρου Ορίζοντα

Παρατηρήθηκε ότι υπάρχει περιθώριο βελτίωσης στον χρόνο επίλυσης του αλγορίθμου Επανάληψης ως προς την Αξία (Value Iteration), προσπαθώντας να φράξουμε την συνάρτηση δράσης-αξίας και κατ΄αυτον τον τρόπο να αποκλειστούν δράσεις οι οποίες εγγυημένα δεν οδηγούν σε βέλτιστη λύση. Συγκεκριμένα, τα άνω και κάτω φράγματα της συνάρτησης δράσης-αξίας υπολογίστηκαν σε:

$$Q^*(s,a) \leq \max_{s',r} r + \gamma \max_{s''} V^*(s'') = Q_u(s,a)$$

$$Q^*(s,a) \geq \min_{s',r} r + \gamma \min_{s''} V^*(s'') = Q_l(s,a)$$

Αρχικό βήμα του αλγορίθμου είναι ο υπολογισμός αυτών των άνω και κάτω ορίων. Αυτό φυσικά πραγματοποιείται αφού έχουν τρέξει ορισμένες επαναλήψεις του Value Iteration, ώστε η συνάρτηση αξίας να έχει ήδη ξεκινήσει να συγκλίνει. Επιπλέον, υπολογίζεται για κάθε κατάσταση το μέγιστο κάτω φράγμα των συναρτήσεων δράσης-αξίας της. Τέλος, για κάθε κατάσταση και για κάθε δράση, αν το άνω φράγμα της συνάρτησης δράσης-αξίας αυτού του συνδυασμού κατάστασης-αξίας είναι μικρότερο από το μέγιστο κάτω φράγμα της κατάστασης, η δράση διαγράφεται εφόσον δεν μπορεί εγγυημένα να οδηγήσει σε βέλτιστο αποτέλεσμα.

Παρόλη την θεωρητική θεμελίωση, η πειραματική αξιολόγηση των αποτελεσμάτων δεν απέδωσε καρπούς, με τους χρόνους που σημειώθηκαν να είναι χειρότεροι ή ίσοι με αυτούς χωρίς την εφαρμογή των φραγμάτων. Ένας πιθανός λόγος είναι ο μικρός αριθμός δυνατών δράσεων σε κάθε κατάσταση, όπως θα δούμε στην Ενότητα 0.5, που είχε ως αποτέλεσμα να μην αποκλείονται δράσεις.

Σημειώνεται ότι στο παρελθόν έχουν προταθεί αρκετά μοντέλα που χρησιμοποιούν την ιδέα φράξης της συνάρτησης αξίας, όπως φαίνεται στα [Mac67], [SB79], [Whi82] and [DWG+11].

0.4 Θεωρητική Θεμελίωση Βελτιστοποίησης Πεπερασμένου Ορίζοντα

0.4.1 Ήδη Υπάρχουσες Λύσεις

Όπως ήδη αναφέρθηκε στην Εισαγωγή, οι ήδη υπάρχουσες λύσεις των προβλημάτων MDP Πεπερασμένου Ορίζοντα επιδέχονται σημαντικής βελτίωσης στις απαιτήσεις σε μνήμη. Για να κατανοηθεί που έγκειται το πρόβλημα μνήμης, θα αναφερθούμε αρχικά σε δυο ήδη υπάρχουσες μεθόδους επίλυσης MDP Πεπερασμένου Ορίζοντα. Η πρώτη εξ αυτών στα πλαίσια αυτής της εργασία αποκαλείται Αφελής Λύση (Naïve Solution), αφού είναι η πρώτη λύση που έρχεται στον νου,

βασισμένη στην μέθοδο Value Iteration που χρησιμοποιείται ήδη για τον Άπειρο Ορίζοντα. Αυτή η λύση υπολογίζει κάθε πίνακα πολιτικής (και συνάρτησης αξίας) για κάθε αριθμό βημάτων που απομένουν, χρησιμοποιώντας κάθε φορά τον προηγούμενο για να υπολογίσει τον επόμενο, και αποθηκεύοντάς τους όλους στη μνήμη. Για κάθε απόφαση που λαμβάνει ο πράκτορας, ο πίνακας πολιτικής που αντιστοιχεί στον αριθμό βημάτων που του απομένουν ανασύρεται από την μνήμη και αφού χρησιμοποιηθεί στην συνέχεια διαγράφεται.

Όπως είναι φυσικό, ο συγκεκριμένος αλγόριθμος έχει χρονική πολυπλοκότητα ανάλογη του μεγέθους του ορίζοντα για σταθερό αριθμό καταστάσεων και δράσεων, αφού εκτελεί τόσες επαναλήψεις όσο και το μέγεθος του ορίζοντα. Η κάθε μια εξ αυτών είναι πολυπλοκότητας $\mathcal{O}(|S|^2|A|)$, οπότε συνολικά η πολυπλοκότητα που προκύπτει είναι : $\mathcal{O}(N|S|^2|A|)$. Η χωρική πολυπλοκότητα από την άλλη είναι επίσης γραμμική ως προς το μέγεθος του ορίζοντα, αφού αποθηκεύεται ένας πίνακας μεγέθους όσες και οι καταστάσεις για κάθε αριθμό εναπομεινάντων βημάτων, και υπολογίζεται σε $\mathcal{O}(N|S|)$.

Μια δεύτερη σκέψη με σκοπό την μείωση της χωρικής πολυπλοκότητας θα ήταν να μην αποθηκεύεται κανένας πίνακας πολιτικής στη μνήμη και να υπολογίζονται όλοι όποτε αυτό είναι απαραίτητο. Αυτή είναι και η λύση In-Place, σύμφωνα με την οποία, σε κάθε αριθμό βημάτων, ο πράκτορας υπολογίζει την πολιτική που αντιστοιχεί σε αυτόν, χωρίς να αποθηκεύει κανένα ενδιάμεσο αποτέλεσμα, την χρησιμοποιεί για να λάβει την απόφαση, και έπειτα προχωρά στην επόμενη.

Όπως καθίσταται σαφές, η χωρική πολυπλοκότητα αυτού του αλγορίθμου είναι $\mathcal{O}(|S|)$, αφού το πολύ 3 πίνακες είναι αποθηκευμένοι στη μνήμη ανά πάσα στιγμή, και εντελώς ανεξάρτητη του μεγέθους του ορίζοντα. Το μεγάλο πρόβλημα αυτής της προσέγγισης, ωστόσο, έγκειται στον απαιτούμενο χρόνο εκτέλεσης. Συγκεκριμένα, αυτός είναι τετραγωνικός ως προς το μέγεθος του ορίζοντα, αφού για κάθε αριθμό εναπομεινάντων βημάτων K απαιτούνται $\mathcal{O}(K|S|^2|A|)$ πράξεις, ενώ το K κυμαίνεται από N έως 1.

0.4.2 Προσεγγιστικές Λύσεις

Κατά την μελέτη των μεθόδων επίλυσης MDP Πεπερασμένου Ορίζοντα ελέγχθηκε η απόδοση και δυο προσεγγιστικών λύσεων, οι οποίες βασίζονται στο Θεώρημα Turnpike που περιεγράφηκε στην ενότητα 0.2. Σύμφωνα με αυτό, για αριθμό βημάτων μεγαλύτερο από έναν ακέραιο n_0 η πολιτική γίνεται στατική ανεξαρτήτως των βημάτων Λαμβάνοντας υπόψιν το παραπάνω, θα μπορούσε κανείς να ισχυριστεί ότι μια θεωρητική αποδοτική μέθοδος για την επίλυση ενός MDP Πεπερασμένου Ορίζοντα θα ήταν να υπολογιστεί μια στατική πολιτική, σαν ο ορίζοντας να ήταν άπειρος, να χρησιμοποιηθεί αυτή από τα εναπομείναντα βήματα N έως τον ακέραιο Turnpike n_0 και έπειτα για τα επόμενα $N - n_0$ να υπολογιστεί μια ξεχωριστή πολιτική για κάθε βήμα κατά τα γνωστά. . Η δυσκολία ωστόσο έγκειται στην εύρεση αυτού του ακεραίου n_0. Για τον λόγο αυτό, οι δυο προσεγγιστικές μέθοδοι που εξετάσαμε υποθέτουν ότι για μεγάλες τιμές του ορίζοντα (μεγαλύτερες των 200 βημάτων) αυτός ο ακέραιος θα έχει σχετικά χαμηλή τιμή, και ως εκ τούτου η ίδια πολιτική μπορεί να χρησιμοποιηθεί και για τα τελευταία βήματα χωρίς μεγάλες απώλειες ακρίβειας.

Η πρώτη μέθοδος που αποκαλούμε Προσέγγιση του Άπειρου Ορίζοντα (ή απλώς Προσέγγιση για συντομία) εκτελεί τον αλγόριθμο Value Iteration κατά τα γνωστά και υπολογίζει μια στατική πολιτική θεωρώντας ότι ο ορίζοντας είναι άπειρος. Έπειτα, αυτή η πολιτική χρησιμοποιείται για κάθε βήμα. Σημασία σε αυτήν την μέθοδο έχει και η επιλογή του παράγοντα έκπτωσης γ, αφού επηρεάζει το βάρος που δίνει ο πράκτορας σε μελλοντικές αποφάσεις του. Όσο μικρότερο το γ, τόσο μεγαλύτερο βάρος δίνεται σε άμεσες αποφάσεις, οπότε μικρές τιμές του γ εξυπηρετούν περισσότερο μικρούς ορίζοντες, ενώ αντίθετα όσο μεγαλύτερος ο ορίζοντας τόσο καλύτερα αποτελέσματα αναμένουμε για τιμές του γ κοντά στην μονάδα. Η πολυπλοκότητα αυτής της μεθόδου είναι όση και αυτή ενός κλασσικού αλγορίθμου επίλυσης MDP Άπειρου Ορίζοντα, δηλαδή $\mathcal{O}(|S|^2|A|)$, ενώ χωρικά είναι ίδια με του In-Place.

Η δεύτερη μέθοδος υπολογίζει κατά τα γνωστά την πολιτική που αντιστοιχεί σε εναπομείναντα αριθμό βημάτων N, όσος δηλαδή και ο ορίζοντας, και χρησιμοποιεί αυτόν για να λάβει όλες τις αποφάσεις. Αυτό αναμένεται να δώσει αρκετά καλά αποτελέσματα για μεγάλες τιμές ορίζοντα, αφού είναι πολύ πιθανό ο ακέραιος Turnpike να εμπεριέχεται στον ορίζοντα, όσο μικρότερη είναι η τιμή όμως τόσο αναμένουμε να πέσει η απόδοση. Η χρονική πολυπλοκότητα εκτιμάται σε $\mathcal{O}(N|S|^2|A|))$ ενώ η χωρική είναι ίδια με του In-Place.

0.4.3 Λύση Ρίζας

Οι λύσεις που προτείνονται για μείωση της χωρικής πολυπλοκότητας χωρίς την τεράστια χρονική επιβάρυνση της In-Place λύσης και χωρίς την απώλεια ακρίβειας των δυο προσεγγιστικών λύσεων στηρίζονται στην ισορροπία μεταξύ της Αφελούς Λύσης και της In-Place Λύσης.

Η πρώτη εξ αυτών ονομάζεται Λύση Ρίζας. Σύμφωνα με αυτήν, κατά τον υπολογισμό του πρώτου πίνακα πολιτικής, αυτού δηλαδή με δείκτη όσο και το μέγεθος του ορίζοντα, αποθηκεύονται $\mathcal{O}(\sqrt{N})$ ακόμη πίνακες συνάρτησης αξίας, αυτοί των οποίων οι δείκτες είναι πολλαπλάσια της ρίζας του μεγέθους του ορίζοντα. Ο κάθε υπολογισμός επόμενου πίνακα ξεκινά από τον αποθηκευμένο πίνακα με τον μεγαλύτερο δείκτη αντί από την αρχή. Επιπλέον, σε κάθε διάστημα μήκους $\sqrt{N}$, την πρώτη φορά που υπολογίζεται ο πίνακας του οποίου ο δείκτης αντιστοιχεί στο τέλος του διαστήματος αποθηκεύονται και όλοι οι ενδιάμεσοι, με επιβάρυνση $\mathcal{O}(\sqrt{N})$ ακόμη πίνακες συνάρτησης αξίας ως προς τη μνήμη. Η συγκεκριμένη λύση παρουσιάζεται σε ψευδοκώδικα στον Αλγόριθμο 2. Σημειώνεται ότι η συνάρτηση $calculateValues$ που αναφέρεται υπολογίζει και επιστρέφει τον πίνακα που αντιστοιχεί στον δείκτη του πρώτου ορίσματος, ξεκινώντας από τον πίνακα που δίνεται ως τριτο όρισμα (με δείκτη τον αριθμό που δίνεται ως δεύτερο όρισμα). Επιπλέον, αν το τέταρτο όρισμα τεθεί σε $true$, αποθηκεύονται όλοι οι ενδιάμεσοι πίνακες συνάρτησης αξίας που υπολογίζονται.

Algorithm 2 Λύση Ρίζας

```
1: procedure rootSolution(N)
2:     V ← zeros(N)
3:     steps = N
4:     for i = 0 to N step ⌊√N⌋ do
5:         calculateValues(i + ⌊√N⌋, i, V, false)
6:     end for
7:     if indexStack.top() < N then
8:         calculateValues(N, indexStack.top(), valueStack.top(), true)
9:     end if
10:    while steps > 0 do
11:        if indexStack.empty() then
12:            calculateValues(steps, 0, zeros(N), false)
13:        else if ((steps + 1)mod⌊√N⌋) == 0 then
14:            calculateValues(steps, indexStack.top(), valueStack.top(), true)
15:        end if
16:        V ← valueStack.top()
17:        chosenAction = calculateBestAction(V[currentState])
18:        takeAction(chosenAction)
19:        steps = steps − 1
20:    end while
21: end procedure
```

Η λύση αυτή αποδεικνύεται ότι έχει χρονική πολυπλοκότητα $\mathcal{O}(N|S|^2|A|)$, ίδια δηλαδή (ασυμπτωτικά) με αυτήν της Αφελούς Λύσης, εφόσον αποθηκεύονται και οι ενδιάμεσοι πίνακες σε κάθε διάστημα μήκους $\sqrt{N}$. Από την άλλη, η χωρική πολυπλοκότητα είναι της τάξης του $\mathcal{O}(\sqrt{N}|S|)$, αφού ανά πάσα στιγμή στη μνήμη βρίσκονται αποθηκευμένοι $2\sqrt{N}$ πίνακες μήκους $|S|$.

0.4.4 Λογαριθμική Λύση

Η δεύτερη λύση που προτείνουμε επιχειρεί να μειώσει ακόμη περισσότερο την χωρική πολυπλοκότητα, με μια μικρή χρονικη επιβάρυνση. Η λύση αυτή, που ονομάζεται Λογαριθμική Λύση, ακολουθεί την λογική της Δυαδικής Αναζήτησης για να αποφασίσει ποίοι πίνακες συνάρτησης αξίας θα αποθηκευτούν στη μνήμη. Για να υπολογίσουμε έναν πίνακα με δείκτη *target* αρχικά δυο δείκτες l, r αρχικοποιούνται σε 0 και N αντίστοιχα. Ο δείκτης k που υποδεικνύει τον πίνακα που εξετάζουμε κάθε φορά ξεκινά από το $\frac{l+r}{2}$, αλλιώς από τον μεγαλύτερο δείκτη ήδη αποθηκευμένου πίνακα στη μνήμη. Έπειτα διακρίνουμε τρεις περιπτώσεις:

- Αν ο δείκτης k είναι **ο ζητούμενος**, υπολογίζουμε την συνάρτηση αξίας του χωρίς να αποθηκεύουμε κανένα ενδιάμεσο αποτέλεσμα. Έπειτα, επιστρέφεται αυτή η συνάρτηση αξίας.
- Αν ο δείκτης k είναι **μικρότερος** από αυτόν που ψάχνουμε, ο πίνακας συνάρτησης αξίας που αντιστοιχεί στον δείκτη k υπολογίζεται και απο-

θηκεύεται στη μνήμη. Έπειτα, ο δείκτης l παίρνει την τιμή $k+1$, ο δείκτης r δεν αλλάζει, ενώ ο δείκτης k υπολογίζεται ξανά ως $\frac{l+r}{2}$. Έπειτα, εφόσον $l \leq r$ η διαδικασία επαναλαμβάνεται έως ώτου βρεθεί ο k.

- Αν ο δείκτης k είναι **μεγαλύτερος** από αυτόν που ψάχνουμε, ο δείκτης r παίρνει την τιμή $k-1$, ο δείκτης l δεν αλλάζει, ενώ ο δείκτης k υπολογίζεται ξανά ως $\frac{l+r}{2}$. Έπειτα, εφόσον $l \leq r$ η διαδικασία επαναλαμβάνεται έως ώτου βρεθεί ο k.

Εφόσον σε κάθε αναζήτηση και υπολογισμό πίνακα πραγματοποιείται μια Δυαδική Αναζήτηση, το πολύ $\log_2(N)$ πίνακες θα ελεγχθούν (και άρα ίσως αποθηκευθούν στη μνήμη). Ένας διαφορετικός τρόπος οπτικοποίησης του αλγορίθμου μπορεί να επιτευχθεί αν σκεφτούμε ένα νοητό δένδρο δυαδικής αναζήτησης, του οποίου κάθε επίπεδο είναι γεμάτο εκτός ίσως από το τελευταίο. Το δένδρο αυτό περιέχει όλους τους δείκτες των πινάκων συνάρτησης αξίας από 1 έως N. Κάθε φορά που χρειαζόμαστε έναν πίνακα με δείκτη k, αποθηκεύονται στη μνήμη όλοι οι πίνακες συνάρτησης αξίας των οποίων οι δείκτες βρίσκονται στο μονοπάτι από την ρίζα προς τον δείκτη k και είναι μικρότεροι του k. Για κάθε επανυπολογισμό πίνακα, ξεκινάμε από τον πίνακα με τον μεγαλύτερο δείκτη που είναι ήδη αποθηκευμένος στη μνήμη. Τα παραπάνω διακρίνονται σε ψευδοκώδικα στον Αλγόριθμο 3. Αυτός ο αλγόριθμος χρησιμοποιείται στον Αλγόριθμο 4 για την επίλυση του MDP.

Algorithm 3 Βοηθητική Συνάρτηση Αποθήκευσης Πινάκων Συνάρτησης Αξίας

```
1: function treeTraversal(targetIndex, N)
2:     l ← 0
3:     r ← N
4:     k ← (l+r)/2
5:     V_tmp ← []
6:     if notindexStack.empty() then
7:         if indexStack.top() == targetIndex then
8:             V_tmp ← valueStack.top()
9:             valueStack.pop()
10:            indexStack.pop()
11:            return V_tmp
12:        else
13:            k ← indexStack.top()
14:        end if
15:    end if
16:    while l ≤ r do
17:        if k == target then
18:            if indexStack.empty() then
19:                calculateValues(k, 0, zeros(N), false)
20:            else
21:                calculateValues(k, indexStack.top(), valueStack.top(), false)
22:            end if
23:            valueStack.pop()
24:            indexStack.pop()
25:            V_tmp ← valueStack.top()
26:            break
27:        else if k < target then
28:            if indexStack.empty() then
29:                calculateValues(k, 0, zeros(N), false)
30:            else if indexStack.top()! = k then
31:                calculateValues(k, indexStack.top(), valueStack.top(), false)
32:            end if
33:            l = k + 1
34:            k = (l+r)/2
35:        else
36:            r = k − 1
37:            k = (l+r)/2
38:        end if
39:    end while
40:    return V_tmp
41: end function
```

Algorithm 4 Λογαριθμική Λύση

```
1: procedure treeSolution(N)
2:     steps ← N
3:     V ← []
4:     while steps > 0 do
5:         V ← treeTraversal(steps, N)
6:         chosenAction = calculateBestAction(V[currentState])
7:         takeAction(chosenAction)
8:         steps = steps − 1
9:     end while
10: end procedure
```

Αποδεικνύεται ότι η χρονική πολυπλοκότητα αυτού του αλγορίθμου είναι $\mathcal{O}(N \log_2(S)|S|^2|A|)$, χειρότερη δηλαδή σε σχέση με αυτήν της Αφελούς Λύσης. Εντούτοις, η χωρική πολυπλοκότητα είναι $\mathcal{O}(\log_2(N)|S|)$, με αποτέλεσμα το μέγεθος της μνήμης που απαιτείται για αποθήκευσης της πολιτικής να είναι σχεδόν μηδαμινό.

0.5 Περιγραφή Υλοποίησης

Προκειμένου να ελέγξουμε τα παραπάνω θεωρητικά ευρήματα καθώς και να συγκρίνουμε τις επιδόσεις των διαφόρων αλγορίθμων, κατασκευάσαμε πειράματα στηριζόμενα στην εφαρμογή που περιγράφεται στην εργασία [LKKK17]. Συγκεκριμένα, το πρόβλημα που πραγματεύεται η συγκεκριμένη εργασία αφορά μια συστάδα υπολογιστών σε νέφος, η οποία διαχειρίζεται μια βάση δεδομένων. Αυτή λαμβάνει αιτήματα ανάγνωσης και εγγραφής δεδομένων, ενώ μπορεί να μεταβάλλει την υπολογιστική της ισχύ προσθέτοντας ή αφαιρόντας εικονικές μηχανές (VMs). Ο πράκτορας σε αυτήν την εφαρμογή δρα ως οργανωτής (coordinator) της συστάδας, προσθαφαιρώντας εικονικές μηχανές κατά τις ανάγκες του συστήματος, ώστε να εξυπηρετείται όσο το δυνατόν αρτιότερα το φορτίο. Οι δυνατές του δράσεις περιορίζονται σε προσθήκη μιας εικονικής μηχανής, αφαίρεση μιας εικονικής μηχανής ή διατήρηση της κατάστασης της συστάδας ως έχει (καμία ενέργεια). Οι καταστάσεις του MDP προκύπτουν βάσει του αριθμού των εικονικών μηχανών που είναι ενεργές και του επερχόμενου φορτίου.

0.5.1 Διαφορές με την Αρχική Υλοποίηση

Αρχικά, είναι σημαντικό να αναφέρουμε το πως προσαρμόσαμε την εφαρμογή στις ανάγκες μας. Τα πειράματα ήταν γραμμένα σε Python 3, οπότε μεταφράστηκαν σε C++, η οποία θεωρούμε ότι προσφέρει περισσότερες δυνατότητες για μελέτη των μεγεθών που μας ενδιέφεραν, δηλαδή χρόνου και μνήμης. Επιπλεόν, στην συγκεκριμένη εφαρμογή, οι ερευνητές ήλεγχαν τέσσερις διαφορετικούς αλγορίθμους επίλυσης του MDP που δημιουργείται από αυτό το πρόβλημα. Για τις ανάγκες του πειράματός μας, επιλέξαμε να χρησιμοποιήσουμε μόνο τον έναν εξ' αυτών, τον απλούστερο, δηλαδή τον κλασσικό αλγόριθμο επίλυσης MDP. Σημειώνουμε ότι

παρόλο που ο αλγόριθμος έτρεχε σαν σε MDP Απείρου Ορίζοντα στα πειράματα των ερευνητών, τα βήματα τα οποία εκτελούσε ήταν πεπερασμένα σε αριθμό. Στα δικά μας πειράματα, θεωρήσαμε ότι ο πράκτορας γνωρίζει εκ των προτέρων το μέγεθος του ορίζοντα, και επομένως εκτελεί αλγόριθμο επίλυσης MDP Πεπερασμένου Ορίζοντα. Τέλος, το μοντέλο δεν ήταν γνωστό εξ αρχής στον πράκτορα, οπότε εκείνος το μάθαινε δυναμικά, ανανεώνοντας τις παραμέτρους αυτού τόσο κατά την εκπαίδευση όσο και κατά την αξιολόγηση. Στην δική μας περίπτωση, θεωρήσαμε ότι μετά το τέλος της εκπαίδευσης του πράκτορα αυτός αποκτά πλήρη εποπτεία του μοντέλου, δηλαδή γνωρίζει όλες τις παραμέτρους αυτού, ενώ αυτές δεν αλλάζουν.

0.5.2 Περιγραφή Κλάσεων

Το μοντέλο που χρησιμοποιήθηκε στα πειράματα στηρίζεται σε μια δομή αποτελούμενη απο κλάσεις. Η βασικότερη εξ αυτών είναι η κλάση $QState$, η οποία περιέχει πληροφορίες που αφορούν σε έναν συγκεκριμένο συνδυασμό κατάστασης και δράσης. Η κλάση $State$ αναπαριστά μια κατασταση του MDP και περιέχει μια $QState$ για κάθε δυνατή δράση. Το ίδιο το μοντέλο, που εκφράζεται μέσω της κλάσης $MDPModel$ αποτελείται από μια $State$ για κάθε κατάστασή του. Η δομή αυτή είναι ίδια με εκείνη που χρησιμοποιήθηκε στο [LKKK17]. Για τις ανάγκες μας, κατασκευάσαμε μια κλάση που επεκτείνει την $MDPModel$ και περιλαμβάνει τους νέους αλγορίθμους επίλυσης του MDP Πεπερασμένου Ορίζοντα, γνωστής ως $FiniteMDPModel$. Για την εκπαίδευση ορίζεται επιπλέον και η κλάση $Complex$, η οποία περιέχει πληροφορίες που αφορούν σε ένα σενάριο ποροσμοίωσης φορτίων για την συστάδα.

0.5.3 Εκπαίδευση

Για την εκπαίδευση χρησιμοποιήθηκε η κλάση $Complex$. Σε κάθε βήμα εκπαίδευσης, ο πράκτορας είτε επιλέγει τυχαία μια δράση από τις εφικτές με πιθανότητα ϵ είτε απλώς υπολογίζει και επιλέγει την βέλτιστη δράση με πιθανότητα $1 - \epsilon$ (μέθοδος ε-greedy). Έπειτα, η δράση εκτελείται και ο πράκτορας λαμβάνει τα αποτελέσματά της μέσω της κλάσης $Complex$, τα οποία και χρησιμοποιεί για να ανανεώσει το μοντέλο. Κάθε 500 βήματα, εκτελείται η συνάρτηση Value Iteration ώστε ο πράκτορας να μπορεί να αποφασίσει την βέλτιστη δράση για το ενημερωμένο μοντέλο.

0.5.4 Αξιολόγηση

Αφού εκπαιδευτεί ο πράκτορας, εκτελεί καθέναν από τους επιλεγμένους αλγορίθμους επίλυσης MDP Πεπερασμένου Ορίζοντα. Κάθε φορά που συλλέγει μια ανταμοιβή την προσθέτει σε μια ειδική μεταβλητή, ώστε να την συγκρίνει με την αντίστοιχη των άλλων αλγορίθμων. Επιπλέον, μετράται η χρονική διάρκεια εκτέλεσης του κάθε αλγορίθμου, καθώς και η μνήμη που χρησιμοποιήθηκε κατά την εκτέλεση (η μέγιστη μνήμη που μετρήθηκε μείον την αρχική μνήμη ακριβώς πριν ξεκινήσει η εκτέλεση). Οι επιδόσεις των αλγορίθμων στα παραπάνω κριτήρια παρουσιάζονται αναλυτικά στην επόμενη ενότητα.

0.6 Πειραματικά Αποτελέσματα

Παρακάτω παρουσιάζονται τα πειραματικά αποτελέσματα των αλγορίθμων που συγκρίθηκαν. Αυτοί είναι: η Αφελής Λύση, η Λύση In-Place, η Λύση Ρίζας, η Λογαριθμική Λύση, η Προσέγγιση Άπειρου Ορίζοντα και η Προσέγγιση Turnpike.

0.6.1 Σύγκριση Ανταμοιβών

Προκειμένου να επιβεβαιωθεί η ακρίβεια των προτεινόμενων λύσεων, έγινε σύγκριση των ανταμοιβών που συλλέγει ο πράκτορας σε πέντε διαφορετικές περιπτώσεις: με έναν οποιονδήποτε αλγόριθμο επίλυσης Πεπερασμένου Ορίζοντα (είναι όλοι ισοδύναμοι), με τρεις Προσεγγίσεις Άπειρου Ορίζοντα με διαφορετικά γ καθώς και με την Προσέγγιση Turnpike. Για κάθε μια εξ αυτών, δοκιμάστηκαν 20 διαφορετικά μοντέλα για 20 διαφορετικές τιμές οριζόντων. Τα αποτελέσματα παρουσιάζονται στην Εικόνα 0.1

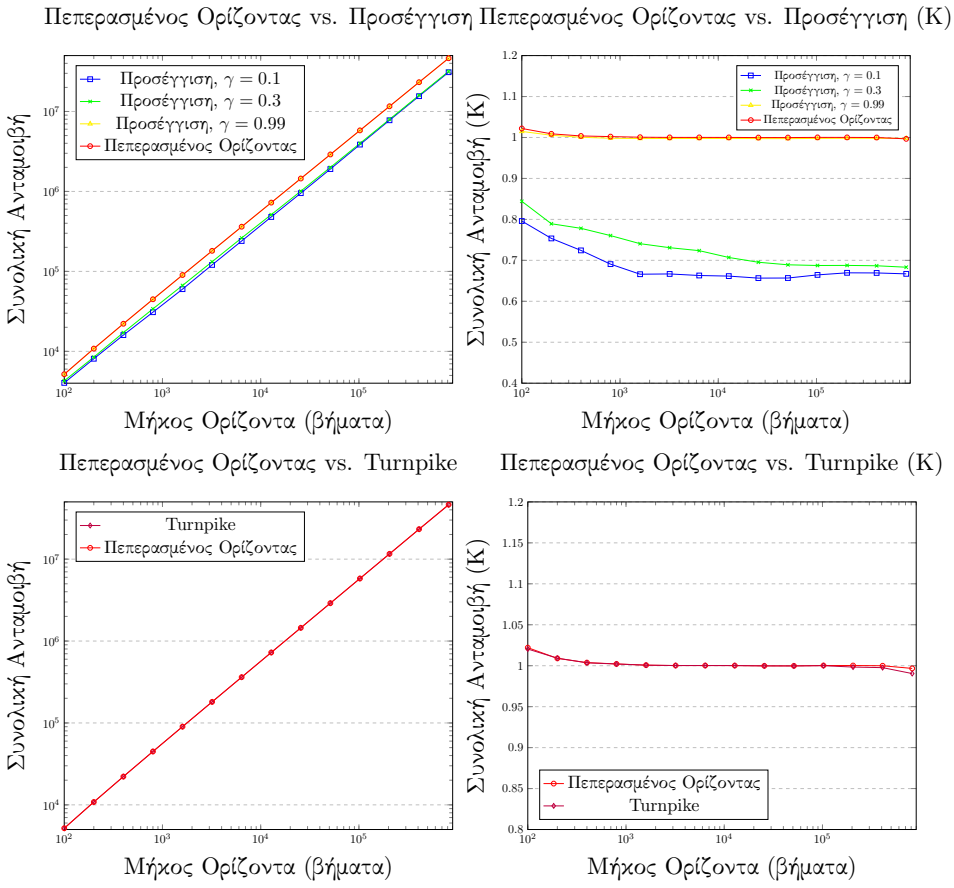


Figure 0.1: Πάνω αριστερά: μέση τιμή της συνολικής ανταμοιβής που συλλέγεται από τον πράκτορα σε 20 διαφορετικά πειράματα ως προς το μέγεθος του ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και τριων Προσεγγίσεων Απείρου Ορίζοντα με μεταβλητό $\gamma = \{0.1, 0.3, 0.99\}$. Πάνω δεξιά: κανονικοποιημένη τιμή συνολικής ανταμοιβής (μέση τιμή συνολικής ανταμοιβής δια την μέση αναμενόμενη ανταμοιβή του Πεπερασμένου Ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και τριων Προσεγγίσεων Απείρου Ορίζοντα με μεταβλητό $\gamma = \{0.1, 0.3, 0.99\}$. Κάτω αριστερά: μέση τιμή της συνολικής ανταμοιβής που συλλέγεται από τον πράκτορα σε 20 διαφορετικά πειράματα ως προς το μέγεθος του ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και της Προσέγγισης Turnpike. Κάτω δεξιά: κανονικοποιημένη τιμή συνολικής ανταμοιβής (μέση τιμή συνολικής ανταμοιβής δια την μέση αναμενόμενη ανταμοιβή του Πεπερασμένου Ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και της Προσέγγισης Turnpike.

Οι κανονικοποιημένες εικόνες δημιουργήθηκαν ώστε να διακρίνεται πιο εύκολα η διαφορά στην ανταμοιβή που συγκεντρώνουν οι αλγόριθμοι, διαιρώντας την μέση

τιμή της συνολικής ανταμοιβής για κάθε τιμή ορίζοντα με την αντίστοιχη αναμενόμενη τιμή που υπολογίζεται από τον αλγόριθμο Πεπερασμένου Ορίζοντα, εφόσον αυτός υπολογίζει την μέγιστη αναμενόμενη ανταμοιβή σύμφωνα με την θεωρία.

Ερμηνεύοντας τα αποτελέσματα, παρατηρούμε ότι οι δυο Προσεγγίσεις με $\gamma = \{0.1, 0.3\}$ υστερούν σημαντικά έναντι του αλγορίθμου Πεπερασμένου Ορίζοντα. Οι καλύτερές τους επιδόσεις επιτυγχάνονται για μικρές τιμές του ορίζοντα, όπως είναι φυσικό, αφού ο πράκτορας γίνεται μυωπικός. Αντίθετα, η Προσέγγιση με παράγοντα έκπτωσης 0.99 συλλέγει ανταμοιβή σχεδόν όση και ο αλγόριθμος Πεπερασμένου Ορίζοντα. Τέλος, το ίδιο ισχύει και για την Προσέγγιση Turnpike, με την καμπύλη της να είναι σχεδόν πανομοιότυπη με αυτην του Πεπερασμένου Ορίζοντα. Μια πιθανή εξήγηση για τις τόσο καλές επιδόσεις των δυο τελευταίων προσεγγίσεων είναι η έλλειψη ποικιλίας δράσεων του μοντέλου, με αποτέλεσμα οι πολιτικές να μην διαφοροποιούνται ιδιαίτερα μεταξύ τους.

0.6.2 Σύγκριση Μεταβλητών Ανταμοιβών

Τα αποτελέσματα αναμένεται να διαφοροποιηθούν σε μοντέλα όπου οι ανταμοιβές είναι μεταβλητές και συναρτήσει της εποχής απόφασης. Προκειμένου να προσομοιώσουμε ένα τέτοιο μοντέλο, θεωρήσαμε ότι σε κάθε εποχή απόφασης n, για μια συγκεκριμένη κατάσταση s και μια συγκεκριμένη πιθανή δράση a μια ανταμοιβή μηδενιζόταν με κυκλικό τρόπο κάθε φορά. Σημειώνεται ότι ο υπολογισμός της βέλτιστης πολιτικής στην περίπτωση των Προσεγγίσεων με μεταβλητό παράγοντα έκπτωσης έγινε θεωρώντας ότι οι ανταμοιβές δεν αλλάζουν (δηλαδή με τις αρχικές τιμές ανταμοιβών), αφού δεν θα γινόταν να επιτευχθή σύγκλιση. Τα αποτελέσματα παρουσιάζονται στην Εικόνα 0.2.

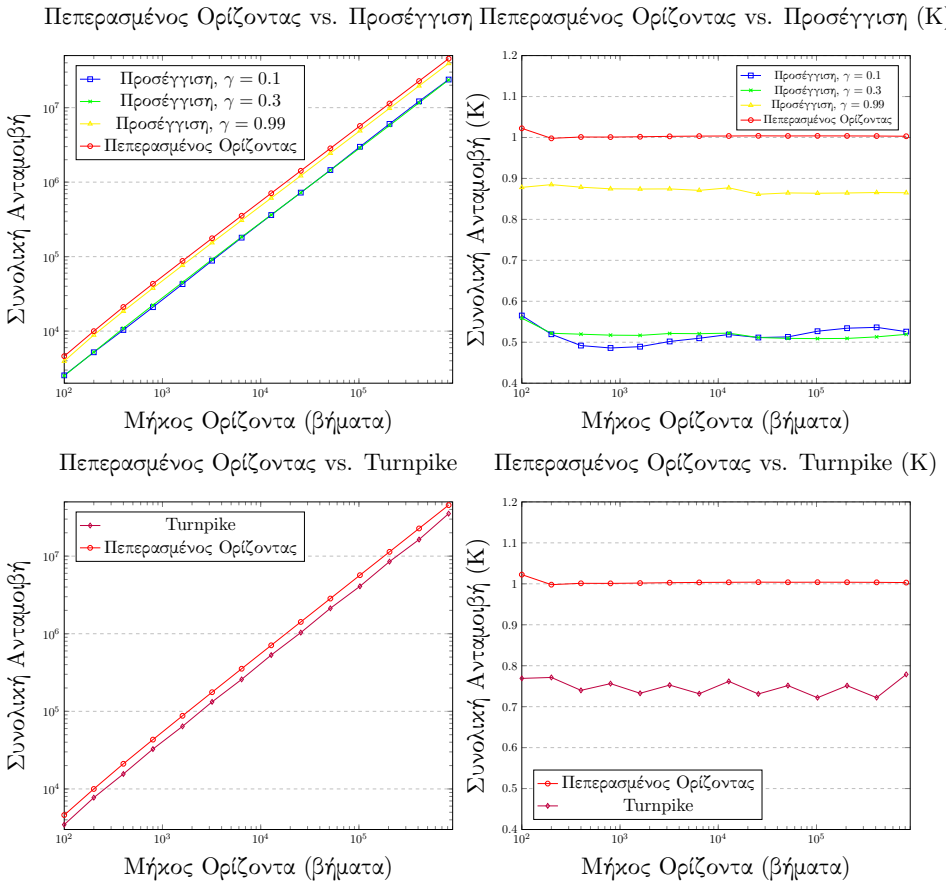


Figure 0.2: Πάνω αριστερά: μέση τιμή της συνολικής ανταμοιβής που συλλέγεται από τον πράκτορα σε 20 διαφορετικά πειράματα ως προς το μέγεθος του ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και τριων Προσεγγίσεων Απείρου Ορίζοντα με μεταβλητό $\gamma = \{0.1, 0.3, 0.99\}$. Πάνω δεξιά: κανονικοποιημένη τιμή συνολικής ανταμοιβής (μέση τιμή συνολικής ανταμοιβής δια την μέση αναμενόμενη ανταμοιβή του Πεπερασμένου Ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και τριων Προσεγγίσεων Απείρου Ορίζοντα με μεταβλητό $\gamma = \{0.1, 0.3, 0.99\}$. Κάτω αριστερά: μέση τιμή της συνολικής ανταμοιβής που συλλέγεται από τον πράκτορα σε 20 διαφορετικά πειράματα ως προς το μέγεθος του ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και της Προσέγγισης Turnpike. Κάτω δεξιά: κανονικοποιημένη τιμή συνολικής ανταμοιβής (μέση τιμή συνολικής ανταμοιβής δια την μέση αναμενόμενη ανταμοιβή του Πεπερασμένου Ορίζοντα, στις περιπτώσεις των αλγορίθμων Πεπερασμένου Ορίζοντα και της Προσέγγισης Turnpike.

Τα διαγράμματα της Εικόνας 0.2 παρουσιάζουν ιδιαίτερο ενδιαφέρον. Όμοια με την περίπτωση των μη μεταβλητών ανταμοιβών, οι Προσεγγίσεις με τις μικρότερες

τιμές του παράγοντα έκπτωσης, δηλαδή 0.1 και 0.3 παρουσιάζουν πολύ χαμηλότερες επιδόσεις συγκριτικά με τον Πεπερασμένο Ορίζοντα. Αντίθετα όμως με προηγουμένως, οι μεταβλητές ανταμοιβές έχουν επίπτωση και στην απόδοση της Προσέγγισης με $\gamma = 0.99$, η οποία φαίνεται μετά βίας να αγγίζει το 90% της αναμενόμενης συνολικής ανταμοιβής. Τέλος, μεγάλη πτώση παρουσιάζει και η απόδοση της Προσέγγισης Turnpike, με την καμπύλη στο κανονικοποιημένο διάγραμμα να είναι ακόμα χαμηλότερη και της Προσέγγισης με $\gamma = 0.99$. Σε κάθε περίπτωση, τα βέλτιστα αποτελέσματα επιτυγχάνονται με χρήση αλγορίθμου Πεπερασμένου Ορίζοντα, γεγονός που τον καθιστά αναντικατάστατο σε περιπτώσεις μεταβλητών ανταμοιβών.

0.6.3 Σύγκριση Χρόνου Εκτέλεσης

Έχοντας εδραιώσει την υπεροχή των αλγορίθμων Πεπερασμένου Ορίζοντα έναντι των Προσεγγίσεων, σημαντική είναι η σύγκριση μεταξύ αυτών, τόσο ώστε να διαπιστωθούν και πειραματικά οι θεωρητικές εκτιμήσεις της χρονικής τους πολυπλοκότητας όσο και να αναδειχθούν τα προτερήματα και αδυναμίες τους. Συγκρίθηκαν ως προς τον χρόνο εκτέλεσης οι αλγόριθμοι Αφελούς Λύσης, Λύσης Ρίζας και Λογαριθμικής Λύσης. Σημειώνεται ότι δεν παρουσιάζονται τα αποτελέσματα της In-Place Λύσης, μιας και ήταν αδύνατο να εκτελεστεί σε ρεαλιστικό χρόνο ακόμη και για σχετικά μικρές τιμές ορίζοντα. Τα αποτελέσματα παρουσιάζονται στην Εικόνα 0.3

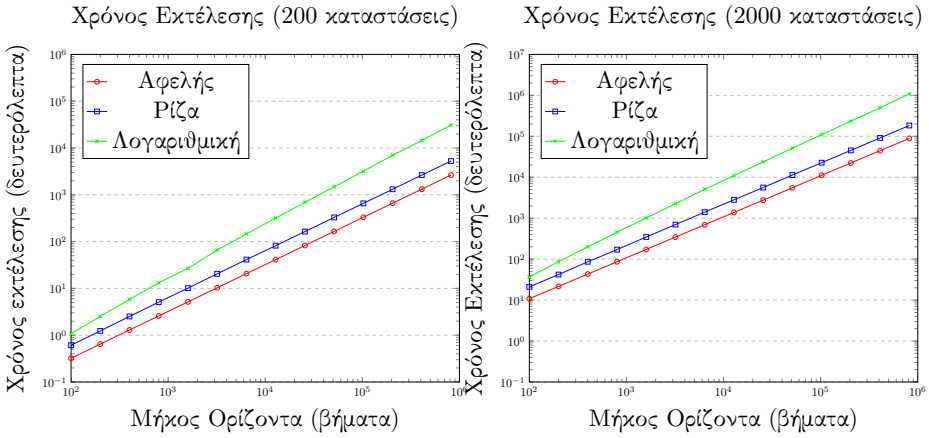


Figure 0.3: Αριστερά: λογαριθμικό γράφημα με τους χρόνους εκτέλεσης των αλγορίθμων Αφέλους Λύσης, Λύσης Ρίζας και Λογαριθμικής Λύσης ως προς το μέγεθος του ορίζοντα σε μοντέλο με 200 καταστάσεις. Δεξιά: λογαριθμικό γράφημα με τους χρόνους εκτέλεσης των αλγορίθμων Αφέλους Λύσης, Λύσης Ρίζας και Λογαριθμικής Λύσης ως προς το μέγεθος του ορίζοντα σε μοντέλο με 2000 καταστάσεις.

Τα αποτελέσματα παρουσιάζονται σε λογαριθμικούς άξονες εξαιτίας της ραγ-

δαίας αύξησης των τιμών. Παρατηρούμε ότι οι ευθείες των αλγορίθμων Αφελούς Λύσης και Λύσης Ρίζας είναι σχεδόν παράλληλες, με αυτήν της Ρίζας να είναι ελαφρώς ψηλότερα, γεγονός το οποίο υποδεικνύει ότι ασυμπτωτικά οι χρονικές πολυπλοκότητες των δυο είνα ίσες, με αυτήν της Ρίζας να είναι μεγαλύτερη κατά μια σταθερά. Η Λογαριθμική Λύση παρατηρείται πως λαμβάνει μεγαλύτερες τιμές σε σχέση με τις άλλες δυο καμπύλες για κάθε τιμή του ορίζοντα, ενώ φαίνεται να έχει μεγαλύτερη κλίση, γεγονός που ερμηνεύεται αν αναλογιστούμε τον λογαριθμικό παράγοντα στην εξίσωση πολυπλοκότητας. Συνολικά, η πειραματική μελέτη ερμηνεύει πλήρως τα θεωρητικά αποτελέσματα της χρονικής πολυπλοκότητας.

0.6.4 Σύγκριση Απαιτήσεων Μνήμης

Το σημαντικότερο κριτήριο για την σύγκριση των παραπάνω αλγόριθμων αποτελεί η μνήμη που απαιτείται από τον καθένα κατά την εκτέλεση. Οι αλγόριθμοι που συγκρίνονται είναι και πάλι η Αφελής Λύση, η Λύση Ρίζας και η Λογαριθμική Λύση. Τα αποτελέσματα που παρουσιάζονται στην Εικόνα 0.4 αφορούν την διαφορά μέγιστης μνήμης που μετρήθηκε κατά την εκτέλεση και αρχικής μνήμης που μετρήθηκε αμέσως μετά την εκπαίδευση.

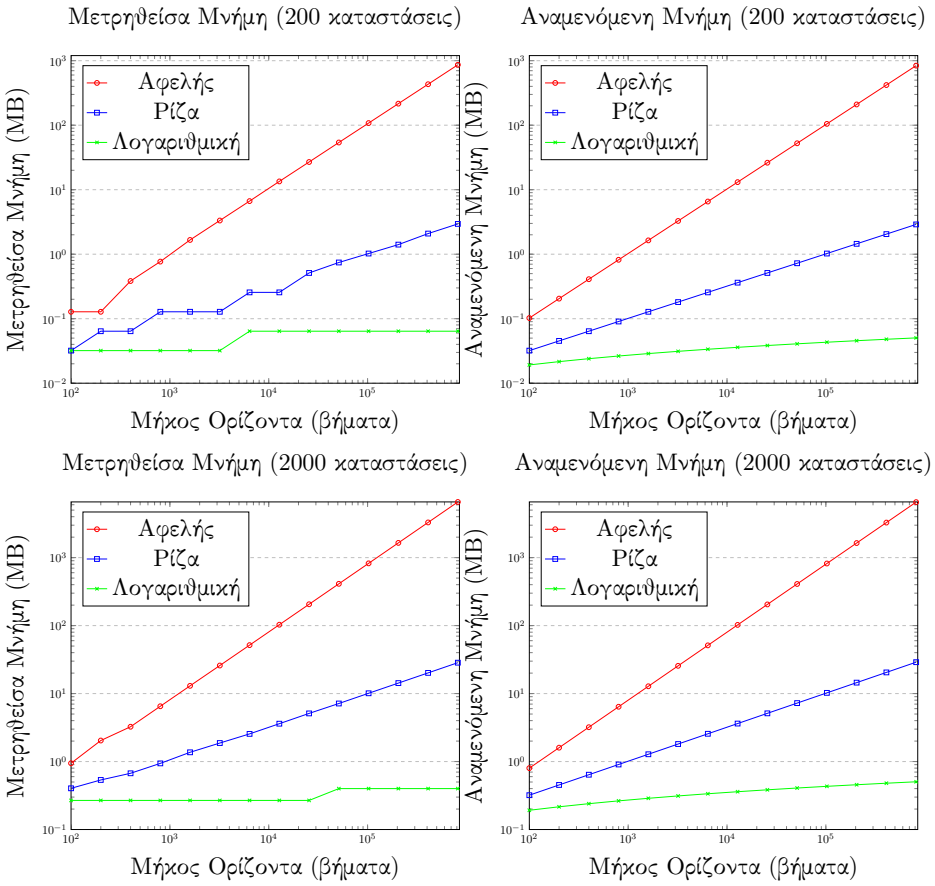


Figure 0.4: Πάνω Αριστερά: Μετρηθείσα διαφορά μέγιστης με αρχική μνήμη σε Megabytes για μοντέλο με 200 καταστάσεις. Πάνω Δεξιά: Εκτιμώμενη διαφορά μέγιστης με αρχική μνήμη σε Megabytes για μοντέλο με 200 καταστάσεις. Κάτω Αριστερά: Μετρηθείσα διαφορά μέγιστης με αρχική μνήμη σε Megabytes για μοντέλο με 2000 καταστάσεις. Κάτω Δεξιά: Εκτιμώμενη διαφορά μέγιστης με αρχική μνήμη σε Megabytes για μοντέλο με 2000 καταστάσεις.

Παρατηρείται ότι τα πειραματικά αποτελέσματα συμφωνούν σε μεγάλο βαθμό με τα εκτιμώμενα. Σημειώνεται ότι για εκτίμηση της μνήμης υπολογίσαμε τον μέγιστο αριθμό πίνακων που είναι αποθηκευμένοι στη μνήμη για κάθε αλγόριθμο και κάθε τιμή ορίζοντα και πολλαπλασιάσαμε αυτόν τον αριθμό επί το μέγεθος του τύπου δεδομένων του (floating point) ώστε να προκύψει η αναμενόμενη τιμή. Όσες διαφωνίες υπάρχουν συναντώνται σε σημεία που η καμπύλη των πραγματικών μετρήσεων είναι σταθερή. Αυτό συμβαίνει διότι η μνήμη ανατίθεται με τη μορφή σελίδων, επομένως είναι πιθανό να ανατεθεί περισσότερη μνήμη από ό,τι είναι απαραίτητο, με αποτέλεσμα η αύξηση του ορίζοντα να μην επιφέρει ανάθεση περισσότερης, αφού αρκεί η ήδη υπάρχουσα. Η Λογαριθμική Λύση φαίνεται να έχει

την καλύτερη επίδοση με διαφορά σε σχέση με τις άλλες δυο, ενώ η Λύση Ρίζας συμπεριφέρεται αρκετά βελτιστότερα σε σχέση με την Αφελή Λύση όσο μεγαλώνει ο ορίζοντας, όπως αναμέναμε από την θεωρία.

0.7 Επίλογος

0.7.1 Συμπεράσματα

Μέσα από αυτήν την εργασία επιδιώξαμε να βελτιστοποιήσουμε αλγορίθμους επίλυσης MDP τόσο ως προς τον χρόνο εκτέλεσης όσο και προς τις ανάγκες σε μνήμη. Συνοψίζοντας:

- Ο αλγόριθμος Value Iteration παρουσιάζει σημαντικά προβλήματα χρόνου εκτέλεσης. Η ιδέα που προτείναμε σε αυτήν την εργασία αφορούσε φράξη της συνάρτησης αξίας-δράσης με σκοπό την απόρριψη δράσεων που εγγυημένα δεν οδηγούσαν σε βέλτιστη λύση. Εντούτοις, τα αποτελέσματα δεν ήταν τα αναμενόμενα, ίσως εξαιτίας της επιλεγμένη εφαρμογής. Παρόλα αυτά, υπάρχει περιθώριο βελτίωσης της μεθόδου σε μελλοντική εργασία με υποσχόμενα αποτελέσματα.

- Οι κλασσικοί αλγόριθμοι επίλυσης MDP Πεπερασμένου Ορίζοντα χωλαίνουν όσον αφορά την χωρική πολυπλοκότητα. Για αυτόν τον λόγο, προτείνουμε δυο νέους αλγόριθμους με σημαντικά μειωμένες χωρικές απαιτήσεις και μηδαμινή χρονική επιβάρυνση για επανυπολογισμούς πινάκων συνάρτησης αξίας.

- Τέλος, εξετάσαμε και προσεγγιστικούς αλγορίθμους επίλυσης MDP Πεπερασμένου Ορίζοντα, με εξαιρετικές επιδόσεις σε μη μεταβλητά μοντέλα. Ωστόσο, τα πράγματα ήταν διαφορετικά σε μοντέλα με μεταβλητές ανταμοιβές, καθιστώντας έντονη την ανάγκη ύπαρξης κλασσικών λύσεων.

0.7.2 Μελλοντική Εργασία

Οι βελτιστοποιήσεις που παρουσιάστηκαν αποτελούν ισχυρές βάσεις για μελλοντικές επεκτάσεις και βελτιώσεις. Συγκεκριμένα, η μέθοδος φράξης για την περίπτωση του Value Iteration θα μπορούσε να δοκιμαστεί σε περισσότερες, πιο κατάλληλες εφαρμογές εφόσον τα θεωρητικά της θεμέλια είναι στέρεα. Τα άνω και κάτω φράγματα θα μπορούσαν να βελτιωθούν, ώστε τα περιθώρια να είναι ακόμη μικρότερα.

Αντλώντας έμπνευση από την ιδέα των φραγμάτων, θα μπορούσε κανείς να υπολογίσει άνω και κάτω φράγματα για την συνάρτηση δράσης-αξίας χωρίς να υπολογίσει την ίδια την συνάρτηση για κάθε κατάσταση. Κάθε επανάληψη του αλγορίθμου θα έφερνε τα φράγματα πιο κοντά το ένα στο άλλο, μέχρι που τελικά θα ισούνταν με την πραγματική τιμή της συνάρτησης δράσης-αξίας, απορρίπτοντας δράσεις στην πορεία.

Αναφορικά με τον Πεπερασμένο Ορίζοντα, το βασικό πρόβλημα των αλγορίθμων που προτάθηκαν είναι η χρονική επιβάρυνση που εισάγεται λόγω

επανυπολογίσμών πινάκων συνάρτησης αξίας. Εφόσον κατά την πρώτη εκτέλεση του αλγορίθμου υπολογίζονται όλοι οι πίνακες συνάρτησης αξίας για κάθε βήμα, θα μπορούσε κανείς να αξιοποιήσει αυτήν την πληροφορία ώστε να επιταχύνει τους επανυπολογισμούς. Έτσι, ο χρόνος εκτέλεσης της Λογαριθμικής Λύσης ίσως πλησίαζε αυτούς της Αφελούς ή της Λύσης Ρίζας, καθιστώντας την την πιο αποδοτική μέθοδο επίλυσης MDP Πεπερασμένου Ορίζοντα.

1 Introduction

In this section, we aim to establish the challenges we are faced with when utilizing the Markov Decision Process framework in use cases such as Reinforcement Learning, while suggesting methods to overcome them. We also present practical applications of the framework and related work that has been executed towards similar optimizations.

1.1 Motivation

Over the last decade, Reinforcement Learning has rapidly gained lots of popularity, finding use in a variety of modern applications [Li19]. This machine learning technique can be used to model real-world problems, with or without a large dataset, as it can also learn from experience. A crucial requirement, however, is that the world modeled must follow the Markov property, further discussed in Section 4, to be represented as a Markov Decision Process.

A Markov Decision Process (to which we will be referring to as a MDP throughout this thesis) is a statistical model, defined by states and actions, which are made by an agent. Every action yields a reward, which can be positive or negative towards the agent. Moreover, an action induces a reaction of the system, which transitions to a new state according to a probability.

Depending on the number of decision the agent is required to make, more specifically whether this number is finite or infinite, the MDP can be characterized as having a Finite Horizon (FHMDP) or Infinite Horizon (IFMDP). In the first case, the agent knows beforehand exactly how many decisions they are required to make and that their decision making process will terminate once they achieve that number. On the other hand, in an IFMDP, the process' termination occurs when the agent reaches a special state known as a terminal state (defined beforehand) or when an outer source instructs them to terminate. Regardless of the size of the horizon, the agent's goal throughout this process is maximizing the total reward they will receive through their actions.

As the use cases of the MDP framework become more demanding and complex, so does the model, resulting in huge state spaces with large numbers of possible actions. As a result, both the memory requirements and the execution time of MDP solving algorithms grow exponentially, often rendering the algorithm practically impossible to execute. In addition, it is possible for many modern applications using the MDP framework to be running in devices not

able to fulfill those demands, such as smartphones and tablets, making the need for optimization of such algorithms more relevant than ever.

This master thesis delves into methods of optimizing FHMDP solving algorithms with regards to the memory requirements, providing support to an issue which cannot otherwise be dealt with, as having memory constraints is usually solved by acquiring extra memory which is often costly.

1.2 Applications

Most modern day applications of MDPs involve Infinite-Horizon MDPs, especially considering the big uprising of Reinforcement Learning, which mostly operates on infinite horizons. However, there are still many use cases for Finite-Horizon models, mainly in resource management, treatment planning and routing problems.

An important application of FHMDP in resource management, which also highlights the importance of the findings of this thesis, is described in [COM+09]. This work suggests a software for optimizing phone call times in mobile phones, by deciding the timing to perform background actions, such as email synchronization. In particular, it is assumed that one of the most important functions of a mobile phone is voice calls, and the ability to perform them is crucial. As a result, the amount of battery remaining should be sufficient enough to perform average voice calls. Every state of the model consists of the current time and battery remaining. An action would be synchronizing emails (in the background) or not, or in a second case, whether to turn on the WiFi radio. The transition probabilities come from the stochastic nature of the duration and occurrence of the phone calls, which could be obtained after user modeling. The finite horizon of the model is the time (partitioned into multiples of a time unit) between two battery recharges. The MDP is then used to calculate an optimal policy array, of size $N|S|$, where $|S|$ is proportional to the remaining battery and time units since the time synchronization (or WiFi activation) was last performed. The agent can then use this stored array to make decisions at every time step. An important issue mentioned in this work was the size of the optimal policy array, as it is proportional to both the horizon and the number of states, while a mobile phone's RAM remains relatively smaller than that of a personal computer. Although the researchers managed to decrease the number of states dramatically (which may also result in a reduction to the model's granularity), we believe that the methods suggested in Section 4 could optimize the performance even further.

Finite-Horizon MDPs are greatly utilized in medicine as well. This framework can be used for scheduling the way and order patients will utilize a medical device in a hospital, as is discussed in [GBGG11]. In this example specifically, there exists one (or more) computed tomography scanner(s) and patients wanting to use them are split into three categories, emergency patients (critical and non-critical), inpatients and outpatients. Choosing which patient will use the machine(s) out of those in the pool of waiting patients can be crucial towards maximizing the revenue of the hospital (which also depends on the waiting time

of each of the patients). The states of the model consist of the number of patients waiting from each category and whether a critical emergency patient is waiting, to whom the highest priority is given. The possible actions refer to which patient will be scheduled immediately (namely to which category they belong). The transitions occur when new patients arrive in between uses of the computed tomography scanner, while the reward is a function of the total cost of the use of the machine, the cost of patients waiting and the revenue the hospital makes from the particular patient using the machine. Finally, the horizon refers to the number of time slots of machine use in a day (each use of the machine is considered to require a constant amount of time). Results showed that this method could, in most cases, maximize the profits when compared to other traditional methods.

Another application of FHMDP in medicine involves cancer treatment planning. In [BL20], researchers used an FHMDP model combined with a preexisting optimization model, to make gastric and gastroesophageal cancer treatment planning more efficient. Such a planning is considered efficient when it minimizes the levels of toxicity the patient suffers and maximizing their survival time. The states of the model represented different levels of toxicity while the actions refer to the different regimens the patient could receive in the next decision epoch. Transition probabilities come mostly from other clinical trials and statistical analyses, while the reward function involves the toxicity levels as well as the expected survival time. The model's horizon refers to the number of cycles of the treatment. This method resulted in higher survival times and lower levels of toxicity, and is deemed particularly useful in sequential treatment decision making in clinical trials. Nevertheless, this model is limited by the data set used for its parameters.

Finite-Horizon Markov Decision Processes can find great use in Game Theory, specifically in Routing Games. In a Routing Game, there exists a population in a graph of nodes and edges. Every individual of the population acts as an agent, which needs to decide the optimal policy to traverse the graph in order to maximize a reward it will receive (or minimize a cost). An example of a Routing Game used in [CS17] is a problem drivers of car sharing services face: which path to follow in order to maximize their earnings while minimizing the cost (waiting time when competing with other drivers, fuel costs etc.). The researchers in this work formulated this problem as an FHMDP Routing Game, where the agents must find the optimal policy that they will follow, while achieving Finite-Horizon Wardrop Equilibrium. While this term deviates from the point of this work and is thoroughly explained in [CS17], we will briefly mention that, in essence, such an equilibrium is achieved when there exists a distribution of the population among the nodes such that for every node and time step, the action chosen by the percentage of the population in that node is the most optimal action possible (the one with the greatest value function). Every agent in this example solves a FHMDP, with every node being a state, every edge denotes a transition probability (could also be deterministic) and the reward function is a summation of all the costs and profits mentioned before. Results occurring when this logic was run in a simulation appeared to be very

promising.

FHMDP approaches have also been proposed as a means of planning airline meal preparations and adjustments. In [GLP04] an airline meal provisioning plan was formulated as a FHMDP in order to minimize the airline's losses. Particularly, meals in a flight are prepared in advance, taking into account the expected amount of passengers, as well as any special requests. Then, adjustments are made and the meals are shipped to the airport to be loaded before the flight. An excess number of meals can be costly to the firm, while a meal shortage can lead to customers being lost, which although difficult to quantify, is used as a metric in this paper. The horizon of the problem consists of only a small number of epochs, which represent points in time prior to departure. It should be noted that, the later an adjustment is made, the more costly it is toward the airline. The model's state take into account the number of meals made/expected for this flight as well as the expected number of passengers. The actions possible in each state represent increases or decreases of the number of meals available. After a certain decision epoch (known as the adjustment period), this number is limited by the meal transport vehicle's capacity. The rewards in this case are negative and include all the costs mentioned above, as well as meal transportation costs, which increase the later the adjustments are made (horizon-dependent rewards, also known as non-stationary). The transitions are stochastic, as they happen because of changes in the expected number of passengers (such as cancellations). The results show that this model could find great use in meal provision scheduling of long and medium duration flights, with the short duration flights yielding worse results. The granularity of states is a serious issue of the model, as the number of states is proportional to the square of the plane's capacity (as it is the upper limit of the number of passengers and number of meals). Although the horizon is small, the number of states is large (before any alterations to lower it, such as aggregation of seats). This method can also be run simultaneously for a large number of flights, resulting in huge needs in execution time and space, showing that the theoretical results in Section 4 could be of value.

1.3 Related Work

While methods to speed up Value Iteration have been thoroughly examined, optimizations on the classic Finite-Horizon algorithms are limited to practical methods and heuristics, such as state space reduction, as applicable.

1.3.1 Time Complexity Optimizations

Value Iteration algorithm (described in depth in Section 2.5) is the most widely used solution for Infinite-Horizon MDP problems, especially in Reinforcement Learning. Nevertheless, its time complexity often proves prohibiting for large MDPs, with huge numbers of states and actions. In Section 3.1 we propose an optimization using value function bounds in order to eliminate suboptimal actions and thus reduce the execution time needed. A plethora of attempts has

been made towards the same goal, using a bounding approach, all regarding the calculation of accurate bounds.

The first time bounding the value function in order to eliminate suboptimal actions was suggested was in 1967 by J. MacQueen [Mac67]. There, the upper and lower bounds of the value function were defined as sequences which monotonically converge to their correct values after a number of iterations. Later, further improvements on this approach were introduced, such as those in [SB79], applicable in both Finite- and Infinite-Horizon cases, including a new test for suboptimal actions, which may be reintroduced at later stages of the iteration process. Other approaches include calculating bounds in order to compute a policy that is approximately optimal, as seen in [Whi82].

Another important optimization on the running time of a MDP solving algorithm involves the concept of aggregation. This idea refers to grouping together states of the MDP in order to reduce the state space and thus the execution time, as most MDPs bear the so called "curse of dimensionality" [Bel15]. One of the earliest important works in state aggregation is that described in [RK02], discussing a way to partition the MDP into subsets of states, aggregate states inside those partitions and perform the known MDP-solving algorithms on the newly-created Aggregated MDP. It was proven, however, that while this method achieved a suboptimal policy (with a certain error tolerance) the time required to perform the partitioning along with the solution was at least as much as that needed to solve the original MDP.

Many methods of state aggregation have been suggested. The authors of [LWL06] present the unified results of those methods, concluding that there exists a constant trade-off between retaining information about the model and reducing the dimensionality of it. In essence, the greater the granularity of the model, the more accurate calculations can be performed and thus the more optimal the resulting policy, while a coarser model with fewer, aggregated states could be solved more efficiently at the cost of accuracy.

The two optimizations of bounding and state aggregation have been used in combination with each other in the algorithms suggested in [DWG$^+$11]. The first algorithm, known as Topological Value Iteration (TVI) performs Value Iteration on the states in an order such that convergence is achieved as fast as possible. This however requires that the graph representing the MDP does not contain cycles. To achieve this, state aggregation is performed on groups of states in which cycles exist and the MDP is transformed to a new aggregated MDP without cycles. Nevertheless, many MDPs contain large cycles resulting in a MDP with few number of aggregated states, containing lots of grouped states. To cope with this problem, the authors suggest performing an action elimination based on upper and lower bounds for each state-action pair. This approach eliminates edges of the graph and hopefully lowers the number of cycles. After the action elimination process, TVI is executed as normal after the states have been grouped appropriately.

Another idea similar to that of aggregation is the idea of partitioning, in other words dividing the state space into groups of states by some criterion. An exceptional use of this concept is made in [WS03], where the authors describe

partitioning the state space of the MDP, and, using a concept known as potential information flow (meaning how drastic the change of the value function is in the next iteration for some state-action pair) update only the states of a single partition. Afterwards, the same process is repeated for the partition with the next largest potential information flow, until the greatest potential information flow of all partitions is below a threshold. This procedure was shown to achieve great computational acceleration, even for MDPs with large state spaces.

Optimizations of time complexity have also been made with regards to Finite-Horizon problems. One important heuristic, known as Temporal Concatenation, is described in [SX20]. This procedure suggests splitting the problem into two (or more) subproblems whose horizons add up to the horizon of the original problem. Those subproblems with smaller horizons could be solved in parallel and, after obtaining their respective policies, concatenate them to create a unified policy for the original problem. This solution of course is not guaranteed to be the optimal solution, but researches managed to find bounds of the difference between the expected value of the optimal policy and the expected value of the concatenated policy (this difference is also known as regret). The acceleration of computations was also important, as the problem was split in two subproblems able to be solved in parallel, with the total execution time being almost half the original (but not quite due to added overhead)

1.3.2 Space Complexity Optimizations

As already mentioned, the only attempts made towards solving the space complexity issue of Finite-Horizon algorithms regard heuristics and case-dependent optimizations. It has also been observed that developers often opt for other solutions, such as linear programming, when applicable. The previous work we shall present, however, regards optimization tactics falling under the first case. It should be mentioned that, most memory optimization techniques regard the model's memory footprint rather than the non-stationary policy stored, unlike our suggestions.

In [GLP04], for example, the authors mention that the state space grows rapidly with the number of passenger seats. Each state of the model represents a combination of seats and meals prepared, with both numbers being at most equal to the maximum seat capacity of an airplane. This results in quadratic increase of the number of states with respect to the number of seats. Even for a small aircraft of only 108 seats, the amount of time needed for the calculation is huge, while the total space required adds up the more models running simultaneously. In order to cope with this issue the authors used state aggregation, meaning their experiments were performed on a model in which the "seats" were bins containing a number of actual plane seats (this number was referred to as bin size).

In [DMW08], researchers suggested a memory optimization on the Value Iteration algorithm. In particular, this optimization is based on the External Memory Value Iteration algorithm presented in [EJB07]. This algorithm represents the MDP model as a graph and operates on its edges, which show how

states are connected through the transition function. While an external memory, such as a hard disk, offers much larger capacity than a RAM, the input-output operations are much slower. To compensate for that, the edges of the MDP graph stored in memory are sorted before each iteration in order to speed up the process. This approach manages to utilize external memory, at the cost of execution time. It should also be noted that if a large portion of memory is available the algorithm cannot utilize it to make input-output operations more efficient. It was shown that this algorithm requires a much larger time to execute compared to Value Iteration.

An improvement of this model was suggested in [DMW08] through an algorithm named Partitioned External Memory Value Iteration. This algorithm partitions the state space into blocks and then loads those from main memory, making I/O operations more efficient. When compared to problems too big for Value Iteration's internal memory, its execution time was faster than that of the EMVI algorithm by an order of magnitude. Both the above mentioned methods are applied on Infinite-Horizon problems, and do not reduce the space complexity of the algorithm, but rather utilize external memory such as hard disks to store larger models. While those partitions described in [DMW08] are meant to be computed manually, the optimizations of [DW$^+$09] include automatic partionining, leading to better overall partitions than those created by human.

1.4 Objective

While solving FHMDPs appears to be efficient in theory, practice has shown otherwise, bearing in mind the challenges mentions in subsection 1.1. This thesis includes suggestions on memory complexity optimization of the MDP solving algorithm in the Finite Horizon case.

In section 4.2 the readers are introduced to two new methods of solving FHMDPs, which are based on the preexisting algorithm known as Value Iteration, analyzed thoroughly in section 2.5. A developer opting for a FHMDP can decide between the preexisting algorithms, as well as the newly introduced ones, depending on their needs, rendering the FHMDP algorithms more flexible and versatile.

2 Definitions and Preliminaries

This section will be covering the theoretical material deemed necessary for better understanding of the ideas introduced in Section 4. These include Markov Decision Process fundamentals, as well as the algorithms used to solve them. In addition, the concepts of time and space complexity are briefly discussed, as they represent the theoretical basis on which the comparisons between methods will be executed. Finally, we will also be referring to Machine Learning concepts, emphasizing on Reinforcement Learning, a framework utilized in great extent by the experiments that were carried out.

2.1 Computational Complexity Theory

There exist problems which are considered decidable, in other words computationally solvable in theory. Realistically, however, the algorithm solving these problems could have huge needs of execution time and memory, thus rendering the problem practically unsolvable. In order to judge whether a solution to such problems are acceptable, we define the metrics of time and space complexity.

Formally defining the concepts of time and space complexity requires having defined the Turing Machine. The Turing Machine was invented by Alan Turing in 1936 and according to himself it is a mathematical model of computation that defines an abstract machine [Min67]. This machine possesses an infinite tape, consisting of cells, from which it can read symbols, as well as write symbols on them. The part of the machine performing these actions is a moving head, which can go both left and right on the tape. There are also special states for rejecting or accepting an input, which automatically terminate the processes once the moving head lands on them. A Turing Machine has the same power as any computer, meaning there exist problems that even it cannot find a solution to.

Such a machine could be either deterministic or non-deterministic. The first case suggests that after every computation the transition of the machine's head is deterministic and known, while in a non-deterministic Turing Machine this transition could result in any state from a certain group, with some probabilities. Nevertheless, it can be proven that any non-deterministic Turing Machine has an equivalent deterministic one [Sip13].

2.1.1 Time Complexity

According to M. Sipser's definition of Time Complexity: "Let M be a deterministic Turing Machine that halts on all inputs. The running time or time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing Machine. Customarily we use n to represent the length of the input."

Calculating the exact running time of algorithm can get very complex and so a metric of estimation is preferred. This form of estimation is known as asymptotic analysis and is mainly focused on the importance of the highest order term of $f(n)$. As a result, we disregard every lower order term, as well as the coefficient of the highest order term, because they affect the Time Complexity far less than the higher order term for large length inputs.

Again, this can be formally expressed using M. Sipser's definition of asymptotic upper bounds: "Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = \mathcal{O}((g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$:

$$f(n) \leq cg(n)$$

When $f(n) = \mathcal{O}((g(n))$ we say that $g(n)$ is an upper bound for $f(n)$, or more precisely, that $g(n)$ is an asymptotic upper bound for $f(n)$, to emphasize that

we are suppressing constant factors." [Sip13].

In a realistic algorithm's case, we estimate the time complexity of an algorithm by counting the number of elementary actions performed by it with regards to the length of the input, treating the time complexity of every elementary action as constant

2.1.2 Space Complexity

Similarly, we use M. Sipser's definition of Space Complexity: "Let M be a deterministic Turing Machine that halts on all inputs. The space complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$" [Sip13].

In other words, the space complexity of an algorithm is the amount of memory it occupies during its execution with respect to the input length. The asymptotic notation described in the above subsection is also used in space complexity in worst-case analysis.

2.2 Markov Decision Processes (MDP)

A Markov Decision Process is used to simulate a problem where a decision making agent interacts with its environment, whose behavior is probabilistic. Every decision the agent makes results in a different behavior of the environment. The agent's goal is to achieve the optimal sequence of behaviors of the environment with regard to to some defined performance metric.

Every decision the agent makes is taking place in a point of time, commonly referred to as a decision epoch. Those points in time could be either discrete or continuous. Our main focus will be on discrete decision epochs, however we briefly mention that if the decision points are continuous, the agent can make decisions continuously (in other words at every decision epoch), or it can make decisions at random points when a specific event occurs or the agent itself could choose when its decision will be made.

The elements of the set of discrete decision epochs are known as stages or periods. These elements could either refer to a single point in time or a time interval. The number of elements in this set could be finite, for example containing $H = \{1, 2, ..., N\}$, where each number refers to the step index of the discrete time element, or it could also be infinite, namely $H = \{1, 2, ...\}$. The first case renders the problem a finite-horizon problem, whereas the second case consists an infinite-horizon problem.

At each decision epoch the agent finds itself in a particular state of the environment. The set of possible states will be denoted as S. At every state of S there exist a finite number of possible actions the agent can make. The set of all possible actions the agent can make from every state will be denoted as A, while the set of possible actions from a particular state s will be written as $A(s)$. By definition, $A = \bigcup_{s \in S} A(s)$.

Choosing an action a from a state s brings certain consequences upon the agent. Firstly, the agent receives a reward r, namely a scalar value which is produced by the real-value function $R(s, a)$. This reward could be either positive or negative.

After the reward is collected, the action the agent has chosen leads it to a different state j accessible by state s and the decision making process begins again. This transition is random and every accessible state from state s is associated with a probability $p(j|s, a)$, namely the probability that the agent will be transferred to state j after choosing action a in state s. We define T to be a function known as the transition probability function with $T(s, a, j) = p(j|s, a)$. It follows that for a particular state s_j:

$$\sum_{\substack{s_i \in S \\ a \in A(s_j)}} p(s_i|s_j, a) = 1$$

We define the above transition probability function as memory-less, meaning that the next state the agent finds itself into depends only on the current state and the current decision made and is completely independent from past states or decisions. This property is known as the Markov property.

Having defined the above objects, a Markov Decision Process is defined as the collection of objects $\{H, S, A, T, R\}$. Every function in this collection is also a function of time step t (decision epoch) [Put94].

2.3 Solving the MDP

Solving a MDP is usually interpreted as finding the best possible action the agent can make in each state and then having them make decisions based on these actions in order to achieve the best possible score with regards to a certain metric.

2.3.1 Rewards and the Discount Factor

In a Markov Decision Process, the agent's goal is maximizing the total reward they will receive, which is the cumulative reward they receive after every interaction with the environment for a particular number of time steps (decision epochs) or until they reach a terminal state. A terminal state is defined as a state in which the agent cannot execute any more actions, cannot receive any more rewards and cannot transition to any other state. If we write the reward the agent receives after the decision epoch t as R_t, then the expected cumulative reward the agent is to receive after decision epoch t can be expressed as:

$$G_t = \sum_{i=t+1}^{T} R_i$$

, where T means the final decision epoch.

An observation that can be made with regards to the above sum is that it is only valid in case where the decision problem can be broken down to subsequences (called episodes), the number of which is finite. Each of these episodes terminates once the agent reaches a terminal state. Those type of problems are known as episodic problems.

Some problems, however, cannot be viewed as compositions of discrete episodes, such as continuous control problems [LHP+19]. In these types of problems, which are known as continuous problems, we assume that the agent will be making decisions potentially indefinitely, rendering the above sum divergent (in some cases). If, for example, every reward the agent can receive in such a problem is a positive integer, the partial sums will keep increasing with the addition of every new term, while the upper limit of the summation will tend towards infinite, for the number of decision epochs is infinite.

A solution to the above problem is discounting the rewards the agent will receive in the future, by introducing a factor γ, known as the discount factor, to the above infinite sum. Using the discount factor, the expected, now discounted, cumulative reward of the agent becomes:

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{i+t+1}$$

. In this case, it can be proven that the series will definitely converge for $0 \leq \gamma \leq 1$ [Ber00].

The value of γ affects how important future rewards are to the agent. In particular, when $\gamma = 0$, the agent becomes myopic, meaning they focus more on maximizing only the immediate reward, which they will receive after executing their next action. This fact can also be mathematically validated, since every term in the expected discounted cumulative reward becomes 0, except the first term:

$$G_t = R_{t+1}$$

. On the other hand, the more γ approaches 1, the more value is given to future rewards. The expected cumulative reward of a Finite-Horizon MDP can be viewed as an expected discounted cumulative reward with a discount factor of 1.

2.3.2 Policies and Value Functions

In the previous subsection, we formally defined the agent's goal during their decision making process. The most logical continuation would be to define the method by which the agent can discriminate between correct and incorrect decisions leading them towards achieving their goal. In order to assist the agent in their decision making process, we assign a value to every state, indicating how beneficial or harmful it is for the agent to find themselves in this state. Having this information available, the agent can choose the action which will lead them to the optimal next state, the one with the largest value.

An appropriate function to express the value of every state is the expected (discounted or not) cumulative of this state, in other words the cumulative reward the agent is expected to collect starting from this state and performing a number of steps thereafter (finite or infinite). The decisions the agent will make are determined by a concept known as a policy, namely a mapping between the states and a probability that the agent will choose a certain (possible) action in that state, for every action of that state [SB18]. We write the probability that action a will be chosen if the agent find themselves in state s as $\pi(a|s)$.

Using the definition of a policy, we can also define the value function of every state with regards to a policy π as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{i+t+1} \middle| S_t = s\right]$$

which refers to the expected value of the total discounted reward the agent will receive starting from state s and following policy π thereafter. This function is known as the state-value function.

Another useful definition is that of the action-value function for a particular given policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{i+t+1} \middle| S_t = s, A_t = a\right]$$

The action-value function of a state s is the expected value of the total discounted reward the agent will collect starting from state s, choosing action a and following policy π thereafter.

It can be observed that the value function is particularly useful towards calculating the agent's optimal course of action. This is because such a function acts as a way of comparing policies. The agent is certain a policy π is better than another policy π' if and only if the value function of π is greater than the value function of π' for every possible state. Formally, this can be express as: $V_\pi(s) \geq V_{\pi'}(s)$ for every $s \in S$. The optimal policy of a MDP is the policy whose value function is greater than or equal to the value function of every other policy at every possible state, or equivalently if it is better than every other policy. It follows that a particular MDP m can have more than one optimal policies, which will of course have equal value functions at every possible state. The value function of all of those optimal policies is known as the optimal state-value function at every state s and can be express as:

$$V^*(s) = \max_{\pi \in P(m)} V_\pi(s)$$

, where $P(m)$ means the set containing every possible policy of this particular MDP m. Those optimal policies also have equal action-value functions at every possible state, the optimal action-value function:

$$Q^*(s,a) = \max_{\pi \in P(m)} Q_\pi(s,a)$$

2.3.3 Bellman Equations and Optimal Policies

An especially useful attribute derived from the above equations is that the value function satisfies a recursive relationship. More specifically, it holds that, for a random policy π and a state s of the model:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)(r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'])$$

$$= \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)(r + \gamma V_\pi(s')) \quad (1)$$

As already established, $\pi(a|s)$ represents the probability that the agent will choose action a if they find themselves in state s, in other words it is the mathematical representation of policy π. The two summations over r and s' have been written as a single sum for the sake of convenience [SB18].

This equation can be interpreted graphically, using what is known as a backup diagram, which can be seen in Figure 2.1. Every diamond-shaped node represents a state of the system, while the circular nodes stand for a state-action pair. In every diamond-shaped node, the agent can choose a possible action, depending on a policy π. The environment, then, reacts to this decision and the agent is transferred to a new possible state, which depends on a probability p, connecting it to the state-action pair. In addition, the agent is given a reward r. Every edge of the diagram from a circular to a diamond-shaped node is associated with a reward r and a transition probability p. In a diamond-shaped node, the agent is expected to choose one of the possible actions and transition to the corresponding circular node of the lower level. In a circular node, the agent will be transferred randomly (depending on the above mentioned transition probabilities) to a new possible state. The second (double) sum of Equation 1 represents the expected value of the agent's gain while they are at a circular node, under a policy which dictates the probability of deciding every action for every possible state.

In a diamond-shaped node, the agent has already decided and executed an action, therefore the only thing left for them is to transition to a new state. This can be expressed by the corresponding recursive equation of the action-value function as:

$$Q_\pi(s, a) = \sum_{r,s'} p(s', r|s, a)(r + \gamma V_\pi(s')) \quad (2)$$

Using this equation, equation 1 can be written as:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s, a) \quad (3)$$

In the previous section we mentioned that the value function is a way of ordering policies, based on which we can define the optimal policy, meaning the

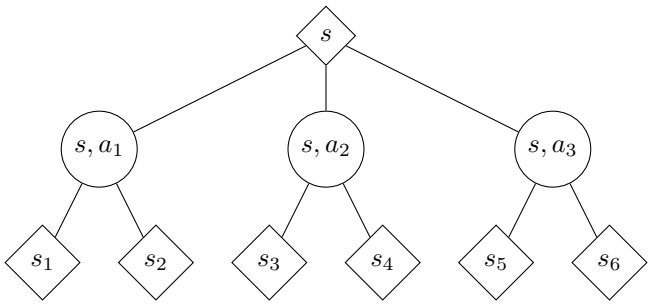


Figure 2.1: Backup Diagram visualizing the Bellman equations. Note that s_i can represent any of the system's state

one which, when followed, yields the maximum possible expected cumulative reward. Utilizing equations 1 and 2, we can define recursive equations for calculating this optimal policy. It can be proven that the optimal value function at a particular state s can be calculated as:

$$V^*(s) = \max_{a \in A} \sum_{r,s'} p(s', r|s, a)(r + \gamma V^*(s')) \tag{4}$$

This equation is known as the Bellman optimality equation regarding the value function. An abstract way to interpret this equation comes naturally, when considering that in every state, the maximum value of the value function of the optimal policy will occur by choosing the optimal action, meaning the one which maximizes both the immediate expected reward as well as the expected future reward (discounted or not). Similarly, the Bellman optimality equation regarding the action-value function is:

$$Q^*(s, a) = \sum_{r,s'} p(s', r|s, a)(r + \gamma \max_{a'} Q^*(s', a')) \tag{5}$$

It can be proven that, for finite MDPs (meaning MDPs with finite state and action spaces, not to be confused with Finite-Horizon MDPs), equation 4 has a unique solution. As long as the optimal value function has a value for every possible state, it follows that equation 4 (similarly equation 5) refers to a system of equations with $|S|$ variables, which is, in theory, solvable. If the agent possesses the optimal value function for every state, then the MDP's solution becomes a simple problem of finding the action which will lead the agent to the optimal state (the one with the maximum optimal value function) from every initial state at which they can find themselves into. Things get even simpler if the agent has the optimal action-value function at their disposal, since it already contains what the optimal action of every state is.

In practice, however, solving the above recursive equations using classical methods is rather computationally complex. Three basic requirements that need to by fulfilled in order for the equation to be solved are:

- Knowing all the system parameters beforehand
- Having the required computational and memory resources to solve the system in a realistic time and space
- The Markov property must hold

Usually the first and third conditions are fulfilled, it is unlikely, nevertheless, that the second one will always hold. As an example, we can consider a simple game of backgammon, which, when modeled, results in a state space of approximately 10^{20} states, rendering the immediate computation of the system practically impossible [SB18]. Naturally, approximate solutions must be used, with the dominant one being Dynamic Programming

2.4 Policy Iteration

The first approximate method for computing the optimal policy using the Bellman recursive equations is based on Dynamic Programming, as already mentioned. Dynamic Programming is an important algorithmic tool, which dictates breaking the problem down to subproblems, finding the optimal solution to every single one of them and then composing those solutions to obtain the optimal solution to the initial problem. If the final step is possible, meaning that the optimal solution to the initial problem can occur as a composition of the optimal solutions of its subproblems, then we say that the initial problem has optimal substructure [CLRS09].

In order to apply a solution that utilizes Dynamic Programming, we must assume that both the first and the third condition hold, meaning we know all of the system's parameters beforehand (the transition probabilities and the rewards), as well as our model satisfies the Markov property. If we make the above assumptions, we can compute the value function for a given policy (which dictates the values of the $\pi(a|s)$ probabilities) by solving the system of $|S|$ equations of 1. The existence of the solution is assured, as long as $\gamma < 1$ or the termination is certain for every possible state s. In the k^{th} step of the iteration, the value function is updated according to the following rule:

$$V_k(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)(r + \gamma V_{k-1}(s')) \tag{6}$$

It can be proven that equation 6 does converge for $k \to \infty$. At the end of the iteration, when convergence will have been achieved, the values of the equation will be the value function under policy π. This method is known as Policy Evaluation (Prediction), as it computes the value function for a particular policy, which may not necessarily be the optimal. In other words, we can use this algorithm to "guess" what the optimal policy is, by trying out (brute force) every single policy and compare their value functions. Note that, in practice, this method requires two arrays (at first sight) to execute, one to store the values of the value function of step $k-1$ and another to store the value function of step k which is computed. It has been proven, however, that the computation can be

executed in-place, with just one array, which will be updated dynamically. This means that during the update of step k, values already computed at step k may be used, which in fact accelerates the convergence. Finally, we must address that, in order to check for convergence, we use the maximum difference between old and new values of the value function, namely:

$$\max_{s \in S} |V_k(s) - V_{k-1}(s)| < \theta$$

where θ is the difference threshold, defined by the user

As already mentioned, while the method described above is useful, it alone is not efficient enough for computing the optimal policy, for it relies solely on guesses. Nevertheless, it is a valuable tool in calculating the value function of a policy and thus helping to find a better one. A better policy than a given π can occur if we observe that, at a particular state s, choosing a different action than the one dictated by π and following π thereafter yields better results than blindly following π everywhere.

The above statement is a special case of a general theorem, called the Policy Improvement Theorem. According to the Policy Improvement Theorem, if for a pair of deterministic policies π, π':

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s)$$

for every state $s \in S$, then policy π' is more optimal than (or as better as) π, meaning that:

$$V_{\pi'}(s) \geq V_\pi(s)$$

for every $s \in S$

If we apply the above exchanging actions logic for every state and every possible action, we obtain a greedy method for computing a more optimal policy:

$$\pi'(s) = \operatorname*{argmax}_a Q_\pi(s, a) = \operatorname*{argmax}_a \sum_{r, s'} p(s', r|s, a)(r + \gamma V_\pi(s')) \tag{7}$$

This new greedy policy chooses the action that maximizes the immediate reward in the next step according to V_π and satisfies the conditions of the Policy Improvement Theorem. The method by which a policy π can be improved and transformed to policy π' is known as Policy Improvement.

Having already improved a given policy π and obtained new, more optimal policy π', one would think to further improve the already improved policy π'. This idea is the foundation of the first approximate algorithm used to solve the Bellman equations, which is known as Policy Iteration

The essence of this iterative method is constantly calculating the value function and then improving the policy, until it converges to the optimal. Starting from a randomly selected initial policy, in every step we calculate its value function using Policy Evaluation. Afterwards, we improve it by applying Policy Improvement. This process is then repeated for the improved policy again and again, until convergence is achieved, which is certain for finite MDPs. This idea is presented in pseudocode in algorithm 5.

Algorithm 5 Policy Iteration

```
1: function policyIteration(θ)
2:     Initialize V(s) and π(s) (arbitrarily)
3:     optimalPolicy = false
4:     while !optimalPolicy do
5:         δ = 0
6:         while δ < θ do
7:             for s ∈ S do
8:                 v ← V(s)
9:                 V(s) ← Σ_{r,s'} p(s', r|s, a)(r + γV(s'))
10:                δ ← max(δ, |v − V(s)|
11:            end for
12:        end while
13:        policyStable = true
14:        for s ∈ S do
15:            oldAction = π(s)
16:            π(s) = argmax_a Σ_{r,s'} p(s', r|s, a)(r + γV(s'))
17:            if oldAction ≠ π(s) then
18:                policyStable = false
19:            end if
20:        end for
21:    end while
22:    return V and π
23: end function
```

2.5 Value Iteration

Policy Iteration, though useful and more efficient than Policy Prediction, oftentimes is still not sufficient, as it involves extensive, repetitive use of Policy Evaluation, which is both time inefficient and also grows rapidly as the number of states grows larger.

The solution to this problem appears by deciding to stop Policy Evaluation prematurely rather than wait for convergence. To be exact, even one iteration is enough, while it can be shown that the convergence conditions are not violated. The above modification leads to a new algorithm, known as the Value Iteration algorithm, which is extensively used in modern problems.

This method is based on transforming the recursive Bellman equation 4 to an update rule of the values of the iteration:

$$V_k(s) = \max_a \sum_{r,s'} p(s', r|s, a)(r + \gamma V_{k-1}(s')) \tag{8}$$

To terminate the recursion, we can again use the maximum difference between old and new value of the value function. It will be compared with an arbitrarily chosen threshold θ. Value Iteration is presented in pseudocode in algorithm 6

Algorithm 6 Value Iteration

```
1: function valueIteration(θ)
2:     Initialize V_tmp(s), V_aux(s), Q(s,a) and π(s) (arbitrarily)
3:     δ ← ∞
4:     while δ ≥ θ do
5:         δ ← 0
6:         for s ∈ S do
7:             for a ∈ A(s) do
8:                 Q(s,a) ← 0
9:                 for s' ∈ S(s) do
10:                    Q(s,a) ← Q(s,a) + T(s,a,s')(R(s,a,s') + V_tmp(s'))
11:                end for
12:            end for
13:            V_aux(s) ← max_a Q(s,a)
14:            π(s) = argmax_a Q(s,a)
15:        end for
16:        V_tmp ← V_aux
17:    end while
18:    return V and π
19: end function
```

With $A(s)$ we represent the set of all available actions at state s, while $S(s)$ indicates the set of states accessible from state s. In algorithm 6, operations have been written in detail (opposite algorithm 5) as this particular algorithm is the main tool this thesis will focus on improving and so parts of it should be able to be seen explicitly.

2.6 Finite-Horizon

All the above algorithms for finding the optimal policy are applied on Infinite-Horizon MDPs, meaning that the agent thinks they will be making decisions indefinitely or until they reach a terminal state.

In the case where the MDP's horizon is predetermined and finite, with a length of N, the agent knows in advance how many steps they will make and could theoretically compute the optimal policy for every possible number of remaining steps with great accuracy. An important factor in this computation is the number of steps remaining, which is non-existent in the Infinite-Horizon case.

Every step of the Value Iteration represents a number of remaining steps of the horizon. As a result, execution until convergence is not required, so we opt for as many iterations as the length of the horizon. However, for every value of steps remaining we must save a different policy, resulting in greater memory needs during the execution of the algorithm, growing along with the length of the horizon (at least in the basic/naive case). We can also omit the discount factor γ, as there is no longer need for convergence, and so every optimal value

function value represents the expected cumulative reward without discount that the agent will collect starting from a particular state and following a particular policy thereafter.

2.7 Turnpikes and Planning Horizons

In many cases, Finite-Horizon MDP problems prove to be absurdly demanding in terms of execution time and memory. Comparing an Infinite-Horizon MDP to Finite-Horizon one, we can easily observe that, if the horizon N is large enough, the times Value Iteration will be executed are much more than those needed for convergence in the IFMDP case. Additionally, an Infinite-Horizon MDP results in only one optimal policy, independent of the actions that have already been made or will be made in the future. As a result, the solution can be achieved by storing $\mathcal{O}(1)$ arrays of length $|S|$ in memory, resulting a total space complexity of $\mathcal{O}(|S|)$. The above facts lead to the conclusion that being able to approximate the Finite-Horizon MDP solution as an Infinite-Horizon MDP will yield the most optimal results, both in terms of time and space.

As it so happens, such an approximation can be made, as proven by Shapiro [Sha68] (also mentioned in [LP19] and [Put94]). According to that work, for any FHMDP (must also be finite and not have variable rewards and/or probabilities) and discount factor γ there exists an integer $N^*(\gamma)$, known as the Turnpike Integer, where the decisions made in steps $k \geq N^*(\gamma)$ are the same as the ones decided in the Infinite-Horizon case. In other words, the agent can solve the MDP (of horizon N) as if its horizon was infinite and come up with an optimal policy, independent of time. Then, they can follow this policy from N to $N - N^*$ steps remaining. Then, they solve again this MDP as an FHMDP with horizon equal to $N - N^* - 1$ and result in a different policy for each time step.

Notice that this result can only be applied if the optimal policy of the corresponding Infinite-Horizon MDP is a single decision rule, meaning it cannot have 2 optimal actions for a single state, as is shown in [Sha68]. The above results naturally raise the question: How can one find the Turnpike Integer (also known as Planning Horizon) ? Unfortunately, a concrete answer does not exist so far, there are, however, ways to bound this number, described in detail in [LP19] and [Put94].

In our experiments in the following sections, we chose many different values of γ and observed approximately what the value of the Turnpike Integer is, by examining the convergence of the optimal policy, which did not yield sufficient results. Another way of executing this approximation algorithm is to run a FHMDP solving algorithm up until the length of the horizon and, assuming the horizon is sufficiently large, use the optimal policy of steps remaining equal to the length of the horizon to make decisions for the whole horizon. This is of course expected to yield suboptimal results, as, after the planning horizon, the optimal policy of the FHMDP should be used.

2.8 Non-Stationary MDP

We define a Non-Stationary MDP (NSMDP) as follows: a NSMDP is an MDP whose transition and reward functions depend on the decision epoch. It is defined by a 5-tuple S, T, A, P_t, R_t where S is a state space; $T = 1, 2, .., N$ is the set of decision epochs with $N \leq \infty$; A is an action space; $P_t = p(s'|s, a, t)$the probability of reaching state s' while performing action a at decision epoch t in state s; $R_t(s, a, s')$ is the scalar reward associated to the transition from s to s' with action a at decision epoch t [LR20]. In other words, NSMDPs' reward and transitions functions are also a function of time. As a result, the policy is also dependent on time, and is thus non-stationary. This type of MDPs has lots of applications in real-world problems, as described thoroughly in Section 1.2. The agent could either know how the values of those functions will change with regards to time in advance, or they could be completely ignorant to the adjustment methods. It is, however, possible for a solution to be found (for the first case) using the FHMDP algorithms described below. There are also other ways of solving this type of MDPs (for the second case), as is explained in [LR20] (using the Risk-Averse Tree-Search Algorithm).

2.9 Machine Learning

The idea of a machine being able to think as a human and solve complex problems has been around since the ancient times, but it was not until 1956 that scientific research was actually made, creating the field of Artificial Intelligence.

Along with Artificial Intelligence, rose another great scientific topic, namely Machine Learning. The belief that a computer could learn and perform tasks as well as the human brain (and even exceed it), although dazzling, soon faded and gave way to more realistic ideas. It was indeed possible to have a computer learn a task, but it would be unimaginably hard to have the same machine perform multiple tasks imitating the human brain.

During the early stages of the development of Artificial Intelligence and Machine Learning, the goal was for the computer to be able to fast and efficiently solve complex problems humans could not. Researchers were faced with a much harder challenge when they tried to have a computer solve a problem which could not be formally defined, although the human brain could easily solve it. This led to the realisation that the way to have the computer solve it was to train it with experience.

To better understand the core concepts of Machine Learning, we adopt T. Mitchell's definition of it: "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" [Mit97].

Machine Learning tasks are categorized in three groups depending on the feedback the computer receives in the current system.

2.9.1 Supervised Learning

In Supervised Learning problems, input data consists of the input samples along with a corresponding label assigned to them by a knowledgeable external source. The learning process then can aim for the computer to assign every new sample that arrives to one of a finite number of labels, occurring from the input training data labels, and is thus a Classification task. If the desired output is a number of values for a finite number of continuous variables instead of labels, the task is considered a Regression task [Bis07].

2.9.2 Unsupervised Learning

In a case where the input data consists only of the input samples, without labels or any other values placed by an external source, the learning task is characterized as Unsupervised Learning. In such problems, the objective of the computer could be to discover groups of data bearing similarities and organize them into clusters (Clustering), or to assume the data follow a set distribution and try to define it formally (Density Estimation) or even to decrease the number of dimensions of the input data and project them to lower dimension spaces in order to visualize them (Visualization) [Bis07].

2.9.3 Reinforcement Learning

Reinforcement Learning is a method for the computer (also referred to as the agent) to learn what to do in an environment, which actions to make in every possible state it finds itself, with the sole purpose of maximizing and obtained scalar reward. Every decision the agent makes affects the reward it receives, as well as the next state it ends up in. Consequently, the agent is in a constant trade-off between exploitation and exploration, with the former meaning choosing actions which it already knows yield an adequate reward, while the latter refers to the agent exploring new paths through different, seemingly worse rewarding actions, in order to possibly discover even greater rewards on the way.

Reinforcement Learning differs greatly from Supervised Learning, as the latter has the computer use labels put on the data by an external knowledgeable source to generalize the way it categorizes each sample as to correctly recognize the category samples not given in the dataset belong to. This way the computer does not learn from its own experience but rather from directions given to it by the external source, making it difficult to learn without large datasets and effort to label them.

It also differs from Unsupervised Learning, which tries to discover hidden patterns in a dataset without labels or any other interference from an external source. In both cases the agent learns and executes its task without labels on the dataset samples or other interference by an external source, but in Reinforcement Learning it tries to maximize an obtained reward rather than trying to find patterns and associations between the samples of the dataset [SB18].

2.10 Reinforcement Learning Methods

In all algorithms described in this Section, it was assumed that the agent knew all of the system's parameters in advance. They could then use those accurate estimations of the parameters to calculate the optimal policy and follow it to obtain the optimal results (maximum cumulative reward). In real-world applications, however, the agent rarely knows all of their environment beforehand. In order to overcome this problem, the agent must of course learn its environment, which is achieved by performing actions. It could be that the agent executes an optimal policy calculation algorithm, based on their current knowledge of the environment and has achieved a great result, but could be missing out on an even greater total reward due to their lack of knowledge of "hidden" system parameters (which are yet to be discovered).

The agent's view towards the system plays a crucial role in how the learning process is carried out. There are two fundamental approaches of Reinforcement Learning, namely model-free learning and model-based learning.

2.10.1 Model-Free Learning

By model-free we refer to an algorithm that does not utilize any reward or transition functions but rather relies solely on the defined states, actions and observations made during the calculation of the optimal policy [SB18].

Perhaps the most common method of model-free learning is the Q-Learning method, proposed by Watkins [WD92]. The goal of this method is to estimate the values of the state-action value function using the agent's experiences. Firstly, states and actions are defined to describe the environment. Afterwards, state-action values are initialized arbitrarily for every state-action pair. The agent can then start making decisions. After they make a decision to follow action a from state s, the obtain a reward r and transition to a new state s'. Utilizing this information, the agent can estimate the value of $Q(s, a)$ as follows:

$$Q_{exp}(s, a) = r + \gamma V(s') = r + \gamma \max_{a' \in A} Q(s', a') \tag{9}$$

This quantity can be added to the corresponding state-action value function's value that the agent already knew, to combine both prior and newly obtained knowledge. The agent, then, obtains a new estimation of the state-action function:

$$Q(s, a) = (1 - a)Q(s, a) + aQ_{exp}(s, a) \tag{10}$$

The newly introduced parameter a is known as the learning rate and affects how fast the agent will learn from experience. It has been proved that the agent can successfully learn and achieve the optimal policy using this algorithm if it is executed for every action of every state an infinite amount of time and a is decayed in an optimal manner.

We can observe that Q-Learning offers a very efficient method of learning and solving dynamically a Reinforcement Learning problem. Its great disadvantage, however, is that a large number of experiences is needed in order for

the estimation to be accurate. This is because at every decision the model only updates the values regarding its previous state and chosen action, while no new information is obtained about any of the other actions or states of the model.

2.10.2 Model-Based Learning

In the context of model-based learning, the agent attempts to utilize experiences as a way of learning the model's parameters and then, after the estimation those parameters is sufficiently accurate, they can attempt to calculate an optimal policy based on this knowledge.

Every time the agent makes a decision, they will receive a reward and transition to a new state. This information can be used to update the reward and transition estimation for this initial state, decision and new state. After a large number of such experiences, the agent can perform an optimal policy calculation algorithm as usual, such as Value Iteration, and use it for its decisions. Then the process is repeated until the estimation of the model parameters is believed to be accurate. Although this method produces an accurate estimation of the environment which can be stored and be re-used for future calculations, it comes with a great computational cost.

Methods to faster approximate the model usually involve updating the value functions directly. The most fundamental is known as Dyna-Q [SB18]. After an experience, the agent updates its corresponding reward and transition values and performs the following update on the state-action value function involved:

$$Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a' \in A} Q(s',a') \tag{11}$$

Then, the same operation is performed on k random state-action values. This speed up the process rapidly. There exist also other methods which are optimizations on the Dyna-Q. Two examples are Prioritized Sweeping [MA93] and Queue-Dyna [PW98], which involve assigning priorities to states and updating those states using these priorities instead of randomly.

2.10.3 Exploration Strategies

In both of the cases described above, the agent could fall into the trap of exploitation, by choosing the best action to their knowledge, neglecting the exploration of different actions which may lead to even greater rewards. In order to achieve optimal results, the agent must find a balance between exploring as best they can and exploiting the most optimal actions to their knowledge.

A simple strategy that achieves this equilibrium is known as the ε-greedy Method. This method dictates that, before every decision the agent faces, instead of deciding the optimal action (to their knowledge) they might make a decision at random, based on a probability ε. In other words, at every decision the agent can either choose a completely random action with probability ε, or execute their decision making process as normal with probability $1 - \varepsilon$. This probability does not necessarily stay constant throughout the decision making

process, but could also be adaptive [dd17] (for example it could diminish the more knowledge the agent acquires).

3 A Note on Optimizing the Value Iteration Algorithm

As already mentioned in Section 2.3.3, an important requirement in order to be able to solve the Bellman Equations and thus the MDP is having the required computational and memory resources. We also established that, even in cases of MDPs that seem simple at first, the execution time needed to solve them is often unrealistic. Considering the above, optimizing the time complexity of the Value Iteration algorithm is a well-known obstacle in MDP optimization theory. In this Section, we discuss attempts already made towards this goal, as well as our own proposed optimization and the results it yielded.

3.1 Proposed Solution

While the main focus of this work is optimizing the space complexity of Finite-Horizon MDPs, an attempt towards improving the execution time of Value Iteration in the case of Infinite-Horizon MDPs was made. The idea of our proposed method was bounding the state-action function in way such that some actions would be discarded altogether. This way, when Value Iteration is executed, some actions will not be considered towards calculating the value function, and, thus, neither will their corresponding transitions, practically reducing the execution time.

The first step to this approach was to calculate sufficient upper and lower bounds for every state-action value. Earlier, in Equation 5, we established that the optimal state-action function depends on the immediate reward as well as the optimal next state's state-action function, or in other words, the optimal next state's value function, as $\max_{a \in A} Q^*(s,a) = V^*(s)$. The first term of Equation 5 can be bounded above as follows:

$$q_1(s,a) = \sum_{r,s'} p(s', r|s, a)r \leq \max_{s',r} r \tag{12}$$

In other words, the expected immediate reward the agent will collect by being in state s and performing action a is always less than or equal to the maximum reward accessible from state s and action a. By symmetry, we obtain an inequality for below bounding:

$$q_1(s,a) = \sum_{r,s'} p(s', r|s, a)r \geq \min_{s',r} r \tag{13}$$

Regarding the second term of Equation 5, an above bounding can be performed:

$$q_2(s,a) = \gamma \sum_{r,s'} p(s',r|s,a)V^*(s') \leq \gamma \max_{s'} V^*(s') \tag{14}$$

This expresses the fact that, the total discounted reward the agent is expected to receive after performing action a from state s and transitioning to a new state s' is always less than or equal to the maximum expected total discounted reward the agent will receive starting from an accessible state. Symmetrically, we can also observe that:

$$q_2(s,a) = \gamma \sum_{r,s'} p(s',r|s,a)V^*(s') \geq \gamma \min_{s'} V^*(s') \tag{15}$$

Adding together inequalities 12 and 14, we obtain an upper bound for the state-action function of a particular state and action pair:

$$Q^*(s,a) \leq \max_{s',r} r + \gamma \max_{s''} V^*(s'') = Q_u(s,a) \tag{16}$$

Similarly, we derive a lower bound:

$$Q^*(s,a) \geq \min_{s',r} r + \gamma \min_{s''} V^*(s'') = Q_l(s,a) \tag{17}$$

Notice that, in both inequalities, the maximum (or minimum) reward might come from a state different than that of the maximum (or minimum) optimal value function.

In practice, we would execute the calculation of upper and lower bounds for every state-action pair of the model, and then, for a particular state s, delete all those actions from the possible actions list, whose upper bound is lower than the maximum lower bound of s, as we are certain that no optimal result could be achieved by opting for those actions. Both the calculation, as well as the deletion are presented in Algorithm 7.

Algorithm 7 Calculate Bounds

```
1: procedure calcBounds
2:     for s ∈ S do
3:         s.maxLBound ← −∞
4:         for a ∈ A(s) do
5:             currMax ← −∞
6:             currMin ← ∞
7:             for s′ ∈ S(s) do
8:                 currMax ← max(currMax, V*(s′))
9:                 currMin ← min(currMin, V*(s′))
10:            end for
11:            Q(s, a).upperBound ← γ currMax + max(Q(s, a).rewards)
12:            Q(s, a).lowerBound ← γ currMin + min(Q(s, a).rewards)
13:            s.maxLBound ← max(maxLBound, Q(s, a).lowerBound)
14:        end for
15:    end for
16:    for s ∈ S do
17:        for a ∈ A(s) do
18:            if Q(s, a).upperBound < s.maxLBound then
19:                deleteafromA(s)
20:            end if
21:        end for
22:    end for
23: end procedure
```

We consider that the QStates are represented as objects in our implementation, that have two members, *upperBound* and *lowerBound* corresponding to the values of the bounds discussed earlier. A State, also represented as an object, has the member *maxLBound*, equal to the maximum lower bound of its QStates. After we calculate those upper, lower and max lower bounds, we delete every action whose upper bound is less than the maximum lower bound of a particular state s from its possible actions list, and we repeat this process of deletion for every state in S.

The time complexity of the first loop is $\mathcal{O}(|S|^2|A|)$, as the first loop iterates over every state, action and neighboring state, while the second loop iterates over every state and action, resulting in $\mathcal{O}(|S||A|)$. In total, the time complexity of *calcBounds* is $\mathcal{O}(|S|^2|A| + |S||A|) = \mathcal{O}(|S|^2|A|)$.

A natural question arising is when to perform the bounds' calculation, as performing them when our information about the value function of nearby states is not sufficient could lead to deletion of optimal actions by accident. As an example, we consider a fully observable MDP, whose value function is set to an initial value (e.g. 0) at every state. One idea is updating the bounds every time the value functions are updated, which does not have optimal performance, as the calculations saved be discarding actions are far less than those required to recalculate the bounds at every iterations.

In the case of a model-based approach, such as the one used in our Finite-Horizon experiments [LKKK17], one could calculate the bounds after the training of the model has been completed. During training, Value Iteration is executed at scarce points in time, thus the agent has a (maybe incomplete) value function value at every state.

We attempted this method in a different experiment. However, the results we obtained were not what we desired. The total execution time seemed to have increased even more with the addition of bounds' calculation. An explanation for this fact could be that, as we will be seeing in Section 5, our model consists of at most 3 actions at each state (add 1 VM, remove 1 VM, no operation). Although the state space might be large, only very few actions were actually deleted, thus resulting in more time spent calculating the bounds than saved after the deletion. Our theory, nevertheless, still has room for improvement and could be the foundation of further optimizations of the Infinite-Horizon Value Iteration's execution time.

3.2 Prior Attempts on Optimizing Time Complexity

While our attempt did not yield any fruitful results, we theorized that it could in other use cases. As already stated in Section 1.3.1, the concept of bounding the value function in order to eliminate suboptimal actions has interested many researchers. Most applications, however, refer to either Linear Programming approaches of solving the MDP or utilize action elimination along with other techniques.

4 Optimizing Finite-Horizon MDP Algorithms

As it has already been established, Bellman recursive equations offer an accurate solution to the Finite-Horizon MDP problems in the case of discrete time. In order to execute those equations in an algorithmic framework, however, a great amount of time and memory resources are required, depending on the application. In this section, we delve into the pre-existing solutions to the problem, as well as proposing two new algorithms with improved space complexity.

4.1 Pre-Existing Solutions

The known methods of solving a fully observable FHMDP are efficient in only one of two ways at a time: either they are as quick as it gets in execution or they occupy the minimum possible amount of memory. Their results will be used as the basis, on which we will construct our models and will also serve as comparison targets for our method's results.

4.1.1 Naive Solution

One of the most prevalent solutions for fully observable FHMDPs is the Naive Solution as we call it. This solution refers to computing every policy for every

decision epoch (meaning every number of steps remaining) and store them in memory after they are computed. When the agent finally starts making decisions, they will be able to very quickly recall the policy for the current number of steps they have available, make a choice using it and then delete it. The process is then repeated for the next number of remaining steps until the agent has completed a number of steps equal to the length of the horizon.

This solution written in pseudocode is presented in algorithm 8. We assume that every policy, which consists of an array of length equal to the number of states, is stored in a stack. As a result, at any given time on top of the stack lies the policy with the largest number of steps remaining. Note that the numbers of steps remaining of the stack's entries are the positive integers 1 through k, in descending order starting from the top (where k is the number of steps remaining currently to the agent). This stack in algorithm 8 is represented by *policyStack*, while the array containing the current policy is represented by *policyArray*. We assume that *policyStack* is initialized outside this function and can be accessed by it globally. We also consider given the *takeAction* function, which takes the chosen action as argument and executes it, rewarding the agent and transferring them to the next state, which is calculated in a probabilistic way. This function is different and depends on the context of the application.

Algorithm 8 Naive Solution

```
1:  procedure naiveSolution(horizon)
2:      V_tmp ← []
3:      for i = 0 to horizon do
4:          policyArray ← []
5:          V_aux ← []
6:          for s ∈ S do
7:              for a ∈ A(s) do
8:                  Q(s, a) ← 0
9:                  for s' ∈ S(s) do
10:                     Q(s, a) ← Q(s, a) + T(s, a, s')(R(s, a, s') + V_tmp(s'))
11:                 end for
12:             end for
13:             V_aux(s) ← max_a Q(s, a)
14:             policyArray[s] = argmax_a Q(s, a)
15:         end for
16:         V_tmp ← V_aux
17:         policyStack.push(policyArray)
18:     end for
19:     for i = 0 to horizon do
20:         chosenAction = policyStack.top()[currentState]
21:         takeAction(chosenAction)
22:         policyStack.pop()
23:     end for
24: end procedure
```

Regarding the time complexity of algorithm 8, we can observe the fact that it contains two for-loops which iterate over the length of the horizon N. The first for-loop contains three nested for-loops, which iterate over the states, actions (only those available at current state s, $A(s)$) and neighboring states (only those accessible at current state s, $S(s)$) respectively. The total worst-case time complexity of the first for-loop is, as a result, $\mathcal{O}(N|S|^2|A|)$. The second for-loop does not contain nested loops, thus its time complexity depends only on the function calls inside it. We consider the time complexity of *takeAction* trivial, and so the time complexity of the Naive Solution comes only from the first for-loop and is equal to $\mathcal{O}(N|S|^2|A|)$.

The space complexity in this case involves everything happening inside the *naiveSolution* process, meaning that, again, the space occupied by *takeAction* is trivial. In total, throughout the whole execution of the algorithm, N policy arrays will be stored, each of length $|S|$. Thus, the total space complexity turns out to be $\mathcal{O}(N|S|)$, which grows proportinally to the horizon's length for constant state spaces.

4.1.2 In-Place Solution

Another method deemed useful towards solving fully observable FHMDPs is the In-Place Solution. The agent calculates only the value function (and thus the policy array) of the decision epoch equal to the number of steps they have left to make. As a result, they only need to store (at most) one policy array and two value function arrays of length $|S|$ in memory at any given point in time. The great drawback of this method, however, is that, after the agent uses a policy array, in order to calculate the next they must again start the calculation from the beginning. As a result, the execution time of this algorithm is immensely greater than that of the Naive Solution. The above statements are written in pseudocode in algorithm 9.

Algorithm 9 In-Place Solution

```
1: procedure inPlaceSolution(horizon)
2:     V_tmp ← []
3:     V_aux ← []
4:     stepsRemaining = horizon
5:     while stepsRemaining > 0 do
6:         for i = 0 to stepsRemaining do
7:             policyArray ← []
8:             V_aux ← []
9:             for s ∈ S do
10:                for a ∈ A(s) do
11:                    Q(s, a) ← 0
12:                    for s' ∈ S(s) do
13:                        Q(s, a) ← Q(s, a) + T(s, a, s')(R(s, a, s') + V_tmp(s'))
14:                    end for
15:                end for
16:                V_aux(s) ← max_a Q(s, a)
17:                policyArray[s] = argmax_a Q(s, a)
18:            end for
19:        end for
20:        V_tmp ← V_aux
21:        chosenAction = policyStack.top()[currentState]
22:        takeAction(chosenAction)
23:        stepsRemaining = stepsRemaining − 1
24:    end while
25: end procedure
```

In algorithm 9, variable *stepsRemaining* is used, indicating the number of steps remaining for the agent to make. This variable is initialized to the length of the horizon and is decremented by 1 each time the agent makes a decision. The total time complexity of this algorithms is $\mathcal{O}(|S|)$, completely independent of the horizon, as at any given time at most 3 arrays are stored in memory, two for the value function of the current and the previous step (V_{tmp} , V_{aux}) and one for the policy of the current step (*policyArray*).

The advantage of the memory needs being independent of the horizon's length is huge, especially in the cases of large horizons. Nevertheless, it comes with a great drawback in execution time. Inside the while-loop in algorithm 9, we observe a for loop similar to the one found in the Naive Solution. This for-loop is executed each time from 0 up to the number of steps remaining to the agent. At the end of every iteration of the while-loop, the number of steps remaining is decremented by 1. This while-loop is thus executed as many times as the horizon's length. Every iteration of this while-loop is of time complexity: $\mathcal{O}(stepsRemaining|S|^2|A|)$. In total, if we write *stepsRemaining*

as j for convenience, the total time complexity turns out to be:

$$N|S|^2|A| + (N-1)|S|^2|A| + ... + |S|^2|A| = \sum_{j=1}^{N} j|S|^2|A| = |S|^2|A| \sum_{j=1}^{N} i$$

This particular sum is known to be evaluated to $\frac{N(N+1)}{2}$, and so its time complexity is quadratic with respect to the length of the horizon and can be written as $\mathcal{O}(N^2|S|^2|A|)$. This solution, consequently, has tiny memory requirements even for large horizon values, it requires, however, a huge time to execute, even for relatively small horizon lengths, rendering it forbidding in cases of large horizons.

4.2 Proposed Solutions

The two methods described above each have their own unique advantages, with the Naive Solution achieving sufficient time complexity at the expense of memory occupied, while the In-Place solution requires the least amount of memory possible, resulting in practically impossible execution times. The two methods introduced in this section offer space complexity improvements with some (if any) execution time burden, depending on the user's needs.

During the discussion of this section, we assume that an index representing the number of steps remaining is assigned to every value function array, which contains the value function calculated for this number of steps remaining. We will be referring to this value function array's index simply as index, for the sake of convenience.

4.2.1 Square Root Solution

Our first suggested solution is an improved version of the Naive Solution with regards to the space complexity. This means that this solution has the same time complexity (asymptotically) as the Naive Solution, while lowering the memory needs at the same time. It is based on the Naive Solution, with a simple twist: instead of storing every policy array (and value function array) in memory, we can store only a few "checkpoint" arrays and recompute every intermediate array we might need.

By this reasoning, a good starting idea would be storing in memory the value function arrays that correspond to indices that are multiples of the square root of the total horizon's length. As a result, the number of arrays stored would no longer be linear with regards to the horizon, but in the orders of the square root of the horizon N.

In practice, we would calculate every array up to the one whose index is equal to the horizon's length, storing every array whose index is a multiple of the square root of the horizon in the meantime. Consequently, we would use the value function array we obtained in the above step to compute the optimal policy for the state the agent is in and the number of steps remaining. The agent will then execute the chosen decision and delete the arrays from memory.

Afterwards, the computation is repeated for the value function array of the next index (which is equal to the index used before minus 1, as the agent just made a step). The big difference here is that this time the computation starts from the last value function array calculated(the one with the greatest index out of those stored in memory), resulting in less recomputations required. Graphically, this can be seen in Figure 4.1.

Nevertheless, this solution does not yield the expected results, for it burdens the algorithm's execution time. In every interval of length $\sqrt{N}$, N total computations of time complexity $\mathcal{O}(|S|^2|A|)$ are needed. The reason for this is that, in order to calculate every array in this interval, we need all of the arrays whose index is lower than the one of the array we are trying to compute. By definition, however, we do not store those arrays in memory, so recomputations have to be carried out. Another way to view this issue would be to treat this process as executing an In-Place Solution at every interval of length $\sqrt{N}$, as every time we need an array in this interval we have to recompute all those that come before it. Making the above assumption, one can clearly observe that the time complexity of every interval of length $\sqrt{N}$ is equal to the time complexity of the In-Place solution for a "horizon" of length $\sqrt{N}$. Using the asymptotic notation, we can write the above result as $\mathcal{O}((\sqrt{N})^2|S|^2|A|) = \mathcal{O}(N|S|^2|A|)$. In addition, these calculations are used in every one of those intervals, $\sqrt{N}$ times in total. In conclusion, the total time complexity turns out to be: $\mathcal{O}(\sqrt{N}N|S|^2|A|) = \mathcal{O}(N^{\frac{3}{2}}|S|^2|A|)$.

Fortunately, this solution can be improved further with regards to the execution time, by storing in memory $\sqrt{N}$ more arrays. In every interval of length $\sqrt{N}$, during the computation of the array with the maximum index (in that interval) we store every intermediate array we compute and delete it after we have used it. This way every array in this interval is calculated only once, resulting in $\mathcal{O}(\sqrt{N}|S|^2|A|)$ time complexity in every one of those intervals, and ,because there are $\sqrt{N}$ such intervals in total, the total time complexity is $\mathcal{O}(\sqrt{N}\sqrt{N}|S|^2|A|) = \mathcal{O}(N|S|^2|A|)$. The memory required increases by $\sqrt{N}$ arrays, making it so that no more than $2\sqrt{N}$ arrays of length $|S|$ can be stored in memory at any given time, resulting in a total space complexity of $\mathcal{O}(\sqrt{N}|S|)$.

Visually, Figure 4.2 depicts the way arrays are stored in the case of a horizon of length 16. On the left lie the value function arrays along with their indices. Those stored in memory during the algorithm's execution are colored in green. On the right, one can observe that in the interval 12 up to 15 (which is of length $\sqrt{16} = 4$), having initially only the array of index 12 stored in memory, we compute the array of index 15, using the array with the greatest index available at any given time, while also storing it. Following this reasoning, at most 8 arrays can be stored in memory at any given point in time in this example, which is exactly twice the square root of the horizon's length.

Figure 4.2 graphically shows exactly how the above described computations are executed at an interval of length $\sqrt{N}$ in the case of horizon equal to 16. Specifically, at the interval starting with index 12 and ending with index 15 the steps followed are:

1. Initially, only the value function array of index 12 is stored.

Index	Value Function
1	...
2	...
3	...
4	...
5	...
6	...
7	...
8	...
9	...
10	...
11	...
12	...
13	...
14	...
15	...
16	...

Figure 4.1: Stored value function arrays (in color) at indexes multiples of 4, square root of horizon length 16.

2. We compute the value function array of index 13 using the stored value function array of index 12 and we store it temporarily

3. We compute the value function array of index 14 using the stored value function array of index 13 and we store it temporarily

4. We compute the value function array of index 15 using the stored value function array of index 14 and we store it temporarily

5. We use the value function array of index 15 to compute the optimal policy, the agent executes the chosen action and the value function array of index 15 is deleted from memory.

6. We use the value function array of index 14 to compute the optimal policy, the agent executes the chosen action and the value function array of index 15 is deleted from memory.

7. We use the value function array of index 13 to compute the optimal policy, the agent executes the chosen action and the value function array of index 15 is deleted from memory.

8. We use the value function array of index 12 to compute the optimal policy, the agent executes the chosen action and the value function array of index 15 is deleted from memory.

This process is repeated at every interval of length $\sqrt{N}$.

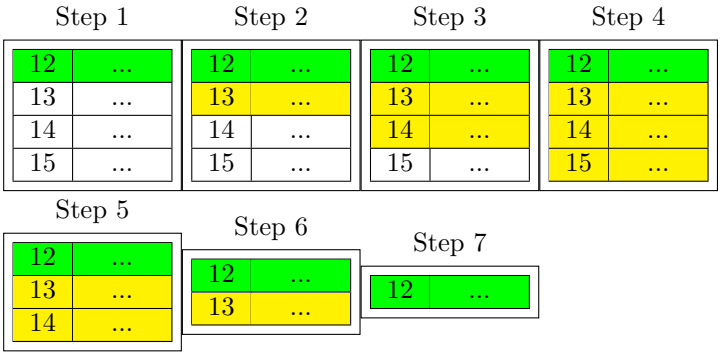


Figure 4.2: Computing value function arrays at indices $15 \dots 12$; array stored in memory in advance in green; arrays stored temporarily in memory throughout the execution in yellow; arrays yet to be computed in white.

Algorithm 11 contains the above process written in pseudocode. Firstly, we have to define the $calculateValues$ function, presented in Algorithm 10, which will be used to compute every value function array needed. The functions arguments are: the index of the value function array we wish to compute (symbolized as $targetIndex$), the largest index of a value function array stored in memory ($startingIndex$), which is assumed to be 0 in the case of no array stored in memory yet, as well as the value function array of $startingIndex$, filled with zeros in the case of $startingIndex = 0$. The fourth argument ($tree$) is a boolean variable, which, when true, forces the function to save every intermediate value function it computes at a stack, called the $valueStack$. On the other hand, when its value is false, the function only stores the final value function array in memory. Note that, whenever a value function array is stored in $valueStack$, its corresponding index is also stored in a second stack, called the $indexStack$.

Algorithm 10 Auxiliary function for calculating value functions

```
1: procedure CALCULATEVALUES(targetIndex, startingIndex, V, tree)
2:     V_tmp ← V
3:     V_aux ← []
4:     for i = startingIndex + 1 to targetIndex + 1 do
5:         for s ∈ S do
6:             for a ∈ A(s) do
7:                 Q(s, a) ← 0
8:                 for s' ∈ S(s) do
9:                     Q(s, a) ← Q(s, a) + T(s, a, s')(R(s, a, s') + V_tmp(s'))
10:                end for
11:            end for
12:            V_aux(s) ← max_a Q(s, a)
13:        end for
14:        V_tmp ← V_aux
15:        V_aux ← []
16:        if tree then
17:            valueStack.push(V_tmp)
18:            indexStack.push(i)
19:        end if
20:    end for
21:    valueStack.push(V_tmp)
22:    indexStack.push(i)
23: end procedure
```

The actual Square Root Solution is executed by Algorithm 11, written in pseudocode. Function *rootEvaluation* takes as arguments the horizon's length and executes every computation necessary. In the first for-loop, the value function arrays whose index is a multiple of $\lfloor\sqrt{N}\rfloor$ are computed, which are stored in both the stacks (as the *tree* argument of *calculateValues* is set to false). Afterwards, if the horizon's length is not a perfect square, the last stored value function array's index will be smaller than the horizon's length, which should be the index of the last stored value function array, so every value function array up to it is computed and stored in both the stacks.

Having stored these arrays, the model's memory state is similar to this of Figure 4.1, meaning there is a value function array stored for each multiple of $\lfloor\sqrt{N}\rfloor$, as well as any excess arrays calculated in the second loop. Variable *steps*, which is initialized to the length of the horizon, indicates the number of steps remaining to the agent. This variable will be used as a counter in the while-loop, which is the core of the algorithm. In particular, if the number of steps remaining is one less than a multiple of $\lfloor\sqrt{N}\rfloor$, the case is similar to the one described in Figure 4.2, thus every intermediate value function up the target one are stored in memory (*tree* variable is set to true). If this is not the case, the value function array will be already stored in the stack. Finally, the value function array is loaded in variable V and used to compute the optimal policy

(*calculateBestAction* function), an action is decided and executed and, after the number of steps is decremented by one and is greater than zero, the agent repeats the process.

Algorithm 11 Square Root Solution

```
1: procedure rootSolution(N)
2:     V ← zeros(N)
3:     steps = N
4:     for i = 0 to N step ⌊√N⌋ do
5:         calculateValues(i + ⌊√N⌋, i, V, false)
6:     end for
7:     if indexStack.top() < N then
8:         calculateValues(N, indexStack.top(), valueStack.top(), true)
9:     end if
10:    while steps > 0 do
11:        if indexStack.empty() then
12:            calculateValues(steps, 0, zeros(N), false)
13:        else if ((steps + 1)mod⌊√N⌋) == 0 then
14:            calculateValues(steps, indexStack.top(), valueStack.top(), true)
15:        end if
16:        V ← valueStack.top()
17:        chosenAction = calculateBestAction(V[currentState])
18:        takeAction(chosenAction)
19:        steps = steps − 1
20:    end while
21: end procedure
```

Observing the above algorithms, the theoretical results regarding the complexities are confirmed. The time complexity of $calculateValues$ is $\mathcal{O}((targetIndex - startingIndex)|S|^2|A|)$, as the outer for-loop depends on the range of computation. In $rootEvaluation$, the first for-loop is executed $\sqrt{N}$ times, while each one of those is of the same time complexity as $calculateValues$ when the range is $\sqrt{N}$, resulting in a total time complexity of $\mathcal{O}(\sqrt{N}\sqrt{N}|S|^2|A|) = \mathcal{O}(N|S|^2|A|)$. Additionally, inside the while-loop, $calculateValues$ is called $\sqrt{N}$ times for intervals of range $\sqrt{N}$, resulting in a term of same time complexity as the former. Consequently, the total time complexity of the algorithm is $\mathcal{O}(2N|S|^2|A|) = \mathcal{O}(N|S|^2|A|)$. Regarding the space complexity, at most $2\sqrt{N}$ value function arrays of length $|S|$ will be stored in memory at any given point, plus their corresponding indices (a total of $2\sqrt{N}$ indices of space complexity $\mathcal{O}(1)$), confirming the total space complexity of $\mathcal{O}(\sqrt{N}|S|)$.

4.2.2 Logarithmic Solution

In cases where space complexity is of the essence, while the execution time needs to be reasonable (with some tolerance), an especially efficient solution is the Logarithmic Solution (or Tree Solution).

The foundation of this idea is the Square Root Solution, discussed in 4.2.1, along with the desire to lower the memory needs even further. This lead to the realization that even less value function arrays need to be stored. a promising asymptotic function with respect to the horizon, indicating the number of value function arrays to be stored, is the binary logarithm of the horizon, or in symbols $\log_2 N$.

In order to better comprehend the solution to be described, we can visualize mentally that the value function arrays are stored in a Binary Search Tree [CLRS09]. Every node of this tree contains a key, which is responsible for the ordering of the nodes, which is equal to the value function array's index. During the computation of a value function array V, only the intermediate value function arrays whose indices lie in the path from the root to the index of V need to be stored. The longest path from the root to a node of the tree is of (at most) length $\log_2 N$ (where N is the number of nodes, in this case the horizon's length), leading (at most) $\mathcal{O}(\log_2 N)$ value function arrays stored at any given point in time. Note that this is achieved only in the case where every level of the tree is full (except maybe the last one). Such a tree can be found in 4.3, with a horizon of length 16.

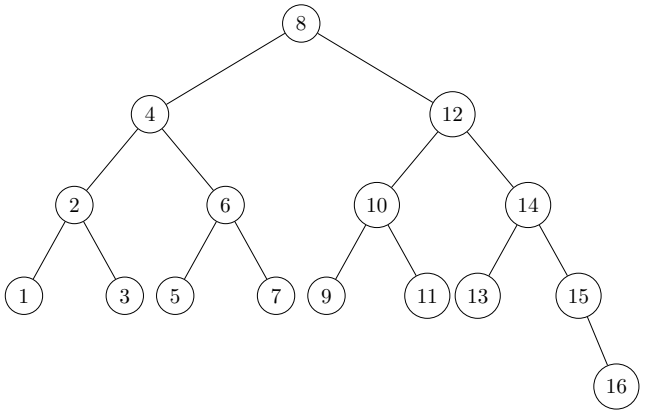


Figure 4.3: Binary tree representing the ordering of array's indices.

We will be examining the example of the complete execution of the Logarithmic Algorithm, in the case of a horizon of length 16. The steps described below are represented graphically in Figure 4.4. Coloring a node with green indicates that the value function array with the corresponding index is stored in memory and ready to be used. On the other hand, the color yellow represents that the corresponding value function array is currently used to compute the optimal policy for this number of steps remaining and will be deleted from memory after an action is decided and executed. Finally, we color white the nodes whose corresponding value function arrays are neither stored in memory, nor

used for calculating the optimal policy. Note that, any computation from those described below uses the value function array with the greatest index stored in memory.

1. We compute up to the value function array of index 8 and we store it in memory
2. We compute up to the value function array of index 12 and we store it in memory
3. We compute up to the value function array of index 14 and we store it in memory
4. We compute up to the value function array of index 15 and we store it in memory
5. We compute up to the value function array of index 16 and we store it in memory
6. We use array of index 16 to calculate the optimal policy for 16 steps remaining and we delete it from memory afterwards.
7. We use array of index 15 to calculate the optimal policy for 15 steps remaining and we delete it from memory afterwards.
8. We use array of index 14 to calculate the optimal policy for 14 steps remaining and we delete it from memory afterwards.
9. We compute the value function array of index 13 and use it to compute the optimal policy for 13 steps remaining.
10. We use array of index 12 to calculate the optimal policy for 12 steps remaining and we delete it from memory afterwards.
11. We compute up to the value function array of index 10 and we store it in memory
12. We compute the value function array of index 11 and use it to compute the optimal policy for 11 steps remaining.
13. We use array of index 10 to calculate the optimal policy for 10 steps remaining and we delete it from memory afterwards.
14. We compute the value function array of index 9 and use it to compute the optimal policy for 9 steps remaining.
15. We use array of index 8 to calculate the optimal policy for 8 steps remaining and we delete it from memory afterwards.
16. We compute up to the value function array of index 4 and we store it in memory

17. We compute up to the value function array of index 6 and we store it in memory

18. We compute the value function array of index 7 and use it to compute the optimal policy for 7 steps remaining.

19. We use array of index 6 to calculate the optimal policy for 6 steps remaining and we delete it from memory afterwards.

20. We compute the value function array of index 5 and use it to compute the optimal policy for 5 steps remaining.

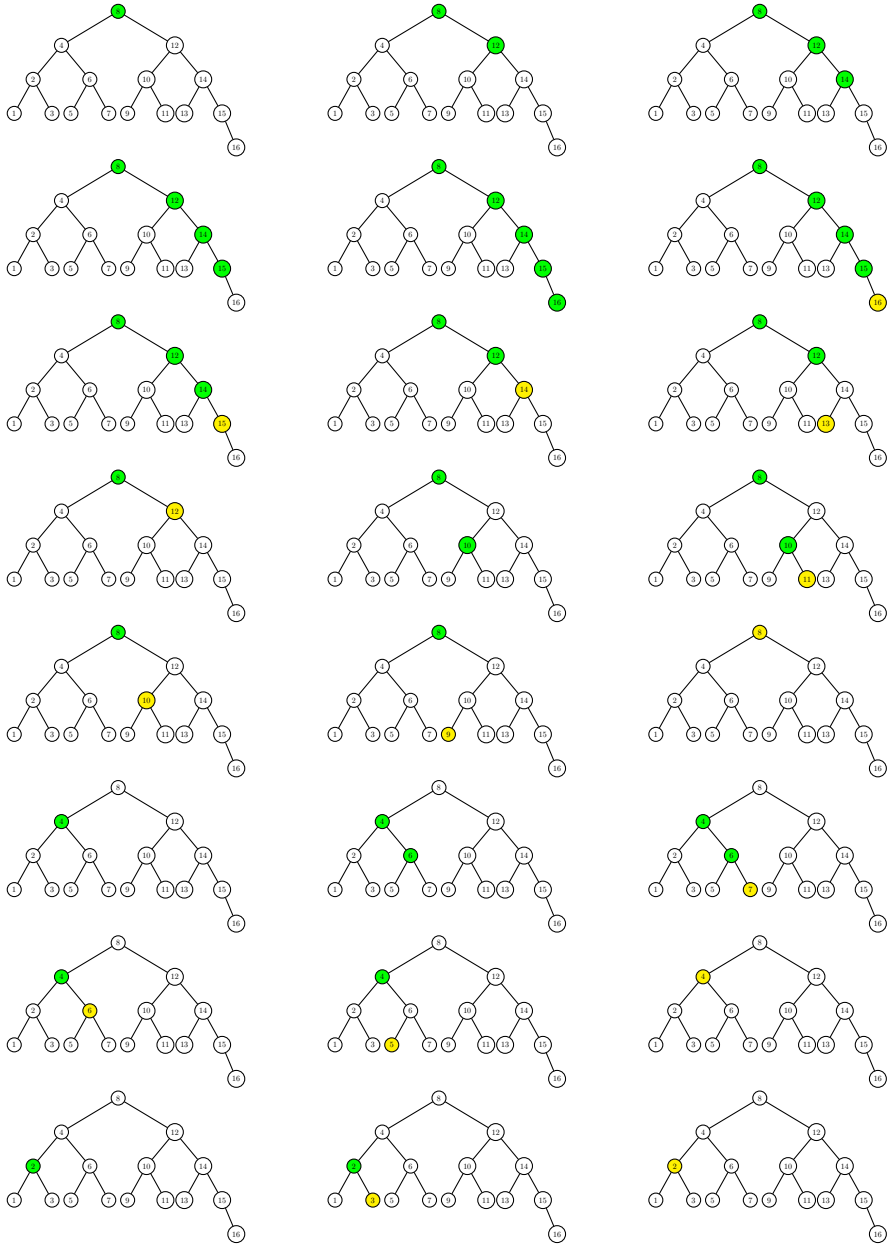
21. We use array of index 4 to calculate the optimal policy for 4 steps remaining and we delete it from memory afterwards.

22. We compute up to the value function array of index 2 and we store it in memory

23. We compute the value function array of index 3 and use it to compute the optimal policy for 3 steps remaining.

24. We use array of index 2 to calculate the optimal policy for 2 steps remaining and we delete it from memory afterwards.

25. We compute the value function array of index 1 and use it to compute the optimal policy for the final step.



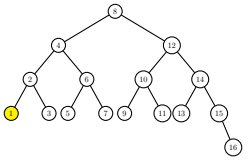


Figure 4.4: Graphical representation of the Logarithmic Solution for the case of horizon of length 16. Each of the graphs corresponds to a step described above. The nodes corresponding to value function array stored in memory are colored green, while those who are currently in use are in yellow.

The algorithm described above stores at most $\lfloor \log_2 N \rfloor$ arrays of length $|S|$ at once at any given point, in other words the same as the height of the imaginary Binary Search Tree, resulting in a space complexity of $\mathcal{O}(|S| \log_2 N)$.

This idea is also contained in Algorithm 13 in pseudocode. It makes use of the $calculateValues$ function already defined in 4.2.1, as well as a new auxiliary function, named $treeTraversal$, aiming to find the indices of the value function arrays that need to be stored during each point of the execution. In particular, this function follows the same logic as Binary Search [CLRS09], using variables l, r as indices for the left and right end of the subarray. Variable k moves as if the key searcing variable in a Binary Search. If there is a value function array already stored in the stack, the search will "continue" from this array's index, as can be seen in line 13, while, if the stack's top is the value function we are trying to calculate, the search is over and this array is returned (lines 8 through 11). If, however, the search is being carried out as normal and pointer l is not greater than pointer r, the algorithm can continue in one of three ways:

- If pointer k is the **requested one**, then we compute the value function we need using $calculateValues$, without storing any intermediate results in memory, starting from the last stored value function array (namely the one with the largest index or none if none is stored). Afterwards, the computed value function array is returned and the function terminates.

- If pointer k is **smaller** than the requested, the value function array corresponding to pointer k will be stored (because it lies on the path from the tree's root to the requested node in the imaginary Binary Search Tree), and it is thus calculated using $calculateValues$, similarly to the previous case and stored. Afterwards, pointer l moves one step to the right of k, while k is updated to the arithmetic mean of l and r. If l is still not greater than r the process is repeated once again.

- If pointer k is **greater** than the requested, the value function array corresponding to k is not on the path from the root to the requested node in the imaginary Binary Search Tree, thus pointer r is set to one position left of pointer k, while k is updated to the arithmetic mean of l and r. If l is still not greater than r the process is repeated once again.

After the while-loop is terminated, the function returns array V_{tmp} containing the requested value function.

Algorithm 12 Auxiliary Function to store Value Function Arrays

```
1: function treeTraversal(targetIndex, N)
2:     l ← 0
3:     r ← N
4:     k ← (l+r)/2
5:     V_tmp ← []
6:     if notindexStack.empty() then
7:         if indexStack.top() == targetIndex then
8:             V_tmp ← valueStack.top()
9:             valueStack.pop()
10:            indexStack.pop()
11:            return V_tmp
12:        else
13:            k ← indexStack.top()
14:        end if
15:    end if
16:    while l ≤ r do
17:        if k == target then
18:            if indexStack.empty() then
19:                calculateValues(k, 0, zeros(N), false)
20:            else
21:                calculateValues(k, indexStack.top(), valueStack.top(), false)
22:            end if
23:            valueStack.pop()
24:            indexStack.pop()
25:            V_tmp ← valueStack.top()
26:            break
27:        else if k < target then
28:            if indexStack.empty() then
29:                calculateValues(k, 0, zeros(N), false)
30:            else if indexStack.top()! = k then
31:                calculateValues(k, indexStack.top(), valueStack.top(), false)
32:            end if
33:            l = k + 1
34:            k = (l+r)/2
35:        else
36:            r = k − 1
37:            k = (l+r)/2
38:        end if
39:    end while
40:    return V_tmp
41: end function
```

Utilizing the above algorithm, the decision making process at each step of the horizon is greatly simplified as can be observed in Algorithm 13. Similarly to

the previous solutions, variable *steps* indicates the number of steps remaining to the agent. For each one of them, the requested value function is computed, using $treeTraversal$ and used to calculate the optimal policy and thus the optimal action.

Algorithm 13 Logarithmic Solution

```
1: procedure TREESOLUTION(N)
2:     steps ← N
3:     V ← []
4:     while steps > 0 do
5:         V ← treeTraversal(steps, N)
6:         chosenAction = calculateBestAction(V[currentState])
7:         takeAction(chosenAction)
8:         steps = steps − 1
9:     end while
10: end procedure
```

In order to acquire the time complexity of the Logarithmic Solution, we must first prove the following Lemma:

Lemma 4.1. *In the Logarithmic Solution with horizon N, when calculating an array of index X, the number of elementary calculations required is always equal to the number of nodes in the left subtree whose root is the node with key $X+1$ in the imaginary Binary Search Tree, except for the array with index N.*

Proof. By elementary calculation we refer to calculating a value function array of index k using a value function array of index $k-1$, which can be completed in $\mathcal{O}(|S|^2|A|)$ time. As we already established, the Logarithmic Solution follows the logic of Binary Search when the calculation of an array with index X is necessary. In any Binary search performed in an array of length N starting from number k and containing every number up to $k+N$, for any number X the following equation is true:

$$X = l_{last}(X) + \frac{N}{2^{iter(X)}} \tag{18}$$

By $l_{last}(X)$ we symbolize the left index of the last interval before finding X, while by $iter(X)$ we denote the number of iterations of Binary Search required to find X. This equation holds because every time we iterate, the length of the array is cut in half, as is denoted by the denominator of the fraction, while in the last step (in which X is actually found) we find X just to the right of $l_{last}(X)$.

As shown in the Logarithmic Solution's explanation, this algorithm performs a Binary Search every time an array needs to be calculated. Every time the index pointer of the Search moves to the right, the array to whose index the left pointer points is stored in memory. This implies that, after calculating an array of index X, on the top of the value function stack lies the value function

array whose index is the left end of the last interval before finding X, or in other words $l_{last}(X)$.

Rearranging Equation 18 we obtain:

$$\begin{aligned} l_{last}(X) &= X - \frac{N}{2^{iter(X)}} \\ X - 1 + 1 - l_{last}(X) &= \frac{N}{2^{iter(X)}} \\ (X-1) - l_{last}(X) &= \frac{N}{2^{iter(X)}} - 1 \end{aligned} \tag{19}$$

The term on the left hand side of this equation is exactly equal to number of elementary calculations that should be performed when requiring the array of index $X-1$, starting from the last array stored in memory after having calculated the array of index X in the previous step, which is precisely $l_{last}(X)$. The term on the right is exactly equal to number of children contained in the left subtree whose root's index is X in the imaginary Binary Search Tree, as the number of iterations to find X using a Binary Search is equal to the depth of X in the Binary Search Tree (with the root being at depth 1 instead of 0). Thus, we proved that the number of elementary calculations needed in order to calculate the array of index $X-1$ using the Logarithmic Solution is equal to number of children nodes in the left subtree whose root is the node with index X. □

Using the above Lemma, the algorithm's time complexity can be calculated as follows: firstly, the array of index N is calculated (N being the horizon's length), storing every necessary intermediate array in memory. Afterwards, for every subsequent calculation, the number of elementary calculations performed is equal to the number of children in the left subtree whose root is the node representing the value function array calculated in the previous step. Every level of the tree contains $2^(i+1)$ nodes, where i is the depth of the tree at each level, with the root being at level 0 (except maybe the last level). The tree contains a total of $\log_2(N)$ levels, as every level is full, except for maybe the last. The total number of elementary calculations is given by:

$$
\begin{aligned}
T(N) &= \sum_{i=0}^{\log_2(N)} 2^i \left(\frac{N}{2^{i+1}} - 1\right) \\
&= \sum_{i=0}^{\log_2(N)} \left(\frac{N}{2} - 2^i\right) \\
&= \frac{N \log_2(N)}{2} + \sum_{i=0}^{\log_2(N)} 2^i \\
&= \frac{N \log_2(N)}{2} + \frac{2^{\log_2(N)+1} - 1}{2 - 1} \\
&= \frac{N \log_2(N)}{2} + 2N - 1 \\
&= \mathcal{O}(N \log_2(N))
\end{aligned}
\tag{20}
$$

Each of these elementary calculations requires $\mathcal{O}(|S|^2|A|)$ time, resulting in a total time complexity of $\mathcal{O}(|S|^2|A|N \log_2(N))$

5 Description of Implementation

In this Section, we will be describing how the environment to test the algorithms mentioned in the previous Sections was set up. We will be discussing the use case of our model, how it was implemented, difficulties encountered along with their solutions, as well as methods used for measuring the needed performance metrics.

5.1 Use Case

The use case we chose for our problem was the one analysed in depth in [LKKK17]. The problem introduced in the above paper regards a computer cluster, which houses a distributed database. This cluster constantly receives read requests, while the number of Virtual Machines, of which it consists, is variable and can be changed by the agent depending on the needs of the system (elasticity property). As a result, the only actions the agent can make are adding a VM, removing a VM or doing nothing. The number of states the model contains varies, depending on the maximum and minimum number of VMs that can exist in the cluster at any time as well as the incoming load. The agent acts a coordinator of the cluster, adjusting the number of active VMs in the cluster as needed to both serve the incoming load and do so efficiently. Every state the model can be at is expressed by a set of parameters, such as the total load and the number of Virtual Machines. Using that information, the agent learns the system's parameters, trains on them and makes the optimal adjusting decision every time it is required (indefinitely in theory).

The authors compared 4 different algorithms, two preexisting and two newly-introduced, to solve the MDP created to simulate the above problem. The first two algorithms are the classic MDP (using Value Iteration in an Infinite Horizon) and Q-Learning, while the two new approaches where both versions of the other two using Decision Trees. We based our work on what was implemented regarding the classic MDP approach with some changes to make the algorithm better fit our approach.

First and foremost, it should be noted that, while the algorithm used in the initial experiments was treated as an Infinite-Horizon MDP solving algorithm (Value Iteration), it actually ran for a finite amount of steps. It was assumed that the agent did not know the horizon's length in advance and thus they could not prepare for it accordingly. In our experiments, we treated the problem as a Finite-Horizon problem, with the agent having prior knowledge about the number of steps they are required to make.

Another important issue was the fact that the agent's training was happening not only throughout the training steps but also during the evaluation. This is of course a valid model-based learning approach, rendering the model partially observable. We, on the other hand, treated the model as fully observable, as we had the agent learn the system's parameters during the training phase as normal, but believe that those parameters are constant and will not change throughout the evaluation period.

Finally, changes were made in the model's structure, regarding both time efficiency and convenience. These are discussed in depth in the following Sections.

In our case, which closely follows the work made in [LKKK17], we assumed that:

- The cluster size can vary anywhere from 1 to 20 VMs
- The only actions available at any point are: increase the cluster size by 1, decrease the cluster size by 1 or do not perform any operation. It goes without saying that those actions are available to the agent only when they can actually be performed. If for example there are already 20 VMs in the cluster, the action "Add 1 VM" is no longer available.
- The incoming load before training is in the form of a sinusoidal function, namely: $load(t) = 50 + 50\sin(\frac{2\pi t}{250})$, while in the evaluation period its frequency doubles.
- The percentage of the incoming load that is read requests is given by a different sinusoidal function: $r(t) = 0.75 + 0.25\sin(\frac{2\pi t}{340})$.
- The RAM size is 1024 for the first 220 steps, then 2048 for the next 220. This pattern continues for any number of steps.
- The I/O operations per second are also given by a sinusoidal function of time: $io(t) = 0.6 + 0.4\sin(\frac{2\pi t}{195})$

- The capacity of the cluster at a given point in time t is given by: $capacity(t) = (10r(t) - io\_penalty - ram\_penalty)vms(t)$, where $vms(t)$ is the number of VMs currently in the cluster at time t. The parameter $io\_penalty$ is 0 when $io < 0.7$, $10io(t) - 0.7$ when $0.7 \leq io(t) \leq 0.9$ and 2 otherwise. Finally, $ram\_penalty = 0.3$ when the RAM size is 1024 and 0 otherwise.

- The reward after executing an action is given by:

  $$reward(t) = \min(capacity(t+1), load(t+1)) - 2vms(t+1)$$

  This implies that the agent is rewarded greatly when the capacity of the system is enough to fully serve the incoming load, while it is penalized if it over- or under-delivers.

All the above assumptions were implemented into a class named *Complex*, as either functions or members. A useful member of this class was the current time step, incremented after every action and used in the above functions. The most important method of *Complex* is the *execute_action* function, which, when given an *action* as input argument (an *action* refers to a pair of string and int, with the first being either "ADD", "REMOVE" or "NO_OP" and the latter "1" or "0") executes it, calculates the current measurements of the system and returns the corresponding reward.

5.2 Classes

Before defining the actual MDP model, we must first discuss the components of which it consists. The *MDPModel* class consists of abstract objects, called *States*, representing the model's states. Each of these *States* contains one *QState* object for each of the possible actions the agent can make while they are at it. This structure was also utilized in [LKKK17], we, however, translated the code from Python to C++, making some changes for convenience and performance. We also built the *FiniteMDPModel* class upon the *MDPModel* as a derived class extending the latter.

5.2.1 QState

We shall first analyze the *QState* object. Its members are:

- pair(int,string) *action*: The action this *QState* refers to. The string could be "ADD", "REMOVE", "NO_OP", while the integer could be 0 or 1.

- int *num_taken*: The number of times this action has been chosen.

- float *qvalue*: The current value of the state-action value function for this state-action pair.

- vector(int) *transitions*: Each index refers to a state, while the value at an index expresses the time this state-action pair has led to this particular state (the one of the index).

- vector(float) *rewards*: Each index refers to a state, while the value at an index expresses the sum of rewards the agent received each time it chose this action and transitioned to a particular state.
- int *num_states*: The total number of states of the model.
- vector(int) **transtate**: Contains the state IDs the agent can transition to.
- vector(float) **trans**: Contains the transition probability to the state corresponding to the same index in *transtate*.

The members written in plain text were present in the implementation used in [LKKK17], while those written in bold were added in our implementation. *transtate* is a vector used to keep track of the possible states this action might lead, without containing information about every single state, resulting in lower vector length. The same goes for *trans*, which contains the transition probabilities corresponding to the states in *transtate*.

$Qstate's$ methods are:

- void *update*(*int state_num*, *float reward*): Updates num_taken, transitions and rewards when the action is taken.
- pair(string,int) *get_action*(): Returns the action of this *QState*.
- $floatget\_qvalue()$: Returns the value of the state-action value function for this *Qstate*.
- boolean *has_transition*(*int state_num*): Returns true if there is a transition to the state whose ID is equal to state_num.
- int *get_num_transitions*(*int state_num*): Returns the number of times the agent has transitioned to the state whose ID is equal to the state_num after choosing this action.
- float *get_reward*(*int state_num*): Returns the reward the agent gains after transitioning to the state whose ID is equal to the state_num.
- float **get_reward**(*int state_num*, *int time_step*): Returns the time-dependent reward the agent gains after transitioning to the state whose ID is equal to the state_num.
- void *set_qvalue*($float\ qvalue$): Sets the $Qstate's$ qvalue equal to the input argument.
- int *get_num_taken*(): Returns the number of times this action was chosen.
- vector(int) *get_transitions*(): Returns the transitions vector.
- vector(float) *get_rewards*(): Returns the rewards vector.

The *get_reward* function written in bold is overloading the preexisting *get_reward* function to extend its functionality to time-dependent rewards. The original *get_reward* function returns the average reward the agent has received when taking a certain transition (to a state whose state ID is given by *state_num*) or zero if this transition has never been taken. The overloaded function takes an extra argument, *time_step*, which expresses the decision epoch for which we would like to know what the reward is. To simulate a time-dependent reward, we chose a Round-Robin method, meaning that at every time step only one of the available $QState's$ rewards is changed. At any given point in time, for a particular *State* and action, one of the available transitions is chosen to not return a reward. This transition leads the agent to the *State* whose *state_num* is equal to *time_step* mod $transtate.size()$. The modulo operator is what ensures that the variability of the rewards affects only one transition at a time in a Round-Robin manner.

5.2.2 State

The next important abstract class is the *State* class. A *State* captures the essence of the model's state, holding important information about it. It's members are:

- vector(QStates) *qstates*: Holds every *QState* of this state.
- int *state_num*: Unique int for every state, also referred to as state's ID.
- int *num_states*: The total number of states of the model.
- float *value*: The value of the value function for this state.
- int *best_qstate*: Index of the *QState* with the maximum qvalue in the *qstates* vector
- boolean **isBestQStateSet**: Indicates whether the best *QState* has been found.
- int *num_visited*: The number of times this state has been visited by the agent.
- map(sting, pair(int, float) *parameters*: Contains the system's parameters in this state.

The *isBestQStateSet* variable indicates whether the best *QState* of the model (the one with the maximum qvaue) has been calculated and set because, if *best_qstate* was not set, this would create problems in a programming language such as C++ in a translated implementation of functions.

A *State* contains of the following methods:

- void $visit()$: Increments *num_visited* by 1.
- int $get\_state\_num()$: Returns this state's *state_num* (ID).

- void *set_num_states*(*int num_states*): Updates *num_states* to equal the input argument.
- float *get_value*(): Returns this state's value.
- int *get_best_qstate*(): Returns this state's *best_qstate*.
- pair(string,int) *get_optimal_action*(): Returns the action of this state's best *QState*.
- int *best_action_num_taken*(): Returns the number of times the optimal action was taken.
- void *update_value*(): Finds and sets the *best_qstate* of this *State* and updates its *value* to equal the corresponding *qvalue*.
- map(string, pair(float, float)) *get_parameters*(): Returns this $State's$ *parameters*.
- void *add_new_parameter*(*string name*, $pair(float, float)$ *value*): Adds a new parameter to this *State*.
- pair(float, float) *get_parameter*(*string name*): Returns the value of a certain input parameter.
- void *add_qstate*(*QState q*): Adds a new *QState* to the *qstates* vector.
- vector(QStates) *get_qstates*(): Returns this $State's$ *qstates*.
- *Qstate *get_qstate*(*pair*(*string*, *int*) *action*): Returns a reference (pointer) to this $State's$ *QState* whose action is equal to the input argument
- vector(pair(string,int)) *get_legal_actions*(): Returns all actions of this $State's$ *qstates*.

As one can see, no additions to the model were needed, just some modifications caused by the changes in some of the $State's$ member's types described earlier.

5.2.3 MDPModel

Having defined both the *QState* and *State* class we are ready to analyze the fundamental class which simulates the system and executes the solving algorithms, the $MDPModel$. The $MDPModel's$ members are:

- float *discount*: The discount factor used in Value Iteration.
- vector(States) *states*: Contains the *States* of the model.
- int *current_state_num*: The *state_num* (ID) of the *State* the agent is currently at.

- int *initial_state_num*: The *state_num* (ID) of the *State* the agent was initially at after training.
- JSON *parameters*: The model's parameters.
- float *update_error*: The threshold for terminating Value Iteration.
- int max_VMs: Maximum number of VMs the cluster can have.
- int min_VMS: Minimum number of VMs the cluster can have.

The members of this class all exist in the initial implementation described in [LKKK17] and no new members were needed. The methods of $MDPModel$ are analyzed below in detail:

- JSON $\_get\_params(JSON\ pars)$: Returns a JSON object containing the input parameters in an appropriate form.
- void $set\_state(JSON\ measurements)$: Sets the $current_state_num$ to that of the *State* whose parameters correspond to the input measurements.
- void $\_update\_states(string\ name,\ JSON\ new\_parameter)$: Creates a new *State* for every combination of the old *states* and new_parameter values. Then, sets the *states* member equal to a vector containing all the newly created *States*.
- int $\_get\_state(JSON\ measurements)$: Returns the *state_num* of the *State* of the model whose parameters correspond to the input measurements.
- void $\_set\_maxima\_minima(JSON\ parameters,\ JSON\ acts)$: If the acts JSON contains the "add_VMs" or "remove_VMs" action(s), set the max_VMs and min_VMs members equal to the corresponding maximum and minimum values in the parameters argument.
- void $\_add\_qstates(JSON\ acts,\ float\ initq)$: For each of the actions in acts creates a *QState* in each of the *States* (if the action is permissible), with a *qvalue* initialized to initq. Then, executes *update_value* for every *State* of the model.
- boolean $\_is\_permissible(State\ s,\ pair(string, int)\ a)$: Checks if the given action a is possible in *State* s.
- pair(string,int) *suggest_action*(): Returns the optimal action of the *State* with *state_num* equal to *current_state_num*.
- vector(pair(string,int)) *get_legal_actions*(): Returns a vector containing all possible actions of the *State* with *state_num* equal to *current_state_num*.

- void *update*(*pair*(*string*, *int*) *action*, *JSON measurements*, *float reward*): Run after choosing an action during training. Visits the current *State*, finds the next *State*, performs *update* and *_q_update* to the *QState* chosen and transfers the agent to the new *State*.

- void **update**(*pair*(*string*, *int*) *action*, *JSON measurements*, *float reward*): Run after choosing an action during evaluation. Finds the next *State* and transfers the agent to the new *State*.

- void *_q_update*(*QState qstate*, *vector*(*float*) *V*): Updates the *qvalue* of qstate argument using the Bellman Equations with discount factor equal to the *discount* member.

- void **_q_update**(*QState qstate*, *vector*(*float*) *V*, *int time_step*): Updates the *qvalue* of qstate argument using the Bellman Equations without discount (Finite-Horizon), while if the time_step argument is given the reward included in the calculation is time-dependent.

- void *value_iteration*(): Executes Value Iteration on the model using the *_q_update* function on all *QStates* of every *State* and the *update_value* method of every *State*. Terminates when the Bellman Residual is smaller than the *update_error* member.

- void **getStateOnlyValues**(*vector*(*float*) *V*): Inserts the *values* of every *State* of the model into V in the corresponding index.

- vector(pair(int,float)) **getStateValues**(*vector*(*State*) *V*): Returns a vector of pairs containing the *best_qstate* (first term of pair) and *value* (second term of pair) members of every *State* in input vector V.

- void **loadValueFunction**(*vector*(*pair*(*int*, *float*)) *V*)): The first term of every pair in the input vector corresponds to a *state_num* and the second to a *value*. Each of the model's *State*$'$*s best_qstate* and *value* is set to the corresponding index pair's values.

- void **loadBestQStates**(*vector*(*int*) *V*): Set every model's *State*$'$*s best_qstate* to the corresponding value in V.

Again, those methods written in bold are the ones added on top of the initial model, as they were deemed useful for the *FiniteMDPModel*$'$*s* methods, described next. The initial model had even more methods and members, which were not included in our version since they could not be of use to us from our theoretical point of view.

5.2.4 FiniteMDPModel

All of the algorithms described in Section 4 were implemented in a new class extending *MDPModel*, called *FiniteMDPModel*. This class naturally inherits every member and method of the base class *MDPModel*. The new members *FiniteMDPModel* contains are:

- stack(int) *index_stack*: A stack used to contain the indexes of the value function vectors stored in memory throughout each of the MDP solving algorithms' execution.
- stack(vector(pair(int,float))) $finite\_stack$: A stack used to contain the value function vectors needed to execute the MDP solving algorithms'. Each of the vectors are of length $|S|$ and contain both the *value* of the corresponding *State* and its *best_qstate*.
- stack(vector(int)) *action_stack*: A stack used to contain the *best_qstate* vectors needed throughout the execution of the Naive MDP algorithm. It was deemed more memory efficient to use a stack containing only the *best_qstate* (equally, the optimal action) of every state and for every number of steps remaining in the case of the Naive algorithm, as no value function will be re-used throughout its execution.
- float *total_reward*: The total reward the agent collects during evaluation.
- int *steps_made*: The number of steps the agent has already made during evaluation.
- float *expected_reward*: The total reward the agent is expected to make starting from the initial *State* they are before evaluation begins. Is initialized using a seed passed to the $FiniteMDPModel's$ constructor as an argument.
- default_random_engine *eng*: A random engine used to produce pseudo-random numbers throughout the experiments.
- uniform_real_distribution(float) $unif$: The distribution which the pseudo-random numbers follow. In this case it is a uniform distribution from 0 to 1, meaning the number produced each time has an equal chance of being any floating point number between 0 and 1.
- int *max_memory_used*: The maximum amount of memory used throughout the execution of one of the algorithms (value either in bytes or kilobytes, depending on the operating system).
- int *init_memory_used*: The initial amount of memory occupied by the model at the beginning of every experiment (value either in bytes or kilobytes, depending on the operating system).

More information about the pseudo-random numbers generating methods can be found in [RNG]. The methods of this class are:

- void $checkMemoryUsage()$: Reads the current value of memory occupied by the process and updates *max_memory_used* accordingly.
- void $\_q\_update\_finite(QState\ q,\ vector(pair(int, float))\ V)$: Updates the *qvalue* of q argument using the Bellman Equations without discount. V argument contains *best_qstate* - *value* pairs for each *State*.

- void *_q_update_finite* (*QState q*, *vector*(*pair*(*int*, *float*)) *V*, *int time_step*): Updates the *qvalue* of q argument using the Bellman Equations without discount using variable rewards (time-dependent). V argument contains *best_qstate* - *value* pairs for each *State*.

- float *calcReward*(*vector*(*float*) *V*): Calculates the average of the values contained in V (sums all values and divides by $|S|$). This is later used to calculate value functions for *States* for which the agent got no information from training (univisited).

- float *calcReward*(*vector*(*pair*(*int*, *float*)) *V*): Calculates the average of the values contained in V's second term of each pair (sums all values and divides by $|S|$). This is later used to calculate value functions for *States* for which the agent got no information from training (univisited).

- void *takeAction*(*int corrAction*, *int time_step*): Used to execute an action. Randomly calculates which transition the agent will follow by choosing input action corrAction, transitions them to the next state and adds the reward to the model's *total_reward*. If the time_step argument (optional) is not set, the reward given is not time-dependent. Otherwise, the time-dependent version of *get_reward* is used.

- void *calculateValues*(*int k*, *int starting_index*, *vector*(*pair*(*int*, *float*)) *V*, *boolean tree*): Executes *calculateValues* algorithm (see Algorithm 10 in Section 4.2.1), with *valueStack* and *indexStack* in the pseudocode being *finite_stack* and *index_stack* members of the model respectively. In addition, the *value* of *States* never visited is set to the result of *calcReward*.

- float *calculatePolicy*(*int horizon*): Used by the Naive algorithm, its functionality is almost identical to that of *calculateValues*, the only difference being that it does not utilize the *index_stack* or *finite_stack*, but rather the *action_stack*, where it stores only the best actions of each value function vector. In addition, it returns the largest index (step) value function's value at the agent's initial *State*, which is equal to the expected reward of the agent.

- void *naiveEvaluation*(*int horizon*): Executes the Naive algorithm (see Algorithm 8 in Section 4.1.1). Firstly, *calculatePolicy* is called. Afterwards, for every time step left to the agent, an action is popped from *action_stack* and executed using *takeAction*.

- void *inPlaceEvaluation*(*int horizon*): Executes the In-Place algorithm (see Algorithm 9 in Section 4.1.2). Calls *calculateValues* once for the largest value of the horizon and uses it to set the *expected_reward* variable. Then, for every step left to the agent, it calculates the value function vector and thus the optimal action using *calcualteValues*, starting from index 0 and saving no intermediate result in memory.

- void *rootEvaluation*(*int horizon*): Executes the Square Root algorithm (see Algorithm 11 in Section 4.2.1). Calls $calculateValues$ for every multiple of the horizon's square root, always starting from the last index stored in memory, and saves every value function vector (whose index is a multiple of the square root of the horizon) calculated in the $finite\_stack$. If the horizon is not a perfect square, and thus some value function arrays have yet to be calculated, it calculates them and stores them in the stack. Then, it starts making actions using the vectors stored in memory. If the needed vector is not stored in memory, it calculates it starting from the largest indexed vector stored, and also storing every intermediate result. Every time and action is made, the corresponding vector is popped from the stack.

- void *treeTraversal*(*int target*, *int horizon*, $vector(pair(int, float))$ V): Executes the tree traversal algorithm which stores every value function vector whose index lies on the path from the imaginary tree's root to the target node (see Algorithm 12 in Section 4.2.2).

- void *treeEvaluation*(*int horizon*): Executes the Logarithmic algorithm (see Algorithm 13 in Section 4.2.2). For every step left to the agent, it calls *treeTraversal* to acquire the desired value function vector. It then uses it to take an action and removes if from the stack.

- void $infiniteEvaluation$(*int horizon*): Executes the first Approximation algorithm of the Finite-Horizon (with discount). Executes $value_iteration$ once and uses the stationary policy calculated to take every action, as many times as the horizon's length.

- void *turnpikeEvaluation*(*int horizon*): Executes the second Approximation algorithm of the Finite-Horizon (Turnpike). Calculates up to the value function vector whose index corresponds to the full horizon's length using $calculateValues$, without storing in memory any intermediate results. Then, every decision is made using this value function vector regardless of the step, for every step of the horizon.

- void *runAlgorithm*(*model_type alg*, *int horizon*): Executes the chosen algorithm depending on the model_type. Also measures performance metrics (execution time and memory) and prints them in an appropriate form.

The *model_type* mentioned in *runAlgorithm* is the name of a custome variable type, using the *enum* keyword of C++. Its values can be: $infinite$, *naive*, *inplace*, *root*, *tree* or *turnpike*. Each of the above evaluations uses $checkMemoryUsage$ at certain keypoints to get as accurate memory readings as possible.

5.3 Training

The model learns the system it simulates by experience, which is provided by the *Complex* class discussed earlier. Before training, a *Complex* scenario is

initialized using user-defined parameters. Afterwards, the $FiniteMDPModel$ is created and its current state is set by receiving measurements from the scenario. The user also provides the model with a number of training steps. For every step in the training steps, the model either selects a legal action at random (with probability ϵ) or simply selects the optimal action (with probability $1 - \epsilon$), as the ε-greedy method dictates. Then, the chosen action is executed in the scenario, which yields a reward using the *execute_action* function, as well as measurements about the current state of the system. The model is then updated using the *update* function, with the input arguments being the chosen action, the collected reward and the new measurements. Every 500 steps, the model executes a single Value Iteration in order for its policy to be up-to-date with the current information about the system.

After the training is completed, the *trans* and *transtate* vectors of the model are set, using those transitions whose transition probability is greater than zero. This is an optimization on the initial model, as will be experimentally verified in Section 5.5. Finally, the model runs the chosen MDP solving algorithm and prints the results.

5.4 Performance Metrics

In order to evaluate the performance of the MDP solving algorithms we utilized three different metrics: the total reward collected, the execution time and the memory occupied at runtime.

The total reward collected was measured by a member variable of the $FiniteMDPModel$ class, namely *total_reward*, as was established earlier. Each time the agent collected a reward during the evaluation period, it was added to its total reward. This metric will later be compared to the expected reward of the agent, also measured using a member of the class (*expected_reward*).

The execution time of each algorithm was performed using the Chrono C++ Library. More information about it can be found in [chr]. In the *runAlgorithm* function, before executing any algorithm, the variable *start* is initialized to the current date and time using a *high_resolution_clock* through the Chrono Library. After the selected algorithm has run, another variable named *stop* gets the value of the current date and time using the same method. The difference of $stop - start$ is stored in the *duration* variable and refers to the elapsed time in microseconds. Note that the *high_resolution_clock* is the clock with the smallest tick period implemented in this library, and should, in theory, yield the most accurate time measurement results.

The most complex measurement was by far the memory occupied measurement. The reason for this is that there is no known method to exactly measure how much memory a certain data structure occupies during runtime. A workaround we managed to perform was to measure the initial memory before any algorithm's execution and periodically re-measure the memory occupied at frequent points during the execution. Then, it was a matter of simply subtracting the maximum of those measurements from the initial memory occupied. Note that this method does not provide the actual amount of memory used by

the data structure containing the value function vectors, but rather an overview of the extra memory the algorithm needed to solve the MDP.

Implementing this measuring method differs between operating systems. An efficient solution for systems running Windows OS is the Process Status API, more specifically the $getProcessMemoryInfo$ function, described in greater detail in [psa]. This function is used to retrieve information about the process that called it. One particularly useful among them is the $WorkingSetSize$ is the size of the pages of virtual memory this process has or is using (in bytes).

On the other hand, a Linux system can refer to a file containing all information about a particular process, such as the one running the algorithms. This file is known as $/proc/[pid]/status$, where pid is the process ID of the process whose information we desire. In our case, $pid = self$. Among other information, the one containing the value of the Virtual Memory size used by the process is the one under $VmRSS$. The value is in kilobytes. This directory's documentation can be found in [lin]. Both of the methods mentioned above are considered relatively insufficient when used for accurate measurements, which is not the case in our experiments.

For better interpretation of the experimental results presented in the next Section, we should discuss how the system's memory is organised. In a typical operating system, each process does not have immediate access to physical memory, but rather to a memory management concept known as Virtual Memory. This concept provides the illusion that the memory available to a process is much larger than the actual physical one [Dei90]. Each physical address is mapped to a virtual address using what is known as a paging table. A page is the smallest unit of data for memory management in such a Virtual Memory system, being a contiguous block of virtual memory. When a process needs access to memory, the memory the system provides it is in the form of pages. This fact will be useful in interpreting the experimental results of the next Section, where the memory measured is always a multiple of the page size, signifying that pages are given to a process whether they actual use it as a whole or not.

The experiments were conducted on a Linux server with Intel(R) Xeon(R) E5-2687W v3 @ 3.10GH CPU and 378 Gigabytes of RAM, while C++11 was used.

5.5 Execution Time Improvement

Observing the initial model used in [LKKK17] revealed that optimizations could be made regarding the actual execution time of the model's algorithms. In particular, the data structure used to iterate over a single state's transitions was the *transitions* vector, containing the number of transitions to each state, even when this number is zero for the majority of them. After training, there are many states which are never visited, resulting in an immediate reward of zero for any transition from them and transition probabilities $\frac{1}{|S|}$. We also noticed that the more states the model has the more sparse it becomes, as most never get visited. The addition of the *transtate* and *trans* vectors made it so that the number of transitions the algorithms iterate over is much smaller than $|S|$.

Furthermore, regarding the non-visited states, we know that the first part of their value function referring to the immediate reward will always be 0, as no information is known about their immediate rewards. As for the second part which involves the value function of neighboring states, it will be the average of all other states of the model, which can be calculated once and reused.

Implementing the above optimizations into the model led to great execution time reduction, as can be seen in Figure 5.1, for the case of 200 and 2000 states.

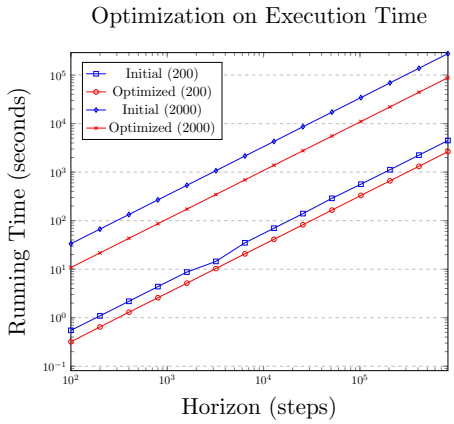


Figure 5.1: Total running time in seconds of the Finite-Horizon algorithm for 200 and 2000 states. The curves colored in blue represent the initial method of implementation, while those colored red are the optimized ones.

In Figure 5.1 the difference between the initial and optimized model is apparent, especially in the case of 2000 states. As the states of the model grow larger, so does the sparsity of those visited, resulting in greater execution time improvements using this optimization technique.

6 Experimental Results

Having established the fundamentals of our implementation, experiments were conducted in order to validate the theoretical results of Section 4. The three categories on which the algorithms competed against each other regard the total reward collected, the total execution time and the memory occupied.

6.1 Models to be compared

First and foremost we must summarize what algorithms will be compared, which include both the preexisting methods as well as the newly proposed ones. Two infinite approximation methods were also included for consistency, based on the Turnpike theorem mentioned in Section 2.7.

6.1.1 First Approximation of the Finite-Horizon

The Finite-Horizon MDP approaches to be compared vary in time as well as space complexity. The first approach is an approximation of the Finite-Horizon, using a type of Infinite-Horizon MDP. In particular, after the training is completed and the model is fully aware of the system and its parameters, a single Value Iteration (until convergence) is executed. The agent then utilizes the values computed by this algorithm to make every decision for a finite number of steps.

This method is an obvious approximation, balancing between infinite and finite horizon. The Infinite-Horizon Value Iteration is used to calculate the values for every state, because of the small time complexity (it is assumed convergence happens in much less steps than the length of the horizon) and small space complexity, as only one list of values for every state needs to be saved in memory and used for every decision. This results in a total space complexity of $\mathcal{O}(|S|^2|A|)$ treating the convergence as almost instantaneous and a total space complexity of $\mathcal{O}(|S|)$.

Of course this shortcut does not come without consequences. Because the agent calculated the values as if the horizon is infinite without prior knowledge of its actual length, it is bound to collect a much smaller reward than the true Finite-Horizon approach, especially for smaller horizon values. In addition, the agent uses the same "guidelines" to make its decisions for every step, without updating them every time the remaining steps get smaller, leaving itself prone to loops of the same rewards when it could be acquiring larger rewards by following an other small path for the final few steps.

It should be noted that, as γ approaches 1, the accuracy of this approximation gets higher. This can be interpreted as an approximation to a Finite-Horizon Value Iteration (which can be considered having γ exactly equal to 1) with a horizon tending to infinity. Consequently, this affects the rate of convergence, which gets significantly slower the closer γ gets to 1.

This method is also expected to not yield optimal results in the case of Non-Stationary MDPs, as it returns a stationary policy. In the case of NSMDPs, the policy must be non-stationary, taking the time dependence into account.

6.1.2 Second Approximation of the Finite-Horizon

The second approximation of the Finite-Horizon MDP as Infinite-Horizon is based on the Turnpike Theorem, described in Section 2.7. According to this theorem, there exists a number of steps remaining n_0 such that for every number of remaining steps greater than n_0 the policy is the same (stationary), given that the horizon is sufficiently large.

To implement this idea, we had the agent calculate the policy up the horizon's length N, storing in memory at most 2 value function arrays, and then using this policy array (of index N) to make every decision until the agent run out of steps. Given that the horizon is sufficiently large and the MDP is Stationary, we expect the total reward collected to be slightly less than that collected

by an actual Finite-Horizon algorithm, as the policy calculated by both algorithms is the same for $n \geq n_0$, but for the remaining $N - n_0$ steps only the actual Finite-Horizon algorithm possesses the correct policy.

This algorithm is of the same time complexity as the Naive Finite-Horizon algorithm $\mathcal{O}(N|S|^2|A|)$, but its space complexity is $|\mathcal{S}|$ as it stores at most 3 arrays of length $|S|$, two value function arrays and the final policy array.

Note that, this method is expected to yield suboptimal results in the case of Non-Stationary MDPs, as it calculates a stationary policy. In the case of NSMDPs, the policy must be non-stationary, taking the time dependence into account.

6.1.3 In-Place Finite-Horizon Algorithm

This algorithm manages to correctly calculate values and actions to be made for the Finite-Horizon MDP problem utilizing the least possible space. This is accomplished by calculating every array of values for every state and every number of steps remaining when it is needed, starting from the beginning. Every new array overwrites the previous, until the decision is made. This results in a space complexity of $\mathcal{O}(|S|)$, as at most one list of values for every state is saved in memory at all times.

Achieving the optimal space complexity of the Finite-Horizon Approximation described in 6.1.1 while making the correct calculations, though, comes at a great temporal cost. For every number of steps remaining, the agent does not have any knowledge of older values and so it must compute them again from the very beginning. As established, this results in a time complexity of $\mathcal{O}(N^2|S|^2|A|)$, rendering this method impractical for large horizons.

6.1.4 Naive Finite-Horizon Algorithm

The term Naive was chosen for this method as it express the first approach that comes to mind for a Finite-Horizon MDP algorithm. As described in Section 4.1.1, this is the least optimal Finite-Horizon method regarding the space complexity. The idea is to simply save every list of values for every number of steps remaining while calculating the N-steps-remaining list of values.

The space complexity of this method is $\mathcal{O}(N|S|)$, growing linearly with the length of the horizon. Nevertheless, because every list of values is calculated only once and saved until used, the time complexity is in the orders of $\mathcal{O}(N|S|^2|A|)$. As it was already mathematically proven in section 4, there is a lot of room for improvement regarding this approach, which will be used as a reference point in the following experiments.

6.1.5 Square Root Finite-Horizon Algorithm

Building on the previous Naive method, this approach achieves the same time complexity of $\mathcal{O}(N|S|^2|A|)$, while lowering the space complexity to $\mathcal{O}(\sqrt{N}|S|)$. As described in section 4, during the computation of the largest number of steps

remaining list of values, a list of values corresponding to multiples of the square root of the horizon N is stored in memory. Then, we treat every interval of length $\lfloor\sqrt{N}\rfloor$ as a Naive Finite-Horizon MDP, starting from index $(k-1)\lfloor\sqrt{N}\rfloor$ up to $k\lfloor\sqrt{N}\rfloor$, where $1 \leq k \leq \lfloor\sqrt{N}\rfloor$, resulting in at most $2\lfloor\sqrt{N}\rfloor$ lists of length $|S|$ to be stored in memory at every point in time. In addition, following this procedure the total time complexity is $\lfloor\sqrt{N}\rfloor$ times the time complexity of the Naive Finite-Horizon MDP, resulting in $\mathcal{O}(N|S|^2|A|)$.

This method should, in theory, always be preferred over the Naive Finite-Horizon MDP, offering less memory consumption in the same execution time.

6.1.6 Logarithmic Finite-Horizon Algorithm

Even greater reduction in space complexity can be achieved using this method described in detail in Section 4.2.2, with a small trade-off in the time complexity of the algorithm. Following the logic of a Binary Search Tree containing the steps remaining, we store in memory the lists of values of every index in the path from the root of the tree up to the index needed. This results in a space complexity of $\mathcal{O}(\log_2(N)|S|)$ and a time complexity of $\mathcal{O}(N\log_2(N)|S|^2|A|)$.

6.2 Reward Comparison

As already mentioned, a random number generator is used whenever needed for the training and decision-making processes. In every execution, a seed is provided to the model in order to ensure consistency in the results. It follows that, for a particular seed, the total reward collected by each of the In-Place, Naive, Square Root and Logarithmic Finite-Horizon algorithms is the same, as they perform the same calculations but store a different number of arrays. Thus, for the following comparison, both Approximations of the Finite-Horizon MDP will be compared to one of the previously mentioned Finite-Horizon algorithms (representing all of them), with regards to the total reward collected.

The values of every state at every number of steps remaining in a Finite-Horizon MDP correspond to the expected total reward the agent is expected to gain after starting from a particular state and making the optimal decision at every state the find themselves into and for every number of steps remaining. In practice, however, the actual reward the agent collects will almost certainly differ from the expected reward. This is justified because no matter what decision the agent makes, the actual state it transitions to (and consequently the reward it collects) is determined by a probability function. If every decision the agent makes is treated as a random experiment, it is almost certain that the agent will, at some point, transition to an unwanted state and will have to continue from there, especially for larger horizon values.

This could result in the agent collecting smaller rewards in the Finite-Horizon case than in the Approximation case, although the first was proven to be more optimal. Thus, the optimality of the Finite-Horizon over the Approximation with regards to the reward collected will be evaluated after a number of repetitions of the experiment with a different seed, in order to eliminate statistical

inconsistencies.

For this purpose, each one of the following three experiments was conducted 20 times with a different seed each time, in order to ensure that the rewards collected are close to the values they would converge to after a large number of experiments and pacify the effect of random events such as the ones described above on the variance of the reward collected.

The parameters of the model used in this experiment are:

- Number of runs in each experiment: 20
- Training steps: 10000
- Minimum VMs: 1
- Maximum VMs: 20
- Initial VMs: 10
- e-probability: 0.7
- Number of States: 200
- Initial QValues: 0
- Actions: {add 1 VM, remove 1 VM, no operation}
- Value Iteration γ: {0.1, 0.3, 0.99}
- Value Iteration threshold: 0.1
- Number of steps before next Value Iteration during training: 500
- Horizon: {100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 409600, 819200}

For better comprehension of the results, we also present a diagram containing the expected reward for each of those algorithms in the case of this Non-Stationary Model in Figure 6.1. The expected reward of the Turnpike Approximation is the same as that of the Finite-Horizon as the same value function array (the one corresponding to index N) is used to perform the calculation.

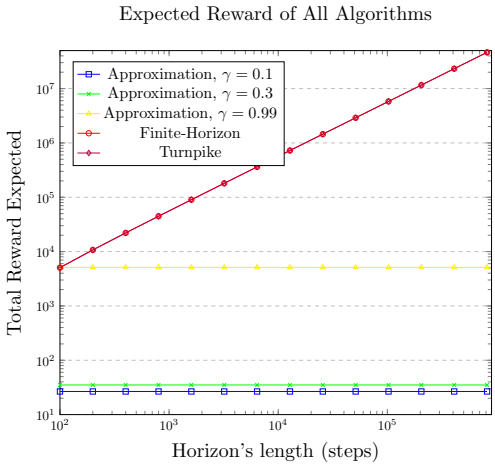


Figure 6.1: Mean value of the total expected reward in 20 different experiments, in the case of the Finite-Horizon algorithm, three Approximations with different values of the discount factor $\gamma = \{0.1, 0.3, 0.99\}$ and the Turnpike Approximation. Both axes are in logarithmic scale.

In Figure 6.1, we can observe that the Finite-Horizon and Turnpike Approximation algorithm's expected reward is identical, which is anticipated, as the expected total reward is the value of the value function at the initial state of the agent for $t = N$ steps remaining. Both of those agents use the same value function for $t = N$. The only difference is that, while the Finite-Horizon agent calculates a new value function array at every step (and thus a different policy), the Turnpike Approximation agent uses the same value function array (with index N) to make each of their decisions.

Interestingly, the expected reward of each of the Approximation algorithms with discount factor is constant regardless of the length of the horizon. This can be interpreted by the fact that each of those agents performs the Value Iteration value until convergence (which may occur at more or less iterations than the horizon size) and the value function calculated is, thus, independent of the horizon.

The average expected reward of each of these algorithms will be particularly useful in the reward comparison performed in these experiments, as is shown in the following two Figures.

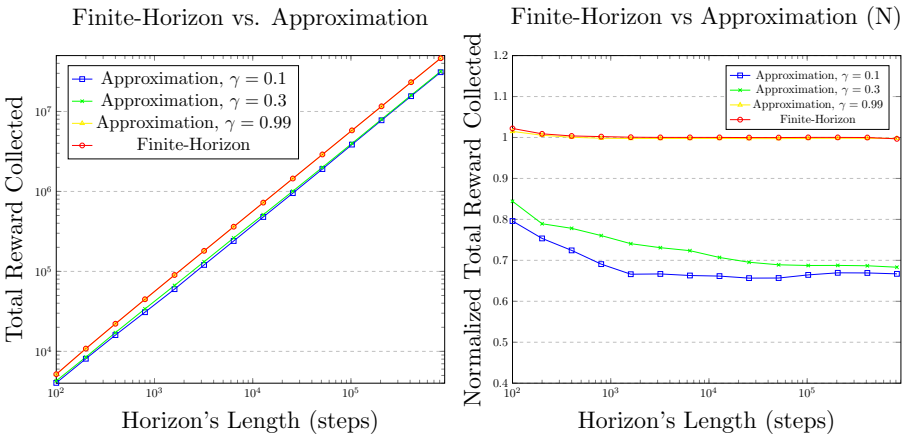


Figure 6.2: Left: average total reward collected by the agent throughout 20 different experiments, with respect to the horizon's length, in the cases of the Finite-Horizon algorithm and three Infinite-Horizon Approximations with variable $\gamma = \{0.1, 0.3, 0.99\}$. Right: ratio of the average total reward collected by the agent over the average expected reward calculated by the Finite-Horizon agent.

The results derived by observing the diagram in Figure 6.2 present great interest. We can observe that in any case, the most total reward accumulated is achieved using a Finite-Horizon algorithm, with the results being identical, if not greater, to those expected. On the other hand, both approximation with low γ values (namely 0.1 and 0.3) seem to yield much lower rewards compared to the high γ approximation and the Finite-Horizon algorithm. The values of the horizon for which they present the greatest results are lower values, such as 100 time steps. In the case of the highly discounted approximation, the results are almost identical to those of the Finite-Horizon. This is in accordance with the Turnpike Theorem, which states that there exists an time step n, after which the policy is stationary, which also depends on the value of γ. Finding this integer, however, has proven to be a difficult problem, as there exists no concrete way of doing so.

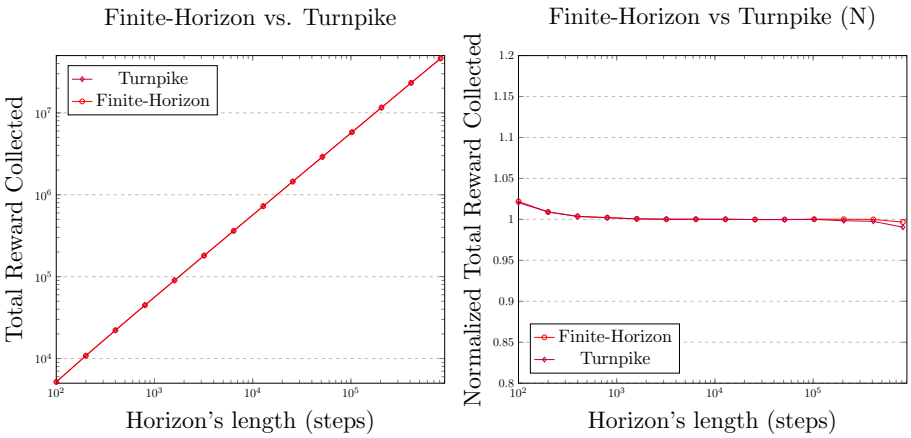


Figure 6.3: Left: average total reward vs. horizon length, in cases of Finite-Horizon and Turnpike Approximation. Right: ratio of average total reward over average expected reward calculated by the Finite-Horizon agent.

Comparing the Turnpike approximation to the Finite-Horizon algorithm in Figure 6.3 shows how well the approximation performs. In the left graph, the difference in the total reward collected is almost non-existent, while, when the results are normalized in the right graph, are indistinguishable. This shows that the policy of the model is almost stationary, independent of the time step. Of course, this is just the case for this particular model, which consists of many states and a very small amount of available actions (at most 3 in each state). As a result, the chance of a policy being completely different from another is tiny. In other use cases, the optimal performance of the Finite-Horizon could be more apparent.

Considering all the above observations, we conclude that, while the Finite-Horizon algorithm is in theory the most optimal, approximations with or without discounting can also yield approximately optimal results while consuming less resources. Nevertheless, it is apparent that the Finite-Horizon algorithm has a better performance regardless of the problem's conditions when compared to a discounted approximation, as in order to perform such an approximation one must calculate the optimal discount factor value, which could be computationally demanding. In comparison to a Turnpike algorithm without discount, while the Finite-Horizon algorithm should in theory be more optimal for small horizon values, because larger horizon's lengths ensure that the Turnpike Integer is contained within the horizon, this is not the case for our model. This is mainly attributed to the scarcity of the model's actions. We would expect different results in more complex MDPs. The most optimal method to solve a MDP, in theory, would be to use the Turnpike approximation up to the Turnpike Integer and then perform a classic Finite-Horizon algorithm for all of the steps remaining. This, however, requires calculation of the Turnpike Integer (or in other words the integer at which the policy converges to an optimal stationary

one) which can be demanding. In most cases of large horizon values, one can omit this calculation and directly use the Turnpike algorithm for every step. As we will see in the next experiment, however, this becomes a lot more inefficient in Non-Stationary MDP problems.

6.3 Variable Reward Comparison

In the previous subsection, we concluded that Turnpike approximation algorithms can be as optimal as accurate Finite-Horizon solutions while requiring less memory resources when the MDP is stationary. Nonetheless, we expect the Finite-Horizon algorithm to yield better results in the cases of Non-Stationary MDPs, which were simulated in the experiments following.

As already described in Section 5.2.1, we chose to simulate variable rewards (dependent on the decision epoch) in the following way: at each time step n and for every state s and (available) action a, the reward the agent will receive after transitioning to a neighboring state s' will be zero if the index of s' in the list of possible transitions is equal to $n \mod |S_{neighbors}|$. We can clearly see that at each time step, only one available transition's reward will be affected.

The parameters of the model used in this experiment are:

- Number of runs in each experiment: 20
- Training steps: 10000
- Minimum VMs: 1
- Maximum VMs: 20
- Initial VMs: 10
- e-probability: 0.7
- Number of States: 200
- Initial QValues: 0
- Actions: {add 1 VM, remove 1 VM, no operation}
- Value Iteration γ: {0.1, 0.3, 0.99}
- Value Iteration threshold: 0.1
- Number of steps before next Value Iteration during training: 500
- Horizon: {100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 409600, 819200}

For better comprehension of the results, we also present a diagram containing the expected reward for each of those algorithms in the case of this Non-Stationary Model in Figure 6.4. Again, the expected reward of the Turnpike Approximation is the same as that of the Finite-Horizon as the same value

function array (the one corresponding to index N) is used to perform the calculation.

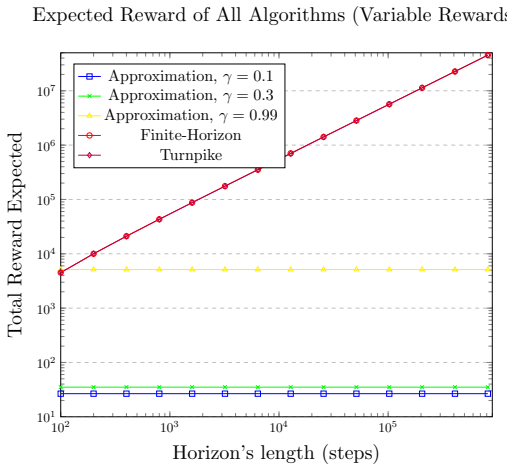


Figure 6.4: Mean value of the total expected reward in 20 different experiments with variable rewards, in the case of the Finite-Horizon algorithm, three Approximations with different values of the discount factor $\gamma = \{0.1, 0.3, 0.99\}$ and the Turnpike Approximation. Both axes are in logarithmic scale.

In Figure 6.4, we can observe that the Finite-Horizon and Turnpike Approximation algorithm's expected reward is identical, which is anticipated, as the expected total reward is the value of the value function at the initial state of the agent for $t = N$ steps remaining. Both of those agents use the same value function for $t = N$. The only difference is that, while the Finite-Horizon agent calculates a new value function array at every step (and thus a different policy), the Turnpike Approximation agent uses the same value function array (with index N) to make each of their decisions.

Interestingly, the expected reward of each of the Approximation algorithms with discount factor is constant regardless of the length of the horizon. This can be interpreted by the fact that each of those agents performs the Value Iteration value until convergence (which may occur at more or less iterations than the horizon size) and the value function calculated is, thus, independent of the horizon.

The average expected reward of each of these algorithms will be particularly useful in the reward comparison performed in these experiments, as is shown in the following two Figures.

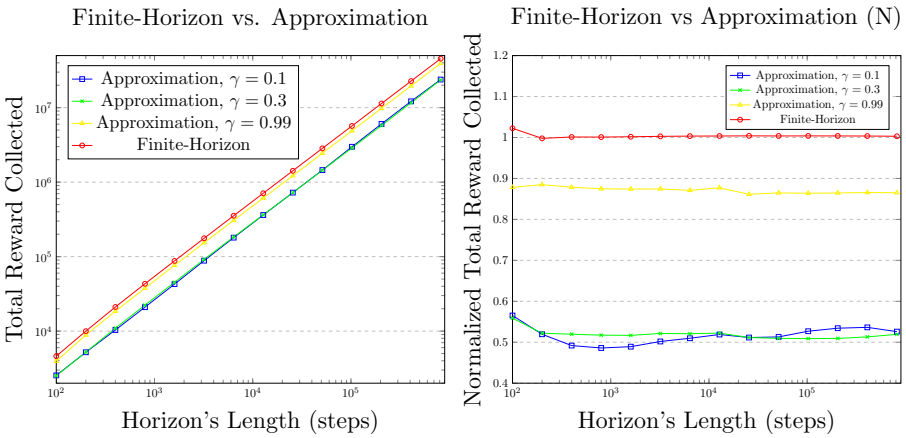


Figure 6.5: Left: average total reward collected by the agent throughout 20 different experiments, with respect to the horizon's length, in the cases of the Finite-Horizon algorithm and three Infinite-Horizon Approximations with variable $\gamma = \{0.1, 0.3, 0.99\}$. Right: ratio of the average total reward collected by the agent over the average expected reward calculated by the Finite-Horizon agent.

In the left graph of Figure 6.5 we present the average reward collected by the agent at each one of the horizon values over the course of 20 experiments, in four different cases: one Finite-Horizon model (using any of the algorithms described in Section 4) and three Infinite-Horizon Approximation algorithms with $\gamma = \{0.1, 0.3, 0.99\}$. An immediate observation that can be made is that, although the difference between the Finite-Horizon and Approximation ($\gamma = 0.99$) curves is almost non-existent, the Finite-Horizon curve seems to always be above it. The other two Approximations exhibit much worse performance, which is obvious even by observation.

In order to further exaggerate the optimal performance of the Finite-Horizon algorithm over those Approximations, we chose to normalize the results, by dividing each average total reward by the corresponding average expected reward the Finite-Horizon algorithm calculated. We opted for this average expected reward, as it is always appears to be more accurate than that calculated by the Approximation algorithms and is considered the most accurate in theory.

The results of this normalization are depicted in the right graph of Figure 6.6. The Finite-Horizon agent is always above the other curves, almost always equal to 1, meaning that the average reward collected is either a little less or a little more than the expected.

That, of course, is not the case for the other three graphs. The two agents using the Approximations with $g\gamma = 0.1$ and $\gamma = 0.3$ seem to collect significantly lower total rewards than the Finite-Horizon algorithm. When compared to each other, there is no clear winner, as their performance is almost identical. The normalized graph clearly depicts that both those approximations manage to

collect around 50% to 60% of the reward expected. In any case, the Finite-Horizon algorithm outperforms both of those approximations by a large margin.

The results are even more interesting when we examine the results of the highly discounted model. Unlike the constant rewards experiment, the agent using an Approximation with $\gamma = 0.99$ does not manage to prepare for the variability of the rewards and thus accumulates a much smaller total, at just below 90% of the reward expected. While this value is still high, it is considered as sub-optimal, especially when compared to the Finite-Horizon algorithm.

We can conclude that, as expected, the Finite-Horizon agent outperforms the three Approximation agents in all cases and should be preferred over them in variable reward cases (NSMDPs), as it is the only way to achieve optimality in reward collection, signifying successful calculation of the optimal policy.

Another important comparison is that between the Finite-Horizon algorithm and the Turnpike Approximation, the results of which are presented in Figure 6.6.

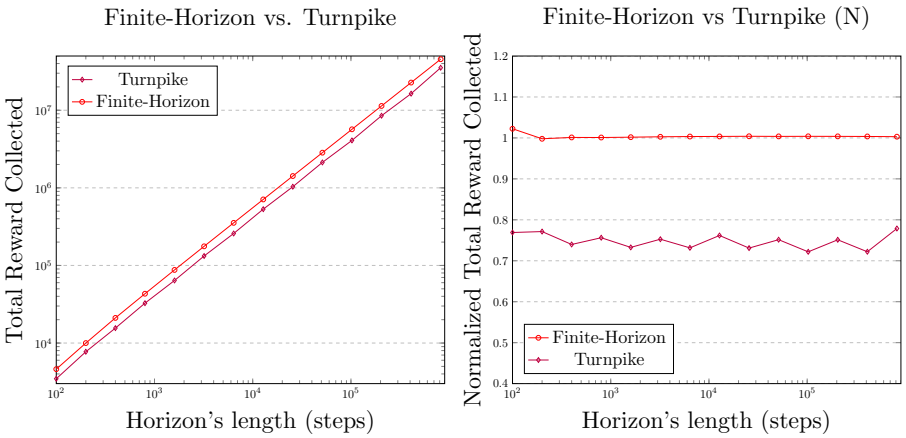


Figure 6.6: Left: average total reward collected by the agent throughout 20 different experiments, with respect to the horizon's length, in the cases of the Finite-Horizon algorithm and the Turnpike Approximation algorithm. Right: ratio of the average total reward collected by the agent over the average expected reward calculated by the Finite-Horizon agent, in the cases of the Finite-Horizon algorithm and the Turnpike Approximation algorithm.

As before, the left graph contains the curves of the average reward collected by the two agents over 20 experiments (log-log plot). In this case, we expect the Turnpike Approximation agent's performance to be close to that of the Finite-Horizon agents', as they both take into account the variability of the model's rewards. Indeed, the two curves differ by a small margin in the left graph, signifying close performance of the two agents.

Performing the same normalization, however, sheds new light to the results of the left graph. Looks can be deceiving, as one can clearly observe in the

right graph that the Turnpike agent's performance is oscillating at around 75%. This performance is surprisingly even worse than that of the highly discounted Approximation model.

Considering the above results, we conclude that any of the Finite-Horizon algorithms proposed in Section 4 yields results in the case of a NSMDP (with variable rewards), achieving performances no Approximation model can. Out of the four Approximation models examined in this Section, the one with $\gamma = 0.99$ had the best performance regarding the total reward collected, its policy, however, is still suboptimal compared to that of the Finite-Horizon.

6.4 Time Complexity Comparison

As already mentioned, execution time is an extremely important metric when it comes to a MDP solving algorithm. In this experiment, all 5 methods were compared with respect to their time complexity. The metric used to perform this comparison was the execution time of the policy evaluation for different values of the horizon length.

The parameters of the model used in this experiment are:

- Training steps: 10000
- Minimum VMs: 1
- Maximum VMs: 20
- Initial VMs: 10
- e-probability: 0.7
- Number of States: {200, 2000}
- Initial QValues: 0
- Actions: {add 1 VM, remove 1 VM, no operation}
- Value Iteration γ: 0.5
- Value Iteration threshold: 0.1
- Number of steps before next Value Iteration during training: 500
- Horizon: {100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 409600, 819200}

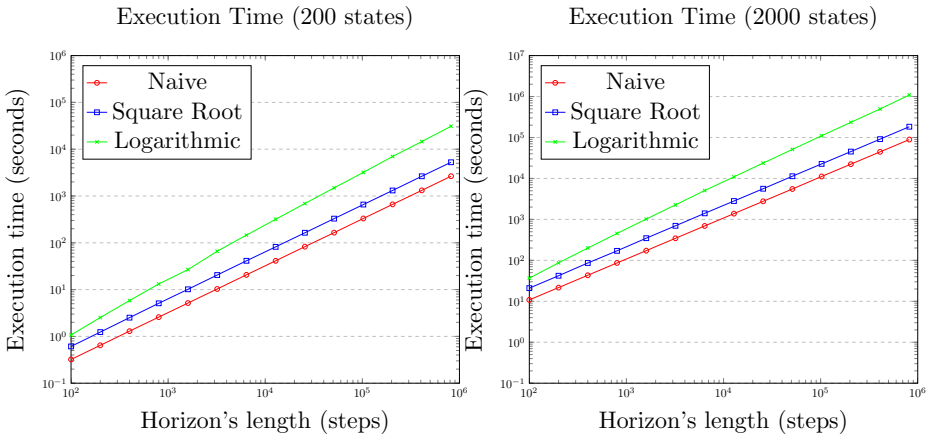


Figure 6.7: Left: log-log graph depicting the total execution time of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 200 states. Right: log-log graph depicting the total execution time of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 2000 states

In Figure 6.7 we observe the results of the comparison between three Finite-Horizon MDP solving algorithms, namely the Naive, Square Root and Logarithmic approaches. We chose not to depict the execution time results of the In-Place algorithm, as it was not physically possible to execute those algorithms because of enormous execution times, even for relatively small horizon values. In addition, neither the Approximation algorithm with discount nor the Turnpike Approximation were included. Algorithms with different discount factors belonging to the first category require tiny, constant amounts of time to execute, as they do not take the horizon's length into account. On the other hand, the Turnpike Approximation algorithm requires approximately the same amount of seconds as the Naive model to execute. Nevertheless, both of these Approximations yield suboptimal results regarding the rewards, as established in the previous Sections and are not preferred over the Finite-Horizon algorithms.

The results in Figure 6.7 are again presented in logarithmic scale (log-log graph) because of how large the values in both axes grow. What can be seen in the graph is completely justified by the theoretical analysis performed in Section 4 regarding the time complexity of the graphs. All models were tested under the same conditions, and so the $|S|^2|A|$ factor of their time complexity could be treated as a constant C (with $|S|$ being 200 in the first experiment and 2000 in the second). It can be observed that, in the case of $|S| = 2000$, the shape of the curves is the same as that of $|S| = 200$ with a small upward shift, which is expected as the constant C is the one changing.

We theorized that the time complexity of the Naive approach is $\mathcal{O}(NC)$. The equivalent time complexity in the logarithmic scale could be given by $\mathcal{O}(\log N +$

$\log C$). Treating $\log N$ as the new dependent variable X, the time complexity becomes $\mathcal{O}(X)$. The graph of a function with this big-O notation is expected to be a straight line, which is the case of the Naive approach as can be seen on the graph.

The same can be said about the Square Root approach, as it was proven to have the same time complexity as the Naive method. In the graph, its execution time in logarithmic scale is correctly represented as a straight line which is approximately parallel to the Naive approach, but differs by just a constant. If we treat the equation of the Naive algorithm's time complexity as $Y = C_1 + X$, where $Y = log(y)$, $C_1 = log(C)$ and $X = log(N)$, then the equation of the line representing the time complexity of the Square Root solution is approximately $Y = C_1 + X + C_2$, as it is almost parallel to the first line but shifted slightly upwards. Replacing those terms with their exponential equivalent, we obtain that the time complexity is $y = CC'N$, where $C' = 10^{C_2}$, which is exactly what was expected in the theoretical analysis. Thus, we validate that the execution time of the Square Root algorithm is asymptotically the same as that of the Naive algorithm.

Finally, the Logarithmic solution graph appears to be above both of the other graphs in each of the two experiments. This is of course expected, as its time complexity was theorized to be of greater order than the one of the other two algorithms. In both log-log plots, the execution time graph of this algorithm is approximately a straight line, above both of the other two lines and with greater slope. We can justify this result by analysing the equation of the Logarithmic solution's time complexity: we theorized that the execution time is asymptotically $y = C_4 N \log_2 N$. Applying log to both sides in order to obtain the equation to be plotted in the log-log graph, we derive: $Y = C_5 + X + \log\log_2 N$, where $Y = \log y$, $X = \log N$ and $C_5 = \log C_4$. Changing the logarithm's base in the third term, we obtain the desired result: $\log\log_2 N = \log\log N - \log\log 2 = \log X + K$, where $K = \log\log 2$. Thus, the log-log equation turns out to be: $Y = C_6 + X + \log X$, where $C_6 = C_5 + K$, which is approximately a straight line.

As far as the time complexity is concerned, we can confirm the results proven in theory in Section 4. In other words, the Logarithmic solution is of slightly higher time complexity than that of the Naive and Square Root solutions, with execution times being tolerable, although higher. The Square Root approach is asymptotically of the same time complexity as the Naive algorithm, as the only differ by a constant. Those results imply that all three algorithms could be of use (depending on the user's needs) regardless, as their time complexity does not differ by a large amount.

6.5 Space Complexity Comparison

The focus of our work is on optimizing the space complexity of the Finite-Horizon algorithms. In the experiments presented in this Section, we managed to measure the memory needs of each of the three Finite-Horizon algorithms, namely the Naive, Square Root and Logarithmic approaches and compare them

against each other, while also presenting how they match against the expected theoretical results.

The parameters of the model used in these experiments are:

- Training steps: 10000
- Minimum VMs: 1
- Maximum VMs: 20
- Initial VMs: 10
- e-probability: 0.7
- Number of States: $\{200, 2000\}$
- Initial QValues: 0
- Actions: {add 1 VM, remove 1 VM, no operation}
- Value Iteration γ: 0.5
- Value Iteration threshold: 0.1
- Number of steps before next Value Iteration during training: 500
- Horizon: {100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 409600, 819200}

Note that in order to acquire the expected memory values, we multiplied the maximum number of value function arrays stored in memory during each of the algorithms by their length times the typical memory footprint of the data type the hold (in this case, a floating point number of single precision).

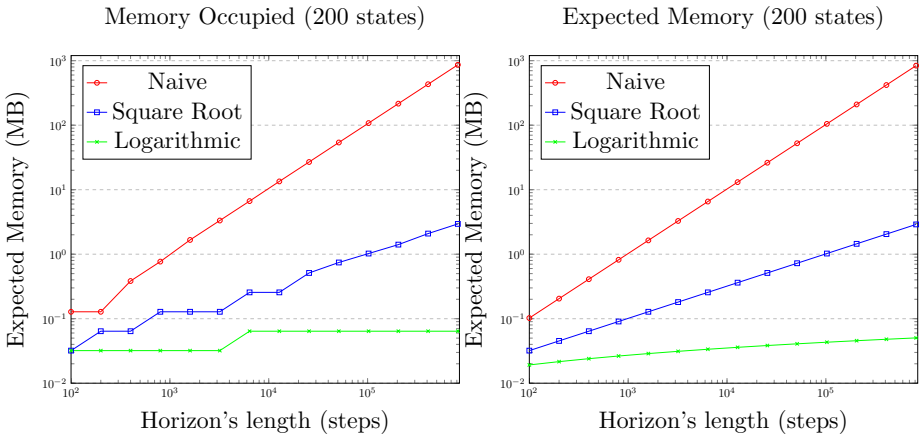


Figure 6.8: Left: difference between the initial memory and maximum memory occupied by each of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 200 states. Right: difference between the initial memory and maximum memory expected to be occupied by each of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 200 states.

The graphs presented in Figure 6.8 depict the results obtained by experiments run on a model with 200 states. The right graph contains the amount of memory expected to be needed in each horizon and for each algorithm. By observation, we can see that the graphs in the left are very close to their theoretically expected form, with the exception of a few flat points. This is because, as already established in Section 5.4, memory is given to an algorithm by the system in the form of pages. Thus, in some cases (especially for smaller horizon values) the extra memory required from a horizon value to the next can be provided within the same page, creating the illusion that no extra memory is needed.

This is especially obvious in the case of the Logarithmic approach. The total memory occupied appears to be constant (and close to 0), with a slight increase from $N = 3200$ to $N = 6400$, after which point the graph continuous to be constant. This fact can be interpreted considering that, up until $N = 3200$ a constant number of pages is sufficient for the algorithm's memory needs. At $N = 6400$, another page is needed, which is enough for the rest of the experiment.

With all the above said, the theoretical results are completely validated. The Naive model requires the most amount of memory, growing linearly with the horizon. On the other hand, the Logarithmic approach has close to non-existent memory needs compared to it, as it requires (almost) constant memory even for large horizon lengths. We must not forget, however, that the execution time required by the Logarithmic algorithm is much larger than that of the Naive model. The Square Root model comes as a compromise between the two, providing lower memory needs than the Naive algorithm, while requiring the

same execution time (asymptotically).

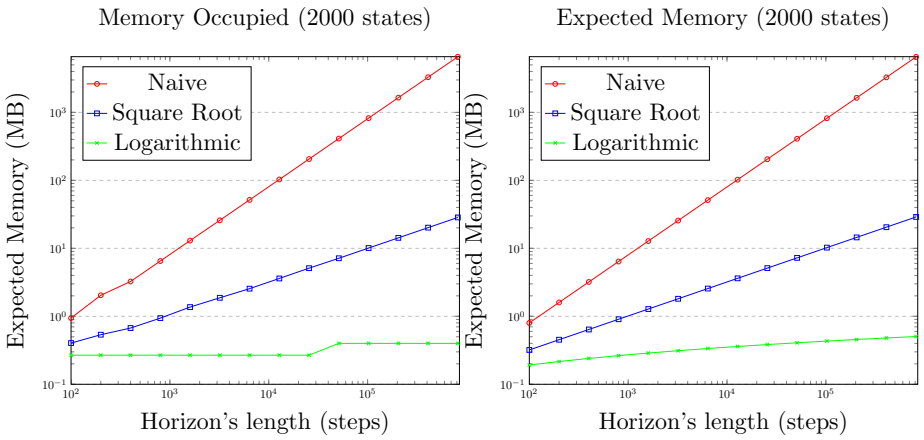


Figure 6.9: Left: difference between the initial memory and maximum memory occupied by each of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 2000 states. Right: difference between the initial memory and maximum memory expected to be occupied by each of the Naive, Square Root and Logarithmic FHMDP algorithms for different horizon values in the case of a model with 2000 states.

The results observed in Figure 6.8 do not change as much in Figure 6.9. Because the number of states is larger, growth in the horizon's length induces greater growth in memory needs, when compared to those of the 200 states model. As a result, the Naive and Square Root graphs appear to go smoother, implying that every time the horizon grows one or more new pages are required. Interestingly, the Logarithmic model continues to require tiny amounts of memory compared to the other two, even in this case and for longer horizons. A constant amount of pages appears to suffice when the horizon ranges from 100 to 25600 steps. After that, one (or more) new pages are required, but their number continues to be constant until the end of the experiment.

Note that, neither the Approximation techniques nor the In-Place algorithms were included in these experiments. All of these models are of constant space complexity with regards to the horizon, $\mathcal{O}(|S|)$, as they only need to store at most two arrays of length $|S|$. The Approximations, however, yield suboptimal results regarding the Reward, while the In-Place model required enormous amounts of time to execute.

In conclusion, the most optimal algorithm regarding the space complexity is the Logarithmic, with the other two Finite-Horizon algorithms failing to even come close to its memory performance. Between the Naive and the Square Root algorithms, the latter shows very promising results, with the memory needs growing much slower than that of the former as the horizon grows larger.

7 Conclusion

7.1 Summary

Throughout this work, we attempted to optimize MDP solving algorithms with respect to both execution time and space and measure the results of our efforts using experiments. To summarize our results:

- Value Iteration, an algorithm used predominantly in the areas of MDPs and Reinforcement Learning, presents serious time complexity issues. We proposed an idea utilizing the concept of bounds of the value function, tested in other forms by other authors. The results of our experimenting, however, were not fruitful, perhaps due to the use case chosen. Nevertheless, there is still room for improvements regarding this technique and future work could make use of it.

- Regarding the general space complexity issues FHMDPs face, we observed that in many cases such MDPs struggle executing algorithms that yield accurate solution due to memory constraints. In these cases, developers tend to opt for heuristic methods for memory optimization as applicable. Thus, an effective universal optimization technique is much needed.

- We also proposed two new algorithms reducing the space complexity of traditional Finite-Horizon MDP solving algorithms as much in theory as in practice. Our methods manage to require drastically less memory to execute the FHMDP solving algorithms, with the small overhead of some recalculations.

- We also examined the behavior of algorithms when compared with Infinite-Horizon Approximation algorithms. The latter yielded promising results when tested in non-variable reward and transition models. In Non-Stationary MDPs, on the other hand, our FHMDP solving algorithms outperformed every approximation we tested by a large margin.

7.2 Future Work

The optimizations presented in the above sections are a concrete basis for further improvements. The bounding optimization we proposed could be tested in other use cases to examine whether the time saved by action elimination is actually more than that spent calculating the bounds. The bounding inequalities could also be fine-tuned for thinner margins.

One could modify the idea of bounding as follows: upper and lower bound for the state-action function are initialized, while we do not compute the actual state-action function. In each of the algorithm's iterations, the upper and lower bound converge towards the same value: the state-action function's value for a particular state and action. In the process, actions would be eliminated, while the bounds are constantly updated.

As for the Finite-Horizon algorithms suggested, the main problem they presented was the overhead introduced by value function array recalculations. As those calculations have happened during the initial run of the algorithm, one could utilize the information provided after the calculation in order to speed up any recalculation of the same value function array afterwards. Such an approach could lead the time complexity of the Logarithmic solution to be close to that of the Naive or Root solution, rendering it the most efficient FHMDP solving algorithm out of those discussed in this work.

References

[Bel15] Richard E Bellman. *Adaptive control processes*. Princeton university press, 2015.

[Ber00] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition, 2000.

[Bis07] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[BL20] Nazila Bazrafshan and M. M. Lotfi. A finite-horizon markov decision process model for cancer chemotherapy treatment planning: an application to sequential treatment decision making in clinical trials. *Annals of Operations Research*, 295(1):483–502, 2020.

[chr] C++ chrono library documentation. https://en.cppreference.com/w/cpp/chrono.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[COM+09] Tang Lung Cheung, Kari Okamoto, Frank Maker, Xin Liu, and Venkatesh Akella. Markov decision process (mdp) framework for optimizing software on mobile phones. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT)*, pages 11–20, 2009.

[CS17] Dan Calderone and S. Shankar Sastry. Markov decision process routing games. In *Proceedings of the 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 273–279, 2017.

[dd17] Alexandre dos Santos Mignon and Ricardo Luis de Azevedo da Rocha. An adaptive implementation of ε-greedy in reinforcement learning. *Procedia Computer Science*, 109:1146–1151, 2017. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

[Dei90] Harvey M. Deitel. *An Introduction to Operating Systems (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[DMW08] Peng Dai, Daniel S Weld Mausam, and Daniel S Weld. Partitioned external-memory value iteration. In *AAAI*, pages 898–904, 2008.

[DW+09] Peng Dai, Daniel S Weld, et al. Domain-independent, automatic partitioning for probabilistic planning. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer, 2009.

[DWG+11] Peng Dai, Daniel S Weld, Judy Goldsmith, et al. Topological value iteration algorithms. *Journal of Artificial Intelligence Research*, 42:181–209, 2011.

[EJB07] Stefan Edelkamp, Shahid Jabbar, and Blai Bonet. External memory value iteration. In *ICAPS*, pages 128–135, 2007.

[GBGG11] Yasin Gocgun, Brian W. Bresnahan, Archis Ghate, and Martin L. Gunn. A markov decision process approach to multi-category patient scheduling in a diagnostic facility. *Artificial Intelligence in Medicine*, 53(2):73–81, 2011.

[GLP04] Jason H. Goto, Mark E. Lewis, and Martin L. Puterman. Coffee, tea, or . . . ?: A markov decision process model for airline meal provisioning. *Transportation Science*, 38(1):107–118, 2004.

[LHP+19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[Li19] Yuxi Li. Reinforcement learning applications, 2019.

[lin] Documentation of the /proc/[pid]/status file used in linux os. https://man7.org/linux/man-pages/man5/proc.5.html.

[LKKK17] Konstantinos Lolos, Ioannis Konstantinou, Verena Kantere, and Nectarios Koziris. Elastic management of cloud applications using adaptive reinforcement learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 203–212, 2017.

[LP19] Mark E. Lewis and Anand A. Paul. Uniform turnpike theorems for finite markov decision processes. *Math. Oper. Res.*, 44:1145–1160, 2019.

[LR20] Erwan Lecarpentier and Emmanuel Rachelson. Non-stationary markov decision processes, a worst-case approach using model-based reinforcement learning, extended version, 2020.

[LWL06] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. *ISAIM*, 4(5):9, 2006.

[MA93] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping : Reinforcement learning with less data and less real time. 1993.

[Mac67] J MacQueen. A test for suboptimal actions in markovian decision problems. *Operations Research*, 15(3):559–561, 1967.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[psa] getprocessmemoryinfo function of the process status api documentation for c++. https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessmemoryinfo.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., USA, 1st edition, 1994.

[PW98] Jing Peng and Ronald Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1, 09 1998.

[RK02] Zhiyuan Ren and Bruce H Krogh. State aggregation in markov decision processes. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 4, pages 3819–3824. IEEE, 2002.

[RNG] Documentation of the random c++ library, used to produce pseudo-random numbers. https://www.cplusplus.com/reference/random/.

[SB79] D Sadjadi and Paul F Bestwick. A stagewise action elimination algorithm for the discounted semi-markov problem. *Journal of the Operational Research Society*, 30(7):633–637, 1979.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[Sha68] Jeremy F. Shapiro. Turnpike planning horizons for a markovian decision model. *Management Science*, 14(5):292–300, 1968.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

[SX20] Ruiyang Song and Kuang Xu. Temporal concatenation for markov decision processes. *Probability in the Engineering and Informational Sciences*, pages 1–28, 2020.

[WD92] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.

[Whi82] DJ White. The determination of approximately optimal policies in markov decision processes by the use of bounds. *Journal of the Operational Research Society*, 33(3):253–259, 1982.

[WS03] David Wingate and Kevin D Seppi. Efficient value iteration using partitioned models. In *ICMLA*, pages 53–59. Citeseer, 2003.