Κοφινάς Νίκος Α.Μ.: 2007030111

Σε αυτή την εργασία υλοποιήσαμε έναν πράκτορα ο οποίος παίζει το παιχνίδι othello. Η γλώσσα την όποια προτιμήσαμε για την διεκπεραίωση της εργασίας είναι η C.

Περιγραφή του παιχνιδιού:

Το othello είναι ένα παιχνίδι που παίζεται μεταξύ δύο αντιπάλων, και ο απώτερος σκοπός του κάθε παίκτη είναι να έχει περισσότερα πούλια πάνω στην σκακιέρα, από τον αντίπαλο, στο τέλος του παιχνιδιού. Το κλασικό παιχνίδι othello αποτελείται από ένα πλέγμα 8*8, όμως στην δική μας άσκηση το πλέγμα αυτό έχει γίνει 10*10 και υπάρχουν κάποια συγκεκριμένα κουτάκια στο πλέγμα, που θεωρούνται illegal, και κανένας παίκτης δεν μπορεί να τοποθετήσει σε εκείνα τα κουτάκια πούλι.

Το othello είναι ένα παιχνίδι με πάρα πολλές πιθανές καταστάσεις, τις οποίες όπως είναι λογικό δεν μπορούμε να εξετάσουμε όλες. Οπότε ο αλγόριθμος που θα υλοποιήσουμε θα πρέπει για κάθε κατάσταση που βρισκόμαστε, να κοιτάζει κάποιες καταστάσεις παρακάτω, και να μας επιστρέφει την βέλτιστη κίνηση που θα μπορούσαμε να κάνουμε εκείνη την στιγμή με τα υπάρχοντα δεδομένα. Ο αλγόριθμος που υλοποιήσαμε είναι ο αλγόριθμος minimax με την επέκταση a-b pruning. Ο αλγόριθμος αυτός μας επιστρέφει την βέλτιστη κίνηση εξετάζοντας όλες τις πιθανές κινήσεις σε κάθε βάθος. Για να επιλέξει μία κίνηση ουσιαστικά φτιάχνει το δένδρο και ξεκινώντας από τα φύλλα, τα οποία αξιολογεί με μία συνάρτηση αξιολόγησης, διαλέγει εναλλάξ (σαν min για τον αντίπαλο και σαν max για εμάς) την βέλτιστη κίνηση. Η συνάρτηση αξιολόγησης είναι φτιαγμένη έτσι ώστε ο αντίπαλος να θέλει να την κάνει minimize και εμείς να θέλουμε να την κάνουμε maximize. Ουσιαστικά, μας επιστρέφει την κίνηση όπου θα είχαμε το βέλτιστο κέρδος αν και ο αντίπαλος έπαιζε βέλτιστα. Το αρνητικό αυτό του αλγορίθμου, είναι ότι δεν εκμεταλλεύεται τις αδυναμίες του αντιπάλου, διότι ακόμα και αν ο αντίπαλος παίζει «χαζά» εμείς θα του συμπεριφερόμασταν σαν να ήταν βέλτιστος.

Minimax αλγόριθμος:

Υλοποιήσαμε τον αλγόριθμο minimax στη γλώσσα προγραμματισμού C, με αναδρομικό τρόπο. Ο minimax αποτελείται από δύο συναρτήσεις, την getMin και την getMax, οπού η getMin προσπαθεί να ελαχιστοποιήσει την eval και η getMax προσπαθεί να την μεγιστοποιήσει. Εμείς είμαστε ο max παίκτης, και άρα ξεκινάμε καλώντας την getMax η οποία ύστερα καλεί την getMin και η μία καλεί την επόμενη, μέχρι να φτάσουμε στο βάθος που θέλουμε να ψάξουμε, όπου τελειώνει και η αναδρομή. Σε κάθε αναδρομική κλήση μειώνουμε και το βάθος κατά ένα, και όταν βρεθούμε σε βάθος ίσο με το μηδέν επιστρέφουμε την τιμή της συνάρτησης αξιολόγησης για το στήσιμο της σκακιέρας σε εκείνο το βάθος. Η τιμή της συνάρτησης αξιολόγησης, ύστερα, προχωράει προς την ρίζα του minimax δένδρου. Οι min κόμβοι (συνάρτηση getMin) διαλέγουν τη μικρότερη τιμή για την eval που μπορούν να πάρουν, και επιστρέφουν αυτήν την τιμή.

Αντίστοιχα οι max κόμβοι, επιλέγουν να επιστρέψουν την μεγαλύτερη τιμή του eval που έλαβαν.

Έτσι στον root κόμβο, που είναι max κόμβος, θα επιλέξουμε την κίνηση, για την οποία λάβαμε τη μεγαλύτερη τιμή από το αναδρομικό κάλεσμα. Έτσι θα έχουμε φροντίσει να επιλέξουμε την καλύτερη τιμή από την αναδρομική κλήση του minimax αλγορίθμου, και αφού και οι δύο παίκτες παίζουν βέλτιστα, τότε η κίνηση που επιλέξαμε, σίγουρα θα είναι η βέλτιστη. Η συνάρτηση αξιολόγησης καλείται, εκτός από όταν φτάσουμε στο μέγιστο βάθος, όταν βρούμε τέλος παιχνιδιού. Τέλος παιχνιδιού έχουμε όταν δεν υπάρχει άλλη κίνηση για κανέναν παίκτη. Σε κάθε μία συνάρτηση, βρίσκουμε όλες τις κινήσεις που μπορεί να κάνει ο παίκτης στην εκάστοτε σκακιέρα, και εκτελούμε κάθε μία από αυτές, και ύστερα καλούμε την getMin ή την getMax (ανάλογα σε ποιον κόμβο είμαστε) για κάθε μία από αυτές τις κινήσεις. Αν ο παίκτης δεν έχει να κάνει κίνηση, τότε απλώς καλούμε την επόμενη συνάρτηση.

α-β pruning:

Ο αλγόριθμος είναι αρκετά αργός και απαιτεί εκθετικό χρόνο για να εκτελεστεί. Με την επέκταση α-β pruning, δεν μπορούμε να διώξουμε τον εκθέτη, αλλά μπορούμε να τον μειώσουμε στο μισό. Η λογική του α-β pruning είναι πολύ απλή, όταν ένας κόμβος καλεί μια συνάρτηση για να αξιολογήσει την κατάσταση στην οποία βρέθηκε μετά από μία κίνηση, ελέγχει τι του επιστρέφει και αν του κάνει, σε σχέση με το προηγούμενο που είχε, το κρατάει και πετάει το προηγούμενο. Με το α-β pruning μπορούμε να προωθούμε την τιμή που έχουμε, στους από κάτω κόμβους, ώστε αν αυτοί βρουν τιμή μεγαλύτερη από αυτή που έχει προωθήσει ο από πάνω κόμβος (στην περίπτωση του max κόμβου) ή μικρότερη (στην περίπτωση του min κόμβου) τότε την επιστρέφουν αμέσως, χωρίς να επεκτείνουν τα εναπομείναντα παιδιά. Ο λόγος που λειτουργεί αυτό, είναι διότι ο minimax αλγόριθμος, θα τα απέρριπτε αυτά τα παιδία από αυτή και μόνο την τιμή, χωρίς να τον νοιάζει αν υπάρχει μεγαλύτερη η μικρότερη τιμή, γιατί σίγουρα δεν θα επέλεγε αυτή την κίνηση, οπότε περεταίρω ψάξιμο είναι άχρηστο. Με αυτή την μικρή βελτιστοποίηση μπορούμε να ψάξουμε σε αρκετά μεγαλύτερο βάθος, στον ίδιο χρόνο.

Από την προγραμματιστική πλευρά, η υλοποίηση είναι πολύ απλή, διότι απλώς διαδίδουμε τα α και β προς τα κάτω, μέσω της κλήσης των συναρτήσεων. Ο έλεγχος για το αν έχουμε cut off γίνεται με το που επιστρέψει η συνάρτηση που καλέσαμε, ανάλογα με την τιμή που μας επέστρεψε, αν έχουμε cut off γυρνάμε την τιμή, αλλιώς κοιτάμε αν πρέπει να αλλάξουμε τα α και β πριν καλέσουμε την επόμενη συνάρτηση. Τέλος, τα α και β προωθούνται μόνο προς τα κάτω.

Συνάρτηση αξιολόγησης:

Η συνάρτηση αξιολόγησης είναι ίσως το πιο σημαντικό στοιχείο για να δουλέψει καλά το πρόγραμμα μας. Από την συνάρτηση αξιολόγησης κρίνονται πολλά πράγματα, αν θα έχουμε εύκολα cut off ή όχι, κάτι το οποίο εξαρτάται από το πόσο καλά έχουμε αξιολογήσει την εκάστοτε κατάσταση. Επίσης, επειδή

πάντα κάνουμε μια εκτίμηση της εκάστοτε κατάστασης, πρέπει να κάνουμε όσο καλύτερη εκτίμηση μπορούμε, ώστε να ξέρουμε όσο καλύτερα γίνεται αν αυτή η κατάσταση είναι θετική για εμάς ή όχι. Επίσης, μπορούμε να χωρίσουμε το παιχνίδι σε φάσεις, όπου στην κάθε φάση ακολουθούμε και άλλη στρατηγική.

Τα βασικά σημεία που κοιτάμε σε κάθε στρατηγική, είναι το πόσες γωνίες έχει πιάσει ο καθένας, πόσα «κακά» κουτάκια έχει πιάσει ο καθένας, πόσες κινήσεις έχει και τέλος όταν πλησιάζει το τέλος του παιχνιδιού, το πόσα πούλια έχει ο καθένας πάνω στην σκακιέρα. Έχουμε χωρίσει σε 3 στάδια το παιχνίδι, στο early game, στο mid game και στο end game. Στο early game κοιτάμε πιο πολύ να μην πάρουμε κάποια bad κουτάκια, το να έχουμε όσες παραπάνω γωνίες γίνεται και κυρίως να έχουμε πιο πολλά moves από τον αντίπαλο. Στο mid game ανεβαίνει το βάρος που δίνουμε στις γωνίες, γιατί αρχίζουν να έχουν μεγαλύτερη σημασία για το stability μας. Στο end game το βάρος που δίνουμε στα bad κουτάκια, πέφτει αρκετά, μιας και πλέον λογικά δεν θα κάνουν τόσο διαφορά οι γωνίες, και προσθέτουμε στην συνάρτηση αξιολόγησης, την διαφορά στα πούλια κάτι το οποίο δεν είχαμε πιο πριν. Η διαφορά αυτή πλέον μας ενδιαφέρει, διότι πλησιάζει το τέλος του παιχνιδιού, και για να φτάσουμε σε νίκη πρέπει να έχουμε περισσότερα πούλια από τον αντίπαλο. Οι τελικές καταστάσεις αξιολογούνται μόνο με την διαφορά στα πούλια, και τις πολλαπλασιάζουμε με έναν μεγάλο αριθμό, ώστε να δηλώσουμε ότι αν νικάει ο min, ο max θα την απορρίψει, αν γίνεται, αυτή την κίνηση, και αν νικάει ο max να την απορρίψει ο min.

Βελτιστοποιήσεις

Το πρόγραμμα μας, με όλα τα παραπάνω, ήταν ικανό να κερδίζει πάντα τους αλγόριθμους greedy και random, άλλα έπρεπε να το βελτιστοποιήσουμε σε διάφορα σημεία, ώστε να μπορούμε να ψάχνουμε σε μεγαλύτερο βάθος, και να μπορούμε να αντιμετωπίσουμε και πιο ισχυρούς αντιπάλους. Παρακάτω, περιγράφονται οι βελτιστοποιήσεις που έγιναν.

Do, Undo and Redo move:

Όταν εκτελούσαμε μία κίνηση, οι συναρτήσεις που την εκτελούσαν, μπορούσαν είτε να μας επιστρέψουν μία νέα σκακιέρα, στην οποία θα έχει γίνει η κίνηση ή να κάνουν την κίνηση στην υπάρχουσα σκακιέρα. Επειδή τα malloc και free είναι ακριβά, επιλέξαμε τη δεύτερη υλοποίηση, στην οποία όλα γίνονται πιο γρήγορα, λόγω έλλειψης κλήσεων malloc και free. Έπρεπε όμως να βρούμε και έναν τρόπο να κάνουμε undo την κίνηση την οποία πραγματοποιήσαμε. Το επιτύχαμε κρατώντας σε μία struct όλες τις μεταβλητές που αλλάζουν, εκτελώντας ένα do. Κρατάμε δηλαδή τα πια πούλια που έγιναν flip, το αν πάρθηκε γωνία ή όχι, τους αρχικούς zobtrist αριθμούς που είχαμε πριν γίνει η κίνηση και τέλος κάποια ακόμα στοιχεία τα οποία μας επιτρέπουν να βρίσκουμε πιο γρήγορα τα legal moves (εξηγούμε παρακάτω τι κρατάμε ακριβώς). Η όλη ιδέα είναι, ότι αφού κάναμε ένα ακριβό do move, πλέον κρατάμε σε έναν πίνακα το ποια πούλια έγιναν flip και έτσι μπορούμε να επιστρέψουμε γρήγορα στην προηγούμενη κατάσταση. Επίσης με το redo μπορούμε να επιστρέψουμε γρήγορα σε μία μεταγενέστερη κατάσταση, εκτελώντας λιγότερα βήματα από

ότι η do move καθώς δεν χρειάζεται να εξετάσει ποια πούλια θα γίνουν flipped, αλλά τα κάνει με τη μία, μιας και τα έχει αποθηκέυσει.

Find Faster The Legal Moves:

Ό τρόπος για να βρίσκουμε τα legal moves ήταν να σαρώνουμε τον 10*10 πίνακα μας και όπου βρίσκουμε empty να κοιτάζουμε αν είναι legal move. Αυτό είναι αρκετά χρονοβόρο, ειδικά στην αρχή του παιχνιδιού, που έχουμε πάρα πολλά empty squares. Η βελτιστοποίηση που υλοποιήσαμε, είναι να κρατάμε κάθε φορά ποια είναι τα empty squares, που συνορεύουν με κατειλημμένα squares. Για να το κάνουμε αυτό, κρατήσαμε έναν ακόμα πίνακα 10*10, όπου «σημαδεύουμε» αυτά τα empty squares, ώστε να εκτελούμε μόνο σε αυτά την isLegalMove. Η ενημέρωση αυτού του πίνακα, γίνεται σε κάθε do και undo move, και ουσιαστικά είναι μόνο οκτώ εντολές if, και άρα είναι αρκετά γρήγορο. Το μόνο πρόβλημα είναι ότι ο πράκτορας μας πρέπει να έχει μπει στο παιχνίδι πριν παιχτεί οποιαδήποτε κίνηση γιατί αλλιώς δεν θα ενημερώσει σωστά τον πίνακα και θα «πέσει» το πρόγραμμα. Όταν πλέον ψάχνουμε για legal moves, κοιτάμε αυτόν τον πίνακα και όπου δούμε κενό τετράγωνο, που συνορεύει με τετράγωνο που έχει πούλι, εκτελούμε την isLegalMove. Έτσι γλυτώνουμε πολλά ακριβά isLegalMove για μεγάλο ποσοστό του παιχνιδιού.

Επίσης παρατηρήσαμε, ότι, η isLegalMove κάνει ό,τι ελέγχους κάνει και η doMove, χωρίς όμως να κάνει αλλαγές, ενώ η doMove αναγνώριζε αν μπορεί να γίνει μία κίνηση ή όχι. Οπότε μπορούμε αντί να καλούμε isLegalMove, και μετά να καλούμε την doMove, να καλούμε κατευθείαν την doMove και εάν επιστρέψει ότι έγινε η κίνηση να καλέσουμε αντίστοιχα την συνάρτηση που θέλουμε και να συνεχίσουμε. Με αυτό τον τρόπο, έχουμε μόνο ένα doMove και ένα undoMove, για κάθε κίνηση κι έτσι γλιτώσαμε τα isLegalMove. Αυτή η λύση είχε πολύ καλά αποτελέσματα, αλλά δεν μας επέτρεπε move ordering, διότι δεν κρατάγαμε κάπου τα πιθανά moves, άλλα τα κάναμε απευθείας. Για να μπορούμε να κρατάμε και κάπου όλα τα legal moves, πριν αρχίσουμε να επεκτεινόμαστε παρακάτω, φτιάξαμε και την redoMove, ώστε να μπορούμε να κάνουμε do και undo για να βρούμε τα legal moves, και μετά redo undo για να επεκταθούμε παρακάτω. Ο λόγος που κάνουμε do undo redo undo και δεν κάνουμε isLegaMove do undo, είναι διότι θέλουμε να βρούμε τον zobrist αριθμό της εκάστοτε σκακιέρας, για να εκτελέσουμε move ordering.

Zobrist Hash Table:

Παρατηρήσαμε ότι, σε μία κατάσταση, μπορούμε να πάμε με διαφορετική ακολουθία καταστάσεων. Οπότε χάνουμε χρόνο ψάχνοντας καταστάσεις που τις έχουμε ήδη ψάξει. Επίσης, παρατηρήσαμε ότι θα μπορούσαμε να έχουμε γρηγορότερα cutoff αν μπορούσαμε να κάνουμε ένα move ordering ,γνωρίζοντας τι περίπου μας είχε επιστρέψει η κάθε κίνηση σε προηγούμενες αναζητήσεις. Η λύση για αυτό είναι να φτιάξουμε ένα μεγάλο hash table , στο οποίο θα κρατάμε τι κόστος είχε κάθε κατάσταση, σε τι βάθος το είχαμε βρει και αν ήταν exact ή όχι τιμή. Για να κάνουμε index στο hast table χρησιμοποιούμε zobrist αριθμούς. Για να βρούμε αυτούς τους αριθμούς, φτιάχνουμε δύο 10*10 πίνακες, οι οποίοι σε κάθε θέση έχουν και έναν random αριθμό. Ύστερα για να βρούμε τον zobrist

αριθμό μιας κατάστασης, διαπερνάμε τον πίνακα της κατάστασης, και για κάθε white πούλι που βρίσκουμε πηγαίνουμε στην αντίστοιχη θέση στον πρώτο πίνακα από τους δύο με τους τυχαίους αριθμούς που φτιάξαμε, και κάνουμε xor τον αριθμό zobrist με αυτόν. Αντίστοιχα κάνουμε και για άμα βρούμε red πούλι, μόνο που εκεί πάμε στον δεύτερο πίνακα. Ο zobrist αριθμός αρχικοποιείται στο 0 και ύστερα ξεκινάμε να διαπερνάμε τον πίνακα. Χρησιμοποιούμε και έναν τυχαίο αριθμό για να υποδηλώνουμε ποιανού είναι η σειρά για να παίξει στην εκάστοτε κατάσταση. Ο zobrist είναι ένας int αριθμός, άρα έχει 32 bit σε όλα τα σύγχρονα pc. Επειδή οι καταστάσεις μας όμως είναι αρκετές παραπάνω (συνολικά όλες οι πιθανές) κάνουμε όλη την παραπάνω διαδικασία για να παράγουμε και έναν δεύτερο zobrist αριθμό. Δημιουργούμε νέους τυχαίους πίνακες για αυτό τον αριθμό. Το να έχουμε ακριβή διαφοροποίηση της κάθε κατάστασης από άλλες, χρειάζεται αν θέλουμε να κάνουμε άμεσο return value από το hash table και άρα πρέπει να είμαστε σίγουροι ότι θα κάνουμε σωστή τιμή return, και όχι την τιμή για κάποια άλλη κατάσταση που έτυχε να έχει ίδιο zobrist αριθμό.

Επειδή για να δημιουργήσουμε έναν zobrist αριθμό, πρέπει να διαπεράσουμε όλο τον πίνακα, αποφασίσαμε ότι είναι καλύτερο να τον διαδίδουμε προς τα κάτω μέσω της do και της undo. Έτσι, όταν βλέπουμε κάποιο flip με την doMove, κάνουμε κατευθείαν τα κατάλληλα xor στους zobrist αριθμούς μας, και έτσι δεν χρειάζεται να ξαναπεράσουμε όλον τον πίνακα. Επίσης, όταν καλείται η undo, επιστρέφουμε την παλιά τιμή του zobrist αριθμού. Το hash table μας έχει μέγεθος δέκα εκατομμύρια, και πιάνει χώρο στην μνήμη ram ίσο με 200MB. Παρόλο που χρειάζεται τόσο πολλή μνήμη ram ανεβάζει κατακόρυφα την απόδοση του προγράμματος.

Move Ordering:

Για να έχει πολύ καλά αποτελέσματα το α-β pruning, πρέπει να έχουμε ένα καλό move ordering , ώστε να προκαλείται γρήγορα cutoff. Οπότε, αφού έχουμε κρατήσει τιμές στο hash table για καταστάσεις που έχουμε ήδη εξετάσει, μπορούμε να της χρησιμοποιήσουμε για να κάνουμε ένα sort στις κινήσεις που έχουμε τώρα. Η διαδικασία είναι, αφού βρούμε όλες τις κινήσεις, και την σκακιέρα την οποία παράγουν αυτές, βρίσκουμε στο hash table την τιμή που έχουμε για αυτή την σκακιέρα (και τίποτα να μην έχουμε δεν υπάρχει πρόβλημα) και ύστερα να κάνουμε move ordering. Οι max κόμβοι, θέλουν να εξετάσουν πρώτα τις κινήσεις που μεγιστοποιούν το κέρδος και αντίστοιχα οι min αυτές που το ελαχιστοποιούν. Οπότε οι κινήσεις που ήταν στο hash table στην σωστή σειρά, και όσες απέμειναν, μπαίνουν με τυχαία σειρά στο τέλος. Έτσι έχουμε επιτύχει να κάνουμε με μεγάλη πιθανότητα γρήγορα cut off.

Iterative Deepening:

Έπρεπε να βρούμε κάπως έναν τρόπο να αξιοποιήσουμε, όλον τον χρόνο που μας δίνεται στο παιχνίδι, ώστε να μπορούμε να βλέπουμε πιο βαθιά ανάλογα με τον χρόνο που έχουμε. Ο τρόπος που σκεφτήκαμε είναι να εκτελούμε iterative deepening, δηλαδή να ορίσουμε ένα χρονικό διάστημα στο οποίο θα τρέχουμε μία αναζήτηση σε ένα βάθος και αν αυτή τελειώσει πριν τον

χρόνο, να αυξάνουμε το βάθος κατά ένα και να ψάχνουμε παρακάτω. Με αυτόν τον τρόπο, αξιοποιούμε σχεδόν όλο το χρόνο που μας δίνεται, χωρίς να βγαίνουμε εκτός χρόνου. Το διάστημα που αφήνουμε για ψάξιμο σε κάθε κίνηση διαφέρει ανάλογα με το στάδιο του παιχνιδιού. Στην αρχή το αφήνουμε για 2 second, το οποίο πολλές φορές παραβιάζεται κατά πολύ, χωρίς όμως να μας πειράζει. Για παράδειγμα, μπορεί να ξεκινάμε σε βάθος 7, να κάνουμε 1 sec για να επιστρέψουμε, μετά να πηγαίνουμε σε βάθος 8 και να κάνουμε 20 second να επιστρέψουμε. Στο τέλος του παιχνιδιού μας νοιάζει να δούμε το τέλος όσο πιο νωρίς γίνεται, οπότε ο χρόνος τον οποίο αφήνουμε για να ψάξει, είναι ο χρόνος που μας έχει απομένει δια δέκα. Με αυτόν τον τρόπο καταφέρνουμε να δούμε αρκετά νωρίς την νίκη, πράγμα που είναι κατά πολύ υπέρ μας.