Αναφορά 2ης Προγραμματιστικής Εργασίας

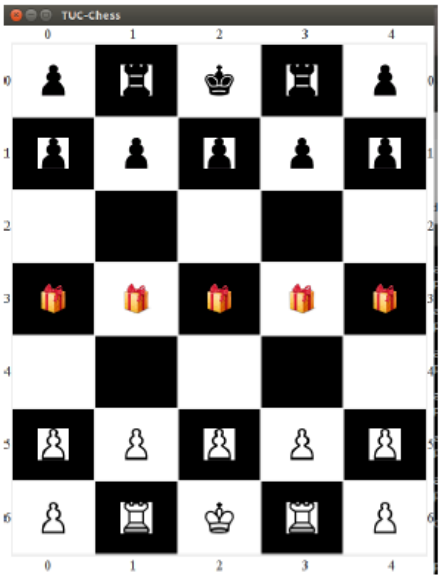
Εαρινό εξάμηνο 2021-22

TUC-CHESS

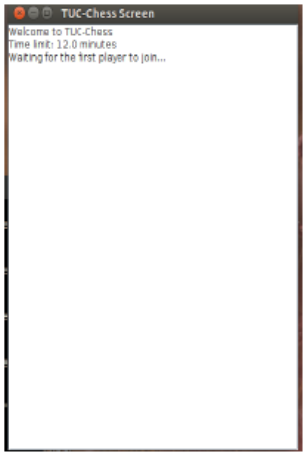
Γιώργος Δούκας AM: 2016030032

Εισαγωγή

Στα πλαίσια της 2ης εργασίας προγραμματισμού σχεδιάστηκε και υλοποιήθηκε ένα πρόγραμμα το οποίο παίζει το παιχνίδι TUC-CHESS, το οποίο αποτελεί μια παραλλαγή του κλασικού παιχνιδιού σκάκι. Το TUC-CHESS παίζεται σε μία σκακιέρα διαστάσεων 7 × 5 και υπάρχουν δύο παίκτες, ο λευκός και ο μαύρος, καθένας από τους οποίους έχει 7 πιόνια, 2 πύργους και 1 βασιλιά.



α. Αρχική κατάσταση της σκακιέρας.



β. Η οθόνη μηνυμάτων του παιχνιδιού κατά την αρχική του κατάσταση.

Οι κανόνες περιγράφονται συνοπτικά παρακάτω:

- Τα πιόνια μετακινούνται πάντα προς τη μεριά του αντιπάλου, είτε ένα βήμα μπροστά αν δεν υπάρχει κάποιος πεσσός σε εκείνη τη θέση είτε ένα βήμα διαγωνίως (αριστερά ή δεξιά) αν υπάρχει κάποιος πεσσός του αντιπάλου σε εκείνη τη θέση.
- Οι πύργοι μετακινούνται μέχρι και κατά τρεις θέσεις προς οποιαδήποτε κάθετη και οριζόντια κατεύθυνση (μη διαγώνια). Αν στις ενδιάμεσες θέσεις κάποιας κατεύθυνσης υπάρχει πεσσός τότε δεν επιτρέπεται η μετακίνηση.
- Οι βασιλιάδες μετακινούνται προς οποιαδήποτε κάθετη και οριζόντια κατεύθυνση κατά μία θέση.
- Τα πιόνια που φτάνουν στην τελευταία γραμμή (γραμμή 0 για τα λευκά και γραμμή 6 για τα μαύρα) απομακρύνονται από τη σκακιέρα, κερδίζοντας 1 πόντο.

- Όλοι οι πεσσοί που αιχμαλωτίζονται απομακρύνονται από τη σκακιέρα διαπαντός.
- Η αιχμαλώτιση ενός πιονιού αυξάνει τη βαθμολογία του παίκτη κατά 1 πόντο.
- Η αιχμαλώτιση ενός πύργου αυξάνει τη βαθμολογία του παίκτη κατά 3 πόντους.
- Η αιχμαλώτιση ενός βασιλιά αυξάνει τη βαθμολογία του παίκτη κατά 8 πόντους.
- Ένα bonus δίνει, στον παίκτη που μετακινείται στη θέση που αυτό βρίσκεται, 1 πόντο με πιθανότητα 0.95 και 0 πόντους με πιθανότητα 0.05.
- Μετά την κίνηση ενός παίκτη ένα νέο bonus εμφανίζεται σε κάποια από τις θέσεις στις οποίες δεν υπάρχει πεσσός ή bonus, με πιθανότητα 0.2.
- Για οποιαδήποτε κίνηση ABCD ισχύει ότι A, C ∈ {0, 1, 2, 3, 4, 5, 6} (7 γραμμές) και B, D ∈ {0, 1, 2, 3, 4} (5 στήλες).
- Κάθε παρτίδα λήγει είτε όταν ένας από τους δύο βασιλιάδες αιχμαλωτιστεί ή όταν στο παιχνίδι δεν έχει μείνει κανένας άλλος πεσσός εκτός των δύο βασιλιάδων ή όταν ξεπεραστεί το χρονικό όριο των 12 λεπτών.
- Για την αποστολή της κίνησής, ο client δεν πρέπει να ξεπερνάει τα 4 δευτερόλεπτα.

Στόχος του κάθε παίκτη είναι να πετύχει μεγαλύτερη βαθμολογία από την αντίπαλο. Ο παίκτης με τη μεγαλύτερη βαθμολογία στο τέλος της παρτίδας ανακηρύσσεται νικητής

Επικοινωνία με server

Η δημιουργία των γραφικών παραθύρων (της σκακιέρας και της οθόνης του παιχνιδιού) γίνεται αυτόματα με την έναρξη της λειτουργίας του server. Για να επιτευχθεί αυτή γράφουμε :

java -jar tuc-chess-server.jar

Στην συνέχεια τρέχουμε δυο clients της επιλογής μας, με τον πρώτο εξ αυτών να είναι ο άσπρος παίχτης και ο δεύτερος ο μαύρος. Ο client είναι σχεδιασμένος για να επικοινωνεί με τον server (στο πρωτόκολλο UDP η επικοινωνία των sockets πραγματοποιείται χωρίς σύνδεση). Για να τρέξει ο client απλά γράφουμε :

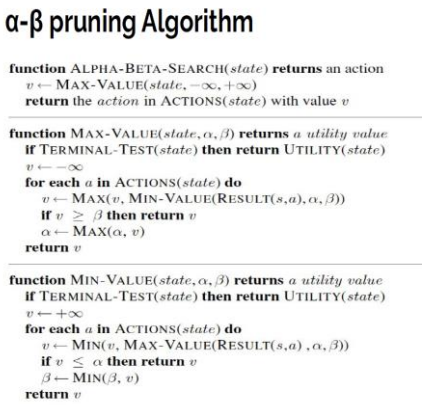
java -jar tuc-chess-client.jar	για τον απλό client που παίζει στην τύχη.
java -jar minimax.jar	για τον client που έχει υλοποιηθεί ο αλγόριθμος minimax με alpha-beta pruning
java -jar montecarlo.jar	για τον client που έχει υλοποιηθεί ο αλγόριθμος monte carlo

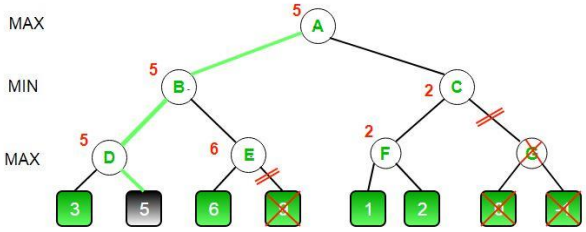
Client

Στο αρχείο *TUC-ChessClient-AI22\src\tuc_chess\World* γίνεται η επιλογή του αλγόριθμου που θα τρέξει ο client με την μεταβλητή *option* από την μέθοδο *selectAction().* Αν option=0 επιλέγεται ο Minimax και αν option= 1 επιλέγεται ο Monte Carlo. Στην μεταβλητή maxDepth ορίζουμε το μέγιστο βάθος του δέντρου του Minimax και με την pruning επιλέγουμε αν θέλουμε να κάνουμε κλάδεμα α-β στους κόμβους του δέντρου. Ακόμη οι μεταβλητές *endTime* και *iterations* είναι για τον αλγόριθμο

Monte Carlo, και χρησιμοποιούνται για να ορίσουμε ποσό χρόνο θα τρέχει ο αλγόριθμος κάθε φορά ώστε να γυρίσει κίνηση, και αν θέλουμε να τρέξει για συγκεκριμένο αριθμό iterations το καθορίζουμε από αυτή τη μεταβλητή.

Minimax





minimax

Ο αλγόριθμος αναζήτησης Minimax είναι ένας backtracking αλγόριθμος ο οποίος χρησιμοποιείται σε παιχνίδια zero sum 2 παικτών, για να βρει την βέλτιστη κίνηση δεδομένου ότι ο αντίπαλος είναι ορθολογικός και αλάνθαστος, δηλαδή παίζει πάντα την βέλτιστη για εκείνον κίνηση. Αν θεωρήσουμε ότι είμαστε ο παίχτης που κάνει κίνηση, σαν maximizer επιλέγουμε την κίνηση που μεγιστοποιεί την minimax τιμή και ο αντίπαλος ως minimizer επιλεγεί την κίνηση που ελαχιστοποιεί την minimax τιμή. Μέσω της συνάρτησης maxValue() επιλέγουμε τον κόμβο με την max τιμή και μέσω της minValue() τον κόμβο με την min τιμή. Κάθε κόμβος στο game tree αντιστοιχεί σε μια κατάσταση του chess board. Αν φτάσουμε σε τερματική κατάσταση όπως αναφέρεται στους κανόνες του παιχνιδιού ή φτάσουμε στο προκαθορισμένο μέγιστο βάθος (maxDepth = 5) τότε αξιολογούμε το board με την evaluation function. Ο αλγόριθμος καλεί μέσα στην maxValue() την minValue() και αντίστροφα, μέχρι να φτάσει σε τερματική κατάσταση, έτσι προσομοιώνεται η εναλλαγή του maximizer/minimizer. Επειδή δεν είναι δυνατόν να δούμε όλο το δέντρο του παιχνιδιού σε λογικό χρόνο καθώς η πολυπλοκότητα του αλγόριθμου είναι:

- χρονική πολυπλοκότητα: $O(b^m)$
- χωρική πολυπλοκότητα: $O(bm)$, b νόμιμες κινήσεις ανά στρώση (ply), m στρώσεις

χρησιμοποιούμε cutoff functions ώστε να σταματήσουμε την επέκταση κόμβων και καλώντας την συνάρτηση αξιολόγησης, αξιολογούμε την κατάσταση μέχρι εκείνο το σημείο. Δεν έχει υλοποιηθεί cutoff function, αντί αυτού ο αλγόριθμος minimax έγινε depth limited και ψάχνουμε το δέντρο μέχρι ένα συγκεκριμένο βάθος που έχουμε καθορίσει αρχικά. Εδώ έγινε επιλογή μέγιστου βάθους maxDepth =5 καθώς μέχρι αυτό το βάθος επέστρεφε ο αλγόριθμος κίνηση μέσα στα 4 δευτερόλεπτα, αλλά εξαρτάται ξεκάθαρα από τον υπολογιστή που τρέχει ο client.

α-β pruning

Βελτίωση του minimax αποτελεί το κλάδεμα α-β. Ο εκθετικός αριθμός καταστάσεων στον minimax συνεπάγεται και εκθετικό χρόνο εκτέλεσης, οπότε με το α-β pruning μπορούμε να κλαδέψουμε κόμβους που δεν επηρεάζουν την τελική απόφαση. Έτσι η ακριβής τιμή στη ριζά υπολογίζεται με λιγότερες επεκτάσεις:

- χρονική πολυπλοκότητα: από $O(b^m)$ σε $O(b^{m/2})$: μέγιστη δυνατή μείωση

- α: τιμή της καλύτερης επιλογής (=μεγαλύτερη τιμή) για τον MAX που έχει βρεθεί οπουδήποτε κατά μήκος της διαδρομής
- β: τιμή της καλύτερης επιλογής (=μικρότερη τιμή) για τον MIN που έχει βρεθεί οπουδήποτε κατά μήκος της διαδρομής

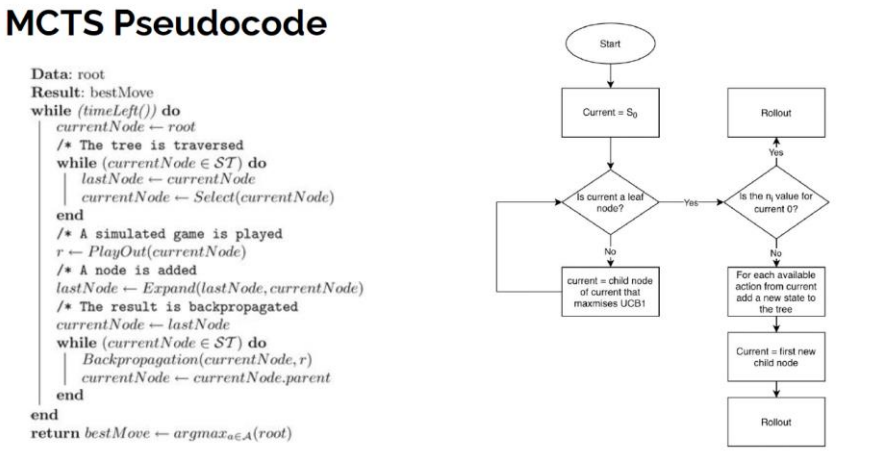
- κόμβοι MAX με τιμή μεγαλύτερη του β κλαδεύονται, διότι ο MIN δε θα επέτρεπε ποτέ να τους φτάσουμε
- κόμβοι MIN με τιμή μικρότερη του α κλαδεύονται, διότι ο MAX δε θα επέτρεπε ποτέ να τους φτάσουμε

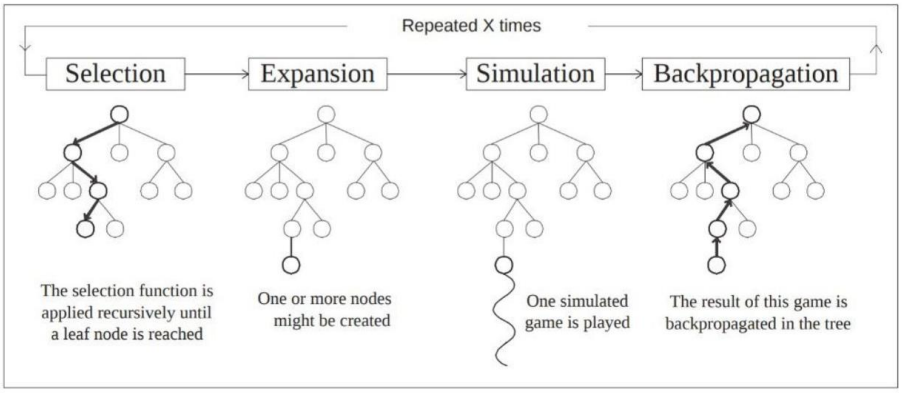
Πρακτικά:

- MAX κόμβος: ενημερώνει το α και κλαδεύει βάσει β
- MIN κόμβος: ενημερώνει το β και κλαδεύει βάσει α
- οι ενημερωμένες τιμές των α, β διαδίδονται μόνο προς τα κάτω

Forward pruning και singular extensions δεν υλοποιήθηκαν, απλά έγινε βελτίωση του αλγόριθμου κάνοντας πειραματισμούς με την συνάρτηση αξιολόγησης ώστε να αξιολογεί καλύτερα κάποιες καταστάσεις, όπως περιγράφεται παρακάτω.

Monte Carlo Tree Search





Ο αλγόριθμος MCTS (Monte Carlo Tree Search) είναι ένας αλγόριθμος πιθανολογικής αναζήτησης όπου αντιμετωπίζει αποτελεσματικά παιχνίδια με υψηλό branching factor, και επιλεγεί κινήσεις οι οποίες είναι πιο πιθανό να είναι καλές. Ο αλγόριθμος, για να πάρει μία απόφαση ενέργειας από μια τρέχουσα κατάσταση, χτίζει προοδευτικά ένα δένδρο αναζήτησης (υπό αντιπαλότητα) από την τρέχουσα κατάσταση, μέσω διαδοχικών επαναλήψεων εντός των οποίων χρησιμοποιούνται στάδια διάσχισης και επέκτασης του δένδρου αναζήτησης με επιλογή ενεργειών και αντίστοιχη προσθήκη κόμβων, καθώς και στάδια προσομοιώσεων μελλοντικών κινήσεων (μέσω εντελώς τυχαίων κινήσεων) ως ότου φτάσουμε σε έναν κόμβο-φύλλο. Η διαδικασία αυτή παράγει διαδοχικές

εκτιμήσεις αξίας για τους κόμβους του δένδρου, οι οποίες αναθεωρούνται μέσω μελλοντικών επαναλήψεων και τελικά επιλεγεί τον κόμβο με την καλύτερη εκτίμηση.

Ο αλγόριθμος λοιπόν αποτελείται από τέσσερις φάσεις:

- Selection
  Σε αυτή την αρχική κατάσταση ο αλγόριθμος ξεκινάει από τον κόμβο ριζά και επιλεγεί κόμβους παιδιά μέχρι να φτάσει σε κάποιο κόμβο φύλλο. Η επιλογή γίνεται με χρήση της συνάρτησης

  $$UCT = V_i + \sqrt{2}\sqrt{\frac{ln\,N}{ni}}$$

  όπου V_i είναι το μέσο reward του παιδιού i του συγκεκριμένου κόμβου, N είναι ο αριθμός των φορών που ο τρέχον κόμβος έχει δεχτεί επίσκεψη, και τέλος n_i είναι ο αριθμός των φορών που ο κόμβος-παιδί i έχει δεχτεί επίσκεψη.

- Expansion
  Αν ο κόμβος φύλλο που έχουμε φτάσει έχει ξαναεπισκεφτεί, επεκτείνουμε τον κόμβο και προσθέτουμε στο δέντρο τους κόμβους παιδιά του.

- Simulation
  Μετά το expansion ο αλγόριθμος επιλεγεί στην τύχη ένα κόμβο παιδί και προσομοιώνει ένα παιχνίδι από τυχαίες κινήσεις, μέχρι να φτάσει σε μια τερματική κατάσταση.

- Backpropagation
  Αφού φτάσει σε τερματική κατάσταση γίνεται αξιολόγηση του παιχνιδιού και το αποτέλεσμα αυτό ανανεώνεται στους κόμβους που κερδίζουν, προς τα πάνω μέχρι την ριζά. Επίσης αυξάνεται ο αριθμός επισκέψεων όλων των κόμβων που επισκέφτηκε ο αλγόριθμος.

  Όπως και στον minimax επιλέχθηκε στην φάση της προσομοίωσης να μην φτάνει μέχρι τέλους το τυχαίο παιχνίδι, αλλά να κάνει μια αξιολόγηση της κατάστασης όταν φτάσει σε βάθος maxDepth = 10. Αυτό έγινε επειδή τα προσομοιωμένα παιχνίδια ξοδεύαν τον διαθέσιμο χρόνο κίνησης και δεν επέτρεπε στον αλγόριθμο να κάνει αρκετές επαναλήψεις έτσι ώστε η πιο πιθανή κίνηση να είναι και η βέλτιστη. Καθώς λοιπόν οι κόμβοι στην προσομοίωση επιλέγονται τυχαία και θα έφταναν σε μεγάλο βάθος, η προσέγγιση τους δεν θα ήταν βέλτιστη λόγω της τυχαιότητας. Με το όριο στο βάθος παίρνουμε μια πιο ρεαλιστική προσέγγιση για την τελική κατάσταση όταν καλούμε την συνάρτηση αξιολόγησης.

  Σαν συνάρτηση αξιολόγησης αρχικά χρησιμοποιήθηκε μια που επέστρεφε μια σταθερή τιμή (+10 για νίκη και -10 για ήττα) για την κατάσταση που αξιολογούμε. Αλλά όταν αντικαταστάθηκε με την συνάρτηση αξιολόγησης του minimax, ο αλγόριθμος MCTS είχε καλύτερα αποτελέσματα νικώντας κάθε φορά τον MCTS με την αρχική συνάρτηση αξιολόγησης. Οι σταθερές τιμές στην αρχική συνάρτηση αξιολόγησης δεν προσέφεραν πληροφορία στο κατά ποσό κερδίζουμε ή χάνουμε, έτσι η μέση αξία κάθε κόμβου μετά τις πολλές επαναλήψεις δεν ήταν προσεγγιστικά καλή. Επομένως και στους δυο αλγορίθμους έχει χρησιμοποιηθεί η συνάρτηση αξιολόγησης που περιγράφεται παρακάτω. Ακόμη καθώς υπάρχει ο κανόνας για κίνηση μέσα σε 4 δευτερόλεπτα, αφήνουμε τον MCTS να τρέξει όσες περισσότερες επαναλήψεις είναι δυνατό μέσα σε 3 δευτερόλεπτα και έπειτα γυρνάει τον κόμβο με την καλύτερη αξία. Έτσι εκμεταλλευόμαστε την υπολογιστική δύναμη του τρέχοντος υπολογιστή, κάνοντας όσες περισσότερες επαναλήψεις είναι δυνατόν για καλύτερη εκτίμηση της βέλτιστης κίνησης.

Evaluation Function

Η συνάρτηση αξιολόγησης που έχει χρησιμοποιηθεί στους δυο παραπάνω αλγορίθμους έχει υλοποιηθεί στην μέθοδο *evaluate().* Αρχικά γίνεται ένα σκανάρισμα του board έτσι ώστε να δούμε ποια πιόνια έχουν μείνει ακόμα και την θέση τους. Η μέθοδος υπολογίζει την τιμή που θα επιστραφεί με βάση την παρακάτω συνάρτηση.

$$V(state) = | my\ (points + pieces) | - | opponent's\ (points + pieces) |$$

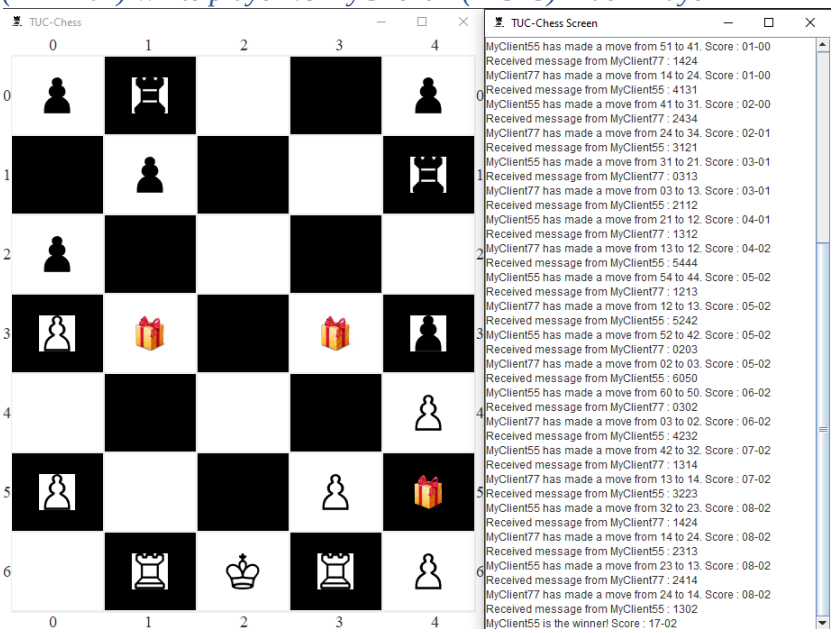
Στα pieces δίνουμε παραπάνω αξία στους πύργους (+10) από ότι στα πιόνια (+1) καθώς έχουν περισσότερη ευελιξία στο board και η αξία τους είναι μεγαλύτερη. Σε περίπτωση που χάνουμε στο σκορ μπορούμε να κυνηγήσουμε πιο εύκολα δώρα με τους πύργους πάρα με τα πιόνια, έτσι ώστε να προηγηθούμε στο σκορ. Επίσης σε ενδεχομένη κατάσταση που προκύπτουν ανταλλαγές με πιόνια, δεν θα θέλαμε να χάσουμε ένα πύργο για ένα πιόνι, οπότε δίνοντας τους μεγαλύτερη αξία προσπαθούμε να τους προστατεύσουμε και να αποφύγουμε τέτοιες ανταλλαγές.

Έπειτα γίνονται δυο έλεγχοι για καλύτερη αξιολόγηση. Ο πρώτος είναι αν έχουμε αιχμαλωτίσει τον βασιλιά του αντίπαλου στην τωρινή κατάσταση αλλά χάνουμε στο σκορ, αφαιρούμε από το υπολογισμένο value την τιμή -20, έτσι ώστε να αποφύγουμε την κατάσταση που οδηγεί σε ήττα. Ο δεύτερος έλεγχος γίνεται με την μέθοδο isKingChecked() για να δούμε αν ο βασιλιάς μας στην τωρινή κατάσταση που αξιολογούμε, είναι check από πεσσούς του αντίπαλου, και ενδεχομένως στον επόμενο γύρο γίνει checkmate. Έτσι μειώνουμε πάλι το value μιας τέτοιας κατάστασης ώστε να την αποφύγουμε.

Αποτελέσματα

Ακολουθούν αποτελέσματα από παιχνίδια μεταξύ των clients. Όπου *MyClient55* είναι ο minimax με α-β pruning και maxDepth = 5, *MyClient77* είναι ο MCTS και όπου *client<number>* είναι client που διαλέγει κινήσεις στην τύχη.

MyClient55(Minimax) White player vs MyClient77(MCTS) Black Player



Σε αυτή την περίπτωση ο MCTS έτρεχε για όσα iterations προλάβαινε μέσα σε 3 δευτερόλεπτα. Νικητής αναδείχτηκε ο client του Minimax.

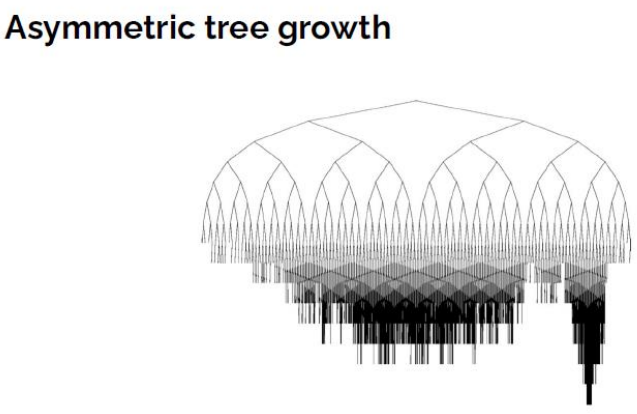
WHITE PLAYER	*SCORE*	*BLACK PLAYER*	*SCORE*	*Iterations of MCTS*
MyClient55	17	MyClient77	2	Time = 3 secs
MyClient77	20	client10	14	10
MyClient77	11	client16	2	50
client16	1	MyClient77	11	100
MyClient77	22	client7	4	1000
MyClient77	14	client6	1	10000
MyClient77	6	MyClient55	18	50
MyClient55	17	MyClient77	3	100
MyClient77	8	MyClient55	26	1000
MyClient77	10	MyClient55	22	10000
MyClient77	8	MyClient55	17	70000

Ο MCTS και με μικρό αριθμό iterations = 10 νικάει τον client που παίζει στην τύχη καθώς έχει την συνάρτηση αξιολόγησης. Παρόλα αυτά βλέπουμε ότι το παιχνίδι εξελίχθηκε αρκετά μέχρι να τελειώσει (20 - 14) διότι με τον μικρό αριθμό επαναλήψεων είχαμε και κακή εκτίμηση μέσης αξίας. Συνεχίζει να τον κερδίζει για όσο αυξάνουμε τις επαναλήψεις, και αφού γίνεται καλύτερος ο αλγόριθμος το παιχνίδι τερματίζεται πιο γρηγορά.

Παρόλα αυτά χάνει σε όλες τις παρτίδες με τον client του Minimax. Όσο περισσότερες επαναλήψεις βάζουμε τόσο πιο ακριβής θα είναι, έχοντας περισσότερες πιθανότητες για μια «δίκαιη» παρτίδα μεταξύ ισάξιων αντίπαλων. Όμως ο χρόνος εκτέλεσης μεγαλώνει εκθετικά που το καθιστά αδύνατο.

Συμπεράσματα

Ο Minimax σε σχέση με τον MCTS είναι καλύτερος αλγόριθμος καθώς είναι πλήρης (σε πεπερασμένα δέντρα) και βέλτιστη μέθοδος (ως προς την χειρότερη περίπτωση). Που σημαίνει ότι κάθε φορά επιλέγουμε την βέλτιστη κίνηση δεδομένου ότι ο αντίπαλος μας παίζει και αυτός εξίσου βέλτιστες κινήσεις. Σε προβλήματα όπως το σκάκι οπού το branching factor είναι μεγάλο και η χρονική πολυπλοκότητα αυξάνεται εκθετικά είναι συντηρητική στρατηγική και δεν μπορούμε να επισκεφτούμε όλο το δέντρο του παιχνιδιού, γιατρό ενδείκνυται ο MCTS οπού με πολλές προσομοιώσεις θα προσφέρει μια εκτίμηση για την καλύτερη επιλογή κόμβου.



Παρόλα αυτά με κατάλληλες τεχνικές όπως το κλάδεμα α-β, singular extensions και χρήση cutoff functions μπορούμε να έχουμε μια βελτιωμένη εκδοχή του minimax ο οποίος θα επιστρέφει βέλτιστη λύση για πεπερασμένο βάθους δέντρο, και χωρίς να επισκέπτεται όλους τους κόμβους του δέντρου παρά μόνο αυτούς που χρειάζονται. Όπως και στην περίπτωση μας οπού ο Minimax κερδίζει σε όλες τις παρτίδες τον MCTS.

Μέθοδοι

Class -> World
performMove(): Μέθοδος για να εκτελούμε κινήσεις στα διαφορετικά worlds του δέντρου
terminalTest(): Μέθοδος για τον έλεγχο τερματικής κατάστασης
evaluate(): Μέθοδος για τον υπολογισμό της συνάρτησης αξιολόγησης
isKingChecked(): Έλεγχος check του βασιλιά
cloneArray2D(): Αντιγραφή του board σε άλλο array[][]

Class -> Minimax
alphaBeta(): Μέθοδος για την εύρεση κίνησης χρησιμοποιώντας τον minimax αλγόριθμο
maxValue(): Επιστρέφει την μέγιστη τιμή των κόμβων παιδιών του
minValue(): Επιστρέφει την ελάχιστη τιμή των κόμβων παιδιών του

Class -> MonteCarloTreeSearch
findNextMove(): Μέθοδος για την εύρεση κίνησης χρησιμοποιώντας τον MCTS αλγόριθμο
selectPromisingNode(): Υλοποίηση της φάσης του selection καλώντας την *findNodeWithMaxUCT*() για την επιλογή του κόμβου με το max UCT.
simulateRollout(): Υλοποίηση της φάσης του simulation
backPropagation(): Υλοποίηση της φάσης του backpropagation

Class -> Node
expandNode(): Υλοποίηση της φάσης του expansion και στους δυο αλγόριθμους, επιστρέφοντας τους κόμβους παιδιά με τις διαθέσιμες κινήσεις.
getRandomChildNode(): Επιλογή τυχαίου κόμβου παιδιού για την φάση selection του MCTS
getChildWithMaxValue(): Τελική επιλογή του κόμβου με το μέγιστο μέσο reward Vi

Πηγές πληροφόρησης:

Minimax

https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

https://courses.cs.washington.edu/courses/csep573/12au/lectures/09-games.pdf?fbclid=IwAR2D-POAWjgYcug_IUijE8FxiPzZl2oHQ9SQxqMAMNXJlcGaYB_wxpC-IbY

Monte Carlo

https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238

https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

https://www.eclass.tuc.gr/courses/HMMY226/