ΠΛΗ311 Τεχνητή Νοημοσύνη
Εαρινό Εξάμηνο 2022 - Διδάσκων: Γιώργος Χαλκιαδάκης

Χρήστος Ιωαννίδης
ΑΜ:2018030006
Παπαδόπουλος Στέφανος
ΑΜ:2018030169

Αναφορά 2ης Προγραμματιστικής Εργασίας

Δομή αναφοράς:

- **Minimax**
  - **Διαδικασία σχεδιασμού minimax**
  - **Minimax Βασική Δομή Κώδικα**
  - **Πειραματισμοί κατα την διάρκεια υλοποίησης του αλγορίθμου Minimax**
  - **A-b pruning**
  - **Cut Off Function**
  - **Heuristic**

- **MCTS**
  - **Διαδικασία σχεδιασμού MCTS**
  - **Πειραματισμοί κατά την διάρκεια υλοποίησης του αλγορίθμου MCTS**
  - **MCTS Δομή Κώδικα**

- **Συγκρίσεις αλγορίθμων**

Σημείωση: Έχουν γίνει 2 υλοποιήσεις του αλγορίθμου MCTS, μια εκ των οποίων είναι η τελική και άλλη μία πειραματική οποία γράφτηκε σε μία προσπάθεια να βελτιωθεί το win rate του αλγορίθμου. Παρόλο που η πειραματική έκδοση είναι ημιτελής αποφασίστηκε να συμπεριληφθεί στο τελικό αρχείο μιας και τρέχει κανονικα απλα δεν παράγει συστηματικά τα επιθυμητά αποτελέσματα. Περισσότερες πληροφορίες για την πειραματική έκδοση έχουν συμπεριληφθεί στην παράγραφο Πειραματισμοί κατά την διάρκεια υλοποίησης του αλγορίθμου MCTS.

● Minimax

Διαδικασία σχεδιασμού minimax:

- Αρχικά προστέθηκε ένα board σαν όρισμα στις συναρτήσεις blackMoves,whiteMoves έτσι ώστε να μην επιδρούν μόνο σε ενα global board αλλα σε οποιοδήποτε board τους δοθεί, κάτι που θα φανεί χρήσιμο αργότερα στις προσομοιώσεις.
- Στην συνέχεια προστέθηκε η συνάρτηση simMakeMove() η οποία βασίστηκε στην makeMove() αλλα με κύρια διαφορά να μπορεί να κάνει αλλαγές σε οποιοδήποτε local board δοθεί και όχι μόνο στο global.
- Στην συνέχεια προστέθηκε η αναδρομική συνάρτηση miniMax(String[][] Simboard,int searchDepth,String playerColour) η οποία αξιοποιώντας τις παραπάνω προσθήκες και μια συνάρτηση αξιολόγησης κάνει αναζήτηση σε όλες τις πιθανές καταστάσεις που μπορούν να προκύψουν απο το δοθέν board μέχρι ένα ορισμένο επίπεδο (το οποίο ορίζεται από το searchDepth) προκειμένου να βρεί και να επιστρέψει την βέλτιστη εκτιμώμενη κίνηση.
- Τέλος προστέθηκε η συνάρτηση selectAction() η οποία είναι υπεύθυνη να αρχικοποιήσει και να καλέσει την συνάρτηση miniMax(String[][] Simboard,int searchDepth,String playerColour) καθώς και να επιστρέψει την τελική λύση.

Minimax Βασική Δομή Κώδικα:

- **selectAction():** Αρχικοποιεί και τρέχει για κάθε πιθανή αρχική κίνηση του χρώματος του παίκτη την συνάρτηση **miniMax**. Απο τις τιμές που επιστρέφονται επιλέγει και επιστρέφει την κίνηση με την καλύτερη δυνατή τιμή για τον client που την κάλεσε.
- **miniMax(String[][] Simboard,int searchDepth):** Προσομοιώνει αναδρομικά μια αλληλουχία εναλλαγων κινήσεων μεταξύ μαύρου και άσπρου παίκτη μέχρι ένα συγκεκριμένο επίπεδο και μετά από αξιολόγηση όλων των τελικών καταστάσεων, επιστρέφει την τιμή της ευρετικής της κατάστασης με το καλύτερο εκτιμώμενο αποτέλεσμα για τον client που την κάλεσε.
- **simMakeMove(Simboard,x1,y1,x2,y2):** Παίρνει σαν είσοδο ένα board και μια κίνηση και την εφαρμόζει πάνω του. Τέλος επιστρέφει το αποτέλεσμα στην μορφή ενός καινούριου board.

Πειραματισμοί κατα την διάρκεια υλοποίησης του αλγορίθμου Minimax :

A-b pruning:

Υπο την θεώρηση οτι ο αντίπαλος θα επιλέξει την βέλτιστη δυνατή κίνηση προκειμένου να αυξήσει τις πιθανότητες νίκης του προκύπτουν κάποιοι υποκλάδοι του δεντρου οι οποίοι δεν αξίζει να εξερευνήθουν μιας και ο αντίπαλος θα φροντίσει να μην φτάσουμε ποτέ στις καταστάσεις που αντιπροσωπεύουν. Αντιθέτως με την προσπέραση τους μπορούμε να γλυτώσουμε πολύτιμο χρόνο ο οποίος μπορεί να αξιοποιηθεί στην εύρεση μιας καλύτερης και πιο ρεαλιστικά εφικτής λύσης. Στην επιλογή ποιών υποκλάδων θα προσπελαστούν επικεντρώνεται και ο παραπανω αλγόριθμος.

Στον κώδικα ο αλγόριθμος υλοποιείται με μερικες προσθηκες στα ορίσματα της συνάρτησης αναζήτησης καθώς και μερικους ελεγχους κατα την επιστροφη της οι οποίοι μπορούν να οδηγήσουν σε τερματισμό αναζήτησης υποκλάδου.
A-b pruning ON/OFF με την χρήση της μεταβλητής **abPruningOn**.

Cut Off Function:

cutoffFunc(int minLevelOfSearch,int maxLevelOfSearch)**,** Συνάρτηση αποκοπής της αναζήτησης, Λειτουργεί με την παραδοχή της υπόθεσης οτι μια αναζήτηση μεγαλύτερου βάθους θα φέρει καλύτερα αποτελέσματα. Παίρνει ώς όρισμα ένα ελάχιστο και ένα μέγιστο βάθος αναζήτησης και επίσης λαμβάνει πληροφορία μέσω global μεταβλητων για το σκορ. Αν ο παίκτης χάνει τότε αυξάνει το βάθος αναζήτησης και αν κερδίζει το μειώνει αλλα πάντα μέσα στα όρια που δίνονται σαν ορίσματα.

Heuristic:

evaluationFunc(String[][] Simboard), η δομή της είναι σχετικά απλή με έξοδο να ακολουθεί την παρακάτω μορφη:

Output=white Points+(est.) presents Taken By White-black Points-(est.) presents Taken By Black
Variable Explanations
white Points,black Points=pawns_in_ play*pawn_Weight+rooks_in_ play*rook_Weight+king_exists*king_Weight
(est.) presents Taken By White,Black= estimated number of presents taken by the time it reaches evaluation state. *(Υπάρχουν μεταβλητές που μετράνε τον αριθμό δώρων που θα έχει μαζέψει ο κάθε παίκτης μέχρι την κατάσταση που καλείται η ευρετική με βάση τις πληροφορίες για τα υπάρχοντα δώρα στην σκακιέρα κατα την διάρκεια της αναζήτησης).*

Επίσης στα Weights των πύργων και των βασιλιάδων δόθηκε σημαντικά μεγαλύτερη τιμή από την αξία αιχμαλώτισης τους μιας και θεωρήθηκε ότι είναι πολύ ευέλικτα και πολύτιμα πιόνια σε σύγκριση με τα απλα στρατιωτάκια, επομένως η διαφύλαξη τους μπορεί να δώσει καλύτερες πιθανότητες νίκης στον αλγόριθμο μακροπρόθεσμα.

Σημείωση:*Τα ονόματα των παραπάνω μεταβλητών είναι γραμμένα με τέτοιο τρόπο ώστε να βοηθήσουν τον αναγνώστη να κατανοήσει πιο εύκολα την δομή του κώδικα και επομένως υπάρχουν διαφορές με τα ονόματα των μεταβλητών του κώδικα.*

● MCTS

Διαδικασία σχεδιασμού MCTS:

- Αρχίσαμε με την υλοποίηση των φάσεων selection & expansion του αλγορίθμου, γράφοντας την αναδρομική συνάρτηση *select()* στην κλάση Node η οποία επιμελείται και τα δύο στάδια, και ταυτόχρονα την αλλαγή του αριθμού των επισκέψεων σε κάθε κόμβο. Φτιάξαμε 2 constructors για την κλάση που χειρίζονται την δημιουργία του αρχικού root και την δημιουργία των παιδιών οιουδήποτε Node. Φτιάξαμε την συνάρτηση *mctsInit()* στην κλάση World, που επιμελείται την δημιουργία του αρχικού root. Τέλος, φτιάξαμε την συνάρτηση *uctScore(int parentVisits)* η οποία ανανεώνει το uct του κόμβου (this) με βάση τις καταγεγραμμένες τιμές επισκέψεων και νικών, και το όρισμα που είναι ο αριθμός επισκέψεων του γονέα.
- Παραλλήλως, φτιάξαμε παραλλαγές των δοσμένων συναρτήσεων *selectAction(), whiteMoves()/blackMoves()*, και *makeMove()*, η οποίες κυρίως λειτουργούν σε ένα όρισμα String[][] anyBoard, για να μπορούμε να κάνουμε κινήσεις πάνω στους κόμβους του δέντρου μας χωρίς να επηρεάζουμε την τρέχουσα κατάσταση. Επιπλέον, προσθέσαμε την δυνατότητα επιστροφής (return) συγκεκριμένων πληροφοριών από κάθε συνάρτηση, οι οποίες θα φανούν χρήσιμες σε διάφορα σημεία του αλγορίθμου, πχ πότε κάνουμε throw την καινούρια μας *GameOverException* για να ειδοποιήσουμε πως ένας Node είναι τελικός.
- Φτιάξαμε την βασική μας συνάρτηση ***mctsAction(String enemyMove)***, η οποία βρίσκει το καινούριο root (τον κόμβο στον οποίο βρισκόμαστε αμέσως μετά την κίνηση του αντίπαλου) σε κάθε γύρο (δεν φτιάχνουμε το δέντρο απο την αρχή κάθε φορά, αλλά πλοηγούμαστε στο κατάλληλο υποδέντρο για να κάνουμε τον αλγόριθμο πιο γρήγορο), ανανεώνει κάποιες πληροφορίες σε κάθε Node σχετικά με τα τυχαία δώρα που εμφανίστηκαν αυτόν τον γύρο, και μετά κάνει έναν αριθμό από **iterations**, που δεν είναι σταθερός κάθε γύρο, αλλά βασίζεται στον διαθέσιμο χρόνο μας (~3 δευτερόλεπτα για ασφάλεια), ουτως ώστε να κάνει όσο το δυνατόν περισσότερα simulations σε κάθε γύρο. Σε κάθε **iteration** πρώτα καλούμε την *Node.select()* πάνω στο root για τις φάσεις selection & expansion, παίρνουμε τον Node που θα χρησιμοποιήσουμε, και μετά υλοποιούμε την φάση simulation σε ένα εσωτερικό loop που κάνει γύρους με τυχαίες κινήσεις από κάθε παίκτη μέχρι να πάρουμε *GameOverException*. Για την φάση backpropagation, απλά παίρνουμε τον γονέα κάθε κόμβου μέχρι να φτάσουμε στο root, ανανεώνοντας των αριθμό νικών και το uct όπως πάμε.
- Ύστερα από όλα τα iterations, επιλέγουμε τον παιδί του root με τις περισσότερες επισκέψεις ως το καινούριο root (τον κόμβο στον οποίο βρισκόμαστε αμέσως πριν την κίνηση του αντίπαλου), κάνουμε την κίνηση στο board της κλάσης World μας, και επιστρέφουμε την ίδια κίνηση για το μήνυμα που στέλνουμε στον server.
- Επίσης φτιάχνουμε διάφορες helper συναρτήσεις στην κλάση World που χρησιμοποιούμε για δημιουργία deep copies, την έξοδο της κονσόλας, και άλλα.

Πειραματισμοί κατά την διάρκεια υλοποίησης του αλγορίθμου MCTS:

Τυχαιότητα:

- Πειραματιστήκαμε με διάφορους τρόπους για να βελτιώσουμε τον αλγόριθμο, 2 εκ των οποίων έχουν να κάνουν με το τυχαίο στοιχείο: Πρώτα υλοποιήσαμε λειτουργικότητα που ελέγχει την απόκλιση στο σκορ και στην εμφάνιση δώρων κάθε γύρο (1 και 1 κίνηση) μεταξύ του πραγματικού και του προβλεπόμενου από τον αλγόριθμο, και ενημερώνει με DFS τα παιδιά του κόμβου ως προς το καινούριο σκορ και τα καινούρια δώρα. Αυτή η αλλαγή καθέστησε τον αλγόριθμο ικανό να μένει λειτουργικός σε μεγαλύτερης διάρκειας παιχνίδια.
- Αργότερα ξαναγράψαμε ολόκληρο τον κώδικα από την αρχή, με την ιδέα να αντικαταστήσουμε το ντετερμινιστικό δέντρο, με ένα μη-ντετερμινιστικό, στο οποίο τα παιδιά έκαστου κόμβου αναπαριστούν μια κίνηση ενός παίκτη, μαζί με την εμφάνιση/μη-εμφάνιση ενός δώρου σε ένα συγκεκριμένο από όλα τα δυνατά (ελεύθερα) σημεία. Αυτό αύξησε το branch factor από ~8 σε ~120, αλλά μας επέτρεψε να έχουμε έναν αλγόριθμο που “ακολουθούσε” το παιχνίδι με ακρίβεια (εκτός της 5% πιθανότητας να μην πάρουμε 1 πόντο για ένα δώρο, που θεωρήσαμε αμελητέα, απλά αλλάζοντας την υπολογιζόμενη αξία κάθε δώρου ως 0.95 αντί για 1).
- Στον καινούριο αλγόριθμο επίσης κάναμε αλλαγή από single-threading σε multi-threading: Η ιδέα είναι πως αντί να περιμένουμε τον αντίπαλο να κάνει την κίνησή του και να αρχίσουμε το loop του αλγορίθμου μόλις πάρουμε το σχετικό μήνυμα (πχ T11424000099), θα αρχίσουμε το loop πριν από την αρχή του παιχνιδιού, μόλις στείλουμε το μήνυμα join στον server, θα το αφήσουμε να τρέχει σε διαφορετικό thread και θα αλλάζουμε την ρίζα του δέντρου όταν παίρνουμε μηνύματα “T1/T2…”, και, ύστερα από ~3.2 δευτερόλεπτα, θα ζητάμε την προτεινόμενη κίνηση, όλα χωρίς να σταματάμε το loop (εκτός από 2 μικρά critical sections). Αυτό μας επέτρεπε να εκμεταλλευτούμε τον διαθέσιμο χρόνο μας στο έπακρο. Ο αλγόριθμος έτρεχε πολύ καλύτερα έναντι του δοσμένου client (τυχαίες κινήσεις), αλλά σε match έναντι του minimax, έμενε από μνήμη πριν προλάβει να κερδίσει.
- Ακολούθησε περίοδος ημερών για debugging, και δοκιμάσαμε διαφορετικούς τρόπους συγχρονισμού (κυρίως polling με while(...){Thread.sleep(...)}, και της κλάσης Semaphore της Java 5, που είναι interrupt-driven) αλλά λόγω της πολυπλοκότητας του debugging σε multi-threaded κώδικα εξαντλήσαμε το χρόνο και την υπομονή μας πριν καταφέρουμε να πάρουμε πλήρως λειτουργικό κώδικα.

MCTS Βασική Δομή Κώδικα:

Στην κλάση World:

- **simulateAction(Simboard):** Το αντίστοιχο της selectAction(), αλλά για δεδομένο board. Επιστρέφει μια τυχαία επιλεγμένη κίνηση, ή πετάει *GameOverException* στις 3 (εκτός της υπέρβασης των 14ων λεπτών) περιπτώσεις που το παιχνίδι δεν μπορεί να συνεχίσει: Ένας Βασιλιάς είναι νεκρός, όλα τα κομμάτια, εκτός από τους 2 Βασιλιάδες είναι νεκρά, όλα τα κομμάτια ενός παίκτη είναι “μπλοκαρισμένα” (δεν υπάρχει νόμιμη κίνηση που μπορούμε να κάνουμε).

- **mctsAction(enemyAction):** Ο αλγόριθμος που τρέχει, με όρισμα την κίνηση που μόλις έκανε ο αντίπαλος (null στον πρώτο γύρο αν είμαστε λευκοί). Περιλαμβάνει το κύριο loop.
- **makeMove(Simboard,x1,y1,x2,y2):** Παίρνει σαν είσοδο ένα board και μια κίνηση και την εφαρμόζει πάνω του. Τέλος επιστρέφει το αποτέλεσμα στην μορφή ενός καινούριου board.

Στην κλάση Node:

- **select():** Η φάσεις selection & expansion, σε μία αναδρομική συνάρτηση που εν τέλει επιστρέφει το επιλεγμένο παιδί της ρίζας πίσω στην mctsAction() για simulation & backpropagation.
- **mctsAction(enemyAction):** Ο αλγόριθμος που τρέχει, με όρισμα την κίνηση που μόλις έκανε ο αντίπαλος (null στον πρώτο γύρο αν είμαστε λευκοί). Περιλαμβάνει το κύριο loop.
- **makeMove(Simboard,x1,y1,x2,y2):** Παίρνει σαν είσοδο ένα board και μια κίνηση και την εφαρμόζει πάνω του. Τέλος επιστρέφει το αποτέλεσμα στην μορφή ενός καινούριου board.

Απόδοση του αλγορίθμου για διαφορετικό αριθμό iterations:

Ο αλγόριθμος χρησιμοποιεί περιορισμό χρόνου, αντί για iterations, και ως εκ τούτου, μέσα σε ~3.2 δευτερόλεπτα μπορεί να κάνει από 10k έως και 100k iterations κάθε γύρο όσο φτάνουμε πιο κοντά στο τέλος του παιχνιδιού. Κατά κανόνα όμως, ~0.8 δευτερόλεπτα αρκούν για να κερδίσουμε τον random moves αλγόριθμο, ενώ >3 δευτερόλεπτα κάνουν μικρή διαφορά έναντι πιο έξυπνων αλγορίθμων.

● Συγκρίσεις αλγορίθμων

VS	Random	Minimax (ab pruning ON)	MCTS (number of iterations=10k)
Random	-	Very high minimax win rate	Very high MCTS win rate
Minimax (ab pruning OFF)	Very high minimax win rate, slow response time due to minimax extensive exploration of cases	About equal win rate for both	Higher win rate for minimax
MCTS(number of iterations=10k)	-	-	About equal win rate for both

VS	MCTS (number of iterations=1k)	MCTS (number of iterations=10k)	MCTS (number of iterations=50k)
MCTS (number of iterations=1k)	-	90% win rate for 10k	100% win rate for 50k
MCTS (number of iterations=10k)	-	-	80% win rate for 50k
MCTS(number of iterations=100k)	-	-	-

Results Explanation

- Και ο minimax αλγόριθμος καθώς και ο MCTS έχουν εξαιρετικό win rate απέναντι σε αλγόριθμο που κάνει τυχαίες κινήσεις. Αυτό το αποτέλεσμα είναι το αναμενόμενο μιας και ο σκοπός αυτών τών αλγορίθμων είναι εξ αρχής να βελτιώσουν το win rate σε παιχνίδια ειδικά αντίπαλα σε απλοικούς αλγορίθμους όπως ο Random.
- Ο (υλοποιημένος) mcts υστερεί του minimax, λόγω της σχετικής απλότητας του παιχνιδιού (λίγοι γύροι, μικρό branch factor, νίκη με score που κάνει εύκολη την λειτουργία της ευρετικής) σε σχέση με πχ, κανονικό σκάκι που τείνει να έχει ~3πλάσιο branch factor. Αυτή η απλότητα επιτρέπει στον minimax να εξετάσει κινήσεις που απαιτούν πολύ συγκεκριμένη στρατηγική (δηλαδή έχουν μικρή πιθανότητα νίκης), τις οποίες ο mcts μπορεί να χάσει με c=1.414 που χρησιμοποποιούμε.

- Στα παιχνίδια mcts vs mcts, φαίνεται η συμπεριφορά των diminishing returns ως προς τον αριθμό των iterations, αφού χρειαζόμαστε 100 iterations/victory percent από την περίπτωση 1k vs 1k (50%) σε 1k vs 10k (90%), αλλά ~1333 iterations/victory percent από την περίπτωση 10k vs 10k (50%) σε 10k vs 50k (80%).