TECHNICAL UNIVERSITY OF CRETE

Τεχνητή Νοημοσύνη
ΠΛΗ 311

2η Προγραμματιστική Άσκηση

Καλαϊτζάκης Παναγιώτης

Προπτυχιακός Φοιτητής - 2018030188

pkalaitzakis@isc.tuc.gr

Λαμπρινάκης Γεώργιος

Προπτυχιακός Φοιτητής - 2018030017

glamprinakis@isc.tuc.gr

Τμημα Ηλεκτρολογων Μηχανικων,
Μηχανικων Υπολογιστων

31 Μαΐου 2022

Εισαγωγή

Το αντικείμενο της παρούσας εργαστηριακής άσκησης είναι η δημιουργία ενός πράκτορα αναζήτησης για το παιχνίδι TUC-CHESS. Το παιχνίδι αυτό είναι μια παραλλαγή του κλασσικού παιχνιδιού σκάκι και οι κανόνες του καλύπτονται άκρως αναλυτικά στην εκφώνηση της άσκησης, συνεπώς δεν θα αφιερωθεί χρόνος στην παρακάτω παρουσίαση για την ανάλυση των κανόνων. Αξίζει να σημειωθεί ωστόσο, ότι το βασικότερο χαρακτηριστικό του παιχνιδιού που το διαφοροποιεί από το σκάκι είναι ότι η αιχμαλώτιση συγκεκριμένων θέσεων επιφέρει πόντους στον παίκτη που την πραγματοποίησε. Το γεγονός αυτό μας επιτρέπει να υλοποιήσουμε τον αλγόριθμο minimax, όπως ζητάτε στην εκφώνηση. Κατόπιν, θα τον βελτιστοποιήσουμε χρησιμοποιώντας τον αλγόριθμο α-β-pruning συνδυαστικά με όποιες άλλες μεθόδους βελτιστοποίησης προκύψουν κατά την εκπόνηση της εργασίας, οι οποίες εξηγούνται παρακάτω.

Για την εκπόνηση του πρότζεκτ, όπως και στην πρώτη προγραμματιστική εργασία, αρχικά χρειάστηκε να αναλύσουμε και να κατανοήσουμε το template του κώδικα που μας δόθηκε έτοιμο. Βασιζόμενοι στην δομή των αρχείων World.java, Client.java, υλοποιήσαμε τελικά τις παρακάτω επιπλέον κλάσεις:

- Move: Η βασική κλάση που χρησιμοποιήθηκε για να μοντελοποιήσουμε μια συγκεκριμένη κατάσταση-κόμβο του παιχνιδιού. Κάθε φορά που ελέγχουμε για μια νέα εφικτή κίνηση για οποιονδήποτε παίκτη (whiteMoves(), blackMoves()) δημιουργείται ένα νέο instance της Move. Το αντικείμενο αυτό περιέχει πληροφορίες σχετικά με τις συντεταγμένες της αντίστοιχης κίνησης, την όψη της σκακιέρας μετά την εκτέλεση της κίνησης (moveBoard -¿ deepcopy), καθώς και άλλες χρήσιμες μεταβλητές που χαρακτηρίζουν την κατάσταση του κάθε παίκτη μετά την συγκεκριμένη κίνηση. Λόγου χάριν, για κάθε ζωντανό βασιλιά, ο παίκτης του αντίστοιχου χρώματος λαμβάνει 7 μονάδες, για κάθε πύργο 3 και για κάθε πιόνι 1. Επιπλέον περιέχουν flags που υποδηλώνουν εάν σε αυτή την κίνηση αιχμαλοτίστηκε κάποιο πίονι από οποιονδήποτε βασιλιά ή πύργο.

- MoveComparator: Κλάση που υλοποιεί τον comparator για δυο κινήσεις ορίσματα. Συγκρίνει τις απόλυτες αξίες τις εκάστοτε κίνησης και επιστρέφει 1 ή -1 αναλόγως το αποτέλεσμα της σύγκρισης. Μας επιτρέπει να σορτάρουμε τις παραγόμενες κινήσεις από μια συγκεκριμένη κατάσταση (Move Ordering).

- Agent: Κλάση που περιέχει όλες τις απαραίτητες συναρτήσεις αναζήτησης που μας ζητούνται. Πιο συγκεκριμένα, οι συναρτήσεις blackMoves() και whiteMoves() που δόθηκαν έτοιμες στην κλάση World.java μετακινήθηκαν στον Agent και τροποποιήθηκαν ελαφρώς ώστε να παράγουν αντικείμενα της κλάσης Move. Οι μεταβλητές που αφορούν ένα instance μιας νέας κίνησης να υπολογίζονται προοδευτικά με κάθε κλήση των blackMove(), whiteMoves(), αλλά τα scores των κινήσεων υπολογίζονται από τους αλγορίθμους minimaxABP(), minimizerABP() και maximizerABP().

Για την εκτέλεση του προγράμματος απαιτείται η χρήση terminal με την χρήση της παρακάτω εντολής:

java - jar MM_Agent-TUC-Chess.jar args, όπου args= "minimax" OR "MCTS"

Αλγόριθμος Minimax

Αρχικά υλοποιήσαμε τον αλγόριθμο Minimax, όπως παρουσιάζεται στον παρακάτω ψευδοκώδικα:

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

---------------------------------------------------

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

---------------------------------------------------

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Η αναζήτηση αρχίζει με τον υπολογισμό όλων των δυνατών κινήσεων που μπορεί να πραγματοποιήσει ο πράκτοράς μας και ταξινομούνται με βάση την αξία τους. Η τεχνική αυτή ονομάζεται Move Ordering και χρησιμοποιείται καθώς σε όλες τις αναζητήσεις η σειρά επέκτασης κόμβων έχει πολύ μεγάλη σημασία για την επίδοση του εκάστοτε αλγορίθμου. Κατόπιν ο αλγόριθμος χτίζει προοδευτικά ένα δέντρο αναζήτησης υπό αντιπαλότητα (min-max), μέχρι ένα συγκεκριμένο βάθος (8). Σε κάθε κίνηση του minimizer (παίκτης), αυτός προσπαθεί να ελαχιστοποιήσει τα κέρδη του αντιπάλου, ενώ ο maximizer να μεγιστοποιήσει τις απώλειες του αντιπάλου του. Ακολουθώντας την λογική αυτή, όταν τελειώσουν οι αναδρομικές κλήσεις η αρχική συνάρτηση καταλήγει στην κίνηση που θεωρεί βέλτιστη. Σημειώνεται, στον υπολογισμό τον μελλοντικών δυνατών κινήσεων δεν αξίζει να συμπεριλαμβάνονται περιπτώσεις όπου εμφανίζονται νέα δώρα σε κενές θέσεις της σκακιέρας, οπότε

εξετάζονται μόνο τα υπάρχοντα δώρα της πραγματικής τρέχουσας κατάστασης. Η ταχύτητα του αλγορίθμου χωρίς Move Ordering και χωρίς περιορισμό του βάθους, οδηγεί σε πολύ αργές αναζητήσεις.

Βελτίωση αλγορίθμου με α-β-Prunning

Στη συνέχεια της εργασίας, με σκοπό να βελτιώσουμε την ταχύτητα εκτέλεσης του αλγορίθμου Minimax, ενσωματώσαμε την λειτουργικότητα του αλγορίθμου α-β-pruning. Ο αλγόριθμος αυτός υλοποιήθηκε βάσει του ψευδοκώδικα που φαίνεται παρακάτω:

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(%α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Αξίζει να σημειωθεί ότι τροποποιήσαμε τον παραπάνω ψευδοκώδικα, αντικαθιστώντας την συνθήκη ελέγχου τερματικής κατάστασης με τον έλεγχο αποκοπής «if CUTOFF-FUNCTION(state,depth) then return EVALUATE(state)». Κατά την evaluate, καλείται

η συνάρτηση αξιολόγησης κίνησης (wEval, bEval) ενώ κατά την CUTOFF-FUNCTION η συνάρτηση που χρησιμοποιεί τα flags που δηλώνουν αιχμαλώτιση πιονιού από κάποιο πύργο ή βασιλία για να αποφασιστεί αν πρέπει να γίνει κλάδεμα την τρέχουσα κατάστασης στην αναζήτηση σε βάθος. Παράλληλα, η μόνη διαφορά σε σχέση με τον απλό αλγόριθμο minimax, είναι η προσθήκη των μεταβλητών α και β, οι οποίες χρησιμοποιούνται ως όρια αποκοπής της εις βάθος αναζήτησης για συγκεκριμένες κινήσεις των minimizer και maximizer που σίγουρα δεν τους συμφέρουν να εξετάσουν. Με αυτόν τον τρόπο επιτυγχάνεται το κλάδεμα (prunning) δέντρο αναζήτησης του παιχνιδιού, και ο χρόνος αναζήτησης της βέλτιστης κίνησης μειώνεται σημαντικά. Το Move Ordering βοηθάει πάρα πολύ τον χρόνο εκτέλεσης στον συγκεκριμένο αλγόριθμο αφού διαπιστώσαμε μέσω του Debugger ότι όταν οι κινήσεις εξετάζονται με σειρά προτεραιότητας βάσει της αξίας τους, το κλάδεμα γίνεται με βέλτιστο τρόπο.

Στοχεύσαμε σε καταστάσεις οι οποίες δεν είναι τερματικές αλλά έχει διακοπεί η περαιτέρω εξέταση τους λόγο περιορισμού βάθους. Σε αυτές τις καταστάσεις θεωρήσαμε ότι αξίζει να εξετάσουμε περισσότερο τις κινήσεις που αφορούν την μετακίνηση ενός βασιλιά ή ενός πύργου πάνω στο board όταν εκείνοι τρώνε άλλα πιόνια. Θεωρήσαμε ότι αυτές οι καταστάσεις είναι σημαντικές καθώς ο ρόλος των συγκεκριμένων πιονιών είναι σαφώς σημαντικότερος από τα άλλα, σε συνδυασμό βέβαια με το γεγονός ότι εάν φάνε κάποιο άλλο πιόνι μπορεί μελλοντικά να απειληθούν. Συνεπώς κατασκευάσαμε μια συνάρτηση η οποία ελέγχει εάν η κίνηση η οποία έχει κοπεί λόγο περιορισμού βάθους αφορά κάποιον βασιλιά ο οποίος τρώει ένα άλλο πιόνι, τότε αυξάνουμε περαιτέρω το βάθος κατά 4 μονάδες ενώ εάν η κίνηση αφορά κάποιον πύργο τότε αυξάνουμε το βάθος κατά 2 μονάδες.

Κάθε φορά που καλείται η evaluate(), επιστρέφεται πάντα η διαφορά της εκτίμησης του παίκτη που παίζει μείον του αντιπάλου του βάσει ορισμένων ελέγχων. Πιο συγκεκριμένα, η συνάρτηση αξιολόγησης κίνησης δέχεται ως όρισμα μια κίνηση η οποία έχει θεωρηθεί ως τερματική είτε επειδή είναι, είτε επειδή η αναζήτηση έχει φτάσει στο μέγιστο προκαθορισμένο βάθος. Για κάθε παίκτη υπολογίζεται η τελική εκτίμηση για το αν η κίνηση είναι προς όφελός του ή όχι, χρησιμοποιώντας τις ήδη υπολογισμένες αξιολογήσεις κατάστασης wEval και bEval και προσθέτοντας το τελικό διαμορφωμένο αρχικό σκορ από την αρχική κατάσταση όπου ξεκίνησε η αναζήτηση μέχρι την τρέχουσα που αξιολογείται. Σε περίπτωση που ο παίκτης που την καλεί προηγείται στην εκτίμηση του διαμορφωμένου σκορ και μπορεί να αιχμαλωτίσει τον βασιλιά του αντιπάλου τότε η κίνηση θεωρείται εξαιρετική και επιλέγεται. Εάν όμως συμβαίνει το αντίθετο, τότε η κίνηση θεωρείται κάκιστη αφού αν γίνει και υπάρχει μεγάλη διαφορά στο σκορ (μεγαλύτερη του 7), το παιχνίδι τερματίζεται. Στην ουσία αυτό που προσπαθούμε να εκτιμήσουμε συνολικά με την evaluate(), είναι αν η τερματική κατάσταση οδηγεί σε μεγιστοποίηση του σκορ υπό συνθήκες ευνοϊκές για τον εκάστοτε παίκτη.

Αλγόριθμος Monte Carlo Search Tree

Η υλοποίηση του αλγορίθμου Monte Carlo Search Tree βασίστηκε στον παρακάτω ψευδοκώδικα:

```
class Node:
  def __init__(self, m, p): # move is from parent to node
    self.move, self.parent, self.children = m, p, []
    self.wins, self.visits  = 0, 0

  def expand_node(self, state):
    if not terminal(state):
      for each non-isomorphic legal move m of state:
        nc = Node(m, self) # new child node
        self.children.append(nc)

  def update(self, r):
    self.visits += 1
    if r==win: self.wins += 1

  def is_leaf(self):
    return len(self.children)==0

  def has_parent(self):
    return self.parent is not None

def mcts(state):
  root_node  = Node(None, None)
  while time remains:
    n, s = root_node, copy.deepcopy(state)
    while not n.is_leaf():    # select leaf
      n = tree_policy_child(n)
      s.addmove(n.move)
    n.expand_node(s)          # expand
    n = tree_policy_child(n)
    while not terminal(s):    # simulate
      s = simulation_policy_child(s)
    result = evaluate(s)
    while n.has_parent():     # propagate
      n.update(result)
      n = n.parent

return best_move(tree)
```

Η λογική του αλγορίθμου χωρίζεται στα εξής τέσσερα στάδια:

- Selection: Γίνεται επιλογή του επόμενου κόμβου προς επέκταση με βάση το πόσο έκτακτο κρίνεται αυτό.

- Expansion: Επέκταση του επιλεγμένου κόμβου επιλέγοντας της καλύτερη εφικτή κίνηση.

- Simulation: Επιλογή τυχαίων κινήσεων (μετά την επέκταση) μέχρι την λήξη του παιχνιδιού.

- Backpropagation: Ενημέρωση των στατιστικών των κόμβων που οδήγησαν στην τρέχουσα κατάσταση.

Κατ' ουσίαν ο αλγόριθμος εκτελείται πολλές φορές και ανανεώνει τα δεδομένα του για το πως καταλήγουν οι διαθέσιμες κινήσεις. Όσες περισσότερες φορές προλάβει να εκτελεστεί, τόσο καλύτερη η εκτίμησή του για το ποια είναι η καλύτερη, εκ των διαθέσιμων κινήσεων. Ακριβώς επειδή η απόδοσή του βασίζεται στο πόσο γρήγορα χτίζεται σε βάθος το δέντρο αναζήτησης, υστερεί συγκριτικά με τους δύο προηγούμενους αλγόριθμους.