ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σκακιστικές Μηχανές: Επισκόπηση Μεθοδολογιών και Υλοποίηση Προσέγγισης PVS/NNUE

Τιμόθεος Γ. Παλαιολόγος

Επιβλέπων **Παναγιώτης Σταματόπουλος**, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2022

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σκακιστικές Μηχανές: Επισκόπηση Μεθοδολογιών και Υλοποίηση Προσέγγισης PVS/NNUE

Τιμόθεος Γ. Παλαιολόγος

Α.Μ.: 1115201700112

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Παναγιώτης Σταματόπουλος**, Επίκουρος Καθηγητής

ΠΕΡΙΛΗΨΗ

Σε αυτή την πτυχιακή εργασία μελετώνται οι κύριες δομές μια σκακιστικής μηχανής καθώς και οι νεότερες τεχνικές εύρεσης βέλτιστης κίνησης με χρήση νευρωνικών δικτύων. Θα μελετηθούν οι κύριοι τρόποι αναπαράστασης της δομής του ταμπλό με ιδιαίτερη έμφαση σε αυτή των bitboards. Στην συνέχεια θα γίνει μια αναφορά στην μεθοδολογία παραγωγής κινήσεων με χρήση προ-υπολογισμένων πινάκων και τεχνικών τέλειου κατακερματισμού. Θα συζητηθούν οι διάφορες παραλλαγές των αλγορίθμων MCTS και PVS και οι τρόποι με τους οποίους μπορούν να παραλληλοποιηθούν για γρηγορότερη εκτέλεση. Τέλος θα παρουσιαστεί η νεότερη αρχιτεκτονική δικτύων NNUE για την στατική αξιολόγηση θέσεων και οι διαφορές της με πιο σύνθετα μοντέλα όπως αυτό του Alpha zero. Στα πλαίσια αυτής της εργασίας υλοποιήθηκε και μια σκακιστική μηχανή με χρήση του μοντέλου NNUE, του αλγορίθμου PVS με τις αντίστοιχες βελτιστοποιήσεις κλαδέματος και της αναπαράστασης bitboards. Οι λεπτομέρειες υλοποίησης παρέχονται στην τελευταία ενότητα.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή νοημοσύνη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σκακιστικό πρόγραμμα, νευρωνικά δίκτυα, αλγόριθμοι αναζήτησης, αναπαραστάσεις ταμπλό, αξιολόγηση θέσης

ABSTRACT

In this dissertation we study the primary components of chess engines, as well as the latest techniques for finding optimal moves using neural networks. The various board representation will be analysed, with special emphasis on that of bitboards. We will refer to the main methods for producing moves using pre-calculated lookup tables with perfect hashing. The various variants of MCTS and PVS algorithms and the ways in which they can be parallelised for faster execution will be discussed. Finally, the latest NNUE network architecture for static position evaluation and its differences with more complex models such as Alpha zero will be presented. As part of this work, a chess engine was developed using the NNUE model, the PVS algorithm with the corresponding pruning optimizations and the representation of bitboards. Implementation details are provided in the last section.

SUBJECT AREA: Artificial intelligence

KEYWORDS: chess engine, neural networks, search algorithms, board representations, positional evaluation

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΠΡΟΛΟΓΟΣ

Μέσω αυτής της πτυχιακής εργασίας κατάφερα να υλοποιήσω μια σκακιστική μηχανή που νικάει συστηματικά έναν μέσο παίχτη. Έμαθα αρκετές νέες ιδέες και αλγορίθμους για την λύση επιτραπέζιων παιχνιδιών και ευελπιστώ να επεκτείνω τις γνώσεις μου στο μέλλον και σε πιο σύνθετα προβλήματα πραγματικού χρόνου.

Για την συνεισφορά του σε αυτό το έργο ευχαριστώ τον επιβλέποντα καθηγητή κ. Σταματόπουλο.

1. ΕΙΣΑΓΩΓΗ

Το σκάκι αποτελεί ένα διαχρονικό επιτραπέζιο παιχνίδι στρατηγικής δύο παιχτών. Παρόλο που οι πρώτες εκδοχές του πρωτοεμφανίστηκαν τον 13ο αιώνα, παραμένει δημοφιλές σε παγκόσμιο επίπεδο μέχρι και σήμερα. Μέσω οργανισμών όπως τον FIDE, έχει γίνει μια εφικτή επαγγελματική καριέρα για τους κορυφαίους παίχτες. Ενώ σαν παιχνίδι δεν περιέχει άμεσα την έννοια του αθλητισμού, μπορούμε να παρατηρήσουμε ότι έχει πολλά χαρακτηριστικά που το καθιστούν ως «άθλημα». Το σκάκι κρύβει ένα κύρος, για αρκετά χρονιά ήταν ένα αυτοκρατορικό παιχνίδι (κάτι που παρατηρούμε από την ισχύ των πιονιών και τους κανόνες του). Σήμερα όταν σκεφτόμαστε μεγάλους σκακιστές, μας έρχονται στο μυαλό οι έννοιες της αφοσίωσης της γνώσης και της ευστροφίας.

Γρήγορα οι πρώτες σκακιστικές μηχανές δημιουργήθηκαν με απώτερο σκοπό να ανταγωνιστούν τον άνθρωπο. Με την ραγδαία τεχνολογική άνθιση της επιστήμης της πληροφορικής, το σκάκι αποτέλεσε και αποτελεί ακόμα αντικείμενο μελέτης του κλάδου της τεχνητής νοημοσύνης. Σήμερα με την υπολογιστική ισχύ και νέες τεχνικές που διαθέτουμε βλέπουμε να έχουν αντιστραφεί οι ρόλοι. Ο άνθρωπος τώρα πια είναι αυτός που συμβουλεύεται τις μηχανές για την ανάλυση των παιχνιδιών του. Πλέον το ζήτημα δεν είναι αν μια μηχανή μπορεί να κερδίσει τον άνθρωπο αλλά αν μια μηχανή μπορεί να ανταγωνιστεί άλλες πιο ισχυρές μηχανές άλλων τεχνολογιών. Πρέπει να επισημανθεί όμως ότι όλες οι μελετώμενες τεχνικές δεν περιορίζονται μόνο στο «βέλτιστο» παίξιμο μιας παρτίδας ενός επιτραπέζιου παιχνιδιού. Τα τελευταία χρόνια βλέπουμε να γίνονται διάσημες πιο γενικευμένες τεχνικές. Σε αφηρημένο επίπεδο το σκάκι αποτελεί καλό παράδειγμα ενός σύνθετου προβλήματος αποφάσεων με ανάγκη για υψηλή ευστοχία, κάτι που μπορεί να επεκταθεί και σε άλλα πιο σημαντικά προβλήματα στην πραγματική ζωή.

“it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance” [1]

2. ΘΕΩΡΗΤΙΚΟ ΣΚΕΛΟΣ

2.1 Βασικά χαρακτηριστικά σκακιού

Για να καταλάβουμε περαιτέρω γιατί ένα επιτραπέζιο παιχνίδι παραμένει ένα τόσο μεγάλο αντικείμενο ενδιαφέροντος μέχρι και σήμερα θα πρέπει να καταλάβουμε πρώτα μερικά βασικά χαρακτηριστικά του.

Οι παρακάτω έννοιες προέρχονται από την θεωρία παιγνίων και συνήθως χρησιμοποιούνται για την περιγραφή οικονομικών μοντέλων. Στην συγκεκριμένη περίπτωση θα τις αξιοποιήσουμε για την ανάλυση των βασικών χαρακτηριστικών του παιχνιδιού του σκακιού.

- Σε μια παρτίδα κάθε παίχτης έχει πλήρη επίγνωση της «κατάστασης» του παιχνιδιού, τις πιθανές κινήσεις του αντιπάλου και τα αποτελέσματα που επιφέρουν αυτές. Δεν υπάρχει ο παράγοντας της τύχης ούτε αποκρύπτεται κάποια πληροφορία από τους παίχτες. Στην βιβλιογραφία αυτό ορίζεται ως τέλεια πληροφόρηση (Perfect information). Το παιχνίδι κρίνεται από την ικανότητα των παιχτών και όχι από εξωτερικούς παράγοντες. Παιχνίδια όπως η τρίλιζα, checkers, go, Xiangqi αποτελούν άλλα παραδείγματα παιχνιδιών με τέλεια πληροφόρηση. Αντιθέτως παιχνίδια όπως το poker, τάβλι, ζάρια έχουν ατελή πληροφόρηση (imperfect information) καθώς είτε υπάρχει ο παράγοντας της τύχης (ζάρι) είτε οι παίχτες αποκρύπτουν πληροφορίες προς τους υπολοίπους (κρυφές κάρτες).
- Το σκάκι είναι ένα διαδοχικό παιχνίδι (Sequential game). Οι παίχτες παίζουν εναλλάξ ο ένας μετά τον άλλον. Σε συνδυασμό με το χαρακτηριστικό της τέλειας πληροφόρησης ο κάθε παίχτης μπορεί στην σειρά του να αναλογιστεί την θέση του αντίπαλου για να λάβει την “βέλτιστη” επιλογή.
- Παίζεται από δύο παίχτες (υπάρχουν εκδόσεις που αυτό δεν ισχύει άλλα αναφερόμαστε για την κλασική περίπτωση). Μια κίνηση που επιφέρει κέρδος για την μια πλευρά έχει αντίστοιχες αρνητικές επιπτώσεις για την άλλη. Όταν ο ένας παίχτης νικάει ο άλλος χάνει. Το σκάκι αποτελεί ένα παιχνίδι μηδενικού αθροίσματος (zero sum game). Για παράδειγμα αν η μια πλευρά κερδίζει κατά 5 πόντους η άλλη χάνει κατά το ίδιο πλήθος με αντίθετο πρόσημο (-5). Στην περίπτωση της ισοπαλίας και οι δυο πλευρές έχουν μηδενικό κέρδος.
- Αποτελεί ένα πεπερασμένο παιχνίδι. Υπάρχουν προκαθορισμένοι κανόνες αυτόματης ισοπαλίας για την αποφυγή «άπειρων» παιχνιδιών (Infinite play) (Πχ ο κανόνας ισοπαλίας για την περίπτωση εμφάνισης πανομοιότυπων θέσεων πάνω από δύο φορές αποτρέπει παιχνίδια κυκλικών μοτίβων.)

2.2 Προβλήματα αναζήτησης

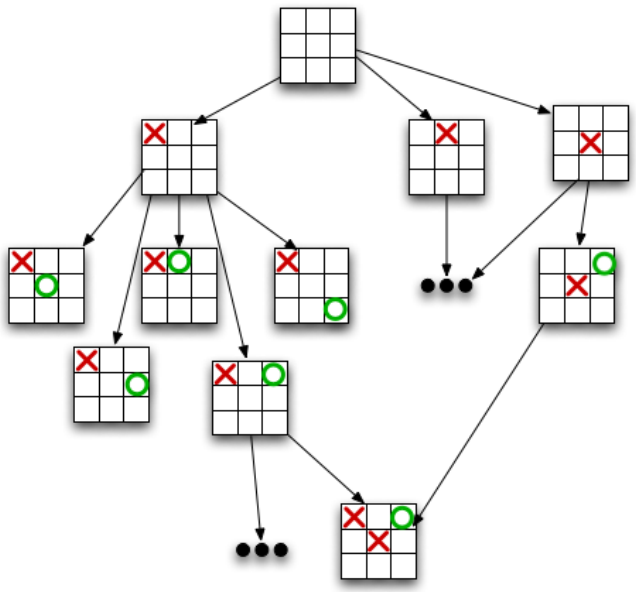
Η εύρεση της «βέλτιστης» κίνησης στα επιτραπέζια παιχνίδια αποτελεί ένα πρόβλημα αναζήτησης. Ως πρόβλημα αναζήτησης ορίζουμε ένα πρόβλημα που μπορεί να περιγραφεί από μια αρχική και μια τελική κατάσταση στόχου καθώς και ένα ενδιάμεσο σύνολο καταστάσεων (χωρίς να είναι απαραίτητο ο τελικός στόχος να είναι μοναδικός καθώς μπορεί να υπάρχουν πολλαπλές βέλτιστες λύσεις για μια πιθανή κατάσταση). Στην περίπτωση των παιχνιδιών, τελικές καταστάσεις ορίζουμε τις καταστάσεις που προκύπτουν μετά από μια νικητήρια κίνηση. Είναι σημαντικό να επισημανθεί ότι τα περισσότερα παιχνίδια μηδενικού αθροίσματος δυο παιχτών, παρουσιάζουν αντίστοιχα χαρακτηριστικά όπως το σκάκι.

2.2.1 Χώρος αναζήτησης

Ως χώρο αναζήτησης ορίζουμε το σύνολο των πιθανών καταστάσεων ενός προβλήματος αναζήτησης. Στην ειδική περίπτωση των turn based επιτραπέζιων παιχνιδιών μπορούμε να σκεφτούμε κάθε κατάσταση ως το ταμπλό του παιχνιδιού με όλα τα απαραίτητα χαρακτηριστικά που το περιγράφουν. (πχ μπορεί να μην αρκεί μόνο η πληροφορία για την θέση των πιονιών. Στο σκάκι ένα τέτοιο χαρακτηριστικό θα ήταν η δυνατότητα για την κίνηση en-passant).

2.2.2 Δένδρο αναζήτησης

Μπορούμε πολύ εύκολα να αναπαραστήσουμε τον χώρο αναζήτησης με μια δεντρική δομή όπου κάθε κόμβος του δένδρου περιγράφει μια πιθανή κατάσταση του ταμπλό και κάθε κόμβος συνδέεται με το επόμενο επίπεδο με την επιλογή μιας κίνησης. Τα φύλλα (τελικοί κόμβοι) του δένδρου θα αποτελούν πιθανές τελικές καταστάσεις. Στην Εικόνα 1 φαίνεται ένα παράδειγμα ενός τέτοιου δένδρου για το πιο απλοϊκό παιχνίδι της τρίλιζας.

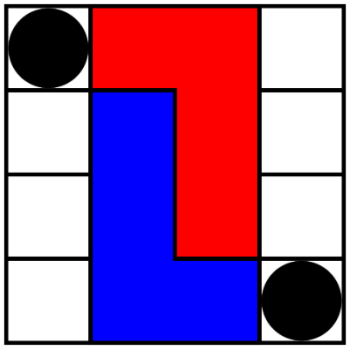


Εικόνα 1: Παράδειγμα δένδρου αναζήτησης τρίλιζας

2.2.3 Πολυπλοκότητα

Πολλά παιχνίδια μηδενικού αθροίσματος έχουν λυθεί με τον αναλυτικό υπολογισμό ολόκληρου του δένδρου αναζήτησης. Έπειτα οι βέλτιστες λύσεις μπορούν να βρεθούν για κάθε κατάσταση και να αποθηκευτούν σε μια βάση δεδομένων. Η μέθοδος αυτή αποτελεί μια προσέγγιση ωμής βίας (brute force) και έχει το πλεονέκτημα ότι δεν χρειάζεται να ξέρουμε τίποτα παραπάνω πέρα από τους κανόνες του παιχνιδιού για να δουλέψει αποτελεσματικά. Η υλοποίηση του αλγορίθμου για τον υπολογισμό του χώρου αναζήτησης είναι επίσης πολύ απλή καθώς απαιτείται να κάνουμε δύο πράγματα. Να επεκτείνουμε κάθε κόμβο του δένδρου ελέγχοντας τις πιθανές κινήσεις που προέρχονται από αυτόν. Να ελέγχουμε αν φτάσαμε σε τελική κατάσταση στόχου. Η διάσχιση του δένδρου μπορεί να γίνει μέσω αναδρομικών κλήσεων.

Μερικά παραδείγματα λυμένων παιχνιδιών είναι η τρίλιζα και το παιχνίδι L.



Εικόνα 2: Το παιχνίδι L

Έχουμε παραλείψει όμως ένα σημαντικό πρόβλημα. Τα παιχνίδια αυτά είναι συγκριτικά πιο απλά. Μπορούμε να μάθουμε την βέλτιστη στρατηγική για το παιχνίδι της τρίλιζας εντός μερικών λεπτών. Για να ποσοτικοποιήσουμε καλυτέρα την δυσκολία αυτή ας δούμε το πλήθος των πιθανών παρτίδων (Στην βιβλιογραφία αυτό ορίζεται ως το State-space complexity [2], δηλαδή το πλήθος των νόμιμων παιχνιδιών που αρχίζουν από την αρχική κατάσταση). Η τρίλιζα έχει ~250.000 διαφορετικά παιχνίδια ενώ το L game μόνο ~2300. Το παιχνίδι του σκακιού αντιθέτως υπολογίζεται ότι έχει πιο πολλά έγκυρα παιχνίδια από τα μόρια, του έως τώρα γνωστού σύμπαντος, με προσεγγιστικό κάτω φράγμα τον αριθμό του Shannon 10^{123} για παιχνίδια που κατά μέσο όρο κρατούν 80 κινήσεις [1]. Στον παρακάτω πίνακα μπορούμε να παρατηρήσουμε την εκθετική αύξηση των πιθανών παιχνιδιών με την αύξηση των κινήσεων. Αντίστοιχα αποτελέσματα βλέπουμε και σε παιχνίδια όπως το Go, με 10^{360} πιθανά παιχνίδια. Για να αντιληφθούμε τι θα σήμαινε αυτό για τον υπολογισμό του χώρου αναζήτησης ας θεωρήσουμε ότι έχουμε έναν υπερυπολογιστή με δείκτη NPS (nodes per second) της τάξης του 10^6 (ως NPS ορίζουμε το πλήθος τον κόμβων (nodes) του δένδρου αναζήτησης που μπορούμε να επισκεφθούμε ανά δευτερόλεπτο). Ο υπολογισμός του δένδρου αναζήτησης για το σκάκι θα έπαιρνε $\sim 10^{110}$ χρόνια. Ακόμα και με κάποια ραγδαία αύξηση της ταχύτητας αντιλαμβανόμαστε ότι ένα τέτοιος υπολογισμός δεν θα ήταν εφικτός ούτε στο σύντομο μέλλον.

Άρα μπορούμε να συμπεράνουμε ότι παιχνίδια σαν το σκάκι αποτελούν πολύ σύνθετα προβλήματα με περιθώρια βελτίωσης και εξερεύνησης νεών τακτικών, ενώ οι προσεγγίσεις ωμής βίας είναι πιο κατάλληλες για παιχνίδια μικρής πολυπλοκότητας χώρου καταστάσεων [2]. Έτσι για την περαιτέρω μελέτη τους στα πλαίσια της τεχνητής νοημοσύνης απαιτούνται στοχαστικές μέθοδοι εξερεύνησης και πιο έξυπνοι τρόποι διάσχισης του χώρου αναζήτησης για την εύρεση βέλτιστων λύσεων.

Πίνακας 1: Πλήθος πιθανών παιχνιδιών

Κινήσεις	Παιχνίδια
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

2.3 Λυμένα παιχνίδια

2.3.1 Ιδανικό παίξιμο

Ως «ιδανικό παίξιμο» (Perfect Play) ορίζουμε την συμπεριφορά ή στρατηγική με την οποία ο παίχτης οδηγείται στο βέλτιστο πιθανό αποτέλεσμα ασχέτως των κινήσεων του αντιπάλου του. Η λύση ενός παιχνιδιού υπονοεί την εύρεση της «ιδανικής στρατηγικής».

2.3.2 Κατηγορίες λύσεων

Διαχωρίζουμε τα λυμένα παιχνίδια σε 3 βασικές κατηγορίες. Τα πολύ αδύναμα (Ultra-weak), αδύναμα (weak) και δυνατά (strong). Κάθε κατηγορία αποτελεί υποσύνολο της προηγουμένης της.

- Πολύ αδύναμα: Όταν μπορούμε να αποδείξουμε (πχ με εις άτοπο απαγωγή όπως το strategy-stealing argument) αν με ιδανικό παίξιμο ο πρώτος παίχτης θα νικήσει, χάσει ή φέρει το παιχνίδι σε ισοπαλία. Δεν περιγράφει το πως θα οδηγηθεί το παιχνίδι σε μια τέτοια κατάσταση. Για παράδειγμα γνωρίζουμε ότι το παιχνίδι της τρίλιζας οδηγείται πάντα σε ισοπαλία.
- Αδύναμα: Όταν υπάρχει ένας αλγόριθμος που μπορεί να κερδίσει (ή να φέρει σε ισοπαλία) ένα παιχνίδι από την αρχή μέχρι το τέλος αποδεικνύοντας ότι οι κινήσεις του είναι οι βέλτιστες
- Δυνατά: Όταν υπάρχει αλγόριθμος που μπορεί να παράξει βέλτιστες κινήσεις από κάθε κατάσταση ασχέτως αν έχουν συμβεί λάθη στο παρελθόν από οποιαδήποτε πλευρά.

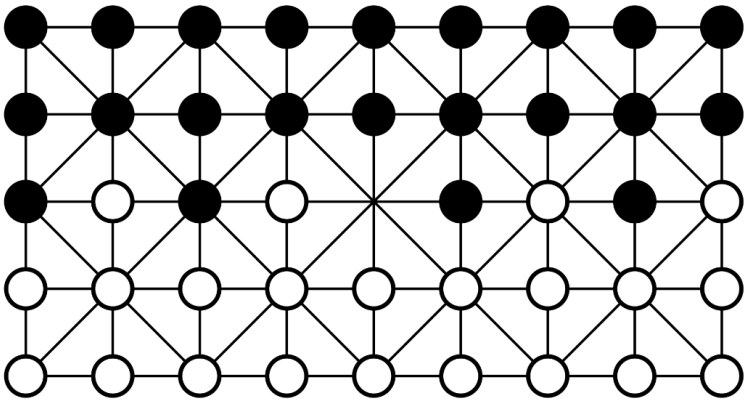
Εικόνα 3: Το παιχνίδι Hex

Ακολουθεί μια συλλογή λυμένων παιχνιδιών και η αντίστοιχη κατηγοριοποίηση τους

Πίνακας 2: Παραδείγματα λυμένων παιχνιδιών

Παιχνίδι	**Κατηγορία**	**Αποτέλεσμα**
Connect four	«Δυνατά» στην κλασική περίπτωση, weakly solved για αυθαίρετο μέγεθος ταμπλό.	1ος παίχτης
Hex	«Πολύ αδύναμα», για κάθε μέγεθος ταμπλό.	1ος παίχτης
Sim	«Αδύναμα»	2ος παίχτης
Fanorona	«Αδύναμα»	Ισοπαλία
Pentomino	«Αδύναμα»	1ος παίχτης

Παρατηρούμε ότι η σειρά με την οποία παίζουν οι παίχτες μπορεί να έχει καθοριστικό ρόλο στο αποτέλεσμα του παιχνιδιού. Τα άσπρα πιόνια στο σκάκι έχουν στατιστικό πλεονέκτημα της τάξης του ~5% [3] ακόμα και σε παιχνίδια μεταξύ υπολογιστών. Αυτό όμως παραμένει ακόμα μια εικασία καθώς δεν έχει βρεθεί κάποια απόδειξη.



Εικόνα 4: Το παιχνίδι Fanorona

2.4 Μερικώς λυμένα παιχνίδια

Όπως είδαμε στην ενότητα 2.2.3 αρκετά παιχνίδια έχουν υψηλή πολυπλοκότητα ώστε να μπορούν να λυθούν πλήρως, αυτό όμως δεν μας αποτρέπει από το να βρούμε μερικές λύσεις για απλούστερες εκδοχές τους. Για παράδειγμα στο σκάκι μπορούμε να μελετήσουμε βέλτιστες κινήσεις σε σενάρια που τα περισσότερα πιόνια έχουν “ανταλλαχθεί” (λεγόμενα ως end games). Με αυτό τον τρόπο ο χώρος αναζήτησης είναι συγκριτικά μικρότερος.

Στις ερχόμενες υπό-ενότητες θα αναλυθεί γιατί αυτό είναι εφικτό βάση της ιδιότητας της σύγκλισης. [2]

2.4.1 Μετατροπή

Μια κίνηση M από την κατάσταση A στην κατάσταση B θεωρείται μετατροπή (conversion) αν δεν υπάρχει καμία διαμόρφωση παιχνιδιών όπου θα μπορούσε να οδηγήσει στην κατάσταση A μέσω της B. Για παράδειγμα στο σκάκι όταν χάνεται ένα πιόνι δεν μπορεί να ξανά εμφανιστεί. Οι μετατροπές συνηθώς συνδέονται με τις προσθήκες και αφαιρέσεις αντικειμένων, χωρίς αυτό να είναι περιοριστικό (πχ στο σκάκι τα απλά πιόνια (pawns) μπορούν να πάνε μόνο μπροστά).

2.4.2 Σύγκλιση

Έστω ότι χωρίζουμε το σύνολο όλων των νόμιμων θέσεων σε ασύνδετα υποσύνολα κλάσεων. Κάθε κλάση περιέχει όλες τις καταστάσεις παιχνιδιών με ισάριθμο πλήθος πιονιών (pieces). Έστω ένα κατευθυνόμενο γράφημα G με κόμβους τις προαναφερόμενες κλάσεις. Οι κόμβοι συνδέονται με ακμές μεταξύ τους μόνο αν υπάρχει μια κατάσταση P στο A και μια νόμιμη κίνηση M όπου να οδηγεί σε μια κατάσταση Q στο B.

Ένα παιχνίδι μπορεί να αποκλίνει να συγκλίνει είτε να είναι αμετάβλητο κατά την εξέλιξη του.

Έχουμε σύγκλιση αν για την πλειοψηφία των ακμών του γράφου από την κλάση A στην κλάση B ο πληθάριθμος του B είναι μικρότερος του A.

Έχουμε απόκλιση αν ισχύει το αντίθετο

Έχουμε αμεταβλητότητα όταν δεν υπάρχουν κινήσεις που οδηγούν σε conversion ή δεν υπάρχει ούτε απόκλιση ούτε σύγκλιση.

Η ιδιότητα αυτή είναι κρίσιμη καθώς για παιχνίδια που συγκλίνουν μπορούμε να παράξουμε βάσεις δεδομένων με βέλτιστες λύσεις. Αντιθέτως στις άλλες δύο περιπτώσεις η πολυπλοκότητα είτε παραμένει ίδια είτε χειροτερεύει και η πλήρης λύση παραμένει αδύνατη για παιχνίδια μεγάλης πολυπλοκότητας χώρου καταστάσεων.

Πίνακας 3: Παραδείγματα παιχνιδιών διαφόρων συμπεριφορών σύγκλισης

Παιχνίδι	**Ιδιότητά**	**Λόγος**
Checkers	Συγκλίνει	Τα πιόνια μειώνονται όσο προχωράει το παιχνίδι
Othello	Αποκλίνει	Κάθε κίνηση προσθέτει ένα πιόνι
Shogi	Αμετάβλητο	Μπορούν να ελευθερωθούν παγιδευμένα πιόνια αρά το πλήθος παραμένει σταθερό.
Connect-four	Συγκλίνει	Πεπερασμένο πλήθος κελιών
Hex	Συγκλίνει	Πεπερασμένο πλήθος κελιών

2.5 Αναπαραστάσεις ταμπλό

Για να μπορούμε να παίξουμε ένα παιχνίδι στον ψηφιακό κόσμο απαιτείται να υπάρχει ένα πρόγραμμα που υλοποιεί το σύνολο των κανόνων που το περιγράφουν και να προσφέρει την δυνατότητα της ανθρώπινης διεπαφής μέσω ενός γραφικού περιβάλλοντος. Το προγράμματα αυτά τα ονομάζουμε συνήθως engines (Μηχανές). Πολλές σκακιστικές μηχανές δίνουν την δυνατότητα στους παίχτες, πέρα από το να παίξουν με άλλους ανθρώπους, να παίξουν και με αντίπαλο τον ίδιο τον υπολογιστή. Όπως αναλύσαμε στα προηγούμενα κεφαλαία η εύρεση βέλτιστων λύσεων είναι ένα δύσκολο έργο σε προβλήματα μεγάλης πολυπλοκότητας. Η δημιουργία ενός τεχνητού αντιπάλου AI (artificial intelligence) προκύπτει από την λύση του προβλήματος της βέλτιστης κίνησης. Πριν όμως μελετήσουμε τις διάφορες τεχνικές για την επίτευξη «τεχνητής νοημοσύνης» πρέπει να αναλύσουμε τα πιο βασικά συστατικά που αποτελούν μια σκακιστική μηχανή. Οι τεχνικές που θα αναλυθούν θα εστιάζουν στο παιχνίδι του σκακιού αλλά μπορούν να γενικευτούν και σε αλλά παιχνίδια παρόμοιου τύπου.

Η σκακιστική μηχανή χρειάζεται μια εσωτερική δομή για την αναπαράσταση του ταμπλό και την αποθήκευση των θέσεων των πιονιών. Να επισημανθεί ότι για τα περισσότερα παιχνίδια θα χρειαστούν και επιπλέον πληροφορίες που δεν εντάσσονται στην δομή αυτή (πχ το en-passant στο σκάκι ή το ροκέ (castling rights)).

Η επιλογή του τύπου δομής αποτελεί κρίσιμο στοιχείο μιας σκακιστικής μηχανής καθώς κάθε κίνηση του παιχνιδιού θα γίνεται μέσω αυτής. Για να έχουμε γρήγορους υπολογισμούς η προσπέλαση της πρέπει να είναι όσο το δυνατόν πιο γρήγορη. Στην περίπτωση παίχτη εναντίον παίχτη η διαφορά δεν θα είναι αντιληπτή. Αν όμως αναφερόμαστε σε ένα AI που απαιτεί εκατομμύρια υπολογισμούς κόμβων, ο χρόνος μπορεί να αυξηθεί σε απαγορευτικό βαθμό. Κατά τον υπολογισμό της βέλτιστης λύσης ενδεχομένως να υπάρχει και χρονικός περιορισμός άρα ο χρόνος αναζήτησης εμμέσως συνδέεται και με την ποιότητα των «βέλτιστων» κινήσεων που θα επιλεχθούν από την σκακιστική μηχανή κατά την διάρκεια μιας παρτίδας. Επίσης η εσωτερική δομή σχετίζεται και με την αναπαράσταση χαρακτηριστικών (feature representation) που μπορεί να παρουσιαστεί αργότερα σε ένα νευρωνικό δίκτυο [4].

Οι αναπαραστάσεις χωρίζονται σε 3 βασικές κατηγορίες. Με επίκεντρο τα πιόνια (Piece centric), με επίκεντρο την θέση (square centric) και υβριδικές.

- Οι piece centric κρατάνε σύνολα πιονιών που είναι ακόμα ζωντανά και την θέση στην οποία βρίσκονται.
- Οι square centric υλοποιούν την ανάποδη λογική και αποτελούν ένα σύνολο από τετράγωνα (πλακάκια ταμπλό) που περιγράφουν το περιεχόμενο τους. Στην περίπτωση που σε ένα τετράγωνο δεν υπάρχει πιόνι, είναι απλά άδειο.
- Οι υβριδικές υλοποιήσεις αποτελούν συνδυασμός των υπολοίπων. Πρακτικά έχουμε πλεονάζουσες δομές που αξιοποιούν τα πλεονεκτήματα της εκάστοτε αναπαράστασης για διαφορετικές χρήσεις εντός της σκακιστικής μηχανής. [5]

2.5.1 Λίστες Πιονιών

Οι «λίστες πιονιών» (Piece lists) αποτελούν μια piece centric υλοποίηση. Χρησιμοποιούνται πίνακες ή λίστες για όλα τα 32 πιόνια. (16 για κάθε πλευρά). Πιόνια άλλων ομάδων μπορούν να διαφοροποιηθούν με την καθιέρωση ενός εύρους τιμών εντός του πίνακα ή λίστας. Διαφορετικά μπορούμε να έχουμε ξεχωριστή δομή ανά ομάδα. Συνδεδεμένες λίστες μπορούν να χρησιμοποιηθούν για την εύκολη εισαγωγή ή εξαγωγή πιονιών.

Η σκακιστική μηχανή Microchess αξιοποιεί την αναπαράσταση αυτή μέσω ενός πίνακα 32 bytes (2 byte ανά πιόνι ανά ομάδα). Οι αριθμοί ανήκουν στο εύρος [0,63] για ζωντανά πιόνια και εκτός αυτού για πιόνια που έχουν αφαιρεθεί. (Υπενθύμιση ότι το σκάκι έχει 64 θέσεις)

Πίνακας 4: Αναπαράσταση Microchess

COMPUTER PIECES		YOUR PIECES
0050	King	0060
0051	Queen	0061
0052	King Rook	0062
0053	Queen Rook	0063
0054	King Bishop	0064
0055	Queen Bishop	0065
0056	King Knight	0066
0057	Queen Knight	0067
0058	K R Pawn	0068
0059	Q R Pawn	0069
005A	K N Pawn	006A
005B	Q N Pawn	006B
005C	K B Pawn	006C
005D	Q B Pawn	006D
005E	K Pawn	006E
005F	Q Pawn	006F

Άλλα παραδείγματα αποτελούν η σκακιστική μηχανή Loop Leiden [6] που έχει ξεχωριστές λίστες μέχρι και για αξιωματικούς (bishops) διαφορετικού χρώματος και το πιο πρόσφατο Giraffe [4] που χρησιμοποιεί λίστες πιονιών για καλύτερη αναπαράσταση χαρακτηριστικών.

2.5.2 Σύνολα πιονιών

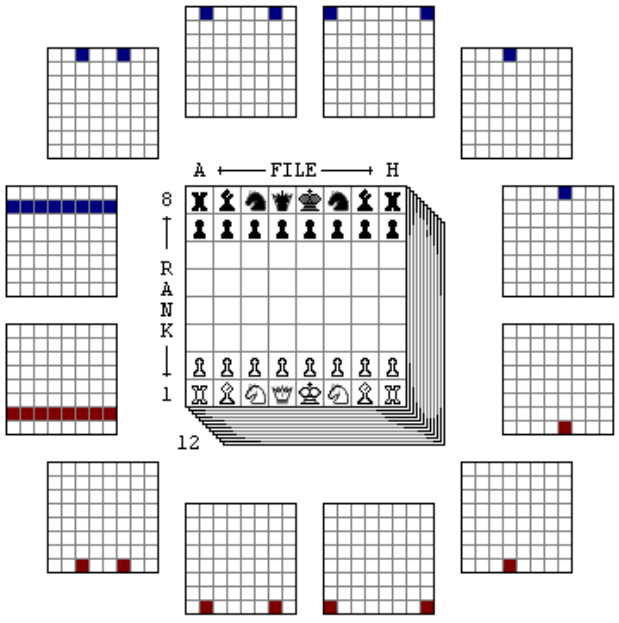
Τα «σύνολα πιονιών» (Piece sets) αποτελούν μια piece centric υλοποίηση που είναι φιλική για 32 bit αρχιτεκτονικές. Έχουμε είτε 2 λέξεις των 16 bits είτε μία των 32 (ανάλογα άμα θέλουμε να διαχωρίσουμε τις λέξεις ανά ομάδα). Τα bits των λέξεων δεν σχετίζονται απευθείας με το τετράγωνο που αντιστοιχούν αλλά αποτελούν δείκτη για μια δομή «λίστας πιονιών». Αυτό είναι και ένα αρνητικό τους καθώς χρειαζόμαστε 2 προσπελάσεις κάθε φορά.

Πιο συγκεκριμένα αν έχουμε K πιόνια και λέξη N bits ($K \leq N$). Αν το λ bit είναι 1 τότε μπορούμε να βρούμε την θέση του πιονιού λ στην δομή «λίστα πιονιών». Βασική προϋπόθεση είναι να υπάρχει μια αντιστοιχία μεταξύ των bits που περιγράφουν τα «σύνολα πιονιών» με την σειρά εμφάνιση τους στην δομή. Στην περίπτωση της μίας λέξης η εύρεση της χρώματος μπορεί να γίνει πάλι με εύρος τιμών (πχ τα 16 πρώτα bit να αντιστοιχούν τα μαύρα πιόνια).

Τα «σύνολα πιονιών» μπορούν να φανούν χρήσιμα σε περιπτώσεις που θέλουμε να περιγράψουμε ποια πιόνια επιτίθενται σε ένα τετράγωνο . Στην ακραία περίπτωση που κάθε πιόνι απειλεί ένα τετράγωνο το «σύνολο πιονιών» θα είναι το $0xFFFFFFFF$.

2.5.3 Bitboards

Τα Bitboards [6] αποτελούν μια ακόμη piece centric υλοποίηση που έχει καθιερωθεί ως ο βασικός τρόπος αναπαράστασης ταμπλό στις σκακιστικές μηχανές. Ένα bitboard αναπαρίσταται από μια 64 bit λέξη. Σε πρώτη ματιά μοιάζουν με αυτή των «συνόλων πιονιών» αλλά με την σημαντική διαφορά ότι περιέχεται και η πληροφορία της θέσης εντός της ίδιας δομής. Σε αφηρημένο επίπεδο τα bitboards αποτελούν διακριτά σύνολα με προκαθορισμένο μέγιστο όριο. Πλεονεκτούν σε υπολογιστές των 64 bit μιας και μπορούν να χωρέσουν εντός ενός καταχωρητή (register). (Κάτι που δεν είναι θέμα για τους περισσότερους σημερινούς υπολογιστές). Μια ολοκληρωμένη αναπαράσταση χρειάζεται ένα bitboard ανά είδος πιονιού ανά ομάδα. Για το σκάκι αυτό θα σήμαινε 12 διαφορετικά bitboards. Η πιο εκτενής ανάλυσης τους θα γίνει σε επόμενο κεφάλαιο.



Εικόνα 5: Bitboards στο σκάκι

2.5.4 Δισδιάστατος πίνακας

Ο δισδιάστατος πίνακας αποτελεί μια square centric υλοποίηση. Συνηθώς προτιμάται (και συνίσταται) για την κατασκευή απλοϊκών σκακιστικών μηχανών οπού η ταχύτητα δεν είναι πρωτίστης σημασίας. Πρακτικά έχουμε ένα δισδιάστατο πίνακα όπου κάθε αριθμός ανταποκρίνεται σε ένα είδος πιονιού. Στην περίπτωση του άδειου τετραγώνου μπορούμε να ορίσουμε μια ειδική κατάσταση με έναν αριθμό εκτός του εύρους τιμών των πιονιών. Αντίστοιχη προσέγγιση μπορούμε να έχουμε και με την χρήση κλάσεων έναντι των αριθμών αν θέλουμε να ακολουθήσουμε το αντικειμενοστραφές paradigm. Το μέγεθος του ταμπλό μπορεί να αλλάξει χωρίς ιδιαίτερη δυσκολία κάτι που την καθιστά εύκολη υλοποίηση για κάθε, τετραγωνικού σχήματος, επιτραπέζιο παιχνίδι.

Παρακάτω βλέπουμε ένα απλό παράδειγμα. Τα μαύρα είδη πιονιών απέχουν από τα άσπρα κατά 6 μονάδες και το 12 αποτελεί το κενό τετράγωνο .

Εικόνα 6: Κανονικό ταμπλό

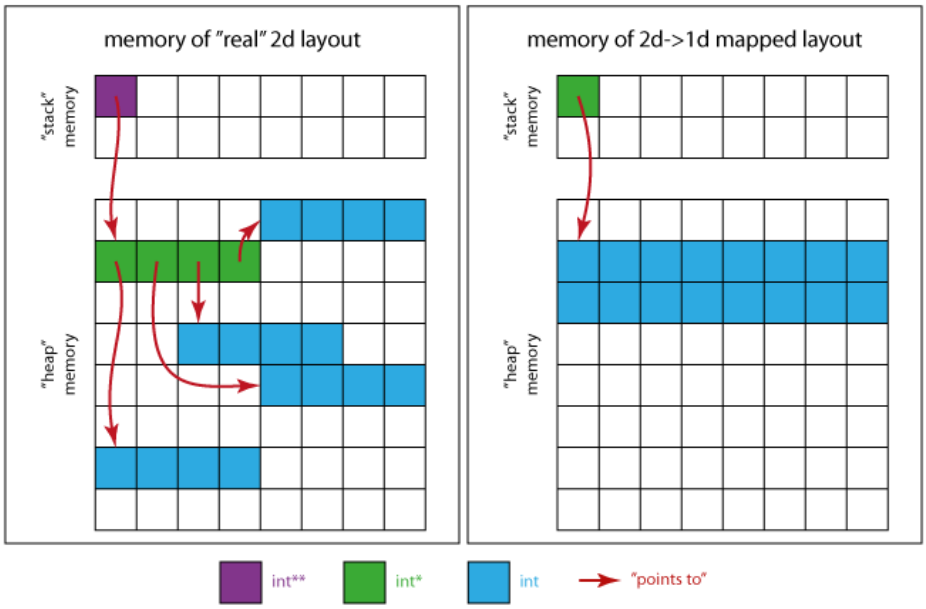
10	9	12	12	6	12	12	10
11	11	11	12	11	12	11	11
12	12	7	11	9	12	8	12
12	12	8	12	12	11	12	12
12	12	12	12	5	12	12	12
12	12	3	5	12	3	12	12
5	5	5	12	2	5	5	5
4	12	2	1	0	12	12	4

Εικόνα 7: Δισδιάστατος πίνακας

Το μεγαλύτερο μειονέκτημα αυτής της τεχνικής είναι η αργή προσπέλαση της. Ένας δισδιάστατος πίνακας αποτελεί ένα σύνολο από pointers. Ο κάθε δείκτης δείχνει σε έναν άλλο αντίστοιχο μονοδιάστατο πίνακα ίσου μεγέθους. Αυτό σημαίνει ότι κάθε 8-αδα του ταμπλό βρίσκεται σε τελείως διαφορετική θέση μνήμης σε σχέση με τις υπόλοιπες κάτι που μπορούμε να δούμε στο αριστερό σχήμα της Εικόνα 8. Για να καταλάβουμε όμως γιατί αυτό αποτελεί πρόβλημα πρέπει να εξηγήσουμε την έννοια της τοπικότητας. Οι επεξεργαστές με κάθε εντολή load φέρνουν στα χαμηλότερα επίπεδα μνήμης (L1, L2, L3 cache) ένα σύνολο συνεχόμενων blocks από bytes. Ο λόγος αυτού είναι για την ορθότερη αξιοποίηση δεδομένων που βρίσκονται τοπολογικά κοντά. (Πχ όταν διασχίζουμε ένα μονοδιάστατο πίνακα θέλουμε να κάνουμε μόνο μία φορά load και όχι για κάθε στοιχείο του για να κερδίσουμε χρόνο.) Οι 2d πίνακες παρουσιάζουν κακή τοπικότητα καθώς χρειαζόμαστε ένα load ανά 8-αδα.

2.5.5 Μονοδιάστατος πίνακας

Ο μονοδιάστατος πίνακας [6] αποτελεί μια square centric υλοποίηση. Βελτιώνει τα μειονέκτημα της δισδιάστατης αναπαράστασης αξιοποιώντας καλυτέρα τις συνεχόμενες θέσης μνήμης. Με αυτόν τον τρόπο έχουμε καλύτερη τοπικότητα και απαιτείται μόνο 1 load για την φόρτωση του ταμπλό. Ο υπολογισμός της νέα θέσης μπορεί να γίνει βάσει της εξίσωσης $f(x, y, W) = y * W + x$, οπου x,y οι συντεταγμένες και W το πλάτος του ταμπλό.



Εικόνα 8: Μνήμη 2d πινάκων

Άμα θέλουμε να κουνήσουμε ένα πιόνι μια θέση αριστερά η δεξιά αρκεί να αθροίσουμε στο εκάστοτε δείκτη θέσης +1 ή -1 μονάδες. Άμα θέλουμε να κινηθούμε στον κατακόρυφο άξονα αρκεί να αθροίσουμε η αφαιρέσουμε W μονάδες. Γενικότερα προκύπτουν οι παρακάτω τύποι για κινήσεις των K τετραγώνων.

$$\mathrm{f(x + k, y, W)} - f(x, y, W) = \mathrm{x + k - x = k}$$

$$f(x, y + k, W) - f(x, y, W) = (y + k) * W - y * W = W * (y + k - y) = k * W$$

Με παρόμοιο τρόπο προκύπτουν και τα Offsets για διαγώνιες κατευθύνσεις.

Το πρωτόκολλο επικοινωνίας για μηχανές παιχνιδιών (game engines), Banksia χρησιμοποιεί 1 δυναμικό πίνακα για την αναπαράσταση του ταμπλό με την χρήση του stl vector της standard βιβλιοθήκης της C++.

Παρ'όλ' αυτά μπορεί να μην είναι αναγκαίο να εφαρμόσουμε εμείς την μετατροπή αυτή αν το μέγεθος του πίνακα είναι στατικά δηλωμένο. Οι σύγχρονοι μεταγλωττιστές μπορούν να κάνουν την βελτιστοποίηση, των 2d πινάκων σε 1d, αυτόματα.

2.5.6 Μονοδιάστατος πίνακας 10x12

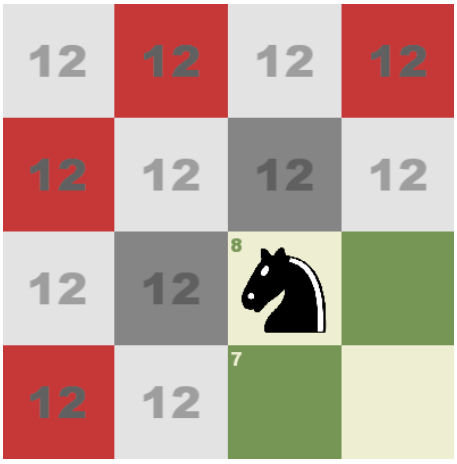
Ο Μονοδιάστατος πίνακας 10x12 [7] αποτελεί μια square centric υλοποίηση. Εντάσσοντας ένα επιπλέον περίγραμμα γύρω από το υπάρχον ταμπλό, μπορούμε ευκολότερα να ελέγχουμε ποτέ βρισκόμαστε εκτός αυτού κατά την παραγωγή κινήσεων. (δηλαδή όταν ένα πιόνι προσπαθεί να κάνει μια κίνηση που θα το οδηγούσε σε μια θέση εκτός πίνακα) Στις προηγούμενες αναπαραστάσεις πινάκων αυτό μπορούσε να επιτευχθεί με 4 ελέγχους.

$$x \geq 0, x \leq width\ y \geq 0, y \leq height.$$

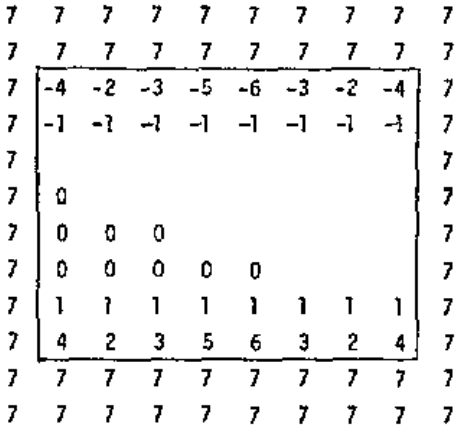
(Να τονιστεί ότι με αυτό τον τρόπο αποφεύγουμε memory violations)

Τώρα αρκεί να κοιτάμε μόνο την τιμή του πίνακα στην νέα θέση. Όπως είχαμε έναν κωδικό για κάθε είδος πιονιού, έτσι και για τετράγωνα εκτός ταμπλό θα υπάρχει μια αντίστοιχη κωδικοποίηση όπου θα μας σηματοδοτεί την λανθασμένη θέση. Βασική προϋπόθεση για την παραπάνω θεώρηση είναι οι κινήσεις να μην μπορούν να οδηγήσουν σε θέση εκτός πλαισίου καθώς τότε θα είχαμε το ίδιο πρόβλημα. Άμεση επίπτωση της τεχνικής αυτής είναι ο ελάχιστα αυξημένος χώρος μνήμης λόγω του περιγράμματος. Στην περίπτωση του σκακιού το πιόνι που μπορεί να κάνει τα μεγαλύτερα άλματα σε μια κίνηση είναι το άλογο. Για να ικανοποιείται η παραπάνω συνθήκη το πλαίσιο πρέπει να είναι τουλάχιστον 2 τετράγωνα σε πάχος. Παρατηρούμε όμως ότι λόγω των συνεχομένων θέσεων μνήμης το δεξιότερο τετράγωνο από το αριστερό τετράγωνο της επομένης λωρίδας απέχουν μόνο μία λέξη αρά μπορούμε να γλιτώσουμε 2 στήλες και να οδηγηθούμε σε ένα ταμπλό 10x12.

Χρήση αυτής της δομής μπορούμε να δούμε στις σκακιστικές μηχανές COKO III, TSCP και Sargon.



Εικόνα 9: Περίγραμμα πάχους 2 τετραγώνων



Εικόνα 10: Ταμπλό 10x12 στο COKO III

Στην πράξη, η επιβάρυνση μιας ακόμα εντολής load δεν προσφέρει το αναμενόμενο όφελος συγκριτικά με τους 4 ελέγχους των άλλων τεχνικών, άρα έχει μόνο ιστορική σημασία σαν μέθοδος στις μέρες μας.

2.5.7 Μέθοδος 0x88

Η μέθοδος 0x88 αποτελεί μια square centric υλοποίηση. Προσπαθεί να λύσει το επιπλέον load της αναπαράστασης 10x12. Βασίζεται στην διαπίστωση ότι η θέση ενός τετραγώνου μπορεί να περιγραφεί από 8 bits. Τα πρώτα τέσσερα αναπαριστούν την στήλη ενώ τα αλλά τέσσερα την σειρά του εκάστοτε τετραγώνου. Έτσι οδηγούμαστε σε ένα ταμπλό 16x16 που υλοποιείται με την χρήση μονοδιάστατου πίνακα όπως και προηγουμένως. Παρατηρούμε ότι το εύρος τιμών της κάθε συντεταγμένης χρειάζεται μόνο 3 από τα 4 bits $(8 = 2^3)$. Βάσει αυτής της παρατήρησης γνωρίζουμε ότι αν το 1ο msb είναι 1 σε οποιαδήποτε 4-αδα, η θέση βρίσκεται εκτός εμβέλειας ταμπλό. Αυτό μπορεί ευκολά να ελεγχθεί εφαρμόζοντας τον δυαδικό τελεστή & (and) μεταξύ της μάσκας 10001000, όπου αποτελεί τον αριθμό 0x88, και του δείκτη της νέας θέσης. Καθώς το πάνω μισό του πίνακα έχει πάντα msb 1, μπορεί να παραλειφθεί. Το τελικό μέγεθος του πίνακα μετά την βελτιστοποίηση είναι 8x16. Το δεξί 8x8 υπο-ταμπλό απαιτείται μόνο για την χρήση του ως padding στις θέσεις μνήμης και δεν χρησιμεύει σαν περιεχόμενο.

Στην ειδική περίπτωση όπου ο δείκτης λάβει τιμές αρνητικού πρόσημου, λόγω της αναπαράσταση των δυαδικών αριθμών σε συμπλήρωμα ως προς 2 το 1ο msb θα είναι πάντα 1. Έτσι αυτομάτως η νέα θέση θα είναι μη έγκυρη λόγω της 1η τετράδας που θα οδηγεί σε μη μηδενικό αποτέλεσμα.

8	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
7	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
6	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
5	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
4	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
3	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
2	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	0 a	1 b	2 c	3 d	4 e	5 f	6 g	7 h	8	9	10	11	12	13	14	15

Εικόνα 11: Αναπαράσταση 0x88

Άρα με 8 παραπάνω bytes έχουμε πολύ γρηγορότερες προσπελάσεις καθώς ο έλεγχος εμβέλειας γίνεται μόνο με 1 branch έναν δυαδικό τελεστή (που είναι μια αρκετά γρήγορη εντολή) και μηδενικά loads εντός του πίνακα.

Για καλύτερη κατανόηση ας μελετήσουμε ένα παράδειγμα. Έστω ότι έχουμε ένα βασιλιά στην θέση 71. Κατά την παραγωγή κινήσεων θα χρησιμοποιήσουμε τις εξισώσεις που αναφέρθηκαν στην ενότητα του μονοδιάστατου πίνακα για τον υπολογισμό των offsets. Οι πιθανές κινήσεις είναι 8 αλλά θα εξετάσουμε τις δύο βασικές περιπτώσεις.

Για τον υπολογισμό της θέσης της κίνησης προς τα κάτω και δεξιά θα πραγματοποιηθεί η πράξη. $newIndex = oldIndex + 1 - width = 71 + 1 - 16 = 56$, κάτι που επαληθεύεται και σχηματικά. Ο αριθμός 56 σε δυαδική αναπαράσταση είναι ο 00111000. Η 1η τετράδα [0011] μας υποδηλώνει ότι βρισκόμαστε στην 4η σειρά (υπενθύμιση ότι μετράμε και το 0 αρά πρέπει να υπολογίσουμε 3 + 1) ενώ η 2η τετράδα [1000] στην στήλη 9 (8+1). Η 9η στήλη είναι εκτός εμβέλειας ταμπλό αρά η κίνηση πρέπει να απορριφθεί. Ο έλεγχος αυτός πραγματοποιείται με τον υπολογισμό της τιμής $newIndex \ \& \ 0x88$

Προκύπτει $[0011][1000] \ \& \ [1000][1000] = [0000][1000]$, το τελικό αποτέλεσμα είναι διαφορετικό του μηδενός αρά η κίνηση όντως απορρίπτεται από το πρόγραμμα.

Τώρα ας εξετάσουμε την κίνηση 1 θέση αριστερά του βασιλιά. Ο νέος δείκτης προκύπτει από τον υπολογισμό $newIndex = oldIndex - 1 = 71 - 1 = 70$, σε δυαδική αναπαράσταση έχουμε 01000110. Εφαρμόζοντας πάλι τον δυαδικό τελεστή & προκύπτει $[0100][0110] \ \& \ [1000][1000] = [0000][0000]$. Το τελικό αποτέλεσμα είναι ίσο με το μηδέν αρά η κίνηση είναι εντός εμβέλειας ταμπλό.

Ένα ακόμα πλεονέκτημα της αναπαράστασης είναι ότι η αφαίρεση δύο έγκυρων δεικτών μας δίνει την σχετική τους θέση. Η ιδιότητα αυτή μπορεί να φανεί χρήσιμη σε αλλά σημεία της σκακιστικής μηχανής.

Μηχανές που χρησιμοποιούν αυτή την υλοποίηση αποτελούν οι Crazy blitz (σε παλιότερη της έκδοση) και Tiny chess.

2.5.8 Συμπυκνωμένα ταμπλό

Τα συμπυκνωμένα ταμπλό [7] (ή αλλιώς CCR (Compact Chess representation)) αποτελούν μια square centric υλοποίηση. Όπως έχουμε ήδη δει σε προηγούμενες ενότητες το σκάκι έχει 6 διαφορετικά πιόνια, 2 για κάθε ομάδα αρά 12 στο σύνολο. Ο αριθμός 12 μπορεί να αναπαρασταθεί από 4 bits. Μια σειρά αποτελείται από 8 τετράγωνα. Μπορούμε να αναπαραστήσουμε μια ολόκληρη λωρίδα με μόλις 32 bits. (όπου είναι και ο λόγος που η τεχνική αυτή υπερτερεί σε 32 bit υπολογιστές έναντι των bitboards). Άρα ολόκληρο το ταμπλό μπορεί να αναπαρασταθεί από 32 bytes. Η υλοποίηση αυτή αξιοποιεί το μέγιστο πιθανό πλήθος bit του ταμπλό για αυτό και ονομάζεται «συμπυκνωμένη».

Είναι κατάλληλη και για 64 bit υπολογιστές καθώς μπορούμε να χρησιμοποιήσουμε αντί τέσσερις 32 bit ακέραιους, δύο των 64.

0000	0000	0000	0000	**1110**	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	**0010**	0000
0000	0000	0000	0000	**0001**	0000	0000	0000
0100	0000	**0011**	0000	**0110**	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

Εικόνα 12: Παράδειγμα αναπαράστασης CCR

Στο παράδειγμα της Εικόνα 12 βλέπουμε τον πίνακα

$CCR = (57344,0,0,0,32,4096,1076912128,0)$ με δεκαεξαδική αναπαράσταση

$CCR(+00)$: $0000E000h$

$CCR(+04)$: $00000000h$

$CCR(+08)$: $00000000h$

$CCR(+12)$: $00000000h$

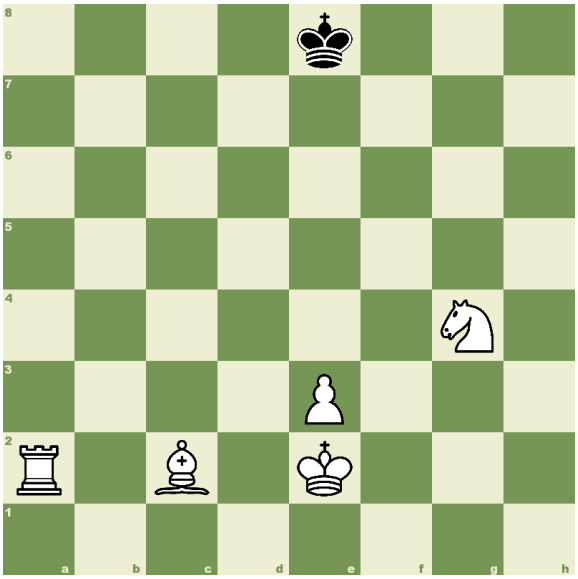
$CCR(+16)$: $00000020h$

$CCR(+20)$: $00001000h$

$CCR(+24)$: $40306000h$

$CCR(+28)$: $00000000h$

οπού σε παραθέσεις αναφέρεται το Offset σε bytes μεταξύ των θέσεων του πίνακα.



Εικόνα 13: Πραγματικό ταμπλό CCR

Αν θέλουμε να παράξουμε της πιθανές κινήσεις του αλόγου αρκεί να πάμε στην θέση $CCR[+16]$ και να ανακτήσουμε την πληροφορία του συγκεκριμένου τετραγώνου. Αυτό μπορεί να επιτευχθεί μέσω μιας μάσκας $0xf = 0b1111$ ολισθημένη κατά 4 θέσεις αριστερά. Με αυτόν τον τρόπο φιλτράρουμε τα υπόλοιπα πιόνια από τον ανακτώμενο ακέραιο. Στο συγκεκριμένο παράδειγμα προκύπτει η πράξη

$00000020\text{h} \,\&\, (0\text{xF} \ll 4) = 00000020\text{h} \,\&\, 000000\text{F0h} = 00000020\text{h}.$

(διαισθητικά παρατηρούμε ότι δεν υπάρχουν άλλα πιόνια στην λωρίδα αρά ο αριθμός παραμένει ίδιος)

Γενικά άμα θέλουμε να κινηθούμε στην πιο πάνω σειρά αρκεί να αφαιρέσουμε από το Offset -4, για να πάμε στο επόμενο προσθέτουμε +4 αντίστοιχα.

Όπως είδαμε ήδη για να πάμε αριστερά ή δεξιά εντός μιας σειράς ολισθαίνουμε την μάσκα σε ποσότητες πολλαπλάσιων του 4.

Για να ελέγξουμε αν το τετράγωνο είναι άδειο αρκεί να δούμε αν το αποτέλεσμα της δυαδικής πράξης & έχει μηδενικό αποτέλεσμα.

Για παράδειγμα αν θέλουμε να ελέγξουμε αν είναι έγκυρη η κίνηση του αλόγου πάνω και δεξιά αρκεί να αφαιρέσουμε από το offset -8 και να ολισθήσουμε την αρχική μάσκα κατά 4 bit δεξιά.

Παράδειγμα κώδικα: $If(CCR[8]\ \&\ (mask \gg 4) == 0)\ legalmove.$

Ο έλεγχος εμβέλειας θέσεων στον κάθετο άξονα μπορεί να πραγματοποιηθεί με τον περιορισμό του offset στο διάστημα [0,28] ενώ στον οριζόντιο άξονα μπορεί να αξιοποιηθεί το γεγονός ότι σε περιπτώσεις underflow / overflow η μάσκα είναι μηδενική.

Σκακιστικές μηχανές που χρησιμοποιούν την αναπαράσταση CCR είναι η Axon και Achilles [7].

PIECE	CODE	Dec.	Hex.
Empty square	0000	0	0
White pawn	0001	1	1
White knight	0010	2	2
White bishop	0011	3	3
White rook	0100	4	4
White queen	0101	5	5
White king	0110	6	6
Black pawn	1001	9	9
Black knight	1010	10	0A
Black bishop	1011	11	0B
Black rook	1100	12	0C
Black queen	1101	13	0D
Black king	1110	14	0E

Εικόνα 14: Κωδικοποίηση πιονιών στην σκακιστική μηχανή Axon

2.5.9 Χρήση δομών σε άλλα παιχνίδια

Όλες οι προαναφερόμενες δομές εξετάστηκαν στα πλαίσια του παιχνιδιού του σκακιού. Αρκετές από αυτές μπορούν πολύ ευκολά να γενικευθούν και για την χρήση τους σε άλλα επιτραπέζια παιχνίδια, ενώ άλλες είναι πιο απαιτητικές στην ορθή προσαρμογή τους χωρίς να είναι ξεκάθαρο το κέρδος απόδοσης.

Πιο συγκεκριμένα:

- Παιχνίδια μικρότερου ή ίσου μέγεθος ταμπλό όπως το connect 4, checkers, Othello, long life [8] μπορούν να υλοποιηθούν εύκολα με την χρήση bitboards. Καθώς τα παιχνίδια αυτά έχουν μόνο 1 είδος πιονιού για κάθε παίχτη, χρειαζόμαστε μόνο 2 bitboards για ολόκληρη την αναπαράσταση (υπενθύμιση ότι στο σκάκι χρειαζόμαστε 12). Αν το μέγεθος του ταμπλό είναι μικρότερο από 8x8 άλλα δεν μπορεί να αναπαρασταθεί από μικρότερου μεγέθους μεταβλητές τότε μπορούμε απλά να μην αξιοποιήσουμε όλα τα bits του 8x8 ταμπλό. Αν το μέγεθος είναι μεγαλύτερο τότε μπορούμε να κάνουμε μια χωρική διάσπαση του ταμπλό σε κομμάτια (chunks) όπου κάθε κομμάτι να αναπαρίσταται από ένα πλήθος από bitboards. Για παράδειγμα αν έχουμε ταμπλό παιχνιδιού 16x16 μπορούμε να έχουμε 4 bitboards, 1 για κάθε υπο ταμπλό 8x8. Με παρόμοιο τρόπο χρησιμοποιούνται στο shogi [9].
- Με «λίστες πιονιών» απλά αρκεί να αλλάξουμε την δομή του πίνακα / λίστας προσαρμόζοντας κατάλληλα την πληροφορία για τα πιόνια του παιχνιδιού που περιγράφουμε.
- Τα «σύνολα πιονιών» μπορούν να αξιοποιηθούν με παρόμοιο τρόπο προσαρμόζοντας το αναμενόμενο πλήθος bits στις λέξεις. Ο μόνος περιορισμός είναι ότι αν έχουμε παραπάνω πιόνια από την μέγιστη λέξη ενδεχομένως να χρειαστούμε και 2^{η} μεταβλητή, κάτι που καταστρατηγεί τα οφέλη της αναπαράστασης.
- Οι υλοποιήσεις μονοδιάστατου και δισδιάστατου πίνακα είναι πλήρως επεκτάσιμες καθώς δεν απαιτούν καμία παραδοχή για το είδος το μέγεθος του ταμπλό.
- Οι μέθοδοι με περίγραμμα όπως η 10x12 μπορούν να αξιοποιηθούν με τον ίδιο τρόπο. Στις περιπτώσεις ποινών που μπορούν να κάνουν μεγαλύτερα άλματα, το πάχος του περιγράμματος πρέπει να ληφθεί υπόψη. Σε ακραίες περιπτώσεις το μέγεθος της δομής μπορεί να υπερβεί τα λογικά μεγέθη και να πρέπει να αναζητηθούν ορθότερες τεχνικές.
- Η δομές CCR και 0x88 μπορούν να αξιοποιηθούν με παρόμοιο τρόπο αν ισχύει η προϋπόθεση ότι τα πιόνια μπορούν να εκφραστούν από μια λέξη με 2^k bits.

Παρατηρούμε ότι οι αναπαραστάσεις «λίστα πιονιών» και του 1d πίνακα είναι η πιο γενικευμένες τεχνικές.

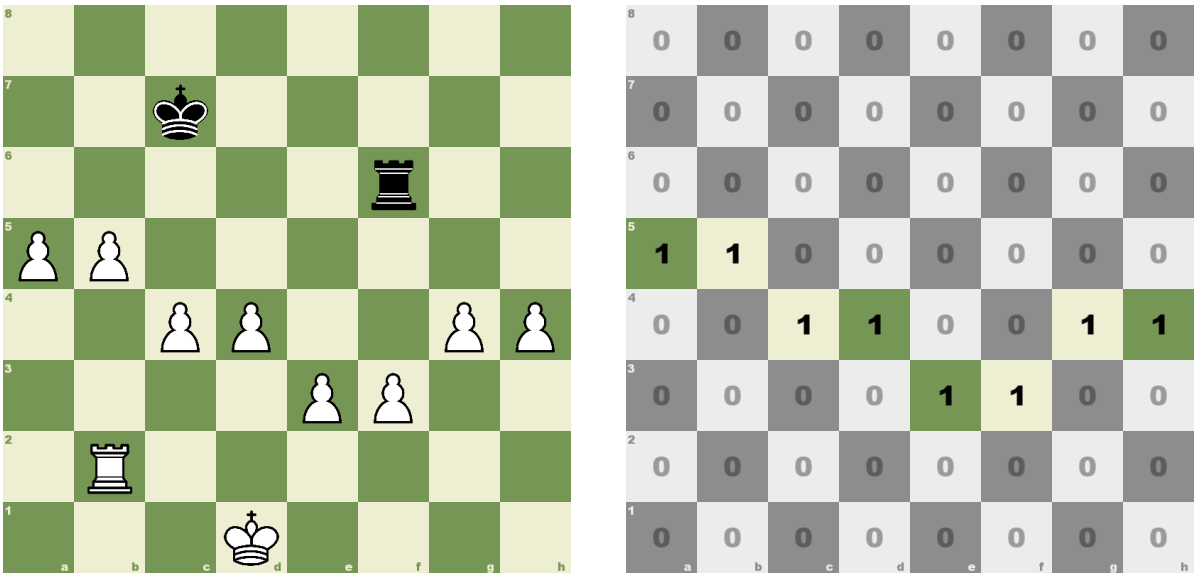
Παρ’ όλ’ αυτά ότι μια δομή μπορεί να προσαρμοστεί για την χρήση της και σε άλλα παιχνίδια δεν την καθιστά απαραίτητα κατάλληλη. Κάθε παιχνίδι πρέπει να εξετάζεται μεμονωμένα βάσει των απαιτήσεών του ώστε να μπορέσει να αξιοποιήσει στο έπακρον την εκάστοτε μέθοδο αναπαράστασης, ενώ άλλες φορές η ταχύτητα δεν είναι η πρώτη προτεραιότητα καθιστώντας την απλοϊκότητα της αναπαράστασης πιο σημαντικό χαρακτηριστικό επιλογής.

2.6 Ανάλυση Bitboards

Όπως είδαμε στην προηγουμένη ενότητα τα bitboards (ή αλλιώς Bitmaps) αποτελούν μια δυαδική αναπαράσταση γνώσης για κάθε τετράγωνο του ταμπλό. Σαν λογική θυμίζουν τα απλοϊκά bit sets. Πρακτικά όταν ένα Bit είναι 1 σηματοδοτείται η ύπαρξη ενός αντικειμένου, ενώ όταν είναι 0 το αντίθετο. Η χρήση τους είναι πολύ ισχυρή σε υπολογιστές των 64 bit καθώς μπορούν να χωρέσουν πλήρως εντός μόνο ενός καταχωρητή. Βασικός στόχος της δομής είναι η μείωση των κύκλων ρολογιού στις προσπελάσεις του ταμπλό μέσω της χρήσης δυαδικών πράξεων. (Υπενθυμίζουμε ότι οι δυαδικές πράξεις σε ατομικούς καταχωρητές μπορούν να υπολογιστούν σε ένα κύκλο λόγω της απλοϊκότητας τους). Στην περίπτωση των 32 bit αρχιτεκτονικών η μέθοδος δεν είναι βέλτιστη καθώς χρειαζόμαστε δύο καταχωρητές ανά σύνολο που οδηγεί στον διπλασιασμό του χρόνου προσπέλασης. Στις μέρες μας η τεχνολογική ανάπτυξη του υλικού έχει οδηγήσει σε σημαντική πτώση των τιμών της μνήμης σε σημείο που μέχρι και οι περισσότεροι οικιακοί υπολογιστές ακολουθούν την 64 Bit αρχιτεκτονική.

Στην πράξη ένα bitboard αναπαρίσταται από έναν 64 bit ακέραιο. Σε γλώσσες όπως την C/C++ αυτό θα σήμαινε ένας τύπος μεταβλητής uint_64 (unsigned long). Παρατηρούμε ότι ένα ταμπλό 8x8 αποτελείται από 64 τετράγωνα. Μπορούμε να αξιοποιήσουμε αυτό το χαρακτηριστικό με το να συσχετίσουμε κάθε bit με ένα από αυτά. Ενεργά bits περιγράφουν την ύπαρξη πιονιού στο εκάστοτε τετράγωνο . Μεμονωμένα το σύνολο είναι αρκετά περιοριστικό καθώς δεν μπορούμε να γνωρίζουμε άμεσα αλλά χαρακτηριστικά του. Το πρόβλημα αυτό λύνεται με το να έχουμε ένα bitboard ανά είδος πιονιού. Έτσι μπορούμε με ρητό τρόπο να προσπελάσουμε μόνο το Bitboard για την κατηγορία πιονιών που επεξεργαζόμαστε. Με παρόμοιο τρόπο διαφοροποιούμε και τις ομάδες του παιχνιδιού. Στο σκάκι αυτό επιτυγχάνεται με 12 bitboards, 6 για τα άσπρα πιόνια και 6 για τα μαύρα

(Υπενθυμίζουμε ότι στο σκάκι έχουμε 6 είδη πιονιών και δύο ομάδες)



Εικόνα 15: Bitboard άσπρων πιονιών

Στην Εικόνα 15 βλέπουμε ένα παράδειγμα. Παρατηρούμε ότι, όπως περιγράφηκε και πριν, περιέχεται πληροφορία μόνο για τα άσπρα πιόνια (pawns). Επίσης κάθε θέση του ταμπλό έχει ενεργό το κατάλληλο bit. Η αντίστοιχη μεταβλητή που θα περιέχει το παρόν ταμπλό θα είναι της μορφής

$$\mathrm{Uint}_{(64)}\ white\ pawns$$

$0\mathrm{b}0000000000000010000100000000000011110011000011000000000000010000010000$

Όπου το 1ο lsb bit είναι το τετράγωνο a1 και το 64ο msb bit το τετράγωνο h8.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Εικόνα 16: Αρίθμηση bits

2.6.1 Δείκτης δυαδικού ψηφιού

Ο έλεγχος ενός συγκεκριμένου τετραγώνου μπορεί να πραγματοποιηθεί με παρόμοιο τρόπο με αυτόν του μονοδιάστατου πίνακα. Μπορούμε να σκεφτούμε την ακολουθία από bits ως μια μονοδιάστατή αναπαράσταση 2 καταστάσεων. Η αρίθμηση του κάθε τετραγώνου εντός της ακολουθίας υποδηλώνει την σειρά εμφάνισης του αντίστοιχου bit και αποκαλείται index / Bitindex.

Για να βρούμε το Bitindex μιας θέσης πραγματοποιούμε τον υπολογισμό

$$Bitindex = rank * 8 + file$$

Η αρίθμηση των σειρών και στηλών ξεκινάει από το 0 (τιμές στο διάστημα $[0,7]$).

Για να φιλτράρουμε το κατάλληλο bit ολισθαίνουμε την μάσκα $0x1$ κατά τον αριθμό εμφάνισης του τετραγώνου μηδενίζοντας όλα τα υπόλοιπα Bits με το δυαδικό &.

$$filteredBitboard = bitboardX\ \&\ (0x1L \ll Bitindex)$$

Για παράδειγμα στο bitboard που εξετάσαμε προηγουμένως, άμα θέλουμε να ελέγξουμε αν υπάρχει πιόνι στην θέση B2 υπολογίζουμε πρώτα το $index = Rank2 * 8 + File = 1 * 8 + 1 = 9$. Ύστερα η μάσκα 0x1L ολισθαίνεται κατά 9 θέσεις και προκύπτει η ενδιάμεση μεταβλητή $mask = 0x1000000000$. Μετά τον υπολογισμό της δυαδικής πράξης με το $\mathrm{Uint}_{(64)}\ white\ pawns$ συγκρίνουμε το αποτέλεσμα με το μηδενικό bitboard,

δηλαδή ένα bitboard με όλα τα bits 0. Στο συγκεκριμένο παράδειγμα προκύπτει ότι δεν υπάρχει άσπρο πιόνι στην θέση B2 κάτι που επαληθεύεται και οπτικά από το σχήμα. Παρατηρούμε ότι ο πύργος δεν λαμβάνεται υπόψη στους υπολογισμούς καθώς αποτελεί πιόνι διαφορετική κατηγορίας.

2.6.2 Συντεταγμένες θέσης από δείκτη

Μερικές φορές είναι απαραίτητο να υπολογίσουμε την ανάποδη διαδικασία, δηλαδή δοθέντος ένα Bitindex να βρούμε το διάνυσμα θέσης στο ταμπλό. Αυτό επιτυγχάνεται με τις παρακάτω εκφράσεις.

$$FileIndex \;=\; bitindex \bmod 8 \;=\; bitindex \,\&\, 7$$

$$RankIndex \;=\; bitindex \;div\; 8 \;=\; bitindex \;\gg\; 3$$

Η χρήση του & και >> αποτελούν βελτιστοποιήσεις των πράξεων mod και div. Οι σημερινοί μεταγλωττιστές εφαρμόζουν τέτοιου είδους μετατροπές αυτόματα άρα συνιστάται η πιο απλοϊκή γραφή σε κώδικα υψηλού επιπέδου.

2.6.3 Πίνακας από bitboards

Τα 16 bitboards πρέπει να αποθηκευτούν σε μια επιπλέον δομή. Η πιο απλή υλοποίηση αποτελείται από έναν πίνακα. Συσχετίζοντας κάθε θέση με μια κατηγορία Bitboard, έχουμε προσπελάσεις σε σταθερό χρόνο. Η διαφοροποίηση ομάδων μπορεί να γίνει με την καθιέρωση μιας σειράς εμφάνισης.

Πιο συγκεκριμένα μπορούμε να χρησιμοποιήσουμε έναν δισδιάστατο πίνακα 2x6 όπου οι σειρές {0,1} δηλώνουν την ιδιότητα της ομάδας ενώ οι στήλες {0,…,6} τις διάφορες κατηγορίες bitboard.

Πχ: $WhitePawns\_Bitboard = Bitboards[WHITE\_INDEX][PAWNS\_INDEX]$

Όπως και στις δισδιάστατες αναπαραστάσεις ταμπλό μπορούμε να μετατρέψουμε το δισδιάστατο πίνακα σε μονοδιάστατο. Η διαφοροποίηση των ομάδων μπορεί να γίνει με την καθιέρωση συνεχόμενων θέσεων μνήμης. Αν για παράδειγμα τα 6 πρώτα bitboards αναφέρονται στην άσπρη ομάδα, για την προσπέλαση των μαύρων bitboard θα χρειαστούμε ένα Offset +6.

Πχ: $Pawns\_Bitboard = Bitboards[PAWNS\_INDEX + SIDE\_OFFSET]$

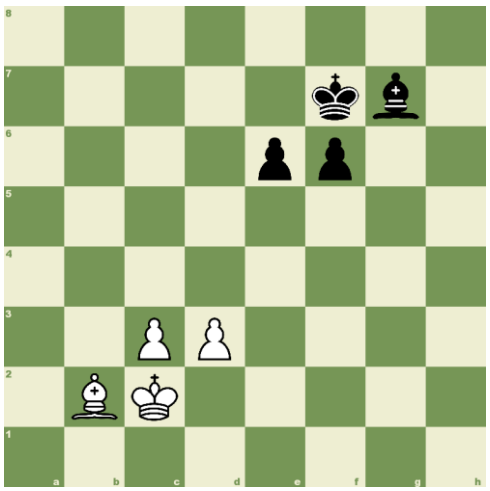
2.6.4 Πυκνά ταμπλό

Μπορούμε να κερδίσουμε λίγο παραπάνω χώρο στην δομή του πίνακα άμα αντί για 16 Bitboards αποθηκεύσουμε μόνο 8 βάσει της ακόλουθης λογικής. Τα 6 πρώτα bitboards όπως και πριν θα αναπαριστούν κάθε είδος πιονιού. Αυτή την φορά όμως δεν θα υπάρχει διαφοροποίηση όσο αφορά την ομάδα στην οποία ανήκουν. Τα νέα bitboards θα αποτελούν ένωση των δύο αντίστοιχων προηγούμενων. [10]

$$Pawns\_Bitboard = WhitePawns\_Bitboard \cup BlackPawns\_Bitboard$$

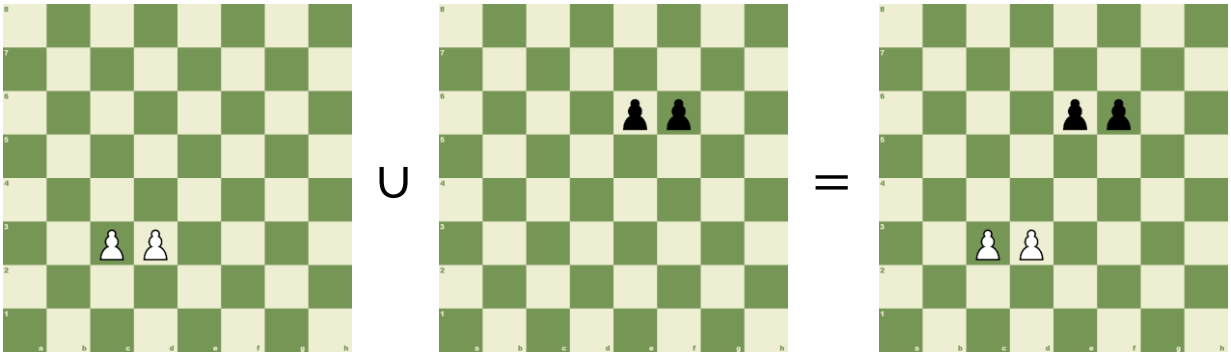
Τα Bitboards {7,8} θα περιέχουν πληροφορία για όλα τα ενεργά πιόνια κάθε ομάδας. Χρησιμοποιώντας τα ως μάσκες μπορούμε να φιλτράρουμε το χρώμα που μας αφορά στην εκάστοτε κατηγορία.

$$WhitePawns\_Bitboard \;=\; Pawns\_Bitboard \,\&\, WhitePawns$$



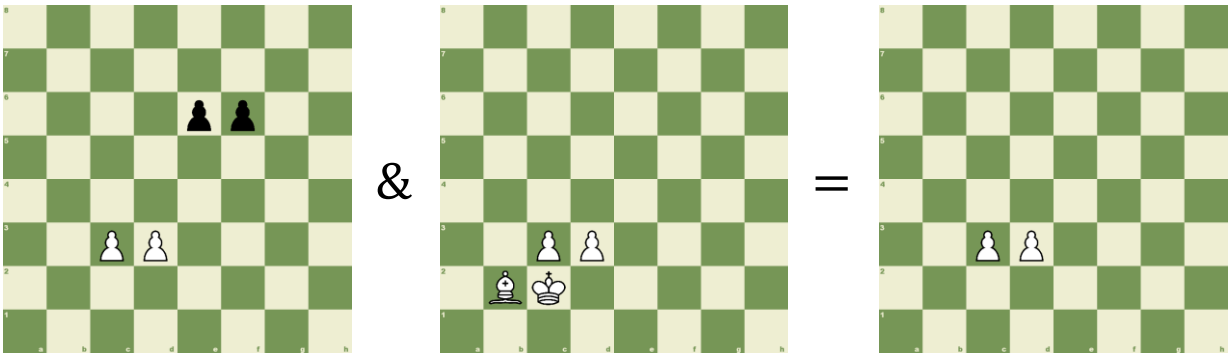
Εικόνα 17: Πυκνά ταμπλό

Παρακάτω παρουσιάζεται ένα παράδειγμα για το ταμπλό της Εικόνα 17. Χρησιμοποιούνται γραφικά πιονιών έναντι δυαδικών αριθμών για καλύτερη αναγνωσιμότητα. Στην πράξη όμως πρέπει να θυμόμαστε ότι οι υπολογισμοί συμβαίνουν μεταξύ δυαδικών αριθμών, όπως και περιγράφηκε εξ αρχής.



Εικόνα 18: Ένωση bitboards πιονιών

Έχοντας το bitboard με τα πιόνια και όλους τους άσπρους πεσσούς μπορούμε να εφαρμόσουμε το δυαδικό τελεστή & για να φιλτράρουμε τα πιόνια της άσπρης ομάδας



Εικόνα 19: Ανάκτηση άσπρων πιονιών

Παρατηρούμε ότι η ένωση ισοδυναμεί με το δυαδικό | (or) και η τομή με το δυαδικό &

2.6.5 Ολισθήσεις

Για την παραγωγή κινήσεων είναι χρήσιμο να μπορούμε να μετακινούμε με ευκολία τα bits της λέξης προς συγκεκριμένες κατευθύνσεις. Αυτό μπορεί να επιτευχθεί με ολισθήσεις κατά συγκεκριμένες ποσότητες. Όπως και στον μονοδιάστατο πίνακα τα αντίστοιχα offsets που απαιτούνται είναι της μορφής

$\pm k, \pm kW, +kW - k, -kW + k, -kW - k.$

Για παράδειγμα η μετακίνηση ενός πιονιού κατά 4 θέσεις προς τα πάνω ισοδυναμεί με την έκφραση $newBitboard = oldBitboard \ll 4 * W$

Οι περισσότερες αρχιτεκτονικές δεν υποστηρίζουν γενικευμένες ολισθήσεις άρα θα πρέπει να γίνει σαφής διαχωρισμός μεταξύ των πράξεων >> (ολίσθηση δεξιά) και << (ολίσθηση αριστερά) καθώς δεν μπορούμε να έχουμε αρνητικές τιμές.

Ένα από τα πλεονεκτήματα των Bitboards είναι το γεγονός ότι οι δυαδικές πράξεις προφέρουν ένα είδος παραλληλίας. Πιο συγκεκριμένα παρατηρούμε ότι η ολίσθηση μετακινεί όλα τα πιόνια / bits προς την ίδια κατεύθυνση. Έτσι μπορούμε να κάνουμε μαζική παραγωγή κινήσεων ή υπολογισμό άλλων χρήσιμων ιδιοτήτων. Επίσης ο έλεγχος εύρους θέσης για μετακινήσεις προς τον κάθετο άξονα δεν είναι αναγκαίος, καθώς τα αντίστοιχα bits θα γίνουν underflow / overflow και θα μηδενιστούν. Για τον οριζόντια άξονα μπορούμε να απορρίψουμε ακραίες περιπτώσεις με χρήση μασκών (οι έλεγχοι αυτοί θα εξεταστούν σε ερχομένη ενότητα πιο αναλυτικά). Άμα θέλουμε να εστιάσουμε μεμονωμένα σε ένα πιόνι θα πρέπει να κάνουμε το αντίστοιχο φιλτράρισμα.

2.6.6 Βασικές πράξεις

Όπως αναφέρθηκε και προηγουμένως τα σύμβολα & και |, στα πλαίσια των δυαδικών συνόλων, περιγράφουν τις πράξεις της ένωσης και της τομής [8]. Το συμπλήρωμα μπορεί να υπολογιστεί με την χρήση του τελεστή ~ (not)

Για να ενεργοποιήσουμε ένα bit σε οποιαδήποτε θέση, μπορούμε να χρησιμοποιήσουμε το δυαδικό τελεστή | μεταξύ του bitboard και του αριθμού 1 ολισθημένου αριστερά κατά Bitindex θέσεις.

$$newBitboard = oldBitboard \mid (1 \ll bitindex)$$

Διαισθητικά είναι σαν να φτιάχνουμε ένα bitboard με ενεργοποιημένη την θέση Bitindex και να παίρνουμε την ένωση του με το ζητούμενο ταμπλό.

Για να αντιστρέψουμε ένα Bit από 0 σε 1 ή από 1 σε 0 αντίστοιχα μπορεί να χρησιμοποιηθεί ο τελεστής ^ (xor) μεταξύ του bitboard και του αριθμού 1 ολισθημένου κατά Bitindex θέσεις. Αυτό προκύπτει από το γεγονός ότι $0 \text{ ^ } 1 = 1$, $1\text{^}1 = 0$, άρα $\text{X ^ } 1 = \bar{X}$

$$newBitboard = oldBitboard \text{ ^ } (1 \ll bitindex)$$

Με παροιμία συλλογιστική μπορούμε να ανταλλάξουμε και δύο τιμές εντός του ιδίου bitboard με την εφαρμογή δυο συνεχόμενων υπολογισμών

$$newBitboard = oldBitboard \text{ ^ } (1 \ll bitindex1)$$

$$newBitboard = newBitboard \text{ ^ } (1 \ll bitindex2)$$

2.6.7 Καταμέτρηση bits

Αρκετές φορές θέλουμε να μετρήσουμε το πλήθος των ενεργών bit εντός ενός bitboard (popCount). (Πχ για τον έλεγχο συνθήκων ισοπαλίας ή μετρήσεων ευρετικής σημασίας). Ο προφανής αλγόριθμος αποτελεί την διάσχιση όλων των 64 θέσεων και την αύξηση ενός μετρητή με την κάθε εμφάνιση ενός ενεργού Bit. Τα Bitboards όμως συχνά αποτελούν αραιές αναπαραστάσεις. Έτσι πληρώνουμε ένα μεγάλο πλήθος αχρείαστων πράξεων.

Μια άμεση βελτιστοποίηση της παραπάνω προσέγγισης είναι η αντιγραφή του ταμπλό σε μια προσωρινή μεταβλητή και η αφαίρεση των ενεργών bit από αυτήν. Όταν η τιμή γίνει μηδενική τότε η επανάληψη μπορεί να σταματήσει. Έτσι γλιτώνουμε ακραίες περιπτώσεις όπως την 000…001 οπού όλα τα bits έχουν καταμετρηθεί από τις πρώτες επαναλήψεις αλλά ο αλγόριθμος συνεχίζει την διάσχιση των υπολοίπων θέσεων. Στην χειριστή περίπτωση όμως έχουμε την ιδιά πολυπλοκότητα αν το 1ο msb bit είναι 1, καθώς η τιμή θα γίνεται μηδενική μόνο μετά την τελευταία επανάληψη.

Ο αλγόριθμος του Kernighan [11] λύνει το παραπάνω πρόβλημα καθώς αποτελεί μια input sensitive υλοποίηση με πολυπλοκότητα O(κ) οπού κ τα ενεργά bits. Η βασική ιδέα είναι η ακόλουθη:

Όταν αφαιρούμε 1 μονάδα από έναν δυαδικό αριθμό το τελευταίο ενεργό lsb bit γίνεται 0 και όλα τα προηγούμενά του 1.

$$XXX1 \ldots 000 \; - \; 1 \; = \; XXX0 \ldots 111$$

Υπολογίζοντας την τομή του αποτελέσματος με την αρχική τιμή της μεταβλητής παρατηρούμε ότι απαλείφουμε το τελευταίο ενεργό lsb bit.

$$XXX1 \; \ldots \; 000 \; \& \; XXX0 \ldots 111 \; = \; XXX0 \; \ldots \; 000$$

Για παράδειγμα $6 \; = \; 0110, \; 6 - 1 \; = \; 5 \; = \; 0101, 6 \; \& \; 5 \; = \; 0110 \; \& \; 0101 \; = \; 0100$

Κοινώς το πλήθος των bits εντός της λέξης μειώνεται κατά 1 μονάδα με κάθε εφαρμογή της έκφρασης $bitboard \; = \; bitboard \; \& \; (bitboard - 1)$. Όταν το ταμπλό λάβει μηδενική τιμή όλα τα Bits έχουν απαλειφθεί και ο αλγόριθμος μπορεί να σταματήσει. Άμα δεν θέλουμε να αλλοιώσουμε την αρχική αναπαράσταση μπορούμε όπως και προηγουμένως να χρησιμοποιήσουμε μια προσωρινή μεταβλητή. Το πλήθος των bits ισοδυναμεί με τις απαιτούμενες επαναλήψεις / αφαιρέσεις.

Μια τελείως διαφορετική προσέγγιση χρησιμοποιεί πίνακες αναζήτησης με προ-υπολογισμένες τιμές για κάθε λέξη. Προφανώς δεν γίνεται να παράξουμε όλους του 2^{64} συνδυασμούς αρά θα βασιστούμε σε μικρότερα μεγέθη. Πιο συγκεκριμένα μπορούμε να υπολογίσουμε το πλήθος των bit κάθε συνδυασμού από 8 bits και να τους αποθηκεύσουμε με την αντίστοιχη σειρά εμφάνισης σε ένα πίνακα 2^8 θέσεων. Έπειτα μπορούμε να σπάσουμε το Bitboard σε 8 λέξεις και να υπολογίσουμε μεμονωμένα το κάθε πλήθος. Είναι σημαντικό η κάθε 8-άδα να μεταφέρεται στις πρώτες θέσεις με χρήση ολισθήσεων για την ορθή αναζήτηση στον πίνακα. Η διάσπαση γίνεται με χρήση της μάσκας 0xff.

$$bitcount[] = 0,1,1,2,1, \ldots ,64$$

$$bitboardCount = bitcount[bitboard \; \& \; 0xff] + \cdots + bitcount[bitboard \gg 56 \; \& \; 0xff]$$

Καθώς απαιτεί 8 Loads θα πρέπει να ελεγχθεί η επίδοση της συγκριτικά με την χρήση επαναλήψεων.

2.6.8 Δείκτης θέσης lsb bit

Στην περίπτωση που θέλουμε να εστιάσουμε σε κάθε πιόνι μεμονωμένα θα πρέπει με κάποιο τρόπο να μπορούμε να μεταβαίνουμε στον αντίστοιχο δείκτη χωρίς να απαιτείται η ολική εξερεύνηση του ταμπλό. Αυτό μπορεί να επιτευχθεί με την εύρεση του 1ου lsb bit στο ζητούμενο bitboard. Έπειτα μηδενίζοντας την συγκεκριμένη θέση και επαναλαμβάνοντας την διαδικασία, μέχρι να μηδενιστεί το ταμπλό, θα έχουμε διατρέξει κάθε τιμή.

$$001101\ (lsb{:}\ 0)\ \rightarrow\ 001100\ (lsb{:}\ 2)\ \rightarrow 001000\ (lsb{:}\ 3)\ \rightarrow 000000\ (none)$$

Η παραπάνω διαδικασία μπορεί να αξιοποιήσει την τεχνική του Bitscan forward. Να τονιστεί ότι υπάρχει και η ανάποδη υλοποίηση Bitscan reverse, όπου στόχος της είναι η εύρεση της θέσης του msb bit. Για τους δικούς μας σκοπούς οι δύο τεχνικές είναι ισοδύναμες.

Ορισμένες αρχιτεκτονικές παρέχουν μέχρι και ειδικά assembly Instructions για τον υπολογισμό αυτό. Για παράδειγμα στον GCC σε x86-64 CPUs μπορούμε να αξιοποιήσουμε την συνάρτηση $\_builtin\_ffsll$. (Για την ορθή λειτουργία του αλγορίθμου υποθέτουμε ότι τα bitboard είναι μη μηδενικά, διαφορετικά το αποτέλεσμα είναι undefined (μη ορισμένο))

Η προφανής λύση του προβλήματος, σε αλγοριθμικό επίπεδο, είναι η σειριακή αναζήτηση. Στην χείριστη περίπτωση όμως απαιτεί έλεγχο κάθε θέσης καθιστώντας τη αρκετά αργή.

Σε αρκετές υλοποιήσεις απομονώνουμε το lsb bit εντός του bitboard πριν την εφαρμογή του βασικού αλγορίθμου. Όταν αλλάζουμε πρόσημο σε έναν αριθμό, η δυαδική αναπαράσταση του σε συμπλήρωμα ως προς 2 έχει όλα τα bits ανεστραμμένα μέχρι το πρώτο lsb. Βάσει αυτής της διαπίστωσης διακρίνουμε ότι παίρνοντας την ένωση του αρχικού ταμπλό με την αρνητική του αναπαράσταση προκύπτει το επιθυμητό φιλτράρισμα. Για παράδειγμα $0110 = 6, 1010 = -6, 0110\ \&\ 1010 = 0010$.

$$isolatedLsb\ =\ bitboard\ \&\ -bitboard$$

Σε 64 bit αρχιτεκτονικές μπορούμε να μετατρέψουμε το απομονωμένο ταμπλό σε αριθμό κινητής υποδιαστολής (double) και να ανακτήσουμε τον εκθέτη της mantissa. Αφαιρώντας το αντίστοιχο bias προκύπτει το ζητούμενο Lsb Bitindex. Η λύση αυτή όμως δεν είναι Portable (φορητή).

Με την χρήση του popCount, μπορούμε να αφαιρέσουμε 1 μονάδα από το απομονωμένο ταμπλό και να μετρήσουμε το πλήθος των ενεργών bits. Για παράδειγμα $0100 - 1 = 0011$, όπου $pop\ count\ =\ 2\ =\ lsb\_bitindex$.

Η πιο δημοφιλής τεχνική είναι ο πολλαπλασιασμός του De Bruijn [12]. Μια 64bit ακολουθία De Bruijn αποτελεί ένα σύνολο 64 αλληλεπικαλυπτόμενων μοναδικών ακολουθιών των 6-bit. Ο πολλαπλασιασμός του μεμονωμένου ταμπλό με την ακολουθία αυτή παράγει νέες υπο-ακολουθίες με τα πρώτα 6 msb bits να είναι επίσης μοναδικά. Οι 2^6 ξεχωριστές τιμές μπορούν να χρησιμεύσουν ως δείκτες σε ένα προκαθορισμένο πίνακα αναζήτησης 64 θέσεων για την εύρεση των ζητουμένων bitboard lsb δεικτών. Η ανάκτηση των πρώτων 6 bit προϋποθέτει και μια επιπρόσθετη ολίσθηση προς τα δεξιά των 58 θέσεων. Έτσι αντί να έχουμε έναν επαναληπτικό αλγόριθμο χρησιμοποιούμε δυαδικές πράξεις και ένα Load.

$$lsbIndex\ =\ lookupTable[isolatedLsb\ *\ debruijn\ >>\ 57]$$

Υπάρχουν πολλές άλλες τεχνικές που δεν αναλυθήκαν όπως το Folding trick, Walter Faxon magic Bitscan, Bitscan με χρήση modulo καθώς και οι αντίστοιχες υλοποιήσεις για το reverse Bitscan.

2.6.9 Πλεονεκτήματα και μειονεκτήματα

Στην παρούσα ενότητα θα συνοψίσουμε μερικά θετικά και αρνητικά της αναπαράστασης των bitboards

Θετικά:

- Αξιοποιούν δυαδικές πράξεις όπως το OR, AND, XOR που μπορούν να εκτελεστούν σε μονάχα ένα κύκλο ρολογιού.
- Οι σημερινοί επεξεργαστές χρησιμοποιούν διοχέτευση για την γρηγορότερη και παράλληλη εκτέλεση εντολών. Καθώς στα Bitboards αποφεύγουμε συνθήκες ελέγχου προκαλείται μειωμένο πλήθος από stalls.
- Ένα bitboard μπορεί να χωρέσει σε ένα 64 bit καταχωρητή
- Παρέχουν καλυτέρους χρόνους εκτέλεσης καθώς απαλείφεται η ανάγκη για επαναλήψεις με την χρήση «παράλληλων» ολισθήσεων και χρήση μασκών.
- Παρέχουν δυνατότητες για τέλειο κατακερματισμό και δημιουργία πινάκων αναζήτησης.
- Αξιοποιούν εντολές χαμηλού επιπέδου για γρηγορότερους υπολογισμούς

Αρνητικά:

- Πολύ πιο εκτενής κώδικας σε επίπεδο πηγαίου κώδικα αλλά και αντικειμενικών αρχείων.
- Πιο σύνθετη συγγραφή κώδικα. Αυξημένη πιθανότητα λάθους
- Απαιτούνται 64 bit αρχιτεκτονικές για βέλτιστη απόδοση
- Απαιτείται πολύ παραπάνω μνήμη καθώς έχουμε αραιές αναπαραστάσεις
- Αύξηση των instruction cache misses λόγω του εκτενέστερου κώδικα (αφορά κυρίως κινητές συσκευές)
- Αύξηση των data cache misses λόγω του αυξανομένου μεγέθους δομής. (Το πρόβλημα αυτό παρουσιάζεται στα επίπεδα L1,L2)
- Απαιτούνται εντολές χαμηλού επιπέδου από την συγκεκριμένη αρχιτεκτονική

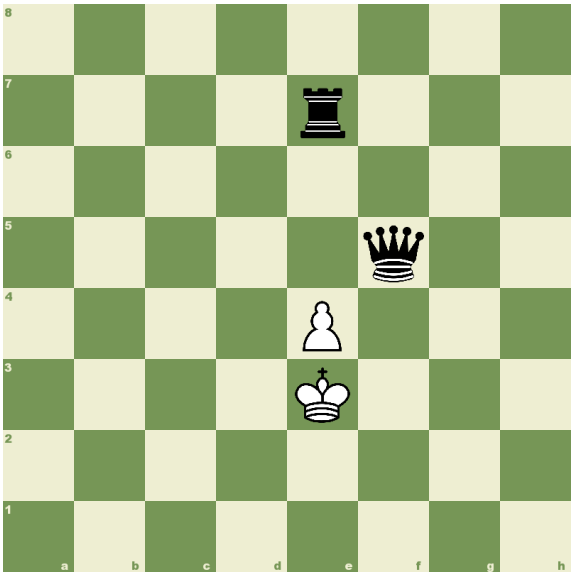
Τα περισσότερα μειονεκτήματα δεν αποτελούν σημαντικό πρόβλημα για τους σημερινούς υπολογιστές. Περιέχουν αρκετά μεγάλες instruction / data cache. Μπορούν να εκτελέσουν όλες τις βασικές πράξεις. Στο σύνολο εντολών συνήθως περιέχονται οι αναγκαίες εντολές χαμηλού επιπέδου για την λύση του προβλήματος Bitscan / popCount. Έχουν αρκετά μεγάλη μνήμη και ακολουθούν την αρχιτεκτονική των 64 bit.

2.7 Παραγωγή κινήσεων

Οι κινήσεις που παράγει μια σκακιστική μηχανή μπορούν να διαχωριστούν σε δύο βασικές κατηγορίες.

- Ψευδο-νόμιμες: ελέγχουμε μεμονωμένα τον τρόπο με τον οποίο μπορεί να κουνηθεί ένα πιόνι βάσει των κανόνων του παιχνιδιού.
- Νόμιμες: ελέγχουμε τον τρόπο με τον οποίο μπορεί να κουνηθεί ένα πιόνι βάσει των κανόνων του παιχνιδιού συνυπολογίζοντας και την απειλή του βασιλιά.

Παρατηρούμε ότι όλες οι νόμιμες κινήσεις είναι ψευδο νόμιμες ενώ το ανάποδο δεν ισχύει.



Εικόνα 20: Παράδειγμα ψευδο-νόμιμων κινήσεων

Για παράδειγμα στην παραπάνω εικόνα η κίνηση του πιονιού από το e4 προς την θέση f5 είναι ψεύδο νόμιμη αλλά όχι νόμιμη γιατί ο βασιλιάς απειλείται από τον πύργο στην θέση e7.

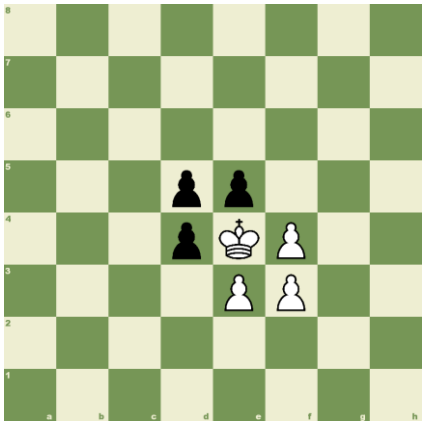
Ο διαχωρισμός αυτός απλοποιεί τους υπολογισμούς καθώς η διαδικασία σπάει σε δύο στάδια. Πρώτα υπολογίζουμε τις ψεύδο κινήσεις και μετά ελέγχουμε, όσες από αυτές μας ενδιαφέρουν, αν είναι έγκυρες. Με αυτόν τον τρόπο μπορούμε γρήγορα να απορρίψουμε κινήσεις μικρής αξίας χωρίς να σπαταλήσουμε επιπλέον χρόνο για την εύρεση εγκυρότητας.

Στις παρακάτω υποενότητες θα μελετήσουμε την παραγωγή κινήσεων με χρήση bitboards. Τα πιόνια του σκακιού διαφοροποιούνται βάσει του τρόπου κίνησης σε

- Leaper πιόνια: πεσσοί που μεταπηδούν ένα σταθερό πλήθος θέσεων ως προς κάποια κατεύθυνση. Σε αυτή την κατηγορία εντάσσουμε τον βασιλιά τα πιόνια και το άλογο.
- Sliding πιόνια: πεσσοί που μπορούν να κουνηθούν ένα συνεχόμενο πλήθος θέσεων μέχρι να συναντήσουν ένα άλλο πιόνι. Σε αυτή την κατηγορία εντάσσουμε την βασίλισσα τον αξιωματικό και τον πύργο.

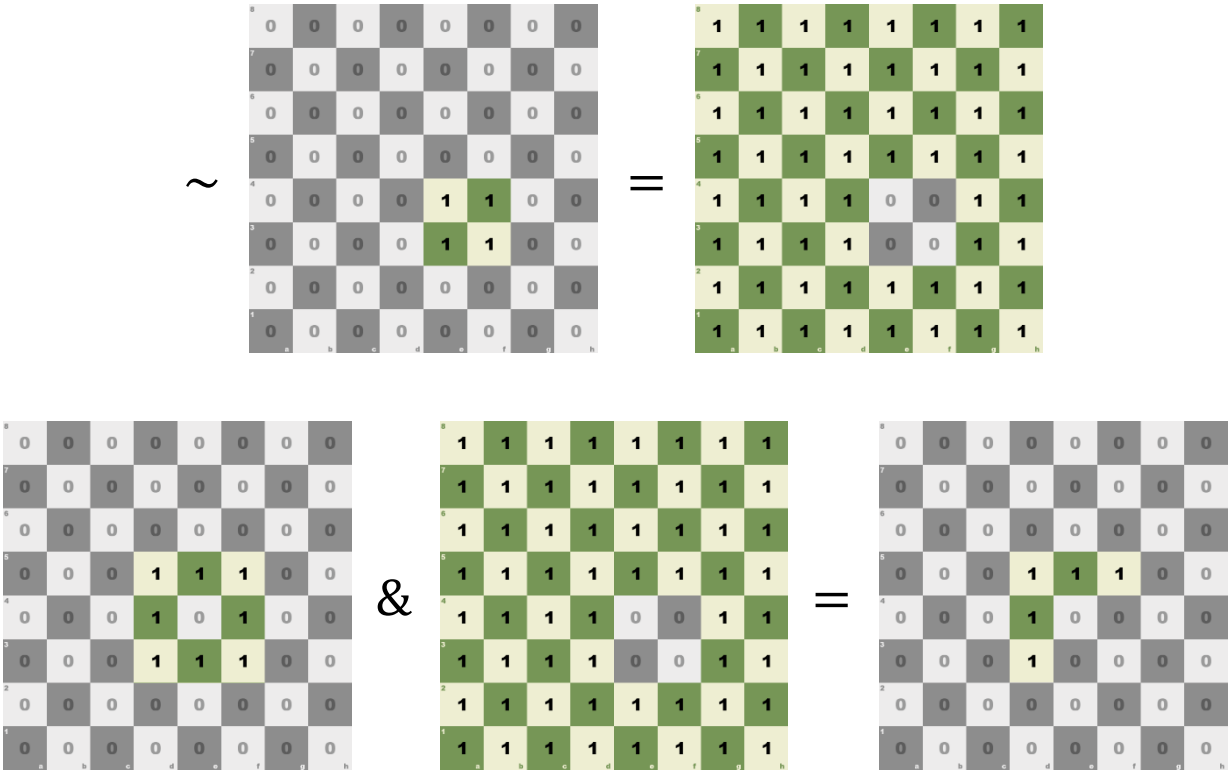
2.7.1 Ταμπλό κατειλημμένων θέσεων

Κατά την παραγωγή κινήσεων είναι χρήσιμο να μπορούμε να ελέγχουμε αν η νέα θέση είναι κατειλημμένη από ένα άλλο πιόνι. Το πρόβλημα αυτό αφορά κυρίως πιόνια ιδίου χρώματος. Στην περίπτωση εμφάνισης αντίπαλου αρκεί απλά να αφαιρέσουμε την πληροφορία από το αντίστοιχο Bitboard. Τα occupancy bitboards αποτελούν ένωση όλων των πιονιών κοινής ομάδας. Δεν μας ενδιαφέρει το είδος πιονιού αλλά μόνο η ύπαρξη του στην εκάστοτε θέση. Στην ενότητα 2.6.4 αναφερθήκαμε στην χρήση τους στην μέθοδο πυκνών ταμπλό (denser boards).



Εικόνα 21: Παράδειγμα occupancy

Κατά αυτόν τον τρόπο άμα θέλουμε να απορρίψουμε κινήσεις που οδηγούν σε κατειλημμένες θέσεις όμοιου χρώματος αρκεί να πάρουμε την τομή του συνόλου με το συμπλήρωμα του occupancy bitboard για το χρώμα αυτό.

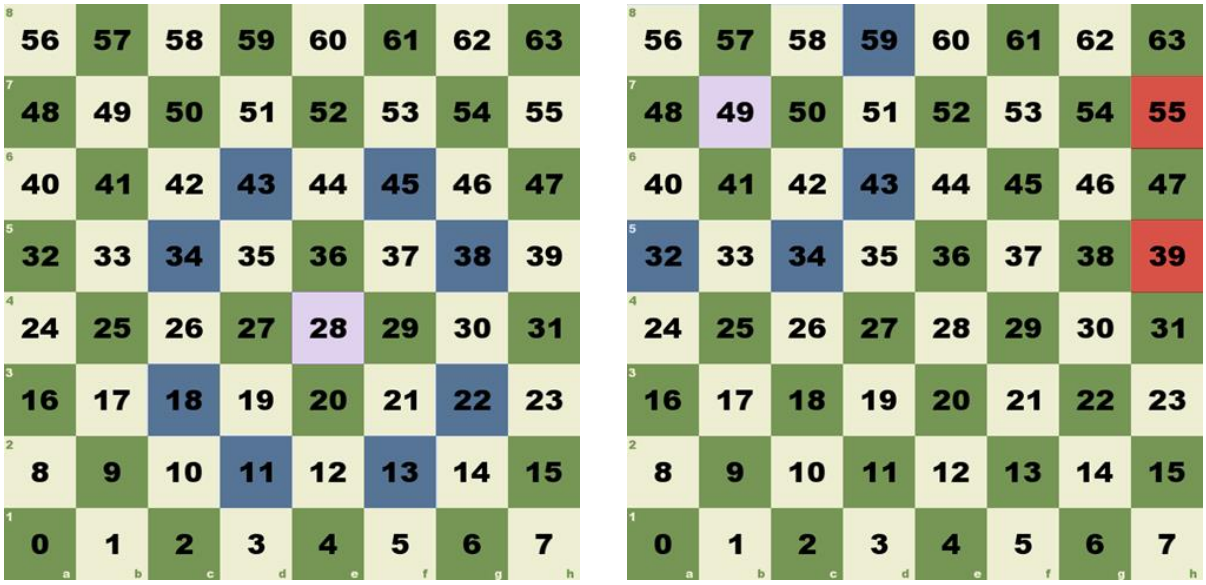


Εικόνα 22: Εφαρμογή occupancy

$$possibleMovesBitboard = movesBitboard \,\&\, \sim occypancyBitboard$$

2.7.2 Leaper πιόνια

Στην ενότητα 2.6.5 είδαμε ότι με χρήση ολισθήσεων μπορούμε να μετακινηθούμε ως προς μια κατεύθυνση βάσει ενός offset. Τα άλογα αποτελούν την πιο απλή περίπτωση για αυτό και θα τα εξετάσουμε ως βασικό παράδειγμα. Τα 8 πιθανά άλματα φαίνονται στην Εικόνα 23.



Εικόνα 23: Κινήσεις αλόγου εκτός εμβέλειας

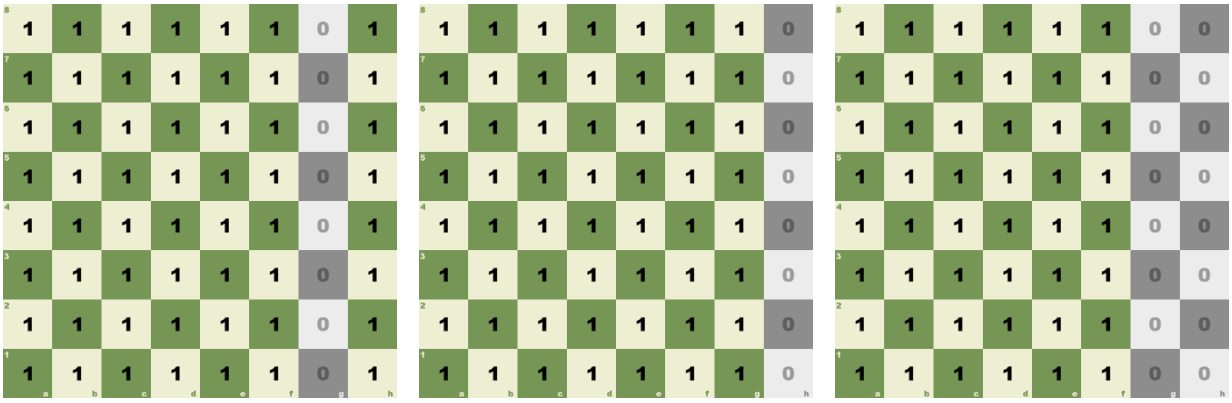
Άμα θέλουμε να μετακινηθούμε πάνω και δεξιά αρκεί να εφαρμόσουμε μια αριστερή ολίσθηση κατά $2 * 8 + 1$ θέσεις. (Το αποτέλεσμα επαληθεύεται με τον υπολογισμό $45 - 28 = 17$)

Με αντίστοιχο τρόπο προκύπτουν όλα τα offsets για κάθε κατεύθυνση. Πρέπει να προσέξουμε στην περίπτωση εμφάνισης αρνητικών τιμών να εφαρμόσουμε ολίσθηση προς τα δεξιά με θετικό πρόσημο. (Κάτι που συμβαίνει για όλες τις κινήσεις μικρότερης σειράς από την αρχική θέση)

Όταν μια κίνηση υπερβαίνει το ύψος του ταμπλό (είτε πηγαίνοντας κάτω από την σειρά 1, είτε πάνω από την σειρά 8) ο έλεγχος εμβέλειας δεν είναι αναγκαίος καθώς τα Bits απλά θα μηδενιστούν ως αποτέλεσμα του underflow / overflow.

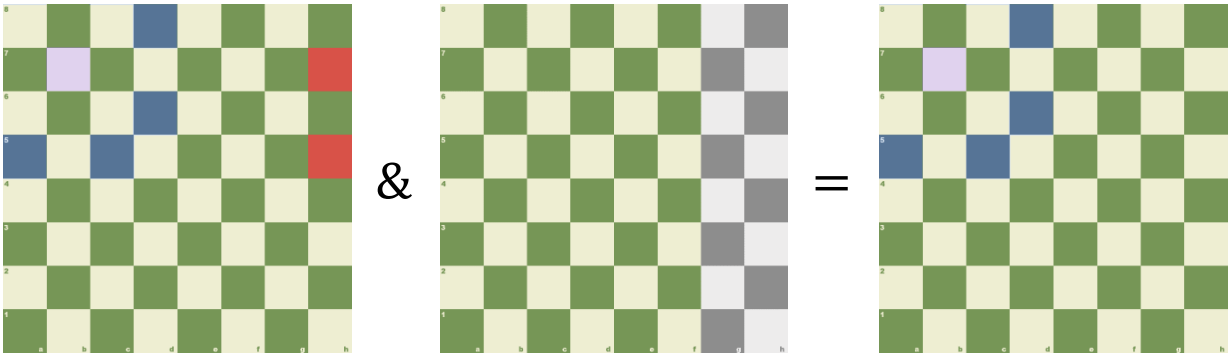
Σε ακριανές θέσεις του ταμπλό βλέπουμε λανθασμένες κινήσεις ως προς τον οριζόντιο άξονα. Το πρόβλημα αυτό αποτελεί άμεση συνέπεια του self-wrapping (αυτό τυλίγματος) των σειρών ως συνεχόμενες θέσεις μνήμης. Για παράδειγμα στην Εικόνα 23 η κίνηση αριστερά και πάνω οδηγεί στην θέση 55 και η κίνηση αριστερά και κάτω στην θέση 39. Αντίστοιχα αποτελέσματα έχουμε και στις θέσεις των στηλών 1, 7 και 8. Παρατηρούμε ότι τα λάθη περιορίζονται πάντα στις δύο ακριανές στήλες της ανάποδης πλευράς. Συμπεραίνουμε ότι μπορούμε να φιλτράρουμε τις μη έγκυρες μετατοπίσεις με χρήση μασκών ανά κατεύθυνση. Οι μάσκες αυτές θα έχουν όλα τα bits 1 εκτός από τις θέσεις των στηλών τις οποίες περιγράφουν.

Στις παρακάτω εικόνες βλέπουμε τις μάσκες notG, notH,notGH, αντίστοιχα υπάρχουν και οι notA, notB, notAB για όλες τις άλλες εναλλαγές κινήσεων.



Εικόνα 24: Μάσκες notG, notH, notHG

Είναι σημαντικό να χρησιμοποιούμε τις ορθές μάσκες για κάθε κατεύθυνση, αν για παράδειγμα παράγουμε κινήσεις προς τα αριστερά ξέρουμε ότι δεν μπορούμε να βρεθούμε σε δεξιά στήλες. Σε διαφορετική περίπτωση το φιλτράρισμα των λανθασμένων κινήσεων δεν θα δουλέψει και θα αφαιρέσουμε μέχρι και ορισμένες έγκυρες κινήσεις.



Εικόνα 25: Φιλτράρισμα λανθασμένων κινήσεων

Το φιλτράρισμα κινήσεων που οδηγούν σε κατειλημμένα τετράγωνα ιδίου χρώματος γίνεται με την χρήση occupancy bitboards με τον τρόπο που αναφέρθηκε στην αντίστοιχη ενότητα.

Ένα μεγάλο πλεονέκτημα των bitboards στην παραγωγή κινήσεων είναι η αντικατάσταση του ελέγχου εμβέλειας θέσης με μια πολύ απλοϊκή δυαδική πράξη. Είναι επιθυμητό να μην χρησιμοποιούμε branches, όπου είναι δυνατό, για την αποφυγή χαμένων κύκλων από πιθανά stalls σε επεξεργαστές με υποστήριξη διοχέτευσης. Παρ'ολ'αυτα οι σημερινές αρχιτεκτονικές έχουν πολύ υψηλά επίπεδα ευστοχίας στα Branch predictions.

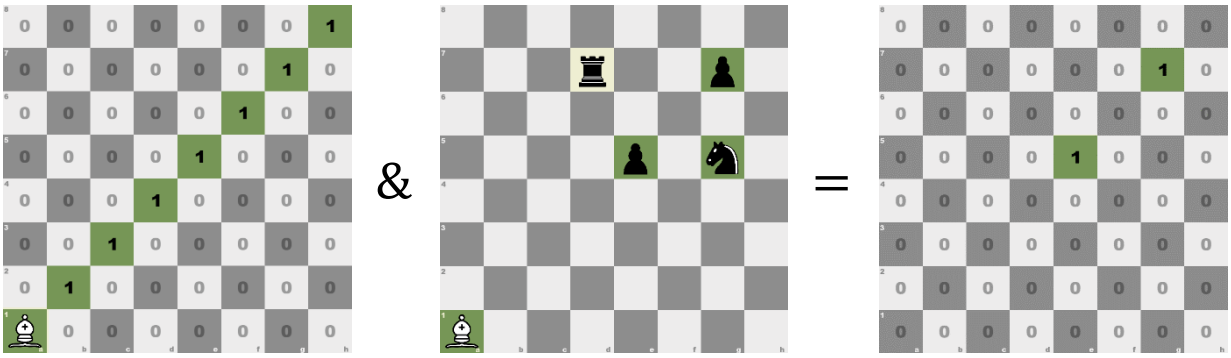
Με αντίστοιχο τρόπο μπορούμε να παράξουμε τις κινήσεις και για τα υπόλοιπα leaper πιόνια με μικρές παραλλαγές στην χρήση των occupancy bitboards. Πιο αναλυτική περιγραφή για κάθε πιόνι θα δοθεί στην ενότητα υλοποίησης.

2.7.3 Slider πιόνια

Μέσω μοναδιαίων αλμάτων, αναζητούμε την ύπαρξη ή μη εμποδίων. Με την χρήση του συμπληρώματος του occupancy bitboard εμποδίζουμε την διάσχιση bits πέρα από ενεργά πιόνια, σε επόμενες επαναλήψεις. Όταν το συνολικό ταμπλό μηδενιστεί τότε ξεκινάει η διαδικασία από την αρχή για την επόμενη κατεύθυνση. Όπως και στα leaper πιόνια αξιοποιούμε κατάλληλες μάσκες για τους περιορισμούς εμβέλειας θέσης στον οριζόντιο άξονα. Η διαδικασία όμως είναι αρκετά χρονοβόρα καθώς απαιτούνται πολλοί κύκλοι για κάθε επανάληψη.

Μπορούμε να περιορίσουμε το πρόβλημα δημιουργώντας ένα πίνακα με όλες τις πιθανές ακτίνες (rays) για κάθε θέση του ταμπλό. Παίρνοντας την τομή της ακτίνας με το occupancy board υπολογίζουμε τα «πιόνια εμπόδια» (blocker pieces) για την κατεύθυνση που μας ενδιαφέρει. Υστέρα αρκεί να βρούμε ποιο από αυτά είναι το 1ο σε ακολουθία και να απορρίψουμε όλα τα προηγούμενα του. Αυτό μπορεί να επιτευχθεί με την χρήση ενός Bitscan forward για θετικές ακτίνες και Bitscan reverse για τις αρνητικές. Η νέα ακτίνα στην θέση lsb Bitindex αποτελείται από τα bits που πρέπει να κάνουμε reset. Η διαδικασία επαναλαμβάνεται για κάθε άλλη κατεύθυνση και η ένωση των τελικών αποτελεσμάτων αποτελεί το σύνολο επιτρεπτών κινήσεων.

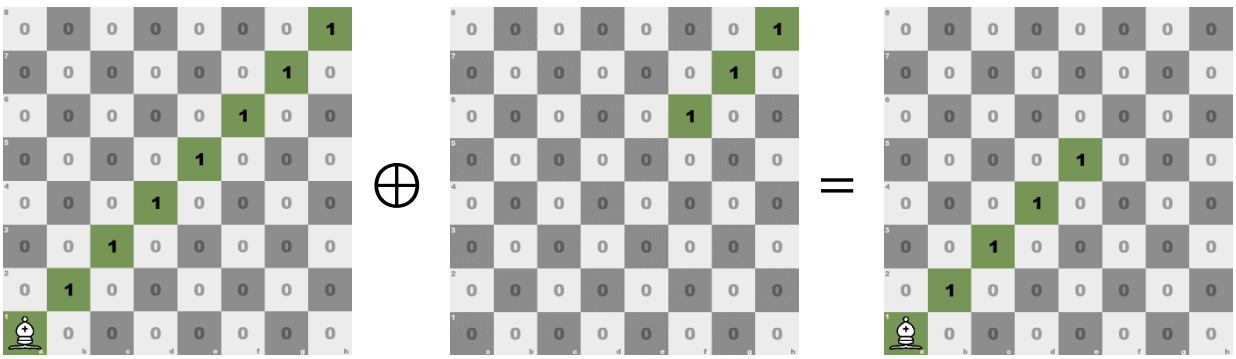
Παρακάτω παρουσιάζεται ένα παράδειγμα για μια από τις κατευθύνσεις του αξιωματικού.



Εικόνα 26: Εύρεση bitboard εμποδίων

$$blockers = rays[direction][A1] \,\&\, occupancyBoard$$

$$attacks = rays[direction][A1] \,\hat{}\, rays[direction][\, Bitscan\_forward(blockers)\,]$$



Εικόνα 27: Εύρεση κινήσεων αξιωματικού

Στην επόμενη ενότητα θα δούμε πώς μπορεί να λυθεί το πρόβλημα με χρήση πινάκων αναζήτησης χωρίς την ανάγκη μεμονωμένων υπολογισμών για κάθε κατεύθυνση.

2.7.4 Απειλή βασιλιά

Για να διακρίνουμε τις ψευδο-νόμιμες κινήσεις σε νόμιμες είναι αναγκαίο να μπορούμε να μετρήσουμε το πλήθος των ενεργών checks σε κάθε κατάσταση. Αυτό μπορεί να επιτευχθεί με τον έλεγχο όλων των πιθανών επιθέσεων των leaper και sliding pieces από την θέση του βασιλιά. Αν ο βασιλιάς επιτίθεται σε ένα πιόνι τότε θα πρέπει να συμβαίνει και το ανάποδο. (Διαισθητικά θεωρούμε τον βασιλιά ένα παντοδύναμο πιόνι που μπορεί να πράξει κάθε κίνηση). Αντιθέτως ένα πιόνι που είναι εκτός εμβέλειας επιθέσεων σίγουρα δεν μπορεί να προκαλεί check. Αφού σχηματίσουμε ένα bitboard με όλες τις πιθανές επιθέσεις ελέγχουμε την ύπαρξη πιονιών στις διάφορες θέσεις με την χρήση του occupancy bitboard της αντίπαλης ομάδας. Το πλήθος των checks υπολογίζεται με την χρήση της μεθόδου popCount που περιεγράφηκε στην ενότητα 2.6.7.

Η μεθοδολογία αυτή είναι προτιμότερη από τον έλεγχο κάθε αντίπαλου πιονιού καθώς γλιτώνουμε άσκοπες προσπελάσεις από απομακρυσμένα πιόνια. Επίσης δεν περιορίζεται μόνο στις απειλές βασιλιά αλλά μπορεί να γενικευτεί για τον έλεγχο επιθέσεων σε κάθε τετράγωνο του ταμπλό.

2.7.5 Perft

Η συγγραφή κώδικα για την παραγωγή κινήσεων αποτελεί λεπτό σημείο, για αυτό είναι κρίσιμο να μην υπάρχουν λάθη. Για τον έλεγχο ορθότητας προελαύνουμε τον χώρο αναζήτησης μέχρι ένα προκαθορισμένο βάθος και συγκρίνουμε το πλήθος των κόμβων φύλλα με έναν προϋπολογισμένο πίνακα τιμών. Με αντίστοιχο τρόπο μπορούμε να μελετήσουμε τους χρόνους προσπέλασης άλλων μηχανών για βελτίωση των αποδόσεων. Κατά τους υπολογισμούς αγνοούμε ισοπαλίες λόγω επαναλήψεων ή ελλείματος πιονιών. Η μέθοδος αυτή είναι χρήσιμη και για άλλα παιχνίδια όπως το checkers [13].

Πίνακας 5: Παράδειγμα αποτελεσματων Perft

Depth	Nodes	Captures	E.p.	Castles	Promotions	Checks	Discovery Checks	Double Checks	Checkmates
0	1	0	0	0	0	0	0	0	0
1	20	0	0	0	0	0	0	0	0
2	400	0	0	0	0	0	0	0	0
3	8,902	34	0	0	0	12	0	0	0
4	197,281	1576	0	0	0	469	0	0	8

2.7.6 Λίστα κινήσεων

Με τον όρο λίστα κινήσεων αναφερόμαστε είτε στην λίστα νόμιμων κινήσεων για τον εκάστοτε γύρο ή στην λίστα κινήσεων που έχουν παιχτεί κατά την διάρκεια του παιχνιδιού. Η δομή υλοποιείται συνηθώς με έναν δυναμικό πίνακα 1024 θέσεων. Ενώ το πραγματικό μέγιστο όριο είναι πολύ μεγαλύτερο, για τα περισσότερα παιχνίδια επαρκεί. (Στην σπάνια περίπτωση που το υπερβούμε, ο πίνακας πρέπει να γίνει reallocate)

2.7.7 Έλεγχος εγκυρότητας

Για να διακρίνουμε αν μια ψευδο-νόμιμη κίνηση είναι έγκυρη την εφαρμόζουμε στο ταμπλό και μελετάμε αν οι επακόλουθες κινήσεις του αντίπαλου μπορούν να οδηγήσουν σε παράνομες καταστάσεις. Το κύριο χαρακτηριστικό που μας ενδιαφέρει είναι η απειλή βασιλιά, μια ψευδο-νόμιμη κίνηση είναι νόμιμη αν δεν φέρνει το ταμπλό σε μια κατάσταση που ένας από τους δύο βασιλιάδες μπορεί να αιχμαλωτιστεί. Διαφορετικά, στην περίπτωση αυτή θα πρέπει να αναιρεθεί.

Η αναίρεση κίνησης μπορεί να γίνει με 2 βασικούς τρόπους.

- Copy-make: Αντιγραφή του ταμπλό και των σχετικών πληροφοριών του σε μια ενδιάμεση μεταβλητή. Η κίνηση εφαρμόζεται μόνο στο αντίγραφο άρα το βασικό ταμπλό παραμένει αμετάβλητο.
- Make-Unmake: Κρατάμε τις απαραίτητες πληροφορίες για κάθε κίνηση ώστε να μπορεί να υλοποιηθεί η αντίστροφη διαδικασία. Για παράδειγμα όταν γίνεται μια αιχμαλώτιση θα πρέπει να ξέρουμε την θέση και το είδος του διαγραμμένου πιονιού για να μπορεί να επαναφερθεί στο ταμπλό.

Η μέθοδος copy-make είναι η πιο εύκολη στην υλοποίηση αλλά υπολογιστικά ακριβή λόγω της άσκοπης αντιγραφής. Η ενδιάμεση μεταβλητή δεν διαφέρει πολύ συγκριτικά με την προηγουμένη κατάσταση. Η νέα κατάσταση δεν ξαναχρησιμοποιείται και μπορεί να προκαλέσει περισσότερες αστοχίες στην cache.

Αντιθέτως η μέθοδος Make-unmake αξιοποιεί τον ίδιο χώρο μνήμης, δεν απαιτεί ενδιάμεση αντιγραφή αλλά είναι πιο συνθέτη στην υλοποίηση καθώς απαιτούνται επιπρόσθετες πληροφορίες για την ανάκτηση της προηγουμένης κατάστασης.

2.7.8 Κωδικοποίηση κίνησης

Οι κινήσεις περιγράφουν τρείς βασικές πληροφορίες. Από πού προήλθε ένας πεσσός, σε ποια θέση κατέληξε και το είδος της προαγωγής. (Στην περίπτωση που δεν αναφερόμαστε σε πιόνια η προαγωγή μπορεί να έχει κάποια κενή τιμή). Η εύρεση του παγιδευμένου υλικού μπορεί να γίνει πριν το παίξιμο της κίνησης και δεν είναι υποχρεωτικό να αποθηκεύεται εσωτερικά για την υλοποίηση της συνάρτησης Unmake. Για αποτελεσματικότερη αποθήκευση η πληροφορία συνήθως πακετάρεται με χρήση μασκών ή bitfields

2.7.9 Μονοχρωματικά ταμπλό

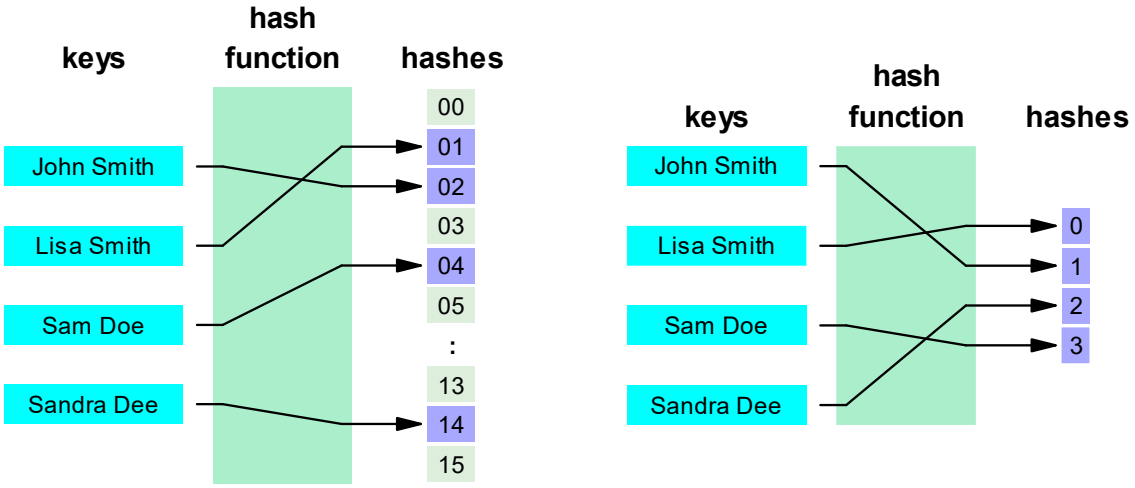
Μπορούμε να θεωρήσουμε το ταμπλό μονοχρωματικό άμα μετά από κάθε κίνηση εφαρμόσουμε μια αναστροφή σε κάθε Bitboard αναπαράστασης. Με αυτόν τον τρόπο τα άσπρα παίζουν πάντα και η παραγωγή κινήσεων απλοποιείται σημαντικά. Για παράδειγμα στην κίνηση πιονιών ελέγχουμε πάντα προς την ίδια κατεύθυνση ανεξαρτήτως χρώματος. Η αναστροφή μπορεί να επιτευχθεί με έξι ολισθήσεις. Η τεχνική αυτή χρησιμοποιείται από σκακιστικές μηχανές όπως η Leela.

2.8 Αναζήτηση κινήσεων βάσει διαμόρφωσης πιονιών

Όπως είδαμε προηγουμένως η παραγωγή κινήσεων για τα slider πιόνια είναι μια πολύ χρονοβόρα διαδικασία που απαιτεί ένα αυξημένο πλήθος επαναλήψεων. Μπορούμε να μειώσουμε την χρονική πολυπλοκότητα σε σταθερό χρόνο, δημιουργώντας πίνακες αναζήτησης με όλες τις πιθανές κινήσεις για κάθε ταμπλό. Με παρόμοιο τρόπο μπορούμε να πράξουμε και για τα leaper πιόνια. Η ουσιώδης διαφορά όμως μεταξύ των δυο κατηγοριών είναι το γεγονός ότι οι κινήσεις των slider πιονιών καθορίζονται από την συστοιχία πιονιών του ταμπλό. Τα leaper πιόνια αντιθέτως έχουν σταθερό πλήθος κινήσεων για κάθε θέση ανεξαρτήτως κατάστασης. Για παράδειγμα οι πιθανές κινήσεις του αλόγου φράζονται από την ποσότητα $64 * 8$ καθώς για κάθε θέση μπορούμε να έχουμε 8 κινήσεις στην βέλτιστη περίπτωση. Χρησιμοποιώντας ένα Bitboard για κάθε τετράγωνο καταλήγουμε στον υπολογισμό μόνο 64 περιπτώσεων. Σε αντιπαράθεση, για τα slider πιόνια θα πρέπει να συνυπολογίσουμε κάθε πιθανή διαμόρφωση ενός καθολικού Occupancy bitboard, παράγοντας ένα άνω φράγμα των $C(64,32) \cong 10^{18}$ περιπτώσεων.

2.8.1 Τέλειος κατακερματισμός

Μια συνάρτηση κατακερματισμού $f(x)$ λέγεται «τέλεια» αν αντιστοιχίζει μονοσήμαντα κάθε αντικείμενο X σε έναν μοναδικό αριθμό K. Ένα hash table που χρησιμοποιεί τέλειο κατακερματισμό έχει μηδενικές συγκρούσεις (collisions) και δίνει πάντα σταθερή πολυπλοκότητα. Αντιθέτως ένα hash table με μη βέλτιστο κατακερματισμό έχει γραμμική πολυπλοκότητα $O(\#X)$ στην χείριστη περίπτωση. Μια τέτοια συνάρτηση είναι δυνατόν να κατασκευαστεί μόνο αν γνωρίζουμε εκ των υστέρων το σύνολο X. Στην δικιά μας περίπτωση οι παραπάνω παραδοχές ισχύουν καθώς το σύνολο των διαμορφώσεων είναι διακριτό και γνωστό πριν την εκτέλεση του προγράμματος. Αν η $f(x)$ δεν δημιουργεί κενά στον χώρο μνήμης τότε θεωρείται και ελάχιστη (minimal). Οι ελάχιστες συναρτήσεις κατακερματισμού είναι επιθυμητές για την βέλτιστη αποθήκευση πληροφορίας και την μείωση των αστοχιών στην cache κατά την προσπέλαση της δομής. Οι παρακάτω τεχνικές αποτελούν μορφές ελάχιστου τέλειου κατακερματισμού.



Εικόνα 28: Ελάχιστος και τέλειος κατακερματισμός

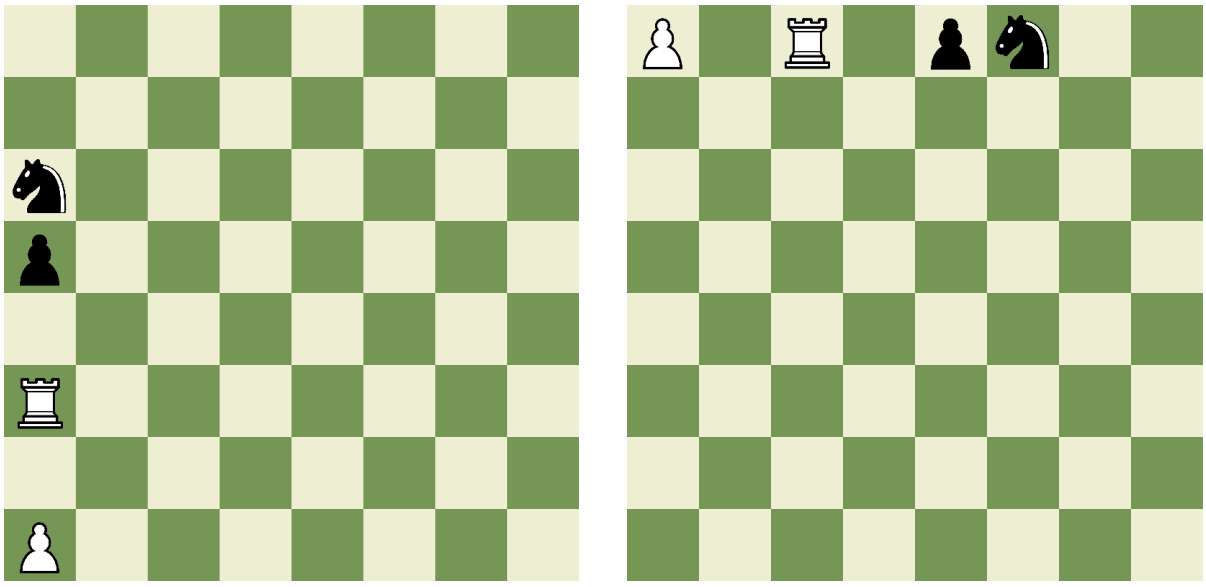
2.8.2 Απλοποίηση χώρου

Βασικό πρόβλημα στην παραγωγή κινήσεων για τα slider πιόνια είναι η αναζήτηση του τελευταίου «πιονιού εμπόδιο» στην ακτίνα κίνησης. Με αυτόν τον τρόπο μπορούμε να αποκλείσουμε κινήσεις που έπονται της θέσης αυτής. Παρατηρούμε όμως ότι τα «πιόνια εμπόδια» που βρίσκονται σε ακριανές θέσεις του ταμπλό δεν επηρεάζουν την διαδικασία παραγωγής. Πιο συγκεκριμένα η ύπαρξη ενός Occupant σε ακριανά τετράγωνα επηρεάζει μόνο την τελευταία θέση του συνόλου και όχι τις κινήσεις πίσω από αυτόν. Στην περίπτωση ιδίου χρώματος αρκεί να φιλτράρουμε το τελικό αποτέλεσμα με την μάσκα των occupants ομάδας. Επίσης σε κάθε κατάσταση μας ενδιαφέρουν μόνο τα bits εντός των ακτίνων κίνησης. Έτσι καταλήγουμε στο ότι μόνο ένα υποσύνολο ενός $6x6$ υπο-ταμπλό είναι χρήσιμο για την παραγωγή των πιθανών διαμορφώσεων.

2.8.3 Περιστρεφόμενα bitboards

Άμα θεωρήσουμε ένα ταμπλό που αποτελείται μόνο από μία σειρά τότε οι πιθανές διαμορφώσεις από occupancies είναι 2^8 (ή 2^6 άμα λάβουμε υπόψη την απλοποίηση χώρου). Η συστοιχία πιονιών μπορεί να χρησιμοποιηθεί ως κλειδί για την αναζήτηση των προϋπολογισμένων κινήσεων για κάθε θέση. Με παρόμοιο τρόπο μπορούμε να συμπεριφερθούμε και στο κανονικό ταμπλό άμα μετατρέψουμε την τομή της ακτίνας κίνησης με το occupancies bitboard σε ένα διάνυσμα διάστασης 8/6. Για μετακινήσεις στον οριζόντιο άξονα η μετατροπή είναι τετριμμένη. Αρκεί απλά να ολισθήσουμε τα bits ακτίνας στις 8/6 πρώτες θέσεις. Το πρόβλημα εμφανίζεται στις κατακόρυφες και διαγώνιες κατευθύνσεις όπου θα πρέπει να μεταφέρουμε την συστοιχία από bits σε συνεχόμενες θέσεις μνήμης για την δημιουργία του κλειδιού. Τα περιστρεφόμενα bitboards (rotated bitboards) λύνουν το πρόβλημα αυτό με το να κρατάνε άλλες 3 αναπαραστάσεις του ταμπλό στις γωνίες {-45,45,90}. Με αυτόν τον τρόπο μπορούμε να συμπεριφερθούμε σε κάθε κατεύθυνση σαν να ήταν οριζόντια.

Η τεχνική των περιστρεφόμενων bitboards εμφανίζεται πιο συχνά σε σκακιστικές μηχανές αλλά έχει χρησιμοποιηθεί και στο παιχνίδι shogi [9] για την παραγωγή κινήσεων των αντίστοιχων slider πιονιών. (Η σημαντική διαφορά του shogi με το σκάκι είναι το μέγεθος του ταμπλό)



Εικόνα 29: Περιστρεφόμενα bitboard 90 μοιρών

Για παράδειγμα στην παραπάνω εικόνα για οριζόντιες επιθέσεις του πύργου στην θέση A2 θα χρησιμοποιήσουμε το 1ο ταμπλό, ενώ για κατακόρυφες το 2ο. Το μεγαλύτερο μειονέκτημα της τεχνικής είναι ότι θα πρέπει να ανανεώνουμε 3 παραπάνω αναπαραστάσεις με κάθε παίξιμο κίνησης. Το πρόβλημα αυτό λύνουν αλγεβρικές υλοποίησης των «rank / file collapses» με χρήση 64 bit πολλαπλασιασμών.

2.8.4 Μαγικά bitboards

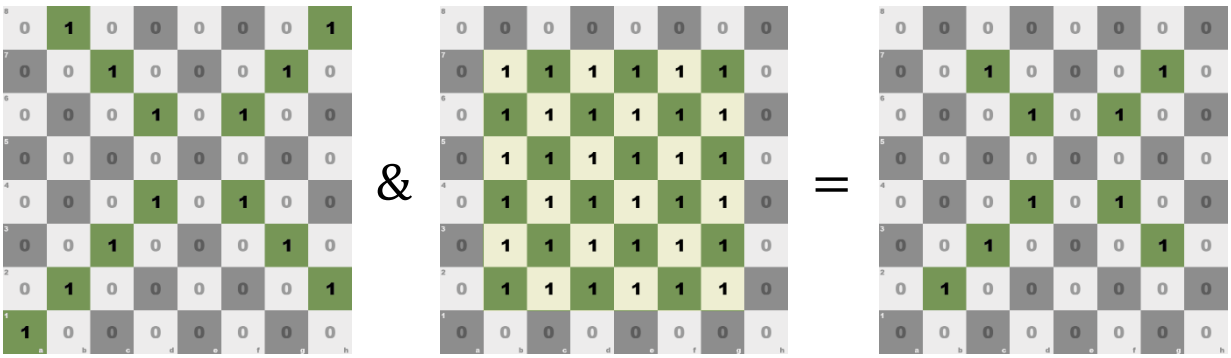
Τα μαγικά bitboards (magic bitboards) [14] θεωρούνται η πιο διαδιδομένη μέθοδος παραγωγής κινήσεων σε σκακιστικές μηχανές όπως την Houdini και το stockfish. Αξιοποιούν γρήγορους πολλαπλασιασμούς σε συνδυασμό με τις σημερινά μεγάλες cache για να υλοποιήσουν έναν αλγόριθμο τέλειου κατακερματισμού. Το ταμπλό occupancy πολλαπλασιάζεται με έναν μαγικό αριθμό για την εύρεση των δεικτών σε ένα προκαθορισμένο πίνακα κινήσεων. Δεν απαιτούνται επιπλέον υπολογισμοί για κάθε ακτίνα ούτε επιπρόσθετες αναπαραστάσεις.

Πιο συγκεκριμένα έχουμε 4 βήματα:

- Ανάκτηση μόνο των χρήσιμων occupancy bits. Στην ουσία αξιοποιούμε την απλοποίηση χώρου μηδενίζοντας τα ακριανά τετράγωνα και τα bits που δεν εντάσσονται στις πιθανές ακτίνες κίνησης. Το βήμα αυτό προϋποθέτει την ύπαρξη πινάκων με όλες τις πιθανές ακτίνες για κάθε θέση για τα πιόνια του πύργου και του αξιωματικού. $raysRook[64], raysBishpo[64]$. Οι ακτίνες της βασίλισσας μπορούν να υπολογιστούν με χρήση και των δύο πινάκων. Το αποτέλεσμα των παραπάνω πράξεων συντελεί στην δημιουργία ενός κλειδιού.
- Πολλαπλασιασμό του κλειδιού με έναν μαγικό αριθμό για την δημιουργία ενός δείκτη.
- Η σχετική πληροφορία του κλειδιού μεταφέρεται σε n συνεχόμενες θέσεις μνήμης εντός της λέξης. Με μια δεξιά ολίσθηση κατά 64-n θέσεις παράγουμε έναν n-bit αριθμό. Όσο μικρότερο το n τόσο πιο συμπυκνωμένη είναι η λύση μας.
- Χρησιμοποιούμε τον τελικό αριθμό για την ανάκτηση των κινήσεων από ένα προκαθορισμένο πίνακα.

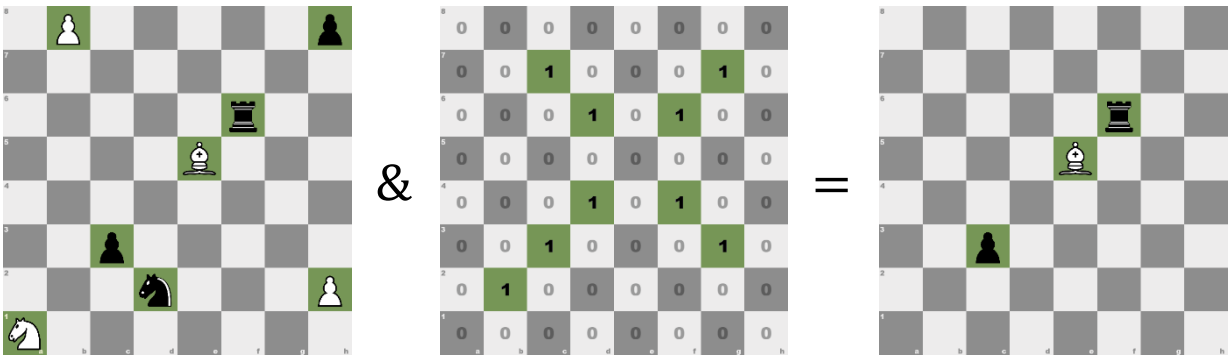
Οι μαγικοί αριθμοί μπορούν να παραχθούν με τεχνικές ωμής βίας (brute force) δοκιμάζοντας επαναλαμβανόμενα τυχαίους αριθμούς. Βασικό χαρακτηριστικό της μεθόδου αποτελεί η απώλεια συγκρούσεων. Αν ένας μαγικός αριθμός συντελεί στην δημιουργία διπλοτύπων συσχετίσεων η διαδικασία πρέπει να ξεκινήσει από την αρχή. Όσο πιο μικρό είναι το πλήθος των απαιτούμενων bits τόσο πιο κοντά οδηγούμαστε στην εύρεση μιας ελάχιστης συνάρτησης κατακερματισμού. Θα πρέπει να τονιστεί ότι ο πληθάριθμος των πιθανών occupancies είναι πολύ μεγαλύτερος από αυτών των πιθανών bitboards κινήσεων. Άρα σε μια βέλτιστη συνάρτηση κατακερματισμού, 2 διαφορετικά occupancies που οδηγούν στο ίδιο ταμπλό κίνησης θα πρέπει να παράγουν τον ίδιο δείκτη για την ελαχιστοποίηση του χώρου μνήμης. Είναι κρίσιμο ο χώρος αυτός να είναι όσο μικρός γίνεται για την μείωση των αστοχιών στην cache κατά την προσπέλαση της δομής. Στις μηχανές σκακιού οι μαγικοί αριθμοί φορτώνονται από τον δίσκο στο άνοιγμα του προγράμματος άρα ο υπολογισμός τους μπορεί να γίνει μεμονωμένα με πιο εξειδικευμένες τεχνικές αναγνώρισης συγκρούσεων. Τα μαύρα μαγικά bitboards, fixed shift fancy, plain και fancy αποτελούν παραλλαγές της βασικής ιδέας.

Παρακάτω ακολουθεί ένα παράδειγμα υπολογισμού της συνάρτησης κατακερματισμού των μαγικών bitboards. Για την καλύτερη κατανόηση των σχημάτων χρησιμοποιούνται γραφικά πιονιών. Η διαφοροποίηση είδους πιονιού και χρώματος δεν προσφέρει επιπλέον πληροφορία καθώς στην πραγματικότητα αναφερόμαστε σε δυαδικά ψηφία.



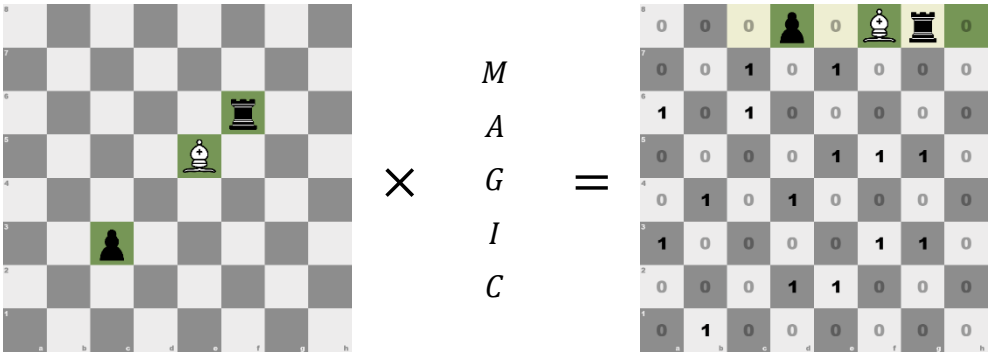
Εικόνα 30: Εσωτερικές ακτίνες

$$simplifiedRays = bishopRays[e5] \,\&\, innerTilesMask$$



Εικόνα 31: Υπολογισμός κλειδιού

$$Key = simplifiedRays \,\&\, occupancy$$



Εικόνα 32: Υπολογισμός δείκτη

$$moves = table[(key * magic) \gg (64-6)]$$

Παρατηρούμε ότι ο τελικός δείκτης περιέχει και σκουπίδια, εμάς μας ενδιαφέρει να υπάρχει αντιστοίχιση στα 6 συνεχόμενα bits, τα υπόλοιπα μηδενίζονται λόγω underflow. Ο τελικός αριθμός ολισθαίνεται στις πρώτες θέσεις και προκύπτει το $0b010110$.

Σε ορισμένες αρχιτεκτονικές η παραπάνω διαδικασία υλοποιείται με την εντολή PEXT

2.8.5 Παραγωγή υποσυνόλων συνόλου

Μελετήσαμε πώς από μια συστοιχία occupancy bits μπορούμε να παράξουμε ένα δείκτη για την αναζήτηση σε ένα πίνακα κινήσεων. Για την αρχικοποίηση του πίνακα απαιτείται ο υπολογισμός όλων των μεταθέσεων κλειδιού. Σε αυτή την ενότητα θα δούμε μια γενικευμένη μεθοδολογία εύρεσης υποσυνόλων συνόλου στον δυαδικό χώρο [15].

Έστω το σύνολο $0b1111$, και τα 3 πρώτα του υποσύνολα $\{0000,0001,0010\}$. Παρατηρουμε ότι για κάθε νέα μετάθεση ισχύει η σχέση:

$$permutation[n] = permutation[n-1] + 1.$$

Στην περίπτωση που θέλουμε να αγνοήσουμε μια υπακολουθία από bits αρκεί να την ενεργοποιήσουμε πριν την αντίστοιχη προσαύξηση (αυτό οφείλεται στην μεταφορά υπολοίπου μεταξύ των δυαδικών ψηφίων).

Αξιοποιώντας την παραπάνω ιδιότητα για σύνολα με ενδιάμεσα κενά παράγεται η νέα έκφραση:

$$Permutations[n] \; = \; [(permutation[n-1] \mid \sim S) \; + \; 1 \;] \; \& \; S$$

Με την πράξη του συμπληρώματος υλοποιείται η επιθυμητή αντιστροφή ψηφίων ενώ η τομή με το σύνολο S φιλτράρει την περιττή πληροφορία.

Για παράδειγμα τα υποσύνολα του συνόλου 1001 υπολογίζονται ως εξής:

$$0000$$

$$0110 \mid 0000 \; + \; 1 \; = \; 0111 \; \& \; 1001 \; = \; 0001$$

$$0110 \mid 0001 \; + \; 1 \; = \; 1000 \; \& \; 1001 \; = \; 1000$$

$$0110 \mid 1000 \; + \; 1 \; = \; 1111 \; \& \; 1001 \; = \; 1001$$

Τέλος, η τελική μορφή μπορεί να απλοποιηθεί περαιτέρω βάσει των ιδιοτήτων $A|B = A + B, \sim A \; = \; -A - 1$, δίνοντας την εξίσωση:

$$Permutations[n] \; = \; (permutation[n-1] \; - \; S) \; \& \; S$$

Στην πράξη κατά τον υπολογισμό του πίνακα $Permutations$ αρκεί να βρούμε, με τις μεθόδους που αναφέρθηκαν στην ενότητα 2.7.3, τα σύνολα ψευδο-νόμιμων κινήσεων για την εκάστοτε διαμόρφωση και να τα αποθηκεύσουμε στην αντίστοιχη θέση πίνακα υπολογίζοντας το δείκτη με χρήση μαγικών αριθμών.

Η διαδικασία είναι αρκετά αργή αλλά είναι σημαντικό να τονιστεί ότι πραγματοποιείται μόνο μία φορά στην αρχή εκτέλεσης του προγράμματος άρα δεν αποτελεί σημαντικό πρόβλημα.

2.9 Αποθήκευση παιχνιδιού

Η αποθήκευση παιχνιδιών γίνεται συνηθώς σε απλοϊκά ascii formats, με τις πιο δημοφιλείς τεχνικές να αποτελούν οι συμβολοσειρές Fen και τα PGN. Αν ο χώρος αποθήκευσης είναι πρωτίστης σημασίας τότε μπορούμε να χρησιμοποιήσουμε τεχνικές συμπίεσης σε δυαδικά αρχεία.

Με αυτόν τον τρόπο μπορούμε είτε να εξάγουμε παιχνίδια από μια σκακιστική μηχανή είτε να τα εισάγουμε για περαιτέρω ανάλυση.

2.9.1 Συμβολοσειρές Fen

Οι συμβολοσειρές fen αποτελούνται από 6 κομμάτια

- Τοποθέτηση πιονιών: κάθε θέση του ταμπλό περιγράφεται από έναν από τους χαρακτήρες rnbqkbnrp, η διαφοροποίηση των χρωμάτων γίνεται με την χρήση κεφαλαίων γραμμάτων. Οι σειρές διαχωρίζονται μεταξύ τους με τον χαρακτήρα ‘/’. Στην περίπτωση κενών τετραγώνων αναγράφεται ο αριθμός αυτών. Για παράδειγμα η αρχική θέση του ταμπλό περιγράφεται από την ακόλουθη συμβολοσειρά rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR
- Ενεργό χρώμα: ο χαρακτήρα w ή b καθορίζει τον ενεργό παίχτη για τον πρώτο γύρο.
- Δικαιώματα ροκέ: τέσσερις χαρακτήρες που περιγράφουν τα δικαιώματα για ροκέ, χρησιμοποιείται η παύλα ‘-‘ όταν ένα από αυτά δεν είναι ενεργά. Για παράδειγμα στην αρχική θέση τα δικαιώματα είναι KQkq (τα κεφαλαία πάλι διαφοροποιούν τις ομάδες)
- Τετράγωνο EnPassant: περιγραφεί σε αλγεβρική σημειογραφία την θέση οπού μπορεί να γίνει enPassant, διαφορετικά περιέχει παύλα ‘-‘.
- Μισή κίνηση: αριθμός κινήσεων από την τελευταία αιχμαλώτιση.
- Ολόκληρη κίνηση: αριθμός πλήρων κινήσεων. Αρχίζει από το 1 και αυξάνεται με κάθε κίνηση των μαύρων.

Ολοκληρωμένο παράδειγμα αρχικού ταμπλό μετά την κίνηση e4:

```
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
```

Οι συμβολοσειρές fen είναι χρήσιμες για την αποθήκευση μόνο της αρχικής κατάστασης μιας παρτίδας καθώς δεν περιέχουν το ιστορικό κινήσεων.

2.9.2 PGN

Αποτελείται από μια σειρά από ζευγάρια που περικλείονται από τετραγωνικές αγγύλες '[ ]'. Διαφοροποιούνται μεταξύ τους με τον χαρακτήρα αλλαγής γραμμής. Τα πρώτα 7 πεδία είναι υποχρεωτικά και περιγράφουν το όνομα της εκδήλωσης, την τοποθεσία, την μέρα, τον αριθμό του γύρου, τα ονόματα των παιχτών καθώς και το αποτέλεσμα του παιχνιδιού. Παρατηρούμε ότι η χρήση του είναι κατάλληλη και για παιχνίδια τουρνουά μιας και υπάρχουν τα αντίστοιχα πεδία πληροφορίας. Ύστερα ακολουθούν ορισμένα προαιρετικά πεδία, εκ των οποίων μερικά περιγράφουν την ώρα του παιχνιδιού, τους κανόνες ρολογιού, πόσες κινήσεις έχουν παιχτεί καθώς και μια συμβολοσειρά fen στην περίπτωση που θέλουμε να κρατήσουμε το ιστορικό κινήσεων μετά από μια ενδιάμεση κατάσταση.

Το κύριο σώμα του αρχείου περιέχει αριθμημένες τις κινήσεις του παιχνιδιού γραμμένες σε αλγεβρικό notation. Υπάρχει η δυνατότητα γραφής σχολίων με την χρήση αγκύλων '{ }'.

Σε αντίθεση με τις συμβολοσειρές fen περιγραφεί εξ ολοκλήρου το παιχνίδι καθώς περιέχεται το ιστορικό κινήσεων και όχι μόνο μια αρχική κατάσταση.

Ακολουθεί ένα παράδειγμα:

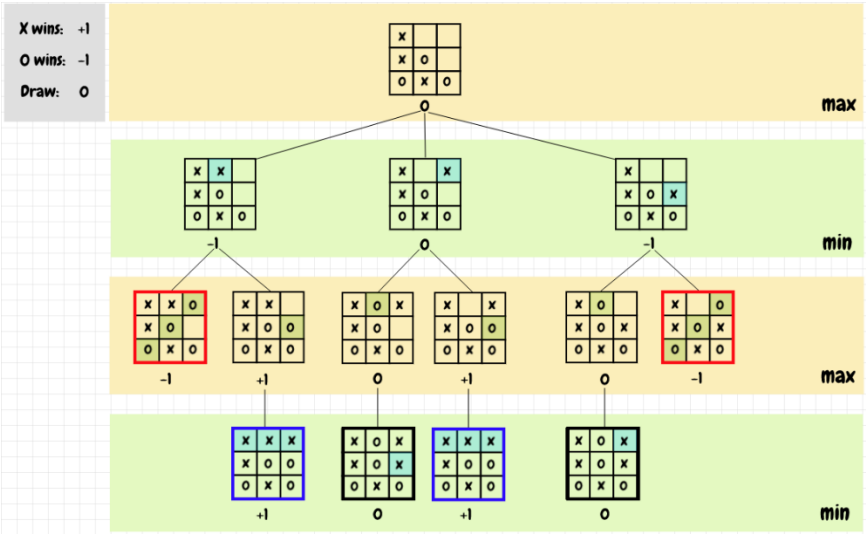
```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 {This opening is called the Ruy Lopez.}
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17.
dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28.
Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2
Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5
41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

2.10 Αλγόριθμοι αναζήτησης τύπου DFS

Σε αυτή την ενότητα θα αναλυθούν οι διάφοροι αλγόριθμοι εξερεύνησης του χώρου καταστάσεων. Η γενική ιδέα είναι ότι μελετώντας μελλοντικές καταστάσεις μπορούμε να έχουμε μια καλύτερη προσέγγιση της βέλτιστης κίνησης για την τωρινή κατάσταση.

2.10.1 Minimax

Ο Minimax είναι ο πιο απλός αλγόριθμος ωμής βίας. Εξερευνούμε όλο το δέντρο μέχρι να φτάσουμε σε τελικές καταστάσεις, τις οποίες αξιολογούμε με ένα αριθμητικό score βάσει του τελικού αποτελέσματος (πχ 1 για νίκες 0 για ισοπαλίες -1 για ήττες). Δεδομένου ότι στα παιχνίδια μηδενικού αθροίσματος, οι παίχτες επιδιώκουν το ιδανικό παίξιμο (perfect play), στην δικιά μας σειρά επιλέγουμε την κίνηση που μεγιστοποιεί το score μας ενώ στην σειρά του αντιπάλου την κίνηση που το ελαχιστοποιεί. Έτσι με αυτή την συλλογιστική σε κάθε ύψος του δέντρου μπορούμε να επιλέξουμε την βέλτιστη κίνηση για κάθε πλευρά. Στον κόμβο ρίζα η κίνηση με το μεγαλύτερο score είναι και η κίνηση που θα παίξουμε (στο παρακάτω παράδειγμα θα επιλέγαμε το 2ο παιδί).

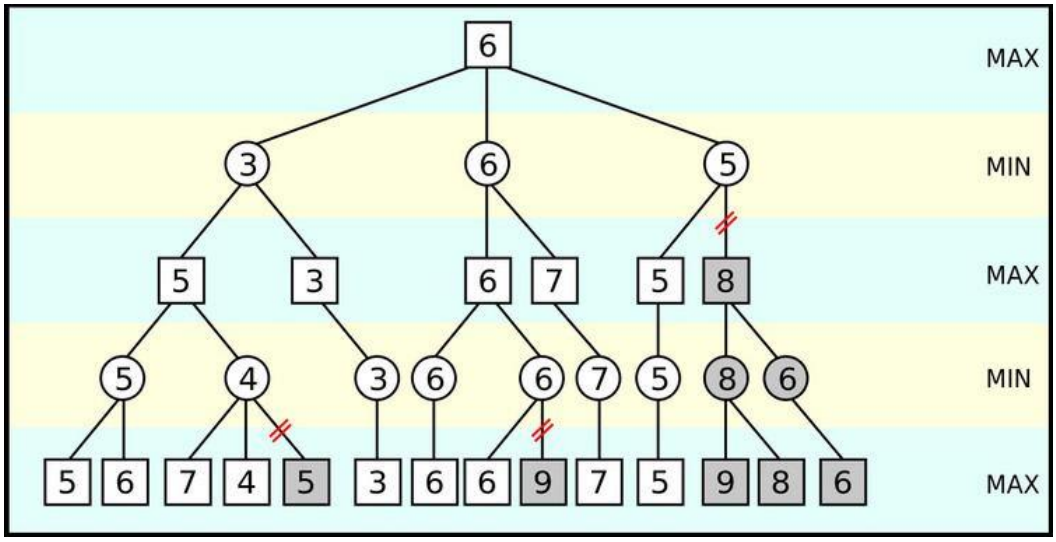


Εικόνα 33: Παράδειγμα Minimax

Όπως αναφέρθηκε προηγουμένως όμως ο χώρος καταστάσεων στο παιχνίδι του σκακιού είναι αρκετά μεγάλος ώστε να μπορεί να υπολογιστεί ολόκληρος. Η άμεση λύση του προβλήματος αυτού είναι ο περιορισμός της αναζήτησης μέχρι ένα προκαθορισμένο βάθος d. Αν φτάσουμε σε τελική κατάσταση μετά από $d' \leq d$ κινήσεις η ιδέα παραμένει η ίδια. Διαφορετικά θα πρέπει να αξιολογήσουμε την θέση προσεγγιστικά με κάποια δικιά μας τεχνική. Η ποιότητα της αξιολόγησης θα κρίνει και την ποιότητα της επιλεγόμενης κίνησης. Διαισθητικά ο περιορισμός αυτός είναι αντίστοιχος της συλλογιστικής των πραγματικών παιχτών οπού μελετούν έναν αριθμό κινήσεων μπροστά για κάθε θέση.

2.10.2 Κλάδεμα Alpha beta

Ακόμα και οι πραγματικοί παίχτες όμως μελετούν συγκεκριμένες κινήσεις και όχι κάθε πιθανό συνδυασμό. Αντίστοιχα μερικές θέσεις μπορούν να αξιολογηθούν κακές πριν φτάσουμε το σύνηθες βάθος. Η ιδέα της αναζήτησης AB είναι ο πρόωρος τερματισμός του υπολογισμού ενός υπο-δέντρου αν αυτός δεν μπορεί να συντελέσει σε αλλαγή του score για τον συγκεκριμένο κόμβο. Για να γίνει πιο αντιληπτό γιατί ισχύει αυτό ας μελετήσουμε το παρακάτω παράδειγμα.



Εικόνα 34: Παράδειγμα κλαδέματος AB

Η αναδρομική διάσχιση του δέντρου θα μας οδηγήσει πρώτα στον αριστερότερο τερματικό κόμβο 5. Καθώς στο προηγούμενο ύψος ελαχιστοποιούμε το κόστος θα υπολογίσουμε την τιμή $min\{5,6\}=5$. Αντίστοιχα στο πιο πάνω επίπεδο ψάχνουμε την τιμή $max\{5,\ x\}$ οπού x η τιμή της επομένης αναδρομικής διάσχισης. Σε αυτό το σημείο πρέπει να παρατηρήσουμε ότι πριν γίνουν άλλοι υπολογισμοί η μέγιστη πιθανή τιμή του κόμβου είναι σίγουρα ≥ 5. Αν κατά την αναζήτηση του x βρούμε έναν αριθμό ≤ 5 δεν χρειάζεται να εξετάσουμε περαιτέρω τους υπολοίπους κόμβους παιδιά καθώς η συγκεκριμένη τιμή μπορεί να γίνει το πολύ μικρότερη του αριθμού αυτού (υπενθύμιση ότι στο τελευταίο επίπεδο ελαχιστοποιούμε την ποσότητα). Η τιμή 5 εν τέλει είναι και η μέγιστη και γλιτώσαμε τον έλεγχο ενός κόμβου καθώς βρέθηκε νωρίτερα η τιμή 4. Το μεγάλο πλεονέκτημα αυτής της ιδέας παρατηρείται σε παραδείγματα όπου αποκόπτονται μεγαλύτερα υπο-δέντρα (όπως αυτό που ξεκινάει από τον κόμβο 8). Η ίδια λογική εφαρμόζεται κατά την διάσχιση και σε εναλλαγές min max έναντι του max min που μελετήσαμε.

Στην πράξη όπως διασχίζουμε το δέντρο κρατάμε 2 τιμές a,b όπου αποθηκεύουν τα έως τώρα γνωστά μέγιστα και ελάχιστα score για να πραγματοποιηθεί ο παραπάνω έλεγχος. Οι τιμές αυτές αρχικοποιούνται με $-\infty,\infty$ ώστε να εγγυηθεί η επιτυχία του 1ου ελέγχου.

2.10.3 Ταξινόμηση κινήσεων

Παρατηρούμε ότι στο προηγούμενο παράδειγμα άμα ο αριθμός 4 και 7 ήταν με ανάποδη σειρά θα είχαμε παραλείψει την εξερεύνηση του κόμβου 7. Συμπεραίνουμε ότι η σειρά εξερεύνησης των κόμβων παίζει σημαντικό ρόλο στο πλήθος των αποκοπών. Επίσης παρατηρούμε ότι στο προτελευταίο επίπεδο προσπαθούμε να ελαχιστοποιήσουμε το score και αποκτούμε την βέλτιστη αποκοπή όταν βρίσκουμε το μικρότερο score, δηλαδή όταν διαλέγουμε την βέλτιστη κίνηση πρώτη. Προφανώς αυτό δεν μπορεί να γίνεται με σιγουριά πάντα καθώς τότε δεν θα υπήρχε λόγος να γίνει εξ αρχής η αναζήτηση αν ξέραμε ήδη την απάντηση. Παρ'ολ'αυτα μπορούμε να ταξινομήσουμε τις κινήσεις με κάποια βασικά κριτήρια ώστε να εξετάζονται πρώτα αυτές που ενδεχομένως είναι καλύτερες.

Για παράδειγμα οι αιχμαλωσίες επιφέρουν πιο δραστικές ανισορροπίες στο score έναντι των ήρεμων κινήσεων (quiet moves) άρα είναι λογικό να εξεταστούν πρώτα.

Υπάρχουν αλγόριθμοι αναζήτησης που αξιοποιούν πιο συνθέτες τεχνικές για την ταξινόμηση των κινήσεων (πχ ο PUCT που αξιολογεί τις κινήσεις με χρήση νευρωνικού δικτύου) αλλά δεν χρησιμοποιούνται σε αλγορίθμους τύπου Minimax λόγο αυξημένης απαίτησης ταχύτητας. Η ανάλυση αυτού του αλγορίθμου θα γίνει σε άλλη ενότητα.

Παρακάτω θα παρουσιαστούν μερικές από τις ευρετικές τεχνικές που χρησιμοποιούνται για την αξιολόγηση της σημαντικότητας κάθε κίνησης.

- **MVV-LVA**: (most valuable victim, least valuable Aggressor) Κατά τον υπολογισμό των αιχμαλωσιών έχουν μεγαλύτερη προτεραιότητα κινήσεις από πιόνια μικρότερης αξίας που κατακτούν πιόνια μεγαλύτερης αξίας. Για παράδειγμα άμα ένα πιόνι μπορεί να παγιδεύσει μια βασίλισσα προφανώς αυτή η κίνηση είναι καλύτερη από το να ανταλλαχθούν ισότιμα πιόνια. Στην πράξη παράγουμε ένα δισδιάστατο πίνακα από scores βάσει του οποίου αξιολογούμε κάθε κίνηση κρίνοντας το από/σε είδος πιονιού.
- **SEE**: (static exchange evaluation) η στατική αξιολόγηση μιας κίνησης δεν συνυπολογίζει πιθανές ανταλλαγές πιονιών που μπορεί να γίνουν στο μέλλον. Η τεχνική αυτή προσπαθεί να λύσει το πρόβλημα δίνοντας μια καλύτερη προσέγγιση στην ταξινόμηση των αιχμαλωτίσεων από την MVV-LVA. Παρ'ολ'αυτα είναι πιο χρονοβόρα καθώς απαιτούνται επιπρόσθετοι έλεγχοι. Ο χρόνος που κερδίζουμε από τις επιπλέον αποκοπές όμως, συνήθως υπερτερεί της ακριβότερης αξιολόγησης.
- **Κινήσεις ιστορικού / δολοφόνου**: (history / killer moves) μια κίνηση που συντέλεσε σε αποκοπή υπο-δέντρων σε κάποιο γειτονικό κόμβο ενδεχομένως να συντελέσει σε ισάξιο αποτέλεσμα και σε άλλους κόμβους. Η παραδοχή αυτή αφορά ήρεμες κινήσεις. Συνήθως το πλήθος των Killer κινήσεων δεν υπερβαίνει τις τρεις.
- **Κινήσεις PV**: Στην περίπτωση που χρησιμοποιούμε τον αλγόριθμο της επαναληπτικής εμβάθυνσης γνωρίζουμε το Principal variation (δηλαδή την ακολουθία βέλτιστων κινήσεων) βάσει μιας αναζήτησης μικρότερου βάθους. Σε αυτή την περίπτωση μπορούμε να δοκιμάσουμε τις κινήσεις αυτές πρώτες καθώς υπάρχει μεγάλη πιθανότητα να εξακολουθούν να είναι βέλτιστες και για μεγαλύτερα βάθη. Ακόμα και αν αυτό δεν ισχύει οι τιμές ab αρχικοποιούνται με ένα καλό κάτω φράγμα που οδηγεί σε αποκοπή κινήσεων που είναι σιγουρά χειρότερες από την προηγουμένη αναζήτηση. (Οι PV κινήσεις έχουν την μεγαλύτερη προτεραιότητα από τις άλλες τεχνικές)
- **Εσωτερική επαναληπτική εμβάθυνση**: Στην περίπτωση που για το εκάστοτε βάθος δεν έχει βρεθεί κάποια PV κίνηση μπορούμε να πραγματοποιήσουμε μια προσωρινή ρηχή αναζήτηση ώστε να αξιολογήσουμε τις κινήσεις ορθότερα. (Η διαδικασία αυτή μπορεί να είναι πολύ αργή όμως καθώς έμμεσα αυξάνουμε το βάθος της κύριας αναζήτησης)

2.10.4 Επαναληπτική εμβάθυνση

Η ιδέα της επαναληπτικής εμβάθυνσης είναι πολύ απλή, αντί να τρέξουμε τον αλγόριθμο αναζήτησης για ένα μεγάλο βάθος τον τρέχουμε επαναληπτικά πολλαπλές φορές αυξάνοντας την απαιτούμενη τιμή κάθε φορά. Για παράδειγμα άμα θέλουμε να μελετήσουμε την βέλτιστη κίνηση ως ένα βάθος n τότε θα τρέξουμε κάθε αναζήτηση $[d_1, \dots, d_n]$. Η υλοποίηση αυτή ίσως να φανεί άσκοπη και χρονοβόρα μιας και υπολογίζουμε $n-1$ αναζητήσεις παραπάνω από ότι χρειαζόμαστε. Βασιζόμενοι όμως στο ότι οι καλύτερες αποκοπές συντελούν σε πιο γρήγορη αναζήτηση μεγαλύτερων βαθών, μπορούμε να παρατηρήσουμε ότι η κάθε προηγουμένη αναζήτηση βελτιώνει την επόμενη. Στην ουσία το όφελος των «άσκοπων» αναζητήσεων αθροιστικά δημιουργεί συνθήκες για καλύτερη ταξινόμηση κινήσεων.

Ένα ακόμα έμμεσο πλεονέκτημα αυτής της υλοποίησης σχετίζεται και με το πρόβλημα του υπολογισμού κινήσεων σε προκαθορισμένο χρόνο. Άμα για παράδειγμα έχουμε t seconds για την εύρεση της βέλτιστης κίνησης δεν μπορούμε να γνωρίζουμε ποιο είναι το βάθος που πρέπει να ελέγξουμε. Κάθε θέση έχει διαφορετική δυσκολία, διαφορετικό πλήθος έγκυρων κινήσεων, η ταξινόμηση μας μπορεί να παράγει καλές αποκοπές ενώ σε άλλες όχι. Άρα για δύο διαφορετικές θέσεις ένα σταθερό βάθος d μπορεί να παράξει τελείως διαφορετικούς χρόνους. Αν όμως αρχίσουμε την αναζήτηση αυξητικά μπορούμε απλά να σταματήσουμε όταν τελειώσει ο χρόνος. Μπορούμε να είμαστε σχεδόν σίγουροι ότι η αναζήτηση με βάθος ένα θα έχει τελειώσει, μιας και οι πιθανοί κόμβοι είναι όσοι και οι κινήσεις (αγνοούμε το σενάριο που και αυτός ο έλεγχος αποτυγχάνει για ένα προκαθορισμένο t γιατί είναι μη ρεαλιστικά αυστηρός).

2.10.5 Aspiration αναζήτηση

Όσο πιο μικρό είναι το παράθυρο [a,b] τόσο πιο πολλές ευκαιρίες για αποκοπές παρουσιάζονται. Η βασική ιδέα είναι ότι μπορούμε να εκτιμήσουμε την τελική αξιολόγηση της αναζήτησης για να ορίσουμε ένα στενότερο παράθυρο. Στην περίπτωση όμως που η τελική κίνηση δίνει ένα score εκτός του παραθύρου αυτού τότε η αρχική μας θεώρηση ήταν λάθος και πρέπει να ξανά υπολογιστεί η διαδικασία από την αρχή με το πλήρες παράθυρο $[-\infty, \infty]$. Γενικά η μέθοδος αυτή επιφέρει βελτιούμενους χρόνους άμα τις περισσότερες φορές δεν απαιτείται επαναϋπολογισμός, αλλά το παράθυρο είναι αρκετά στενό ώστε να αποφεύγουμε και πολλούς κόμβους. Η προσέγγιση του score συνήθως γίνεται με βάση το αποτέλεσμα της προηγουμένης αναζήτησης στον αλγόριθμο της επαναληπτικής εμβάθυνσης. Το παράθυρο μπορεί να είναι στατικό είτε δυναμικά μεταβαλλόμενο με βάση τις πιθανές αποτυχίες κατά την αναζήτηση. (Μεγαλύτερες εμβέλειες μειώνουν την πιθανότητα σφάλματος)

2.10.6 Πίνακες μεταθέσεων

Μερικές θέσεις στο σκάκι μπορούν να εμφανιστούν πολλαπλές φορές. Αυτό σημαίνει ότι η αναζήτηση σε αυτούς τους κόμβους δεν χρειάζεται να επαληθευτεί αν έχουμε κρατήσει την πληροφορία που μας αφορά. Στην ορολογία αυτά τα σενάρια ονομάζονται μεταθέσεις (transpositions). Στην ουσία το δέντρο καταστάσεων μας είναι ένας γράφος με πιθανές κυκλικές σχέσεις. Για να γίνει πιο κατανοητό ας δούμε ένα παράδειγμα. Έστω ότι τα άσπρα ξεκινούν με την κίνηση ενός τυχαίου πιονιού κατά μία θέση προς τα πάνω. Τα μαύρα απαντούν με αντίστοιχη συμπεριφορά. Ύστερα η διαδικασία επαναλαμβάνεται και τα ίδια πιόνια πάνε μια ακόμα θέση μπροστά. Τώρα ας σκεφτούμε ένα διαφορετικό παιχνίδι όπου και οι δύο παίχτες επιλέγουν να κουνήσουν τα ίδια πιόνια δύο θέσεις προς

τα πάνω κατευθείαν. Παρατηρούμε ότι φτάσαμε στον ίδιο τελικό σχηματισμό πιονιών αλλά με διαφορετικό πλήθος κινήσεων. Στις περισσότερες περιπτώσεις η διαφορά αυτή δεν παίζει σημαντικό ρόλο στην αξιολόγηση θέσης. (Συγκεκριμένα το πλήθος των κινήσεων μας ενδιαφέρει μόνο σε περιπτώσεις ισοπαλίας για τον κανόνα 50 κινήσεων η επαναλαμβανομένης θέσης). Έτσι καταλήγουμε στην εξής παρατήρηση. Αν κατά την εξερεύνηση ενός κόμβου γνωρίζουμε ότι έχει ήδη επισκεφθεί στο παρελθόν με βαθύτερο βάθος τότε η αναζήτηση μπορεί να σταματήσει για το συγκεκριμένο υπο-δέντρο μιας και ξέρουμε ήδη το ακριβές αποτέλεσμα (άμα η προηγουμένη εξερεύνηση ήταν ιδίου βάθους) ή ένα ακόμα καλύτερο (αν η εξερεύνηση ήταν μεγαλύτερου βάθους).

Η ιδέα αυτή είναι ακόμα πιο ισχυρή άμα σκεφτούμε ότι κατά τον υπολογισμό της βέλτιστης κίνησης συνυπολογίζουμε και την κίνηση του αντίπαλου. Αυτό σημαίνει ότι άμα η κίνηση που θα παιχτεί είναι από τις πιθανές εκδοχές που έχουν ήδη εξεταστεί θα γλιτώσουμε τον υπολογισμό ενός συνόλου κόμβων για την επόμενη αναζήτηση. Συγκεκριμένα αφού συνηθώς αξιοποιείται ο αλγόριθμος της επαναληπτικής εμβάθυνσης, είμαστε σίγουροι ότι οι περισσότεροι κόμβοι θα βρεθούν για τις πρώτες επαναλήψεις μικρότερου βάθους. Στην περίπτωση που φτάσουμε σε έναν κόμβο που έχει ήδη υπολογιστεί αλλά με μικρότερο βάθος από το επιθυμητό τότε δεν μπορούμε να ξέρουμε αν το τελικό αποτέλεσμα θα παραμείνει ίδιο άμα εξερευνούσαμε λίγες κινήσεις παραπάνω. Ενδεχομένως μια κίνηση που φαίνεται καλή για βάθος 5 να αποδειχθεί κακή για βάθος 6. (πχ για βάθος 1 όποιο πιόνι και να κατακτήσουμε καταλήγουμε σε αξιολόγηση πιο ισχυρής θέση αγνοώντας το γεγονός ότι μπορεί να είναι προστατευμένο)

Σε αυτή την περίπτωση όμως μπορούμε να αρκεστούμε στην βέλτιστη κίνηση για την ταξινόμηση των κινήσεων όπως είχε αναφερθεί και προηγουμένως. Πρακτικά τώρα δεν κρατάμε μόνο τις κύριες κινήσεις PV της προηγούμενης επανάληψης αλλά την βέλτιστη κίνηση για κάθε κατάσταση που έχει εξεταστεί στις αναζητήσεις.

Ο πίνακας που κρατάει το σύνολο των πληροφοριών ονομάζεται πίνακας μεταθέσεων και αποτελεί μια μορφή Hash table. Κατά την εξέλιξη του παιχνιδιού μπορεί να γεμίσει με άχρηστες πληροφορίες για καταστάσεις που δεν μπορούν να συμβούν πια. Στην περίπτωση αυτή μπορούν να υπάρξουν πολιτικές διαγραφής πεδίων που εφαρμόζονται όταν γεμίσουν τα Buckets σε περίπτωση σύγκρουσης.

Πιο συγκεκριμένα οι πιο συνήθεις πρακτικές είναι οι ακόλουθες:

- Άμα βρεθούμε σε πεδίο ιδίου κόμβου με μικρότερο βάθος τότε πάντα προτιμάται ο κόμβος με την μεγαλύτερη τιμή καθώς περιέχει καλύτερη αξιολόγηση της θέσης.
- Κόμβοι μεγαλύτερου βάθους προτιμώνται έναντι κόμβων μικρότερου βάθους για τον ίδιο λόγο
- Κόμβοι κοντά στα φύλλα του δέντρου είναι πιο πιθανό να αναζητηθούν περισσότερες φορές σε μελλοντικές αναζητήσεις.
- Κόμβοι που εξετάστηκαν πιο πρόσφατα έχουν μεγαλύτερη πιθανότητα να αναζητηθούν ξανά.

Οι πιο πρόσφατες υλοποιήσεις χρησιμοποιούν μια μίξη των παραπάνω με χρήση σχετικών flags. Είναι αναγκαίο όμως το κάθε entry να είναι αρκετά μικρό ώστε το φαινόμενο να παρουσιάζεται με πιο αργούς ρυθμούς. (Να σημειωθεί ότι καθώς ο πίνακας συνήθως είναι στατικού μεγέθους είναι αναμφίβολο ότι θα εμφανιστεί λόγω του υψηλού αριθμού κόμβων που εξερευνούνται στην αναζήτηση)

2.10.7 Κλειδιά Zobrist

Για να μπορέσουμε να αποθηκεύσουμε έναν κόμβο στον πίνακα απαιτείται μια διαδικασία κατακερματισμού για να διασφαλιστεί η πολυπλοκότητα του $O(1)$. Ο πιο γνωστός τρόπος που μπορεί να γίνει αυτό είναι τα κλειδιά Zobrist.

Ξεκινώντας από μια 64 bit τιμή $\zeta = 0$ μπορούμε να εισάγουμε στοιχεία σε αυτή εφαρμόζοντας τον τελεστή xor με μεταβλητές που έχουν αρχικοποιηθεί με τυχαίους αριθμούς. Πιο συγκεκριμένα στην αρχή του προγράμματος παράγουμε:

- ένα τυχαίο αριθμό για κάθε θέση του ταμπλό για κάθε χρώμα και είδος πιονιού.
- ένα τυχαίο αριθμό για την σειρά του παίχτη.
- 4 αριθμούς για τα δικαιώματα ροκέ. (Στην πράξη είναι 16 για πιο γρήγορη προσπέλαση αλλά αυτό αποτελεί λεπτομέρεια υλοποίησης)
- 8 αριθμούς που σηματοδοτούν την ενεργή στήλη en passant (άμα αυτή υπάρχει).

Όλες οι παραπάνω μεταβλητές επιδιώκουμε να έχουν όσο το δυνατόν περισσότερα δυαδικά ψηφιά για αποφυγή συγκρούσεων.

Για να παράξουμε το κλειδί Zobrist μια θέσης διατρέχουμε κάθε θέση (ή άλλη πληροφορία) του ταμπλό και επαυξάνουμε την μεταβλητή ζ με τον ανάλογο τυχαίο αριθμό.

Το μεγαλύτερο πλεονέκτημα αυτής της τεχνικής είναι ότι δεν απαιτείται ο επαναϋπολογισμός του κλειδιού για κάθε ταμπλό από το μηδέν.

Η ιδέα βασίζεται στο γεγονός ότι $(a \otimes b) \otimes b = a$, κοινώς αν σε ένα σύνολο εφαρμόσουμε την πράξη xor με ένα από τα αντικείμενά του, τότε αφαιρείται από αυτό.

Για παράδειγμα αν $a = 0b001, b = 0b1010$ τοτε

$a \otimes b = b001 \otimes 0b101 = 0b100, a \otimes b \otimes b = 0b100 \otimes 0b101 = 0b001 = a$

Αυτό αποδεικνύεται ευκολά άμα σκεφτούμε ότι $a \otimes a = 0, a \otimes 0 = a$ αρα $(a \otimes b) \otimes b = a \otimes (b \otimes b) = a \otimes 0 = a$.

Στην πράξη αφού γνωρίζουμε το κλειδί Zobrist της προηγούμενης θέσης, κατά την διάσχιση του δέντρου μπορούμε να υπολογίσουμε τα καινούργια αφαιρώντας και προσθέτοντας τα κατάλληλα random values από το σύνολο. Έτσι σε αναλογία έχουμε πράξεις της μορφής

$$(\dots\ xor\ Black\_pawn\_e1\ xor\ White\_pawn\_e8\ xor\ \dots)\ xor\ Black\_pawn\_e1\ =$$

$$(\dots\ xor\ White\_pawn\_e8\ xor\ \dots)$$

Η χρήση των κλειδιών Zobrist δεν περιορίζεται μόνο στους πίνακες μεταθέσεων αλλά μπορεί να φανεί χρήσιμη και στον έλεγχο ισοπαλίας λόγω επαναλαμβανομένης θέσης όπου απαιτείται ο έλεγχος ομοιότητας δύο θέσεων. Αντί να ελέγχουμε κάθε πιθανό χαρακτηριστικό του ταμπλό συγκρίνουμε δύο 64 bit αριθμούς.

2.10.8 Φαινόμενο του ορίζοντα και quiescence αναζήτηση

Οι αλγόριθμοι αναζήτησης παρουσιάζουν το φαινόμενο του ορίζοντα (horizon effect). Καθώς μελετούμε ένα μέρος του δέντρου καταστάσεων μπορεί να νομίζουμε ότι έχουμε αποφύγει μια μελλοντική κατάσταση επειδή δεν κοιτάξαμε αρκετές κινήσεις μπροστά. Το πρόβλημα αυτό προσπαθεί να λύσει ο αλγόριθμος quiescence. Πρακτικά μετατρέπουμε τις τελικές καταστάσεις σε πιο σταθερές αναπαραστάσεις με το να ελέγχουμε πιθανές αιχμαλωσίες. Με αυτόν τον τρόπο επεκτείνουμε έμμεσα το βάθος αναζήτησης αλλά η αξιολόγηση θέσης είναι πιο ποιοτική και το πρόβλημα περιορίζεται. Ο αλγόριθμος έχει ιδία μορφή με αυτή της αναζήτησης AB με την μόνη παραδοχή ότι δεν ελέγχουμε τελικές καταστάσεις και μελετάμε μόνο κινήσεις αιχμαλωσίας. Όπως και πριν η ταξινόμηση των κινήσεων με κριτήρια όπως το MVV-LVA είναι απαραίτητη για την μέγιστη αποκοπή κόμβων. Ο λόγος που μια πλήρης αναζήτηση είναι απαραίτητη είναι γιατί η σειρά με την οποία θα πραγματοποιηθούν οι κινήσεις είναι σημαντική για το τελικό αποτέλεσμα της θέσης.

Όσο καλή και να είναι η συνάρτηση αξιολόγησης θέσης, παραμένει να είναι στατική για αυτό και η μέθοδος είναι καλό να εφαρμόζεται ανεξαρτήτως τεχνικής.

2.10.9 Negamax

Ο negamax αποτελεί παραλλαγή του minimax και δεν προσφέρει κάποια ιδιαίτερη βελτίωση πέρα από ευκολότερη συγγραφή κώδικα. Όπως είδαμε προηγουμένως ο κλασικός minimax συμπεριφέρεται διαφορετικά ανάλογα με το αν μεγιστοποιεί η ελαχιστοποιεί τον εκάστοτε κόμβο. Οι διαφορές όμως μεταξύ των 2 διαδικασιών είναι αρκετά πανομοιότυπες. Ειδικά αν προσθέσουμε και την βελτίωση κλαδέματος ab παρατηρούμε ότι έχουμε σχεδόν διπλότυπο κώδικα. Ακολουθεί ένα παράδειγμα ψευδοκώδικα.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value ≥ β then
                break
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value ≤ α then
                break
            β := min(β, value)
        return value
```

Παρατηρούμε όμως ότι ισχύει η ακόλουθη σχέση: $max(a,b) = -min(-a,-b)$. Καθώς η κύρια διαφορά μεταξύ των δύο καταστάσεων είναι οι σχέσεις Min, max μπορούμε να αντικαταστήσουμε κάθε συνάρτηση min με την αντίστοιχη ισοδυναμία της.

Έτσι καταλήγουμε σε μια πιο απλή συγγραφή του ψευδοκώδικα.

```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node

    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := −∞
    foreach child in childNodes do
        value := max(value, −negamax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break
    return value
```

2.10.10 PVS

Ο αλγόριθμος PVS (Principal variation search) αποτελεί επέκταση της αναζήτησης ab. Οι περισσότερες κινήσεις δεν είναι αποδέκτες και για τους 2 παίχτες άρα δεν χρειάζεται να κάνουμε μια ολόκληρη αναζήτηση για να βρούμε το πραγματικό score. Το πραγματικό score απαιτείται μόνο σε κινήσεις του principal variation. Η ιδέα θυμίζει λίγο την aspiration αναζήτηση αλλά υλοποιείται εντός του κεντρικού αλγορίθμου και όχι σε εξωτερικές κλήσεις της επαναληπτικής εμβάθυνσης. Πιο συγκεκριμένα αφού ελέγξουμε την 1η κίνηση στον εκάστοτε κόμβο μπορούμε να θεωρήσουμε ένα μηδενικού εύρους παράθυρο για τις αναδρομικές κλήσεις στους κόμβους παιδιά με κάτω φράγμα την τιμή του α. Αν το score μετά από αυτή την παραδοχή εξακολουθεί να είναι εντός του παραθύρου τότε γλιτώσαμε άσκοπες πράξεις καθώς κερδίσαμε επιπλέον αποκοπές με ένα στενότερο παράθυρο. Αν βρεθεί εκτός αυτού τότε σημαίνει ότι υπάρχει πιθανότητα για εύρεση καλύτερου score και τότε πραγματοποιείται ένας πλήρης έλεγχος από την αρχή.

```
function pvs(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    for each child of node do
        if child is first child then
            score := −pvs(child, depth − 1, −β, −α, −color)
        else
            score := −pvs(child, depth − 1, −α − 1, −α, −color)
            if α < score < β then
                score := −pvs(child, depth − 1, −β, −score, −color)
        α := max(α, score)
        if α ≥ β then
            break (* beta cut-off *)
    return α
```

Όπως και στην aspiration αναζήτηση η βελτίωση στους χρόνους προέρχεται από το γεγονός ότι οι κόμβοι που αποκόπτονται υπερτερούν τους επανυπολογισμούς που θα πρέπει να γίνουν με πλήρες παράθυρο. Ο αλγόριθμος Negascout είναι πανομοιότυπος του PVS με μικρές διάφορες σε λεπτομέρειες υλοποίησης.

2.10.11 Άλλες τεχνικές αποκοπής

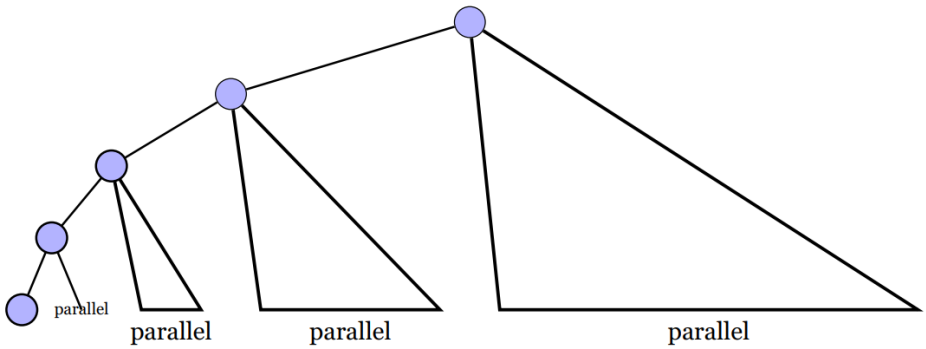
Πέρα από την βελτίωση που προσφέρει το κλάδεμα ab υπάρχουν και άλλες πιο άπληστες (greedy) τεχνικές. Καθώς πολλές από αυτές βασίζονται σε «ευρετικά κλαδέματα», δεν μπορούμε να είμαστε σίγουροι αν παραμελούμε κάποια καλύτερη κίνηση που έχει αγνοηθεί. Παρ'ολ'αυτα καθώς ο αριθμός αποκοπών είναι μεγάλος μας δίνεται ευκαιρία για αναζήτηση μεγαλύτερου βάθους. Κατά μέσο ορό το πλεονέκτημα αυτό υπερτερεί τυχόν λανθασμένων αποκοπών. [16]

- **Κλάδεμα κενής κίνησης (null move pruning):** Αν ήταν εφικτό να μην παίξουμε στην σειρά μας, τότε αυτό θα μας οδηγούσε σε μια χειρότερη θέση από το να παίζαμε οποιαδήποτε κίνηση [17]**.** Αυτή η ιδέα μας επιτρέπει να ορίσουμε ένα κάτω φράγμα για την τιμή του α κάνοντας μια αναζήτηση κενής κίνησης (δηλαδή να αλλάξουμε απλά την σειρά του παιχνιδιού). Δεν πραγματοποιείται η τεχνική άμα βρισκόμαστε σε check γιατί τότε θα είχαμε λανθασμένη θέση, ούτε γίνεται δύο φορές καθώς τότε δεν θα είχε νόημα να χάσουν και οι δύο παίχτες την σειρά τους. Το παράθυρο αναζήτησης είναι 1 και κοντά στην τιμή του b. Το βάθος της αναζήτησης είναι μειωμένο κατά μια μεταβλητή R. Ύστερα μπορούμε να συγκρίνουμε την τιμή του αποτελέσματος με το τωρινό b για πιθανές αποκοπές. Διαισθητικά η ιδέα είναι ότι άμα με το να επιλέξουμε την χειρότερη κίνηση εξακολουθούμε να βρισκόμαστε σε ισχυρή θέση τότε σε αυτή την θέση σιγουρά θα έχουμε αποκοπή κόμβων. Μια από τις περιπτώσεις που η τεχνική αυτή δεν πρέπει να χρησιμοποιηθεί είναι σε θέσεις zugzwang οπού η κενή κίνηση είναι προτιμότερη από οποιαδήποτε άλλη κίνηση.
- **Κλάδεμα δέλτα (delta pruning):** Κατά την quiescence αναζήτηση ελέγχουμε άμα η τωρινή αξιολόγηση θέσης + ένα margin μπορεί να οδηγήσει σε καλύτερη θέση. Αν κάτι τέτοιο δεν ισχύει τότε σταματάμε πρώρα τον έλεγχο (Πχ η τιμή του margin μπορεί να είναι όσο η αξία μιας βασίλισσας, άμα ακόμα και με το να κατακτήσουμε μια βασίλισσα δεν βρισκόμαστε σε καλύτερη θέση μάλλον αυτή η κίνηση δεν είναι αξία για περαιτέρω αναζήτηση).
- **Κλάδεμα ματαιότητας (Futility pruning):** Η ίδια τεχνική του κλαδέματος δέλτα μπορεί να εφαρμοστεί σε κόμβους όπου το βάθος είναι κοντά στον ορίζοντα (τυπικά ο αλγόριθμος εφαρμόζεται για βάθος 1 αλλά μπορεί να εφαρμοστεί και για μεγαλύτερες τιμές όπως 7 ή 8). Ορίζουμε ένα futility_margin βάσει του οποίου βαθμολογούμε την αξία της κίνησης. (Καθώς μια τέτοια αξιολόγηση είναι προσεγγιστική, οι τεχνικές αυτές μπορούν να οδηγήσουν σε εύρεση μη βέλτιστης λύσης). Δεν χρησιμοποιείται άμα βρισκόμαστε σε check ή μια κίνηση μπορεί να οδηγήσει σε check για καλύτερη σταθερότητα στην εύρεση τακτικών.
- **Μείωση αργοπορημένων κινήσεων (Late move reduction):** Άμα θεωρήσουμε ότι η ταξινόμηση των κινήσεων είναι καλή, η $1^{η}$ κίνηση που θα ελεγχθεί είναι και η καλύτερη. Αυτό σημαίνει ότι για όλες τις υπόλοιπες μπορούμε να κάνουμε μια αναζήτηση μικρότερου βάθους και παραθύρου ώστε να αποδείξουμε ότι οι υπόλοιποι κόμβοι θα αποκοπούν γλιτώνοντας επιπλέον υπολογισμούς. Αν όμως η τιμή a μεγαλώσει τότε πρέπει να κάνουμε επανέλεγχο από την αρχή με το κανονικό βάθος και εύρος παραθύρου.

Παρατηρούμε ότι οι αλγόριθμοι που περιγράφηκαν, κατά κύριο λόγο έχουν ακολουθιακή μορφή. Η μελλοντική εξερεύνηση κόμβων εξαρτάται από τις τιμές των a,b που έχουν βρεθεί κατά προηγούμενες εξερευνήσεις. Άμα κάνουμε παράλληλη αναζήτηση για κάθε κλαδί του δέντρου με τυχαίο τρόπο μπορεί να καταλήξουμε να κάνουμε υπολογισμούς για υποδέντρα που θα είχαν απορριφθεί με πιο στενά παράθυρα. Παρακάτω θα παρουσιαστούν αλγόριθμοι που προσπαθούν να λύσουν το πρόβλημα αυτό.

2.10.12 Διασπάσεις PV

Ο αλγόριθμος principal variation splitting [18] αποτελεί μια επέκταση των αλγορίθμων Minimax εισάγοντας το στοιχείο του παραλληλισμού. Η ιδέα είναι ότι δεν μπορούν να γίνουν αποκοπές σε κανένα υποδέντρο ενός κόμβου άμα δεν είναι γνωστό τουλάχιστον ένα κάτω φράγμα της τιμής a στην ρίζα του κλαδιού αυτού. Επίσης όταν είμαστε στο ίδιο επίπεδο η τιμή b παραμένει σταθερή άρα οι πιθανές αποκοπές δεν επηρεάζονται από την εξερεύνηση sibling κόμβων. Η διαδικασία παραμένει ακολουθιακή μέχρι να υπολογιστεί ο αριστερότερος τερματικός κόμβος του δέντρου.



Εικόνα 35: Παράδειγμα διασπάσεων pv [19]

Ύστερα με την άνοδο στο επόμενο επίπεδο αναθέτουμε σε αδρανή νήματα την αναζήτηση των υπολοίπων κινήσεων αφού ενημερώσουμε την τιμή a. Όταν τελειώσουν τους υπολογισμούς όλα τα νήματα, γίνεται ένας συγχρονισμός. Επαναληπτικά πηγαίνουμε στο επόμενο επίπεδο επαναλαμβάνοντας την διαδικασία μέχρι να φτάσουμε την ρίζα του δέντρου.

Τα μεγαλύτερα μειονεκτήματα αυτής της τεχνικής είναι:

- Η εξάρτηση των υπολογισμών μεταξύ υψηλοτέρων επίπεδων. Πριν παραλληλοποιηθεί το βάθος 6 πρέπει να έχει τελειώσει ο υπολογισμός του βάθους 5.
- Ο αριθμός των πιθανών νημάτων φράσσεται από το πλήθος κινήσεων του εκάστοτε επιπέδου.
- Κάθε νήμα λαμβάνει δέντρα διαφορετικού μεγέθους. Μικρότερα δέντρα θα τελειώσουν πιο γρήγορα με συμπέρασμα, τα αντίστοιχοι νήματα να παραμένουν ανενεργά μέχρι να συγχρονιστούν ξανά.

Ο αλγόριθμος Young brothers wait concept (YBWC) αποτελεί παραλλαγή του principal variation splitting και προσπαθεί να λύσει μερικά από τα παραπάνω προβλήματα. Πιο συγκεκριμένα κάθε νήμα πράττει και αυτό μια ολόκληρη αναζήτηση μέχρι να φτάσει στην αριστερότερη τερματική κατάσταση. Τα υπόλοιπα παιδιά εντάσσονται σε μια ουρά οπού ανενεργά νήματα μπορούν να «κλέψουν» δουλειά. Έτσι αξιοποιούνται καλύτερα τα νήματα που τελείωσαν γρηγορότερα την αναζήτηση τους λόγω μικρότερων υποδέντρων.

2.10.13 Lazy SMP

Ο Lazy SMP είναι ο πιο πρόσφατος παράλληλος αλγόριθμος. Αντικατέστησε τον YBWC στην μηχανή stockfish το 2016. Δεδομένου ότι ο πίνακας μεταθέσεων αποτρέπει την εξερεύνηση κόμβων που ήδη έχουν υπολογιστεί, μπορούμε να παρέχουμε ένα κοινό hash σε όλα τα νήματα μέσω του οποίου θα γίνονται οι κατάλληλες ενημερώσεις και έλεγχοι. Με αυτόν τον τρόπο μπορούμε να δημιουργήσουμε όσα νήματα επιθυμούμε με αρχικό κόμβο την ρίζα του δέντρου. Εφόσον οι υπολογισμοί δεν γίνονται με την ίδια σειρά ο πίνακας TT θα ανανεώνεται και κάθε νήμα θα ψάχνει αυτόνομα διαφορετικά υποδέντρα. Αυτό λύνει τον μεγάλο περιορισμό που είχαν οι προηγούμενοι αλγόριθμοι που δεν επέτρεπαν μεγάλο πλήθος από νήματα. Επίσης δεν υπάρχει περιορισμός για την σειρά που μπορούν να υπολογιστούν οι κόμβοι παράλληλα (όπως πχ ανά επίπεδο όπως πριν). Στην πράξη η αναζήτηση δεν είναι τελείως αυτόνομη καθώς η χρήση ενός κοινού hash για όλα τα νήματα υποδηλώνει την χρήση μηχανισμών Locks για την ορθή ανανέωση των στοιχείων του πίνακα. Αυτός ο περιορισμός μπορεί να προκαλέσει αρκετές καθυστερήσεις λόγω του υψηλού αριθμού από loads / reads στην δομή TT. Βασιζόμενοι στο ότι τις περισσότερες φορές δεν θα παραχθεί αλλοίωση (corruption) είτε επειδή οι ανανεώσεις του πίνακα δεν είναι σε ίδιες εγραφές είτε λόγω χρονισμού, έχουν προταθεί lockless υλοποιήσεις [20] [21]. Το προφανές μειονέκτημα είναι η εισαγωγή σφαλμάτων στον πίνακα.

Ο έλεγχος ορθότητας μπορεί να γίνει με την χρήση checksum. Άμα κατά την αναζήτηση ενός στοιχείου ανακαλύψουμε ότι έχει γίνει corrupt λόγω πολλαπλών παραλλήλων εγγράφων τότε η εγγραφή μπορεί απλά να απορριφθεί.

Ενδεχομένος ένα νήμα να αρχίσει την εξερεύνηση ενός κλαδιού προτού ένα άλλο προλάβει να ανανεώσει τον πίνακα. Σε αυτή την περίπτωση μπορεί να γίνουν άσκοπες πράξεις. Μια λύση που μειώνει το φαινόμενο αυτό είναι η διαφοροποίηση των βαθών για κάθε νήμα. Παρατηρούμε όμως ότι έτσι δεν αξιοποιείται πλήρως ο αλγόριθμος της επαναληπτικής εμβάθυνσης καθώς μπορεί να μην έχουν προλάβει να υπολογιστούν όλες οι ρηχές αναζητήσεις και να μην μπορούν να χρησιμοποιηθούν τα αποτελέσματα αυτών σε αποκοπές κόμβων με χρήση καλύτερων ταξινομήσεων κινήσεων.

Πίνακας 6: Βελτιώσεις ταχύτητας με Lazy SMP [19]

Threads	Time (ms)	Speedup
1	5209	1
2	4586	1.14
4	2590	2.01

2.11 Αλγόριθμοι αναζήτησης τύπου BFS

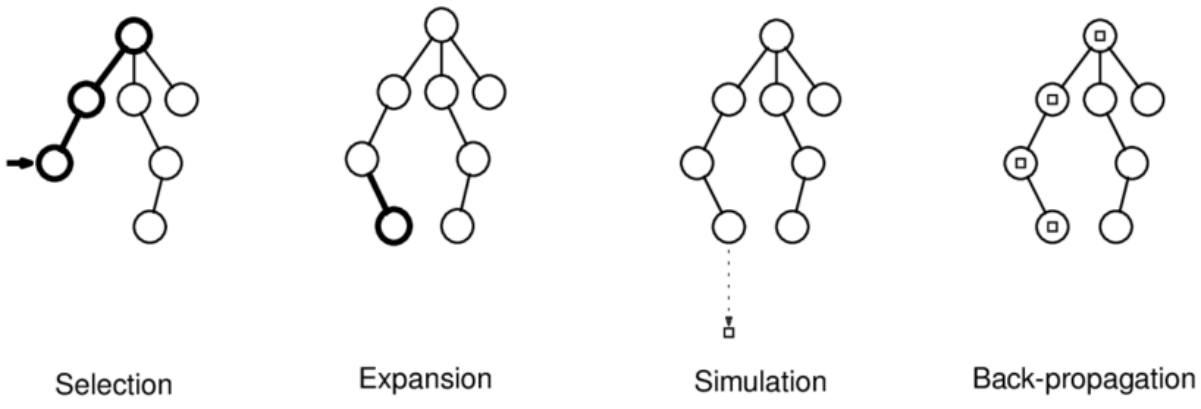
Η κύρια διαφορά μεταξύ των αλγορίθμων DFS και BFS (best first search και όχι breadth first search) είναι ότι αντί να εξερευνούμε το δέντρο καταστάσεων με έμφαση το βάθος, κάθε φορά πηγαίνουμε στον κόμβο του έως τώρα γνωστού δέντρου με το καλύτερο score. Διαισθητικά ελέγχουμε την θέση που φαίνεται να παρουσιάζει την καλύτερη συμπεριφορά. Όσο αφορά το κόστος των τεχνικών αυτών παρατηρούμε ότι

- Για να γνωρίζουμε ποιος είναι ο επόμενος καλύτερος κόμβος χρειάζεται να κρατάμε όλες τις έως τώρα τελικές καταστάσεις στην μνήμη (frontiers). Στους αλγορίθμους dfs υπάρχουν μόνο οι κόμβοι που βρίσκονται εντός της κλήσης στοίβας. Οι κόμβοι συνόρου συνήθως αποθηκεύονται σε μια ουρά προτεραιότητας για γρήγορη εύρεση.
- Ενώ στους αλγορίθμους DFS αξιολογούμε την θέση μόνο σε προκαθορισμένα depths ή τελικές καταστάσεις, οι αλγόριθμοι BFS απαιτούν στατική αξιολόγηση για κάθε κόμβο ώστε να μπορεί να βρεθεί ο επόμενος κόμβος εξερεύνησης.

2.11.1 MCTS

Ο MCTS (monte carlo tree search) [22] είναι ένας πιθανοτικός αλγόριθμος. Η βασική ιδέα είναι ότι άμα παίξουμε άπειρα τυχαία παιχνίδια από την κατάσταση στην οποία βρισκόμαστε σε άπειρο χρόνο σε έναν υπολογιστή με άπειρη μνήμη η βέλτιστη κίνηση θα έχει παράξει τις περισσότερες νίκες. Διαισθητικά μια καλύτερη θέση έχει περισσότερες πιθανότητες να καταλήξει σε νίκη από μια χειρότερη θέση όταν οι 2 παίχτες παίζουν τυχαία. Άρα συμπεραίνουμε ότι το πλήθος των παιχνιδιών που θα δοκιμάσουμε έχει άμεση συσχέτιση με την σύγκλιση του αλγορίθμου στην βέλτιστη λύση.

Ο αλγόριθμος αποτελείται από 4 βήματα



Εικόνα 36: Βήματα MCTS

- **Selection**: Κάθε επανάληψη του αλγορίθμου επαυξάνει ένα δέντρο καταστάσεων όπου κάθε κόμβος περιέχει χρήσιμες πληροφορίες για επόμενες αναζητήσεις. Στην αρχή μιας νέας επανάληψης ξεκινούμε από την ρίζα του δέντρου και προσπαθούμε να φτάσουμε σε κόμβους φύλλα, βάσει ενός κριτήριου Tree policy. Το policy πρακτικά μας λέει σε κάθε κόμβο ποιο παιδί πρέπει να ακολουθήσουμε. Για να μην καταλήξουμε να κάνουμε μυωπική αναζήτηση μόνο σε συγκεκριμένους κόμβους που φαίνονται να έχουν καλά αποτελέσματα λόγω του «φαινομένου του ορίζοντα», ενθαρρύνουμε και την εξερεύνηση κόμβων που δεν έχουν δεχτεί πολλές επισκέψεις. Η πιο συχνή μέθοδος που χρησιμοποιείται είναι η UCB.
- **Expansion**: Αφού φτάσουμε σε ένα φύλλο του δέντρου, αν δεν βρισκόμαστε σε τελική κατάσταση, παράγουμε όλες τις κινήσεις για την θέση αυτή και επιλέγουμε μια από αυτές (ενδεχομένως τυχαία).
- **Simulation**: Το βήμα αυτό συνήθως λέγεται και roll out. Πρακτικά μεταβαίνουμε στον νέο κόμβο που επιλέξαμε στο προηγούμενο βήμα και παίζουμε το παιχνίδι μέχρι να φτάσουμε σε τελική κατάσταση (δηλαδή ισοπαλία νίκη ή ήττα). Κάθε φορά επιλέγουμε κινήσεις με τελείως τυχαίο τρόπο από κάποια τυχαία κατανομή.
- **Backpropagation**: Μεταφέρουμε το αποτέλεσμα του rollout στους εμπλεκομένους κόμβους μέχρι το βήμα του expansion. Δηλαδή αν η ρίζα είναι ο κόμβος R και το rollout ξεκίνησε από τον κόμβο C, ενημερώνουμε τις πληροφορίες των κόμβων στο μονοπάτι $R - C$.

Οι πληροφορίες που αποθηκεύουμε στους κόμβους του δέντρου διαφέρουν ανάλογα με την τεχνική policy tree αλλά συνήθως περιέχουν: πόσες φορές έχουμε επισκεφθεί τον εκάστοτε κόμβο και πόσες φορές έχουμε κερδίσει το παιχνίδι.

Αφού επαναλάβουμε την διαδικασία ένα προκαθορισμένο αριθμό από επαναλήψεις, διαλέγουμε την κίνηση από τον κόμβο R που οδηγεί σε παιδί κόμβο με το μεγαλύτερο πλήθος επισκέψεων. (Διαισθητικά ο κόμβος με τις περισσότερες επισκέψεις είχε την μεγαλύτερη επιτυχία καθώς διαδέχτηκε πολλές φορές από το βήμα του selection). Δεν επιλέγουμε τον κόμβο με το μεγαλύτερο ποσοστό νικών γιατί μπορεί το αποτέλεσμά του να είναι αβέβαιο και ασταθές αν δεν έχει εξεταστεί πολλές φορές.

Μερικά βασικά πλεονεκτήματα αποτελούν:

- Μπορούμε να σταματήσουμε την διαδικασία όποτε θέλουμε (αν πχ υπάρχει χρονικός περιορισμός) και να μην χαθεί έργο. Σε αντίθετη με τους αλγορίθμους DFS μια πρόωρη διακοπή μπορεί να αναιρέσει τους υπολογισμούς για ένα ολόκληρο βάθος στην διαδικασία της επαναληπτικής εμβάθυνσης.
- Καθώς στο βήμα rollout υπολογίζουμε το τελικό score βάσει του πραγματικού αποτελέσματος του παιχνιδιού δεν απαιτείται κάποια συνάρτηση αξιολόγησης θέσης. (μπορεί να χρησιμοποιηθεί σε παιχνίδια που δεν υπάρχει εκτενής γνώση)
- Καθώς δεν ελέγχουμε ολόκληρο το δέντρο αλλά επιλεκτικά επισκεπτόμαστε κόμβους που φαίνονται να επιφέρουν καλά αποτελέσματα, έχουμε καλυτέρα αποτελέσματα σε παιχνίδια με μεγάλο δείκτη διακλάδωσης.

Ένα σημαντικό μειονέκτημα, είναι ο χειρισμός ειδικών καταστάσεων με παγίδες, όπου πρέπει να ακολουθηθεί μια συγκεκριμένη ακολουθία κινήσεων (Ενδεχομένως αυτός να ήταν ένας από τους λογούς που έχασε το alpha go το τελευταίο παιχνίδι εναντίον του Lee Sedol).

2.11.2 UCB / UCT

Όπως περιγράφηκε προηγουμένως ο UCB / UCT [23] (upper confidence bound) αποτελεί ένα tree policy. Προσπαθεί να ισορροπήσει την εξερεύνηση κόμβων που έχουν παρουσιάσει καλή συμπεριφορά στο παρελθόν χωρίς να παραμελεί την εξερεύνηση εναλλακτικών που δεν φαίνονται βραχυπρόθεσμα καλές. Ο τύπος που χρησιμοποιείται είναι ο ακόλουθος:

$$S_i = \frac{w_i}{n_i} + C\sqrt{\frac{ln(N_i)}{n_i}}$$

- Ο όρος w_i περιέχει το πλήθος νικών μετά την κίνηση i.
- Ο όρος n_i περιέχει το πλήθος των επισκέψεων μετρά την κίνηση i.
- Ο όρος N_i περιέχει το συνολικό πλήθος επισκέψεων μετρά την κίνηση i του πατέρα κόμβου του κόμβο στον οποίο αναφερόμαστε.
- Ο όρος c αποτελεί παράμετρο του αλγορίθμου και καθορίζει τον βαθμό εξερεύνησης. Συνηθώς παίρνει την τιμή $\sqrt{2}$.

Άρα το w_i/n_i περιγράφει το ποσοστό νικών των κόμβων μετά την κίνηση i. Υψηλές τιμές σημαίνει ότι μετά την κίνηση i έχουμε νικήσει πολλές φορές παιχνίδια και αξίζει να την ερευνήσουμε περαιτέρω ως πιθανή βέλτιστη κίνηση.

Όταν αυξάνεται το πλήθος των επισκέψεων για έναν κόμβο, η ποσότητα $1/n_i$ μικραίνει ενθαρρύνοντας την επίσκεψη κόμβων που έχουν δοκιμαστεί λιγότερες φορές.

2.11.3 MCTS με Minimax

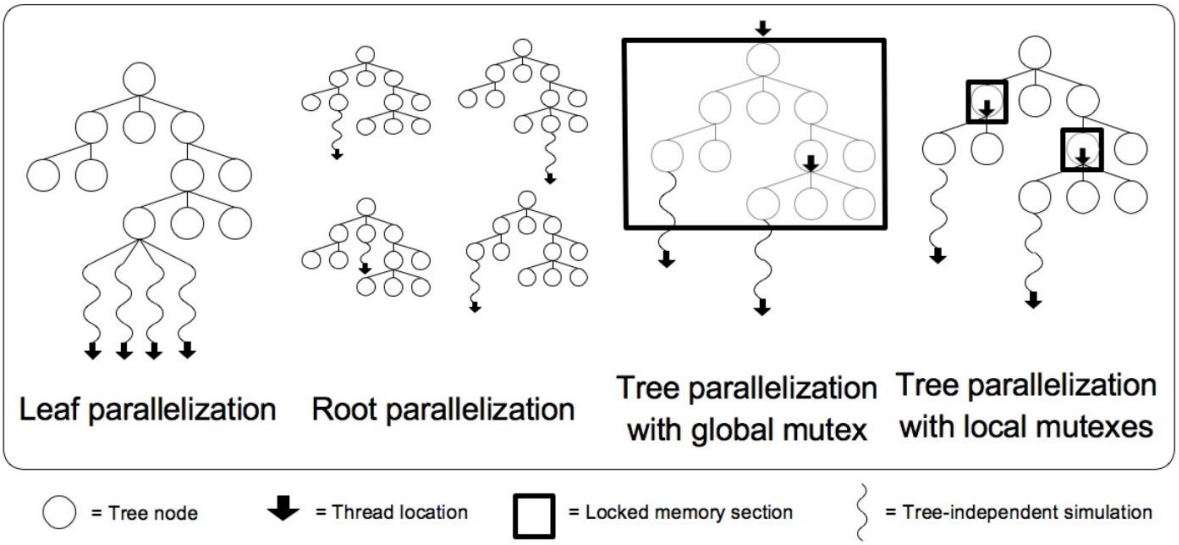
Ένας τρόπος να περιοριστεί η μυωπική συμπεριφορά του κλασικού MCTS είναι η υλοποίηση υβριδικών αλγορίθμων με χρήση παραλλαγών του minimax όπως αυτές που προτάθηκαν στην προηγούμενη ενότητα. Παρακάτω θα παρουσιαστούν τέσσερις διαφορετικές προσεγγίσεις [24] [25].

- **MCTS-IR**: (Informed rollouts) προσπαθεί να κάνει πιο εύστοχη την διαδικασία των rollouts. Κατά την διάρκεια ενός simulation αντί να διαλέγουμε κινήσεις από μια τυχαία κατανομή, κάνουμε μια αναζήτηση Minimax μέχρι ένα προκαθορισμένο βάθος (συνήθως ρηχό). Για να αποφύγουμε τον ντετερμινισμό οι κινήσεις εντός της αναζήτησης παράγονται με τυχαία σειρά. Επίσης έχουμε μια μικρή πιθανότητα ε να επιλέξουμε μια τυχαία κίνηση. Με αυτό τον τρόπο παράγουμε ένα πιο στοχευμένο δέντρο αλλά πληρώνουμε επιπλέον χρόνο για την εύρεση της επόμενης κίνησης σε κάθε rollout step.
- **MCTS-IC**: (Informed cut-offs) Σε κάθε rollout step μπορούμε να τρέξουμε μια αναζήτηση Minimax. Αν η αξιολόγηση είναι ισχυρή και εντός μια εμβέλειας τότε μπορούμε να θερίσουμε το τελικό αποτέλεσμα του παιχνιδιού χωρίς να παίξουμε όλες τις υπόλοιπες κινήσεις. Παίζοντας ένα σταθερό πλήθος rollouts m μπορούμε να εισάγουμε περισσότερη τυχαιότητα. Αν κατά μέσο όρο ο χρόνος υπολογισμού της αξιολόγησης είναι μικρότερος από αυτόν που θα χρειαζόταν για την ολοκλήρωση του υπόλοιπου rollout αυξάνουμε τον ρυθμό δειγματοληψίας.
- **MCTS-IP**: (Informed priors) Όταν παράγουμε έναν νέο κόμβο ή επισκεπτόμαστε έναν κόμβο n φορές, εισάγουμε την ευρετική αξιολόγηση στην εξίσωση του Policy. Πιο συγκεκριμένα πολλαπλασιάζουμε τα «virtual wins / losses» κατά μια παράμετρο w.

Έτσι ενθαρρύνουμε την ορθότερη εξερεύνηση του δέντρου. Όταν επισκεφθούν πολλές φορές οι κόμβοι, η απόδοση τής ευρετικής τιμής φθίνει.

- **AB rollouts:** Τα rollouts μπορούν να γίνουν είτε με τον κλασικό τρόπο είτε με την χρήση αναζήτησης AB. Κάθε κόμβος μπορεί να συνυπολογιστεί με χρήση και των δύο μεθόδων. Όταν μια επανάληψη στον αλγόριθμο επαναληπτικής εμβάθυνσης τελειώσει, προσθέτουμε στον κόμβο που οδήγησε στην βέλτιστη κίνηση έναν αριθμό από MCTS νίκες. Κάθε φορά χρησιμοποιούμε αποκλειστικά μία μέθοδο αλλά μπορούν να υπάρξουν πολλές παραλλαγές με μικτές λύσεις. Η επιλογή του είδους αλγορίθμου rollout μπορεί να γίνει με τυχαία πιθανότητα για κάθε επανάληψη του βασικού αλγορίθμου MCTS.

2.11.4 Παράλληλες μέθοδοι MCTS



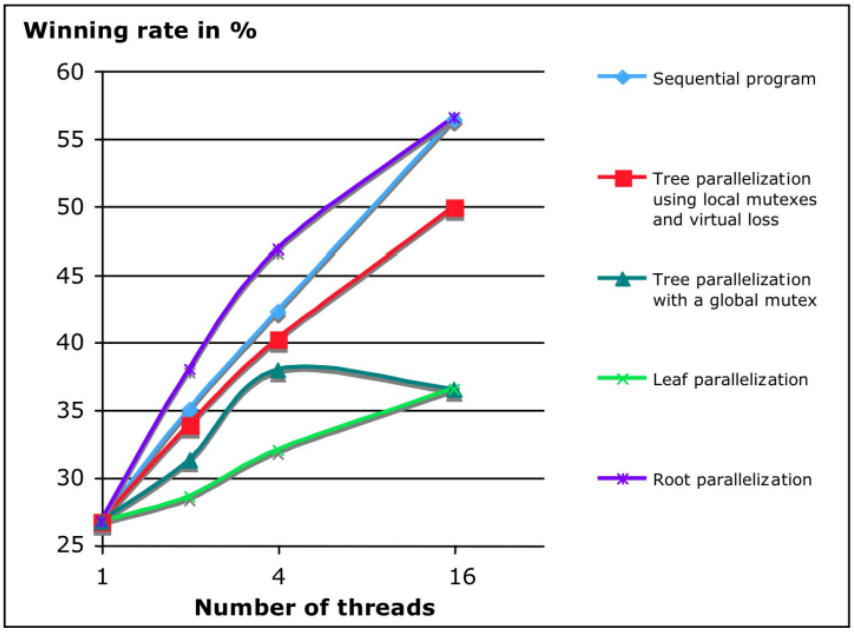
Εικόνα 37: Παράλληλοι αλγόριθμοι MCTS

Ο αλγόριθμος MCTS μπορεί να παραληλλοποιηθεί με τους ακόλουθους 3 τρόπους [26]

- **Παραλληλοποίηση φύλλων:** (Leaf parallelization) η πιο απλή τεχνική παραλληλοποίησης. Η διάσχιση του δέντρου γίνεται από το κύριο νήμα. Όταν φτάσουμε σε κόμβο φύλλο μετά το βήμα 1,2 ξεκινούμε n rollouts όπου n το πλήθος των ενεργων νημάτων. Καθώς κάθε rollout μπορεί να οδηγήσει σε τελείως διαφορετικές καταστάσεις με διαφορετικό μήκος παιχνιδιών, κάθε φορά πρέπει να περιμένουμε το μεγαλύτερο παιχνίδι να τελειώσει (αυτό αποτελεί σημαντικό Bottleneck στην διαδικασία). Επίσης τα νήματα δεν μοιράζονται μεταξύ τους πληροφορίες, άρα αν για μια κίνηση έχουμε πολλές ήττες σπαταλούμε άσκοπα τον χρόνο περιμένοντας για την ολοκλήρωση των υπόλοιπων rollouts. Στο τέλος όλων των παιχνιδιών το αποτέλεσμα διαδίδεται στο δέντρο από το κύριο νήμα.
- **Παραλληλοποίηση ρίζας:** (Root parallelization) όπως και προηγουμένως δεν έχουμε μεταφορά πληροφορίας μεταξύ των διαφορετικών νημάτων. Κάθε νήμα παράγει το δικό του MCTS δέντρο. Μετά την ολοκλήρωση του προκαθορισμένου αριθμού επαναλήψεων αναμιγνύουμε τα αποτελέσματα από κάθε ξεχωριστή αναζήτηση. Με αυτόν τον τρόπο όμως κάθε νήμα δεν εκμεταλλεύεται την γνώση που

έχουν αποκτήσει τα αλλά νήματα. Η βέλτιστη κίνηση επιλέγεται με βάση το αθροιστικό αποτέλεσμα όλων των δέντρων.

- **Παραλληλοποίηση δέντρου:** (Tree parallelization) χρησιμοποιείται ένα κοινό δέντρο μεταξύ πολλαπλών simulations. Τα περισσότερα νήματα ξεκινούν rollouts από διαφορετικά φύλλα. Μόνο 1 νήμα επιτρέπεται να αναζητήσει την δομή κάθε φορά για αποφυγή corruptions. Στην περίπτωση που εφαρμόσουμε πολλαπλά mutex (ένα για κάθε κόμβο, έναντι ενός καθολικού) μπορούμε να αυξήσουμε το πλήθος των νημάτων εντός του δέντρου με το να κλειδώνουμε κάθε κόμβο όταν αυτός επισκέπτεται από ένα νήμα. Για να αποφύγουμε κάθε νήμα να επιλέγει το ίδιο μονοπάτι με όλα τα υπόλοιπα, αναθέτουμε μια virtual loss τιμή σε κάθε κόμβο. Στο τέλος του rollout αφαιρείται κατά την διάρκεια του Back propagation.



Εικόνα 38: Απόδοση παράλληλων αλγορίθμων MCTS

Σε περιβάλλοντα περιορισμένου χρόνου, πιο γρήγοροι υπολογισμοί συντελούν σε περισσότερα rollouts άρα και εύρεση καλυτέρων κινήσεων.

Η καμπύλη sequential περιγραφεί τον ακολουθιακό αλγόριθμο αν του δώσουμε παραπάνω χρόνο αντί για παραπάνω υπολογιστική ισχύ, αποτελεί μετρό σύγκρισης για όλους τους υπολοίπους αλγορίθμους.

2.12 Προσαρμοσμένες συναρτήσεις αξιολόγησης θέσης

Όπως είδαμε προηγουμένως, οι αλγόριθμοι αναζήτησης τύπου DFS απαιτούν την αξιολόγηση της θέσης μετά από ένα προκαθορισμένο βάθος κινήσεων. Στην περίπτωση που βρισκόμαστε σε τερματικό κόμβο μπορούμε να αξιολογήσουμε με ακρίβεια το αποτέλεσμα μιας και είναι γνωστό, διαφορετικά πρέπει να χρησιμοποιήσουμε ευρετικές τεχνικές.

Γενικά ψάχνουμε μια συνάρτηση $Eval = \sum_{\mathrm{i}=1}^{\mathrm{n}} F_i * W_i$, όπου οι τιμές F_i αποτελούν τα χαρακτηριστικά του ταμπλό και οι W_i τα βάρη που καθορίζουν την σημαντικότητα των χαρακτηριστικών αυτών. Οι τιμές W_i αρχικοποιούνται εμπειρικά. Στόχος είναι η συνάρτηση μας να είναι όσο πιο κοντά γίνεται στο πραγματικό score, διαφορετικά μια λανθασμένη άποψη για την κατάσταση του παιχνιδιού θα οδηγήσει σε εύρεση λάθος βέλτιστων κινήσεων.

Παρακάτω ακολουθούν μερικές από τις πιο βασικές ιδέες:

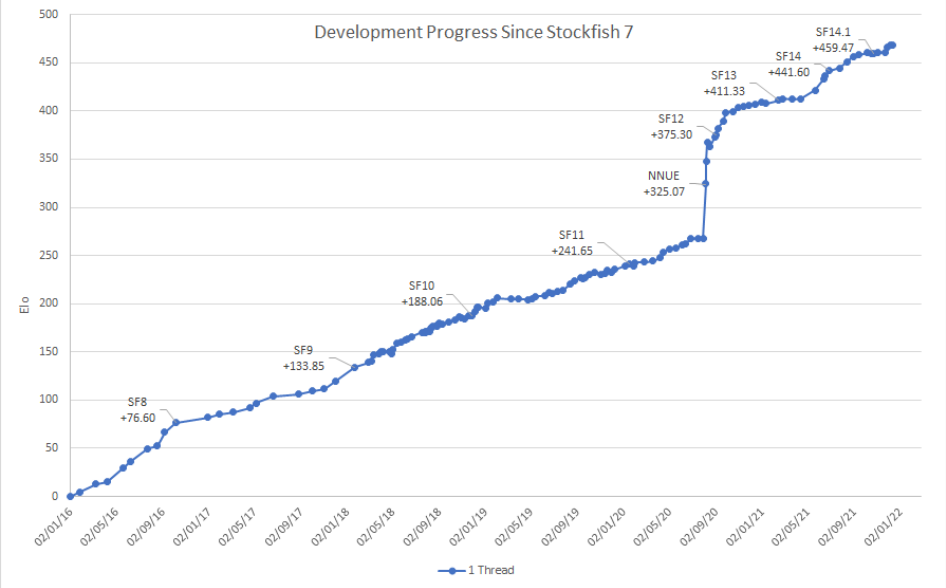
- **Αξία υλικού:** Θεωρούμε ότι κάθε πιόνι έχει μια στατική αξία. Αν για παράδειγμα έχουμε δύο πύργους και θεωρήσουμε ότι αξίζουν 5 πόντους ο καθένας η τελική αξιολόγηση θα είναι είναι $2 * 5 = 10$. Για να συνυπολογίσουμε και το score του αντιπάλου κάνουμε την ίδια διαδικασία αλλά αφαιρούμε το τελικό αποτέλεσμα από το score της δικιάς μας ομάδας. $Eval = our\_score - enemy\_score$. Διαισθητικά ο παίχτης με τα περισσότερα πιόνια είναι σε πιο ευνοϊκή θέση.
- **Κεντροποίηση υλικού:** Εμπειρικά γνωρίζουμε ότι το κέντρο της σκακιέρας έχει στρατηγική σημασία. Για κάθε είδος και χρώμα πιονιού παράγουμε έναν πίνακα που βαθμολογεί την κάθε θέση. Διαισθητικά ενθαρρύνουμε κινήσεις προς το κέντρο.
- **Δομή πιονιών:** Τα πιόνια αποτελούν το πιο αδύναμο υλικό αλλά παίζουν πολύ σημαντικό ρόλο στην διαμόρφωση του παιχνιδιού. Μας ενδιαφέρει ο έλεγχος σχηματισμών / μοτίβων. Μερικά παραδείγματα αποτελούν τα πιόνια που δεν μπορούν να αμυνθούν από άλλα πιόνια, πιόνια που βρίσκονται στην ίδια στήλη, πιόνια που δεν έχουν αντίπαλο πιόνι στην ίδια στήλη, πιόνια που δεν έχουν αλλά συμμαχικά πιόνια σε διπλανές στήλες.
- **Κινητικότητα:** Ο παίχτης με τις περισσότερες κινήσεις ενδεχομένως να βρίσκεται σε καλύτερη θέση. Άλλες υλοποιήσεις πέρα από την απλή καταμέτρηση, βαθμολογούν κάθε κίνηση διαφορετικά ανάλογα το είδος του πιονιού ή το είδος της κίνησης. Για παράδειγμα οι διαγώνιες βαθμολογούνται πιο υψηλά από τις κάθετες ή οριζόντιες.
- **Ασφάλεια βασιλιά:** Η ασφάλεια του βασιλιά παίζει σημαντικό ρόλο ώστε να αποφύγουμε καταστάσεις checkmate. Είναι από τις πιο δύσκολες διαδικασίες αξιολόγησης, συνηθώς ελέγχουμε την ύπαρξη πιονιών ιδίου χρώματος σε κοντινά τετράγωνα, πόσο κοντά είναι πιόνια αντίπαλου χρώματος, πόσα κοντινά τετράγωνα δέχονται επιθέσεις, η κινητικότητα του βασιλιά εφόσον ήταν βασίλισσα.

Παρατηρούμε ότι όλες οι παραπάνω τεχνικές απαιτούν καλή γνώση του παιχνιδιού και κατάλληλη προσαρμογή των βαρών w_i για να έχουμε καλά αποτελέσματα. Αυτό αποτελεί σημαντικό πρόβλημα καθώς τέτοιες ιδιότητες δεν είναι γνωστές για κάθε πρόβλημα που προσπαθούμε να λύσουμε. Επίσης η ανθρώπινη κατανόηση περιορίζει τις στρατηγικές που θα μπορούσε να ακολουθήσει ο αλγόριθμος μας. Σε επόμενες ενότητες θα μελετηθεί πώς το πρόβλημα αυτό μπορεί να λυθεί με χρήση νευρωνικών δικτύων.

2.13 NNUE

Όπως είδαμε προηγουμένως οι «Hand crafted» συναρτήσεις αξιολόγησης απαιτούν ειδικευμένη γνώση, κάτι που δεν είναι πάντα εφικτό. Επίσης η γραμμική σχέση των χαρακτηριστικών περιορίζει τον βαθμό εκφραστικότητας της συνάρτησης.

Το NNUE (efficiently updatable neural network) [27] [28] είναι ένα νευρωνικό δίκτυο που προσπαθεί να αντικαταστήσει τις απλοϊκές συναρτήσεις αξιολόγησης. Πρώτο εφαρμόστηκε στο παιχνίδι shogi αλλά η χρήση του μπορεί να γενικευτεί και για άλλα παιχνίδια. Η σημερινή έκδοση του stockfish 14 χρησιμοποιεί μια υβριδική υλοποίηση που αξιοποιεί είτε και τις δύο μεθόδους ταυτόχρονα (με την μορφή σταθμισμένου αθροίσματος) ή μια από αυτές ανάλογα την φάση του παιχνιδιού.



Εικόνα 39: Αύξηση elo λόγω NNUE

2.13.1 Διαφορές με άλλα μοντέλα

- Σε αντίθεση με άλλες αρχιτεκτονικές (όπως του Alpha zero) έχει λίγα επίπεδα (ρηχό μοντέλο). Καθώς αξιοποιείται σε αλγορίθμους αναζήτησης DFS είναι σημαντικό το inference time (ο χρόνος ενός forward pass στο μοντέλο) να είναι αρκετά μικρός. Πιο σύνθετα μοντέλα θα οδηγούσαν σε πολύ μικρό δείκτη Nps. Υπάρχει το trade off μεταξύ υψηλού nps και υψηλής ακρίβειας αξιολόγησης. Πιο συνθέτες αρχιτεκτονικές χρησιμοποιούνται σε παραλλαγές του MCTS όπου γίνεται πιο έξυπνα η επιλογή του επομένου κόμβου προς αναζήτηση. Το ιδανικό θα ήταν να μπορούμε να χρησιμοποιήσουμε σύνθετα μοντέλα για να μεγιστοποιήσουμε την ποιότητα της συνάρτηση μας χωρίς να έχουμε ισχυρή πτώση στην ταχύτητα. Στην πράξη όμως η πτώση του αριθμού κόμβων που μπορούν να εξερευνηθούν, είναι σημαντικά μικρότερος.
- Τα περισσότερα μοντέλα αξιοποιούν την GPU για γρήγορους παραλλήλους υπολογισμούς. Πριν μπορέσει όμως η GPU να παράξει έργο, απαιτείται η μεταφορά των δεδομένων στην αντίστοιχη vram. Ο χρόνος μεταφοράς έχει μεγάλη σημασία αν συνεισφέρει σημαντικά στον συνολικό χρόνο αξιολόγησης. Καθώς το μοντέλο NNUE

είναι αρκετά ρηχό, είναι προτιμότερο η διαδικασία αυτή να γίνει εξολοκλήρου στην cpu ώστε να απαλειφθεί η ανάγκη για μεταφορά σε $2^{η}$ μνήμη. (Μια καλή cpu ξεπερνάει τις επιδόσεις μιας καλής GPU στο συγκεκριμένο μοντέλο). Σε πιο σύνθετα μοντέλα CNNs παρατηρείται η αντίθετη συμπεριφορά. Καθώς τα βαθιά νευρωνικά δίκτυα έχουν πολύ συνθέτους υπολογισμούς, ο χρόνος μεταφοράς είναι συγκριτικά μικρότερος του συνολικού χρόνου αρά η χρήση της GPU προσφέρει σημαντικά οφέλη.

- Δεδομένου ότι οι κάρτες γραφικών είναι λιγότερο προσβάσιμες σε καθημερινούς χρήστες, η αποκλειστική χρήση της cpu καθιστά την σκακιστική μηχανή πιο εύκολη στην χρήση από περισσότερο κόσμο.
- Μεγάλα βαθιά νευρωνικό δίκτυα απαιτούν πάρα πολύ μεγάλη υπολογιστική ισχύ. Το μοντέλο NNUE μπορεί να εκπαιδευτεί από το μηδέν εντός βδομάδων λόγω του μικρού αριθμού hidden layers από μια μικρή ομάδα ανθρώπων. Συγκριτικά ο alpha zero χρειάστηκε πολλαπλές TPUs (tensor processing unit) για να ολοκληρώσει την διαδικασία εκπαίδευσης. Αντίστοιχα η υλοποίηση lc0 αξιοποιεί συνεισφορές χρηστών.

Πίνακας 7: Σύγκριση nps

Engine	Nps	Algorithm
DeepBlue	200.000.000	Alpha-beta
Stockfish 8	70.000.000	Alpha-beta
Alpha zero	80.000	MCTS

2.13.2 Βασικές ιδιότητες

Το NNUE προσπαθεί να αξιοποιήσει τις παρακάτω ιδιότητες:

- **Χαμηλό πλήθος μη μηδενικών στοιχείων**: τα χαρακτηριστικά εισόδου (input features) θέλουμε να είναι όσο πιο αραιά γίνεται. Καλές υλοποιήσεις παρουσιάζουν ποσοστά της τάξης του 0.1%. Μικρά ποσοστά φράζουν τον χρόνο ανα-υπολογισμού μιας θέσης από την αρχή.
- **Μικρές αλλαγές μεταξύ συνεχομένων εισόδων:** Τα χαρακτηριστικά εισόδου μεταξύ δύο συνεχόμενων θέσεων πρέπει να παρουσιάζουν μικρές διαφορές. Ως συνεχόμενες θέσεις εννοούμε δύο θέσεις που διαφέρουν μεταξύ τους κατά μία κίνηση. Με αυτόν τον τρόπο μπορούμε να αξιοποιήσουμε βηματικούς υπολογισμούς.
- **Κβαντισμένες τιμές:** οι τιμές που εξάγονται από τα ενδιάμεσα Hidden layers είναι κβαντισμένες ώστε να μπορούν να αξιοποιηθούν SIMD (single instruction multiple data) instructions για πιο γρήγορους παράλληλους υπολογισμούς εντός της CPU.
- **Δυαδικά δεδομένα:** οι τιμές των χαρακτηριστικών εισόδου αποτελούν δυαδικές τιμές. Με αυτόν τον τρόπο απλοποιούνται μερικοί υπολογισμοί.

2.13.3 HalfKP

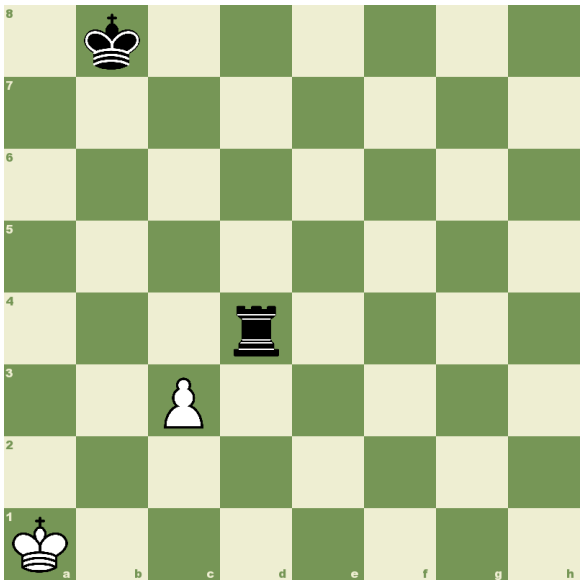
Το HalfKP είναι η πιο γνώστη μέθοδος αναπαράστασης της εισόδου στο NNUE. Περιγράφει την θέση του ταμπλό με επίκεντρο την θέση του βασιλιά για την κάθε ομάδα. Πιο συγκεκριμένα έχουμε την θέση του βασιλιά ακολουθούμενη από μια τριάδα χαρακτηριστικών. Κάθε τιμή είναι δυαδική, δηλαδή αναπαριστά την ύπαρξη ή όχι κάθε χαρακτηριστικού.

$$(own\_king\_position, own\_piece\_position, own\_piece\_type, piece\_color)$$

Αντίστοιχα επαναλαμβάνουμε την ίδια διαδικασία και για τα πιόνια της αντίπαλης ομάδας

$$(own\_king\_position, enemy\_piece\_position, enemy\_piece\_type, piece\_color)$$

Με παρόμοιο τρόπο υπολογίζουμε τα χαρακτηριστικά από την σκοπιά του αντίπαλου βασιλιά. Με τον τρόπο αυτό η αναπαράσταση είναι πάντα από την μεριά των άσπρων και για τα 2 χρώματα όπως και στην μονοχρωματική αναπαράσταση. (Στα enemy_piece δεν συμπεριλαμβάνουμε τους βασιλιάδες). Διαισθητικά προσπαθούμε να μοντελοποιήσουμε την σχέση του βασιλιά με τα υπόλοιπα πιόνια.



Εικόνα 40: Παράδειγμα halfKP

$$(a1, c3, pawn, white), (a1, d4, rook, black)$$
$$(b1, c6, pawn, black), (b1, d5, rook, white)$$

Παρατηρούμε ότι οι πιθανοί συνδυασμοί για κάθε χρώμα είναι $64 * 5 * 2 * 64 = 40960$ (στην πράξη η 3η τιμή έχει 63 πιθανές θέσεις καθώς μια από αυτές είναι πιασμένη από τον βασιλιά, η λεπτομέρεια αυτή αγνοείται για ευκολότερη υλοποίηση). Καθώς έχουμε δύο ομάδες καταλήγουμε σε $2 * 40960 = 81920$ συνδυασμούς.

Μια κανονική θέση περιέχει λιγότερο από 32 πιόνια άρα η αναπαράσταση μας είναι αρκετά αραιή. Τα μοντέλα έχουν καλύτερη γενίκευση και ικανότητα μάθησης όταν περιέχουν πολλαπλές παραμέτρους, συνήθως η υπερ-παραμετροποίηση γίνεται σε επίπεδο hidden layers. Για λόγους βελτιστοποίησης η συγκεκριμένη αρχιτεκτονική το κάνει μέσω υψηλών διαστάσεων χαρακτηριστικών εισόδου.

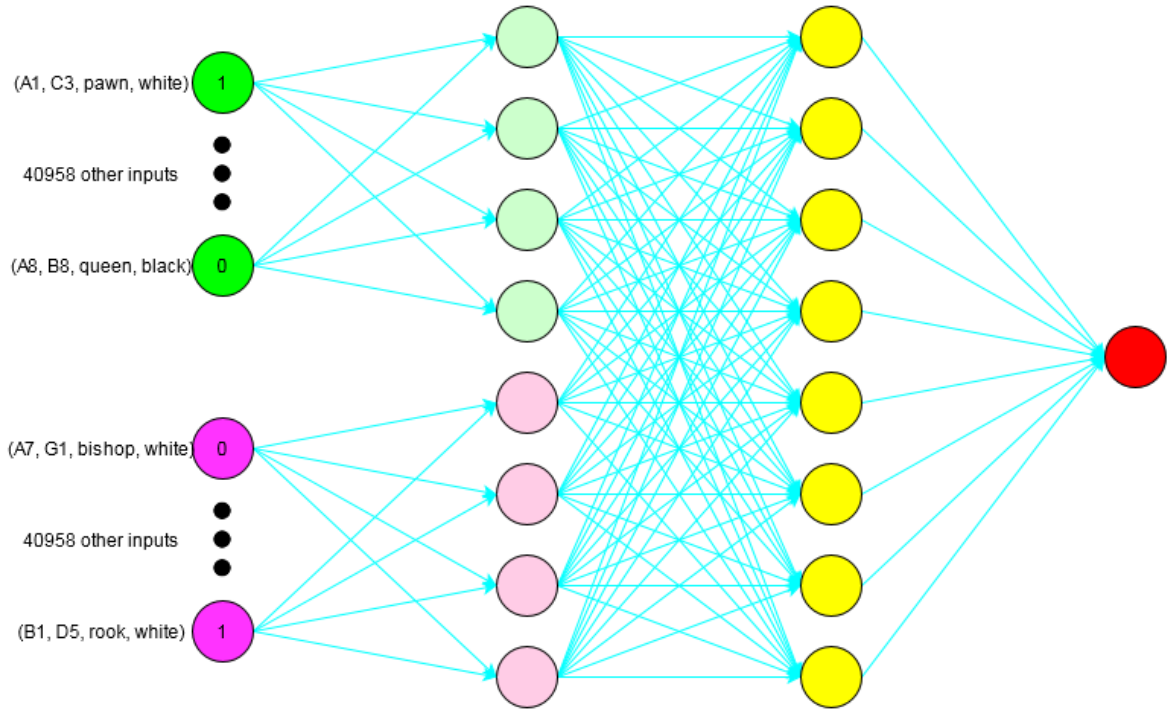
Αφού παράξουμε τα διανύσματα A_w, A_b (όπου A_x το διάνυσμα για ένα μεμονωμένο χρωμα) υπάρχουν 3 τρόποι με τους οποίους μπορούμε να συνθεσουμε το τελικο input.

- Να βάλουμε το A_w και μετά το A_b.
- Να βάλουμε το A_w πρώτο αν είναι η σειρά των άσπρων, αλλιώς το A_b αν είναι η σειρά των μαύρων.
- Μια από τις παραπάνω λύσεις, άλλα αντί να βάλουμε τα διανύσματα το ένα μετά το άλλο, να μπαίνει κάθε τετράδα εναλλάξ με αυτή του άλλου χρώματος.

Τα A_w, A_b μοιράζονται τα ίδια βάρη στο 1ο layer του δικτύου. Διαισθητικά θέλουμε το μοντέλο μας να μαθαίνει ανεξαρτήτως χρώματος παίχτη. Πάραυτα μερικές υλοποιήσεις κάνουν αυτόν τον διαχωρισμό δεδομένου ότι τα άσπρα παίζουν διαφορετικά από τα μαύρα λόγω της 1ης κίνησης.

2.13.4 Αρχιτεκτονική

Ο αριθμός των νευρώνων ανά Hidden layer μπορεί να διαφέρει από υλοποίηση σε υλοποίηση. Η βασική αρχιτεκτονική που προτείνεται για βέλτιστο performance / accuracy είναι 2 input vectors των 40960 χαρακτηριστικών, ενδιάμεσα layers των [256,32,32] νευρώνων. Τελική έξοδος σε 1 νευρώνα (τιμή αξιολόγησης). Ενδιάμεσες συναρτήσεις ενεργοποίησης Relu ή clipped Relu ανάλογα αν χρησιμοποιείται η τεχνική της κβάντισης.



Εικόνα 41: Αρχιτεκτονική NNUE

Η παραπάνω εικόνα δεν παρουσιάζει όλα τα ενδιάμεσα Hidden layers για λόγους απλότητας.

Γενικά η αρχιτεκτονική έχει την μορφή

1. L_0: $Linear\ N -> M$
2. C_0: $Clipped\ ReLu\ of\ size\ M * 2$
3. L_1: $Linear\ M * 2 -> K$
4. C_1: $Clipped\ ReLu\ of\ size\ K$
5. L_2: $Linear\ K -> 1$

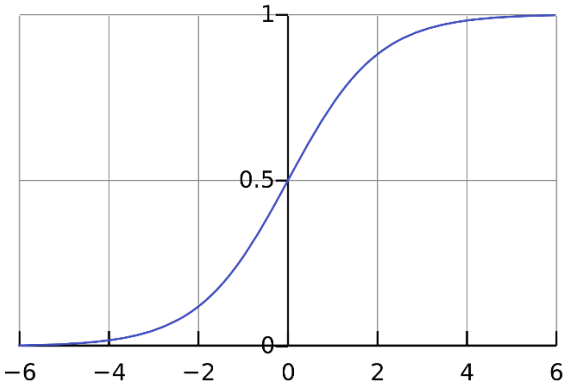
Όπου N το πλήθος των χαρακτηριστικών εισόδου.

2.13.5 Εκπαίδευση μοντέλου

Η εκπαίδευση του μοντέλου απαιτεί πολλαπλά ζεύγη θέσεων και των αντίστοιχων scores αξιολόγησης τους. Μπορούμε να ακολουθήσουμε δύο τεχνικές:

- Να πάρουμε τα ήδη υπάρχον παιχνίδια και να αξιολογήσουμε κάθε θέση με χρήση συναρτήσεων αξιολόγησης άλλων σκακιστικών μηχανών.
- Να αξιολογήσουμε κάθε θέση με χρήση του NNUE με αναζήτηση μεγαλύτερου βάθους. Διαισθητικά το μοντέλο μας εξετάζει μεμονωμένα την θέση με βάθος 0. Όσο μεγαλύτερο το βάθος της αναζήτησης τόσο καλύτερη η προσέγγιση.

Το τελικό αποτέλεσμα μεταφέρεται σε centi pawns (δηλαδή τιμές 0,0.5,1) με χρήση σιγμοειδούς συνάρτησης για αποφυγή του φαινομένου των exploding gradients.



Εικόνα 42: Σιγμοειδής συνάρτηση

Το stockfish στην πράξη αξιοποιεί παιχνίδια της σκακιστικής μηχανής lc0 που έχουν παραχθεί μέσω self-play. Παρόλου που το μοντέλο υλοποιείται εξ ολοκλήρου εντός του engine, η εκπαίδευση γίνεται με χρήση Pytorch όπου προτιμάται η χρήση της GPU.

Για να μπορεί να εκπαιδευτεί το μοντέλο χρειαζόμαστε και μια συνάρτηση σφάλματος. Οι πιο συνήθεις επιλογές είναι η MSE (mean squared error) και η $cross\ entropy\ loss$. Διαισθητικά το πρόβλημα μας μπορεί να θεωρηθεί πρόβλημα κατηγοριοποίησης παιχνιδιού σε τρείς πιθανές κλάσεις $\{win, draw, defeat\}$.

Η ελαχιστοποίηση του σφάλματος γίνεται με κλασικές τεχνικές gradient descent.

2.13.6 Βηματικές ενημερώσεις

Η δομή των χαρακτηριστικών εισόδου, μας επιτρέπει να ενημερώσουμε βηματικά τον υπολογισμό του 1ου γραμμικού layer μεταξύ δύο συνεχόμενων θέσεων (η ιδέα θυμίζει τα κλειδιά Zobrist). Υπενθυμίζουμε ότι τα υπόλοιπα Layers είναι μικρότερου μεγέθους άρα η βελτιστοποίηση έχει πρακτικό νόημα.

Οι μόνες φορές που ο επαναϋπολογισμός είναι αναγκαίος είναι αν:

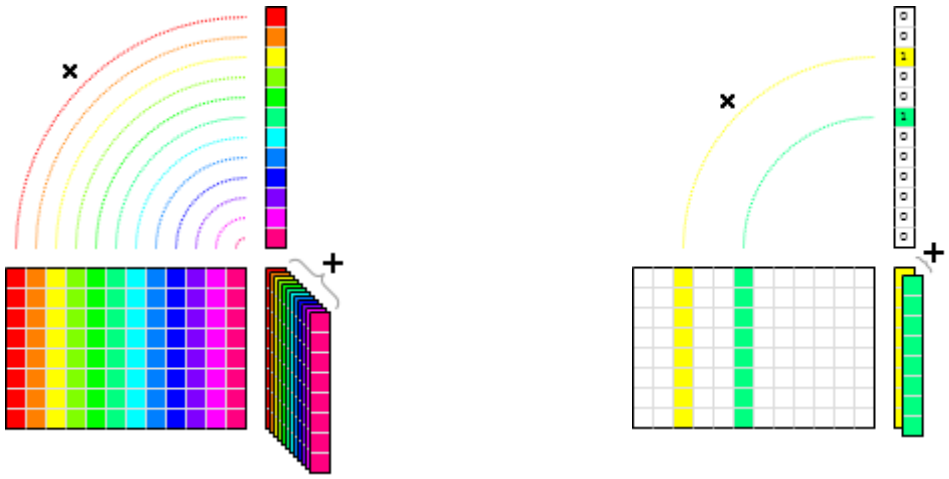
- Δεν υπάρχει προηγούμενη θέση
- Κουνήθηκε ένας από τους δύο βασιλιάδες. Καθώς η αναπαράσταση περιγράφει την συσχέτιση των πιονιών με την θέση του βασιλιά, πρέπει να αλλάξουμε κάθε τετράδα. Ευτυχώς για το μεγαλύτερο κομμάτι ενός παιχνιδιού οι βασιλιάδες παραμένουν στάσιμοι.

Για να γίνει κατανοητή η διαδικασία, πρέπει πρώτα να εστιάσουμε στον τρόπο με τον οποίο υπολογίζεται ένα γραμμικό Layer.

Έστω η έκφραση $y = Ax + b$, όπου x διάνυσμα διάστασης $in\_features$, A ο πίνακας βαρών διάστασης $(out\_features, infeatures)$, b το διάνυσμα «bias» διάστασης $out\_features$ και y το διάνυσμα αποτελέσματος διάστασης $out\_features$.

Καθώς οι ποσότητες A, x δεν είναι βαθμωτές, η πράξη Ax αποτελεί γινόμενο πινάκων.

Πιο συγκεκριμένα, όταν η τιμή $x[i]$ είναι μη μηδενική, αρκεί να πάρουμε την i-οστη στήλη του A να την πολλαπλασιάσουμε με το $x[i]$ και να την αθροίσουμε στο τελικό αποτέλεσμα.



Εικόνα 43: Γραμμικό Layer

Στην περίπτωση του 1ου layer, καθώς τα χαρακτηριστικά εισόδου περιέχουν μόνο δυαδικά στοιχεία, αρκεί να αθροίσουμε τις στήλες παραλείποντας τον πολλαπλασιασμός με την ποσότητα $x[i]$. Ως εκ τούτου, μπορούμε να εστιάσουμε στον υπολογισμό μόνο των μη μηδενικών στοιχείων. Καθώς τα χαρακτηριστικά εισόδου είναι αραιά, γλιτώνουμε την πλειοψηφία των πράξεων.

Με την ίδια λογική μπορούμε να υπολογίσουμε το αποτέλεσμα του 1ου linear layer άμα ξέρουμε την προηγουμένη τιμή y και την τελευταία κίνηση που παίχτηκε. Στην χείριστη περίπτωση θα έχουμε τρεις ταυτόχρονες αλλαγές (διαγραφή από αρχική θέση, εμφάνιση σε νέα θέση, αιχμαλωσία). Η τιμή y περιέχει υπολογισμούς από δείκτες εισόδου που έχουν διαγραφεί, για αυτό και στην νέα τιμή πρέπει να αφαιρέσουμε τις διαγραμμένες

στήλες. Αντίστοιχα η τιμή y δεν περιέχει υπολογισμούς από δείκτες που έχουν προστεθεί, για αυτό και στην νέα τιμή πρέπει να προσθέσουμε τις καινούργιες στήλες.

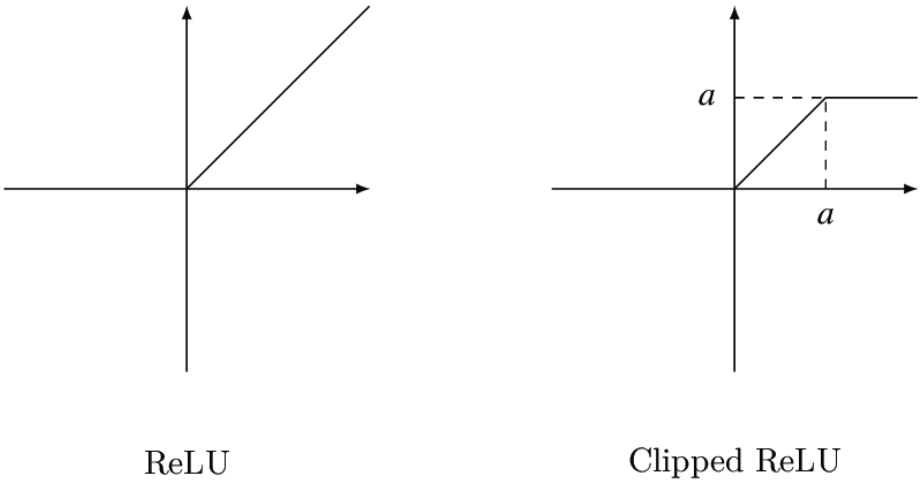
Καθώς οι διαφορές μεταξύ δύο συνεχόμενων θέσεων είναι αρκετά μικρές, οι παραπάνω υπολογισμοί είναι φθηνοί.

Ο πίνακας A στην πραξη αποθηκεύεται «column major» για γρηγορότερη φόρτωση από την cache.

2.13.7 Κβάντιση και SIMD

Το βήμα της κβάντισης υλοποιείται καθαρά για λόγους ταχύτητας και μόνο κατά το Inference (όχι στην εκπαίδευση). Καθώς οι αριθμοί κινητής υποδιαστολής απαιτούν παραπάνω κύκλους ρολογιού ανά πράξη, είναι προτιμότερο να μεταφέρουμε τον χώρο τιμών από τους δεκαδικούς στους ακέραιους. Οι περισσότερες αρχιτεκτονικές προσφέρουν εντολές SIMD (AVX2, SSE) όπου μπορούν να κάνουν παράλληλους υπολογισμούς σε 8,16,32 ή ακόμα και 64 ακέραιους των 8 bits παράλληλα. Ως εκ τούτου θα περιορίσουμε τις μεταβλητές των βαρών, εισόδων σε 8 bits ή 16 όπου χρειάζεται. Καθώς με 8 bits μπορούμε να αποτυπώσουμε τους αριθμούς $\{-128, \dots, 127\}$ η διαδικασία αυτή παράγει σφάλμα ακρίβειας.

Η συνάρτηση ενεργοποίησης (activation function) clipped ReLu συνήθως χρησιμοποιείται για αποφυγή προβλημάτων λόγω υψηλών παραγώγων.



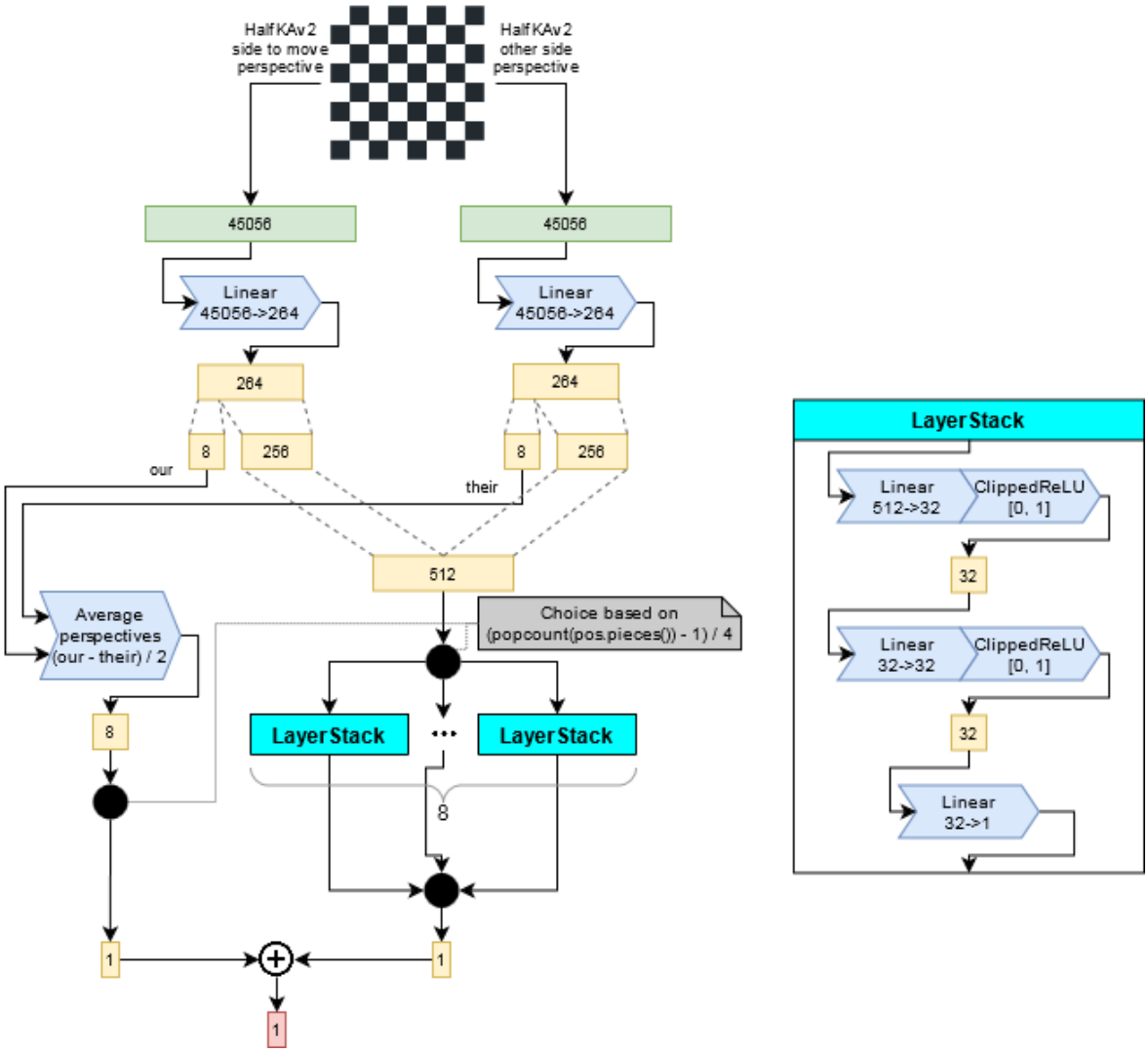
Εικόνα 44: Διαφορά ReLu, Clipped ReLu

Στην δικιά μας περίπτωση θέλουμε να φράξουμε τις ποσότητες που εισέρχονται στο επόμενο layer ώστε να μπορούν να αποθηκευτούν στις 8bit μεταβλητές. Η μεταβλητή α ισοδυναμει με τον αριθμο 127.

Αντίστοιχα όταν πολλαπλασιάζουμε με τις τιμές των βαρών / biases πρέπει να είμαστε προσεκτικοί να περιορίσουμε το εύρος τιμών. Η διαδικασία αυτή προσθέτει παραπάνω σφάλμα αλλά όχι αρκετό ώστε να επηρεάσει δραστικά το αποτέλεσμα της πρόβλεψης.

2.13.8 Βελτιώσεις stockfish

Το stockfish το 2021 εφάρμοσε μερικές αλλαγές στην βασική αρχιτεκτονική του μοντέλου. Δημιούργησε μια παραλλαγή του HalfKP ονόματι HalfKAv2 [29]. Επειδή τα μοντέλα δυσκολεύονται να εκτιμήσουν θέσεις με υψηλή αστάθεια πιονιών (material imbalance) ή γενικότερα να κάνουν υψηλές προβλέψεις, εντάχθηκε μια ενδιάμεση τιμή για κάθε layer των 2 κεφαλών όπου σηματοδοτεί την τιμή PSQT (ως PSQT εννοούμε την στατική αξιολογική θέση πιονιών που περιγράφηκε στην ενότητα 2.12.) Η μέση τιμή των δύο αυτών τιμών τροφοδοτείται απευθείας στο αποτέλεσμα και δεν περνάει από τα υπόλοιπα layers. Για να έχουμε διαφορετική συμπεριφορά του μοντέλου ανάλογα των αριθμών των διαθέσιμων πιονιών εντάχθηκαν 8 διαφορετικά υπο-δίκτυα. Η επιλογή του δικτύου που εκτελείται γίνεται δυναμικά κάθε φορά βάσει της ποσότητας $(αριθμός\_πιονιών - 1)/4$. Στην πράξη καθώς η εκπαίδευση ενός μοντέλου με δυναμικές επιλογές δεν μπορεί να παραληλλοποιηθεί στην GPU, μπορούμε απλά να τα υπολογίσουμε όλα και να διαλέξουμε τα αποτελέσματα που μας ενδιαφέρουν, δεδομένου ότι κάθε δίκτυου είναι αρκετά μικρό (3 layers).



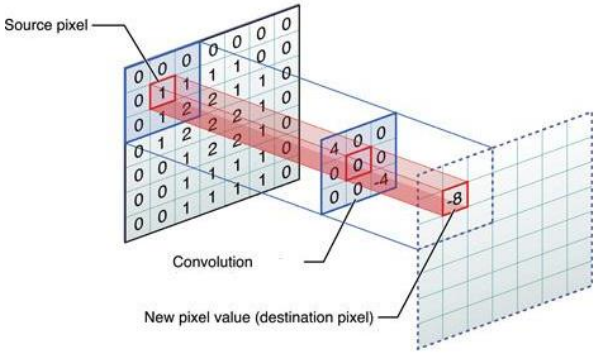
Εικόνα 45: Stockfish NNUE

2.14 Μοντέλα μηχανικής όρασης

Σε αυτή την ενότητα θα περιγράφουν πολύ συνοπτικά τα μοντέλα CNNs και οι παραλλαγές τους ResNets. Οι αρχιτεκτονικές αυτές προέρχονται κυρίως από τον τομέα της μηχανικής όρασης αλλά μπορούν να χρησιμοποιηθούν και σε αλλά προβλήματα όπου μας ενδιαφέρει η χωρική συσχετική των χαρακτηριστικών εισόδου .

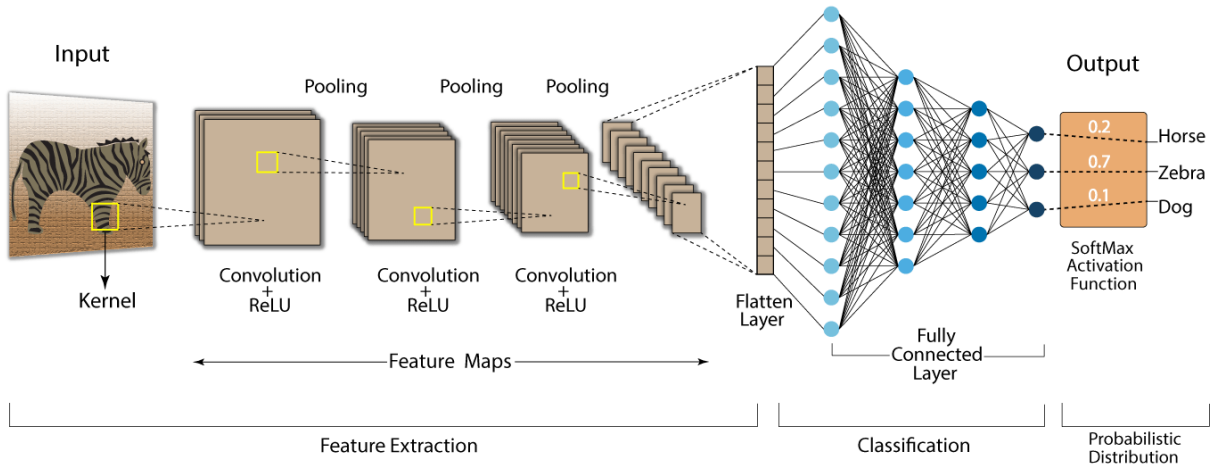
2.14.1 CNNs

Το κύριο πρόβλημα των απλών feed forward NN είναι η έλλειψη συσχετίσεων μεταξύ γειτονικών χαρακτηριστικών. Για παράδειγμα μια γραμμή σε μια φωτογραφία αποτελεί σύνολο πολλαπλών εικονοστοιχείων. Η βασική ιδέα των CNNs (convolutional neural networks) είναι ότι μπορούμε να χρησιμοποιούμε φίλτρα συνέλιξης για να εξάγουμε τα βασικά χαρακτηριστικά εισόδου.



Εικόνα 46: Φίλτρο συνέλιξης

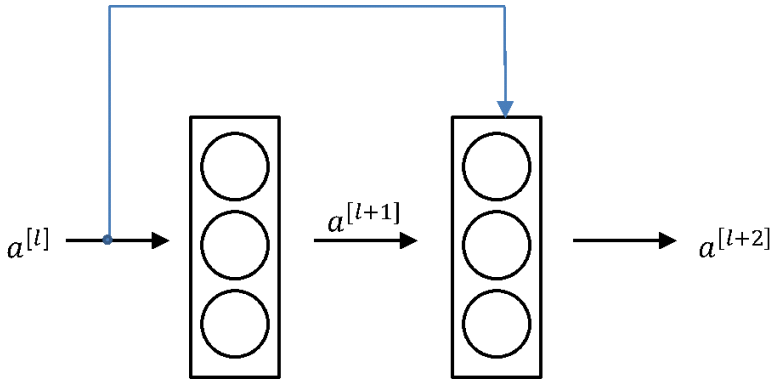
Οι παράμετροι των φίλτρων εκπαιδεύονται κατά την διάρκεια της μάθησης του μοντέλου. Η είσοδος αποτελείται από πολλαπλά επίπεδα που σχηματίζουν έναν όγκο (για παράδειγμα στις έγχρωμες εικόνες έχουμε ένα επίπεδο ανά κανάλι χρώματος rgb). Σε ενδιάμεσα Layers του CNN έχουμε pooling layers που προσπαθούν να δώσουν έμφαση στις πιο σημαντικές τιμές. Αφού γίνει η εξαγωγή χαρακτηριστικών μπορούμε να περάσουμε το αποτέλεσμα σε ένα κανονικό feed forward δίκτυο για την επιλογή ενός προβλήματος κατηγοριοποίησης ή παλινδρόμησης.



Εικόνα 47: Αρχιτεκτονική CNN

2.14.2 ResNets

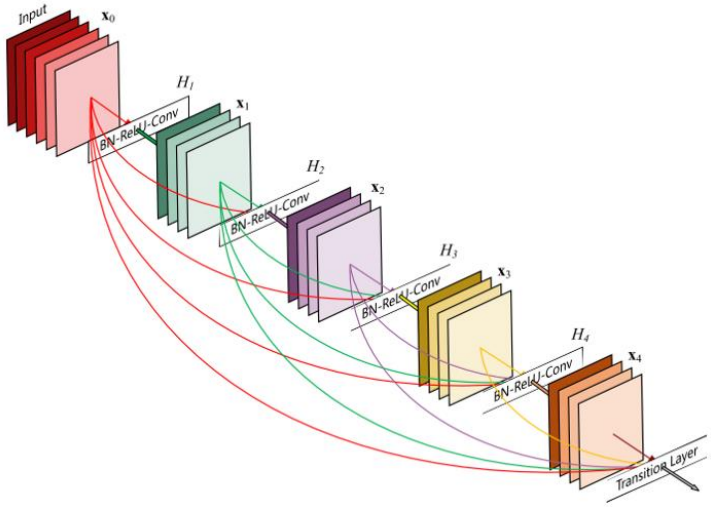
Ένα μεγάλο πρόβλημα των βαθιών νευρωνικών δικτύων είναι το φαινόμενο του vanishing gradient. Καθώς οι μερικές παράγωγοι γίνονται όλο και πιο μικρές, τα βάρη των πιο βαθιών hidden layers μεταβάλλονται πιο αργά. Ως εκ τούτου, μετά από ένα σημείο το μοντέλο σταματάει να εκπαιδεύεται. Για να λύσουμε το πρόβλημα αυτό μπορούμε να εισάγουμε συνδέσεις μεταξύ μελλοντικών layers (skip connections).



Εικόνα 48: Residual block

Ως residual block ορίζουμε ένα σύνολο συνεχόμενων layers όπου η αρχή του 1ου layer συνδέεται με την είσοδο της συνάρτησης ενεργοποίησης του τελευταίου.

Στην πιο εξειδικευμένη περίπτωση μπορούμε να προσθέσουμε μια σύνδεση για κάθε επόμενο layer. Τότε το μοντέλο μας ονομάζεται dense ResNet.



Εικόνα 49: Dense ResNet

2.15 AlphaGo και παραλλαγές του

Ο AlphaGo ήταν το πρώτο AI που κατάφερε να φτάσει super human performance στο παιχνίδι Go. Σε αυτή την ενότητα θα μελετηθεί συνοπτικά ο τρόπος λειτουργίας του και οι διαφορές του με τις βελτιώσεις AlphaGo zero, Alpha zero, MuZero [30] [31] [32] [33].

2.15.1 Δίκτυα

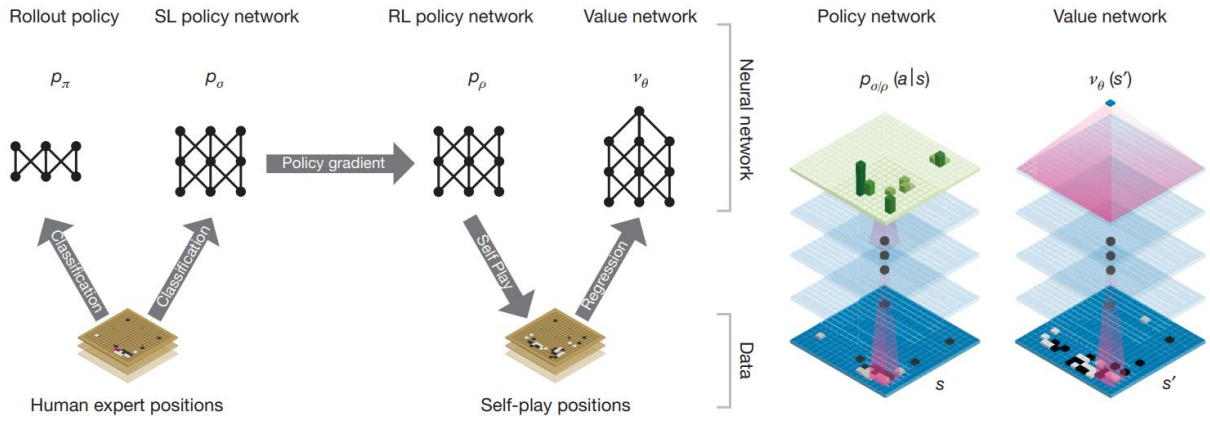
Ο AlphaGo αποτελείται από τέσσερα βασικά δίκτυα CNN.

Έχουμε το SL policy network p_σ όπου εκπαιδεύτηκε με επιβλεπόμενη μάθηση από 30 εκατομμύρια θέσεις πραγματικών παιχνιδιών. Ο στόχος του δικτύου είναι η πρόβλεψη μιας κατανομής πιθανοτήτων $P(a|s)$ για κάθε κίνηση a στην δοθείσα θέση s. Θεωρούμε κάθε κλάση την κίνηση που παίχτηκε στο κανονικό παιχνίδι. Η αντίστοιχη συνάρτηση σφάλματος που ελαχιστοποιούμε είναι η cross entropy loss όπως και στα περισσότερα προβλήματα κατηγοριοποίησης. Χρησιμοποιείται ένα softmax layer για μετατροπή των τιμών σε πιθανότητες.

Με ακριβώς παρόμοια λογική εκπαιδεύτηκε ένα μικρότερο γραμμικό δίκτυο Roll out policy p_π με πολύ λιγότερα data. Η κύρια διαφορά μεταξύ των δύο δικτύων είναι η ταχύτητα εκτέλεσης και η ποιότητα του αποτελέσματος. Το Rollout policy έχει πολύ χαμηλή ακρίβεια ($\sim 24\%$) αλλά και πολύ μικρό inference ($2\mu s$), καθιστώντας το κατάλληλο για χρήση εντός της φάσης rollout στον MCTS.

Ένα τρίτο δίκτυο RL policy network p_p αρχικοποιείται βάση το RL p_σ. Υστέρα εκπαιδεύεται περαιτέρω μέσω παιχνιδιών self-play. Οι κινήσεις του κάθε παιχνιδιού βαθμολογούνται θετικά ή αρνητικά βάσει του τελικού αποτελέσματος.

Τέλος με χρήση του RL policy network εκπαιδεύουμε ένα Value network v_θ όπου ο κυρίως στόχος του είναι η αξιολόγηση θέσης (δηλαδή η εκτίμηση του τελικού αποτελέσματος του παιχνιδιού). Αυτή την φορά προσπαθούμε να ελαχιστοποιήσουμε το MSE loss και η εκπαίδευση γίνεται με τυχαίες θέσεις από τα παιχνίδια που παράχθηκαν κατά το self-play. Χρησιμοποιείται μία θέση ανά παιχνίδι για αποφυγή του φαινομένου overfitting.



Εικόνα 50: Δίκτυα AlphaGo

2.15.2 Χαρακτηριστικά εισόδου

Οι είσοδοι που περνούνται στα CNNs είναι πολλαπλά δισδιάστατα επίπεδα. Θεωρούμε το $19x19$ ταμπλό μια $19x19$ εικόνα. Κάθε τιμή είναι δυαδική άρα επιπρόσθετη πληροφορία πρέπει να συμπεριληφθεί σε διαφορετικό επίπεδο κάθε φορά. Ο AlphaGo πέρα από την αναπαράσταση θέσης είχε και επιπλέον hand crafted χαρακτηριστικά στην είσοδο.

Τα χαρακτηριστικά αυτά δεν θα αναλυθούν περαιτέρω γιατί αποτελούν domain specific γνώση για το παιχνίδι go. Στην συνέχεια θα δούμε ότι η αφαίρεση τους στις επόμενες παραλλαγές του μοντέλου προσέφερε καλύτερα αποτελέσματα.

Τα πολλαπλά επίπεδα ακολουθούν την ίδια συλλογιστική των καναλιών rgb όπως περιγράφηκε στην ενότητα 2.14.1. Συνολικά έχουμε έναν όγκο $19x19x48$. Καθώς το παιχνίδι go παρουσιάζει συμμετρίες, εφαρμόζοντας κατάλληλες περιστροφές στα ήδη υπάρχοντα χαρακτηριστικά μπορούμε να επαυξάνουμε το σύνολο δεδομένων μας (data augmentation).

2.15.3 MCTS και PUCT

Ο συνάρτηση PUCT αποτελεί μια παραλλαγή της συνάρτησης UCB/UCT στον αλγόριθμο MCTS. Στην φάση selection επιλέγουμε την κίνηση a με βάση το παρακάτω κριτήριο.

$$a_t = argmax_a\big(Q(s_t,a) + U(s_t,a)\big)$$

$$Q(s,a) = (1-\lambda)\frac{W_v(s,a)}{N(s,a)} + \lambda\frac{W_r(s,a)}{N(s,a)}, U(s,a) = c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

- Η τιμή $P(s,a)$ αποτελεί την prior πιθανότητα και υπολογίζεται από το SL policy network. (δεν χρησιμοποιήθηκε το RL policy γιατί οδηγούσε σε χειρότερα αποτελέσματα)
- Η τιμή $N(s,a)$ περιέχει το πλήθος των επισκέψεων ενός κόμβου μετά την κίνηση a στην κατάσταση s.
- Η τιμή W_v περιέχει το πλήθος νικών βάσει του Value network ενώ η W_r βάσει των rollouts.
- Οι τιμές λ, C_{puct} αποτελούν υπερπαραμέτρους (στην τελική υλοποίηση ήταν 0.5 και 5 αντίστοιχα).

Στην πράξη η ποσότητα Q περιγράφει ένα μέσο όρο μεταξύ των ποσοστών νικών που υπολογίστηκαν από τα κανονικά MCTS rollouts και αυτών που υπολογίστηκαν μέσω του δικτύου αξιολόγησης. Η ποσότητα U επηρεάζεται από prior πιθανότητες ώστε η εξερεύνηση του δέντρου να είναι στραμμένη ως προς κινήσεις που προτείνονται από το SL policy network έναντι μιας τυχαίας κατανομής. Όσο περισσότερες επισκέψεις έχουν οι υπόλοιποι κόμβοι του πατέρα κόμβου (μεγάλη ποσότητα $\sum_b N(s,b)$) τόσο πιο πολύ αυξάνεται η πιθανότητα να επισκεφθούμε τον κόμβο αυτό (μικρή ποσότητα $1+N(s,a)$).

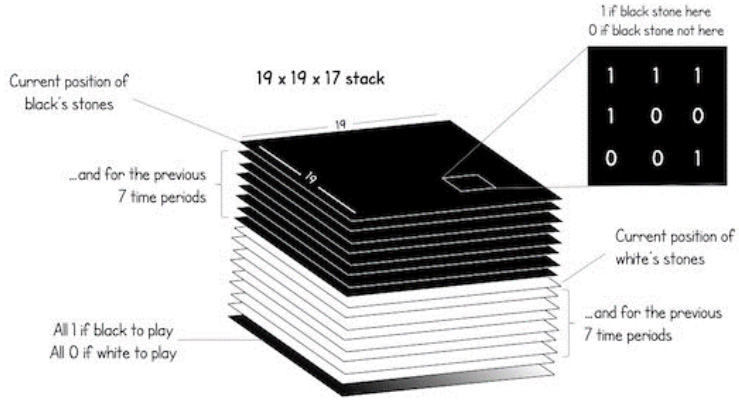
Στην ουσία οι ποσότητες Q+U εκφράζουν το exploitation-exploration trade off όπως περιγράφηκε και στον απλό UCT με την επιπλέον εισαγωγή νευρωνικών δικτύων.

Όσο αφορά την εκτέλεση του αλγορίθμου MCTS έχουμε τις εξής επεκτάσεις. Όταν επεκτείνεται ένας κόμβος, για κάθε νέα κατάσταση υπολογίζονται οι τιμές W_v και $P(s,a)$ με χρήση των δικτύων όπως περιγράφηκε. Στην διαδικασία rollout αξιοποιείται το δίκτυο Rollout policy. Για λόγους παραλληλίας χρησιμοποιείται η τεχνική virtual loss.

2.15.4 AlphaGo Zero

Ο AlphaGo Zero αποτελεί συνέχεια του AlphaGo. Το μεγαλύτερο κομμάτι της διαδικασία απλοποιήθηκε επιφέροντας καλυτέρα αποτελέσματα.

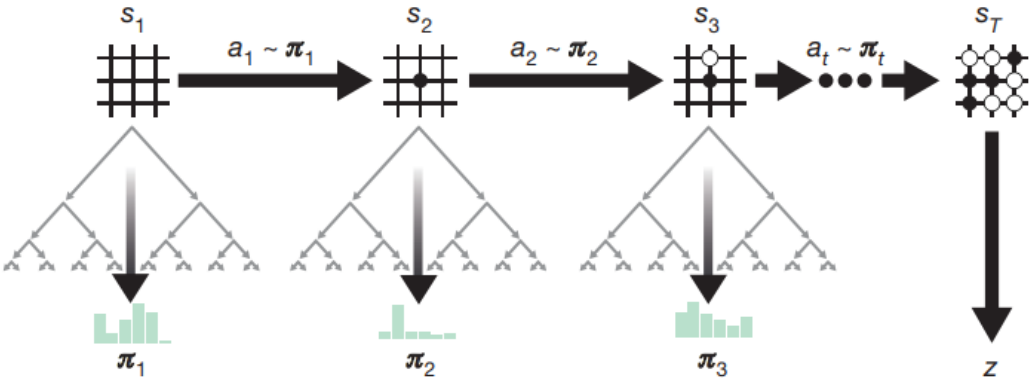
Πιο συγκεκριμένα στα χαρακτηριστικά εισόδου αφαιρέθηκαν τα «hand crafted» επίπεδα και προστέθηκαν αναπαραστάσεις από 7 προηγούμενες θέσεις.



Εικόνα 51: AlphaGo Zero χαρακτηριστικά εισόδου

Δεν χρησιμοποιήθηκε καθόλου ενισχυτική μηχανική μάθηση με πραγματικά παιχνίδια άρα τα Rollout,SL δίκτυα καταργήθηκαν. Παρατηρήθηκε ότι με μηδενική γνώση για το παιχνίδι το μοντέλο κατάφερε να μάθει ήδη υπάρχουσες και καινούργιες στρατηγικές επιφέροντας καλύτερες επιδόσεις από όταν εκπαιδεύτηκε με ανθρωπινά δεδομένα.

Στα παιχνίδια που παράγονται μέσω self-play εντάχθηκε ο αλγόριθμος MCTS για την επιλογή κινήσεων. Στον Alpha go χρησιμοποιούταν μόνο η αρχική πρόβλεψη του δικτύου. Με αυτόν τον τρόπο έχουμε πιο ποιοτικά δείγματα καθώς η κατανομή π προσεγγίζει καλύτερα την βέλτιστη κίνηση.

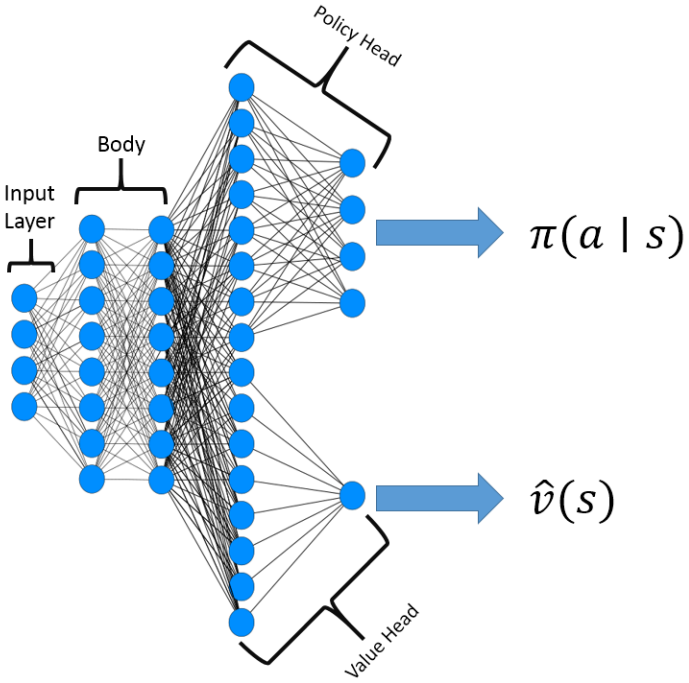


Εικόνα 52: MCTS self-play

Η επιλογή της κίνησης μετά τον αλγόριθμο MCTS μπορεί να γίνει με δύο τρόπους. Σε αγωνιστικά παιχνίδια, επιλέγουμε την κίνηση με τις περισσότερες επισκέψεις N. Σε self-play εντάσσουμε μια θερμοκρασία τ και επιλέγουμε μια κίνηση βάση την κατανομή $\pi \sim N^{1/\tau}$. Με αυτόν τον τρόπο ενθαρρύνουμε την εξερεύνηση κατά την εκπαίδευση.

Τα Policy / Value networks ενσωματώθηκαν σε ένα κοινό ResNet με δύο κεφαλές. Με αυτόν τον τρόπο εκπαιδεύονται παράλληλα και οι δύο ποσότητες με την ίδια είσοδο. Το μοντέλο περιγράφεται ως $(p, v) = f_\theta(s)$ με συνάρτηση σφάλματος $l = (z - v)^2 - \pi^T log p + c\|\theta\|$, όπου z το τελικό αποτέλεσμα του παιχνιδιού, v η πρόβλεψη του

αποτελέσματος, π η κατανομή από τον MCTS, p η προβλεπόμενη κίνηση από το policy head και c regularization υπερπαράμετρος. Παρατηρούμε ότι το δίκτυο προσπαθεί να μάθει τις κινήσεις από την κατανομή π και όχι βάσει του αποτελέσματος του παιχνιδιού, πέρα από αυτό έχουμε πάλι εφαρμογή MSE+Cross Entropy.



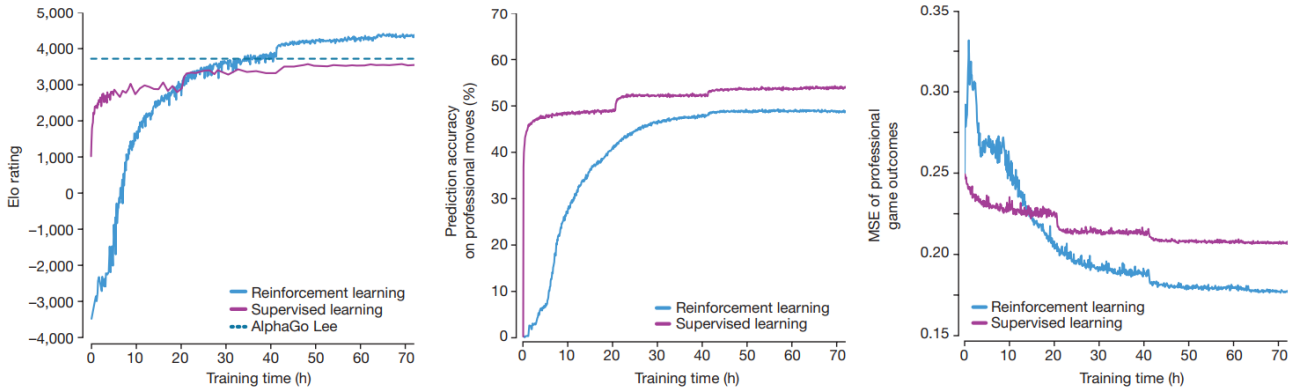
Εικόνα 53: Παράδειγμα απλού feedforward NN με 2 κεφαλές

Στον αλγόριθμο MCTS δεν πραγματοποιούνται rollouts αλλά υπολογίζεται μόνο η αξιολόγηση κάθε κόμβου.

$$Q(s,a) = \frac{W_v(s,a)}{N(s,a)}$$

Σε συγκεκριμένα σημεία ελέγχου πραγματοποιούνται δοκιμές του τωρινού δικτύου με το δίκτυο του προηγούμενου σημείου ελέγχου. Για να συνεχίσει την διαδικασία το τελευταίο δίκτυο πρέπει να καταφέρει ποσοστό νικών της τάξης του 55%. Με αυτόν τον τρόπο αποφεύγουμε κακά δείγματα εκπαίδευσης.

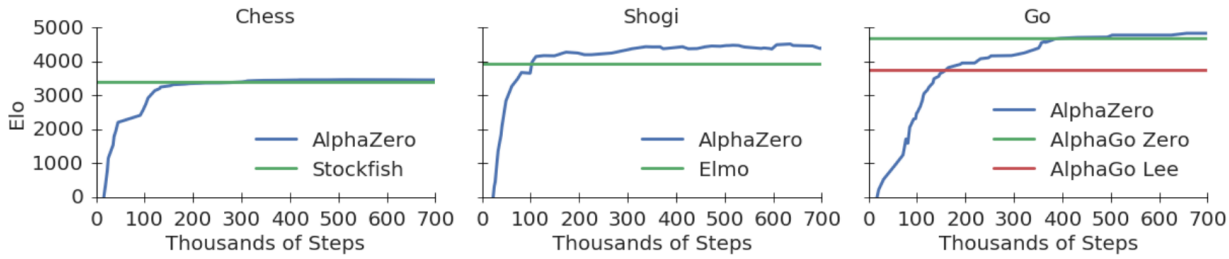
Τέλος στις prior πιθανότητες προστέθηκε Dirichlet θόρυβος για περαιτέρω έμφαση στην εξερεύνηση.



Εικόνα 54: Σύγκριση μοντέλων AlphaGoZero-AlphaGo

2.15.5 Alpha Zero

Ο Alpha Zero αποτελεί βελτίωση του AlphaGo Zero. Εκπαιδεύτηκαν 3 διαφορετικά μοντέλα για τα παιχνίδια chess,go,shogi. Σε θεωρητικό επίπεδο δεν διαφέρει αρκετά από την προηγούμενη υλοποίηση. Οι κύριες διαφορές είναι ότι εκπαιδεύεται κάθε φορά το ίδιο μοντέλο (αφαιρέθηκε η ιδέα με το 55% winrate). Δεν χρησιμοποιήθηκαν «data augmentation» τεχνικές στα παραγόμενα δεδομένα καθώς δεν έχουν και τα 3 παιχνίδια συμμετρίες. Ο θόρυβος Dirichlet προσαρμόζεται ανάλογα με το πλήθος των κινήσεων ανά είδος παιχνιδιού.



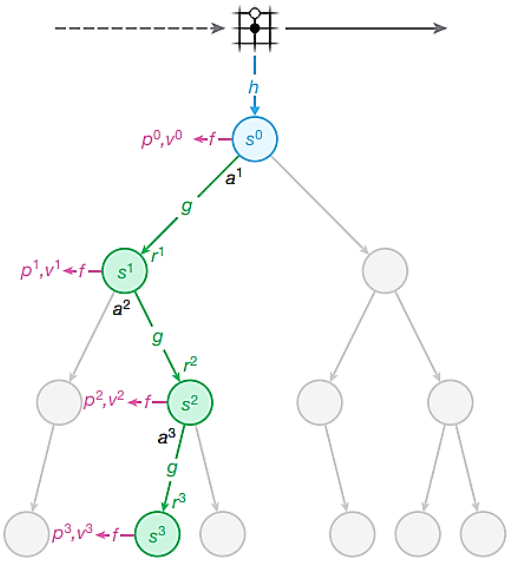
Εικόνα 55: Επιδόσεις Alpha Zero

(Να τονιστεί ότι στην παραπάνω μετρική φαίνεται η επίδοση του πιο αδύναμου stockfish 8, το NNUE εντάχθηκε στην έκδοση 12)

2.15.6 MuZero

Ο MuZero αποτελεί συνέχεια του Alpha Zero. Σε αυτή την παραλλαγή οι κανόνες του παιχνιδιού δεν θεωρούνται γνωστοί. Ως εκ τούτου το αποτέλεσμα μιας κίνησης α σε μια κατάσταση εντός του MCTS αποτελεί προσέγγιση του μοντέλου μας. Με τον τρόπο αυτό μας δίνεται η δυνατότητα να παίξουμε πιο αφηρημένα παιχνίδια όπως αυτά του Atari.

Ένα ResNet h προσπαθεί να αναπαραστήσει μια κατάσταση S ως μια ενδιάμεση κρυφή κατάσταση. Το μοντέλο h δέχεται έναν όγκο που περιγράφει την αναπαράσταση του παιχνιδιού σε πολλαπλές χρονικές στιγμές (πχ frames ή turns) και το μετατρέπει σε έναν όγκο προκαθορισμένου μεγέθους προσεγγίζοντας την αναπαράσταση του πραγματικού περιβάλλοντος.



Εικόνα 56: MuZero κρυφές καταστάσεις

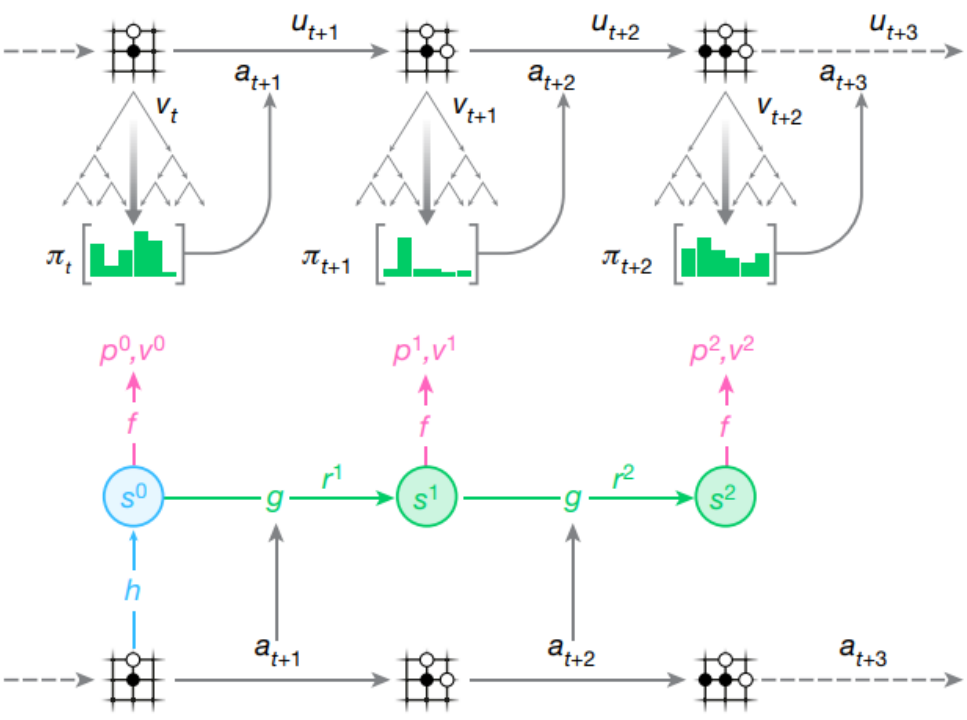
Μέσω μιας συνάρτησης g μπορούμε να μεταβούμε με μια κίνηση α από μια κρυφή κατάσταση στην επόμενη. Η μονή φορά που παράγουμε τις κινήσεις με χρήση σκακιστικών μηχανών είναι σε κόμβους ρίζα. Οι κινήσεις α δεν είναι πάντα έγκυρες.

Η υπόλοιπη διαδικασία PUCT-MCTS παραμένει ίδια.

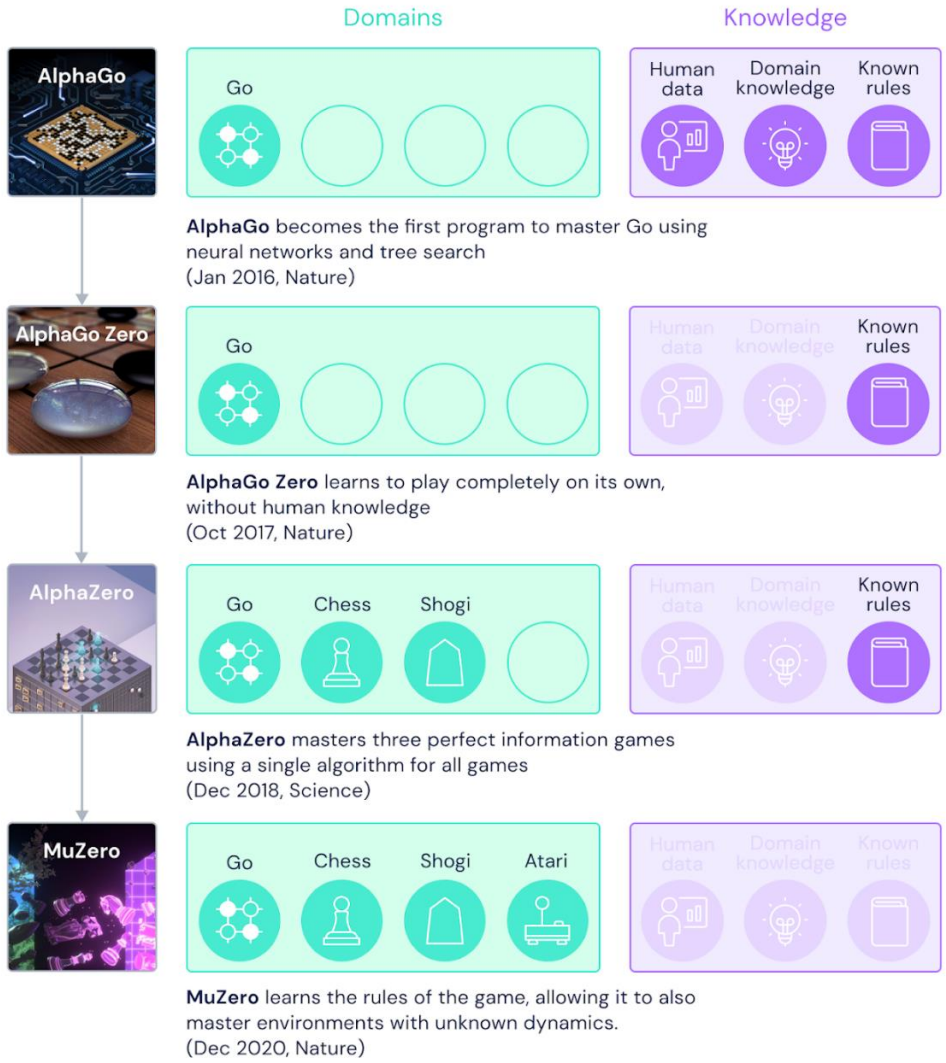
Για την εκπαίδευση του μοντέλου παίρνουμε θέσεις από παιχνίδια που παράχθηκαν μέσω self-play, παράγουμε τις κρυφές καταστάσεις και εφαρμόζουμε μέσω της συνάρτησης g, K διαδοχικες κινήσεις. Για κάθε κρυφή κατάσταση έχουμε μια αντιστοιχία με την πραγματική κατάσταση του παιχνιδιού. Στόχος είναι οι πιθανότητες πολιτικής (policy probabilities) των κρυφών καταστάσεων να είναι ίδιες με αυτές των πραγματικών καταστάσεων και η αξιολόγηση να είναι ίδια με το τελικό αποτέλεσμα του παιχνιδιού. Οι συσχετίσεις των δυάδων αυτών δημιουργούνται λόγω της συνάρτησης σφάλματος και δεν υπάρχει κάποιος άλλος ρητός περιορισμός. Προκύπτει η συνάρτηση:

$$l_t(\theta) = \sum_{k=0}^{K} (CrossEntropy(\pi_{t+\kappa}, p_t^k) + MSE(z_{t+k}, v_t^k)) \; + \; c\|\theta\|$$

Όπου p, v οι τιμές των policy/evaluation δικτύων για την εκάστοτε κρυφή κατάσταση και π, z το policy της πραγματικής κατάστασης και το αποτέλεσμα του παιχνιδιού. Οι τιμές t, K καθορίζουν την χρονική στιγμή του παιχνιδιού και το πλήθος των διαδοχικών κινήσεων αντίστοιχα. Για τα Atari παιχνίδια χρησιμοποιούνται και ενδιάμεσες ανταμοιβές (για λογούς απλότητας δεν συμπεριλήφθηκαν).



Εικόνα 57: MuZero εκπαίδευση / selfplay



Εικόνα 58: Εξέλιξη AlphaGo

2.15.7 Υπολογιστική ισχύς

Ένα από τα μεγαλύτερα προβλήματα των παραπάνω μοντελοποιήσεων είναι η ανάγκη για μεγάλη υπολογιστική ισχύ για την διαδικασία της μάθησης. Συγκεκριμένα η Google κατασκεύασε τα δικά της TPUs που έως και σήμερα δεν είναι διαθέσιμα στην αγορά. Το κόστος του συστήματος του AlphaGo Zero εκτιμάται ότι ήταν κοντά στα $25m\$$. Για τον περισσότερο κόσμο δεν είναι προσβάσιμοι τόσο μεγάλοι πόροι.

Πίνακας 8: Υπολογιστική ισχύς εκδόσεων AlphaGo

Version	Hardware
AlphaGo Fan	176 GPUs
AlphaGo Lee	48 TPUs
AlphaGo Zero	4 TPUS
Alpha Zero	4 TPUs

3. ΥΛΟΠΟΙΗΣΗ

Σε αυτήν την ενότητα θα αναλυθεί η υλοποίηση της σκακιστικής μηχανής που παράχθηκε για αυτή την πτυχιακή. Συνοπτικά, για την εύρεση βέλτιστης κίνησης αξιοποιείται ένα νευρωνικό δίκτυο NNUE για στατική αξιολόγηση θέσης εντός ενός PVS αλγορίθμου. Τα πιόνια αναπαρίστανται μέσω της δομής bitboards. Η διεπαφή γίνεται μέσω του πρωτοκόλλου UCI.

Καθώς ο κώδικας είναι αρκετά εκτενής δεν θα περιγραφεί κάθε συνάρτηση εφόσον η υλοποίηση της είναι αρκετά τετριμμένη και αυτονόητη βάση ονοματολογίας.

3.1 Γλώσσα επιλογής

Οι περισσότερες μηχανές είναι γραμμένες στην γλώσσα C++ (πχ lc0, stockfish). Η C++ απαιτεί πιο σύνθετη σύνταξη κώδικα αλλά προσφέρει και πολύ γρήγορη ταχύτητα εκτέλεσης. Καθώς ο δείκτης nps έχει άμεση συσχέτιση με την ποιότητα της επιλεγόμενης κίνησης, είναι σημαντικό ο χρόνος εκτέλεσης κάθε αναζήτησης να είναι όσο πιο μικρός γίνεται. Επίσης η αναπαράσταση των Bitboards προϋποθέτει τον χειρισμό bytes, κάτι που είναι αρκετά φυσικό στις γλώσσες C/C++. Τέλος ένα από τα μεγαλύτερα πλεονεκτήματα των γλωσσών χαμηλού επιπέδου είναι οι πιθανές βελτιστοποιήσεις σε επίπεδο μεταγλωτιστή (κάτι που δεν μπορεί να γίνει ευκολά σε διερμηνευμένες γλώσσες όπως την Python). Ως εκ τούτου επιλέχθηκε και για αυτή την μηχανή η γλώσσα C++.

3.2 Build system, μεταγλωττιστής και σχετικά flags

Το build system που χρησιμοποιήθηκε είναι το cmake 3.16.13 καθώς μπορεί με ευκολία να παράξει makefiles αυτόματα.

Η μεταγλώττιση έγινε μέσω g++ με το Standard C++20 για ευκολότερο χειρισμό tuples ή άλλων καινούργιων βελτιώσεων.

Καθώς χρησιμοποιείται η Standard βιβλιοθήκη, η libstd γίνεται στατικά link για αποφυγή υπολειπόμενων dlls στα windows.

Για την χρήση βελτιστοποιήσεων σε επίπεδο SIMD εντολών γίνονται link και οι σχετικές intrinsic συναρτήσεις. Στην περίπτωση που δεν υποστηρίζεται μια από τις mavx2 msse4.1 mss3 mss2 mss3 μπορεί απλά να μην συμπεριληφθεί το κατάλληλο define / linking flag και θα χρησιμοποιηθεί μια εναλλακτική υλοποίηση εντός της βιβλιοθήκης NNUE. Άμα μια εντολή δεν υποστηρίζεται από τον επεξεργαστή τότε το πρόγραμμα θα οδηγηθεί σε «invalid instruction error». Για βέλτιστη ταχύτητα συνιστώνται όλα τα flags εφόσον είναι δυνατόν.

Για optimization παρατηρήθηκε 5-7 φορές ταχύτερη εκτέλεση με το flag -Ofast χωρίς αλλοίωση της ροής του προγράμματος.

Για αποφυγή άσκοπων calls σε συναρτήσεις assert ενεργοποιείται η καθολική σταθερά NDEBUG. (Τα asserts κατά κύριο λόγο είναι χρήσιμα για χειρισμό λανθασμένων καταστάσεων κατά την υλοποίηση του προγράμματος και όχι το τελικό release.)

3.3 Ιεράρχηση τύπων και namespaces

Ολόκληρο το project είναι εντός του namespace ChessEngine για αποφυγή Name collisions με άλλες συναρτήσεις / types. Αν μια οντότητα δεν απαιτεί κάποια αντικειμενοστραφή μοντελοποίηση τότε παράγεται ένα ξεχωριστό sub namespace ή εντάσσεται στο ίδιο επίπεδο αν δεν είναι απαραίτητη η ιεραρχική δόμηση. Διαφορετικά απλά χρησιμοποιούνται classes / structs / enums. Αν μια οντότητα χρησιμοποιείται μόνο εντός μια άλλης τότε μοντελοποιήται και αυτή σαν sub class / struct / enum. Εντός cpp αρχείων χρησιμοποιούνται ανώνυμα namespaces για απόκρυψη των συναρτήσεων εντός του translation unit. Γενικά έχουμε την εξής ιεραρχία:

- Namespace ChessEngine
    - Namespace AttackTables
    - Namespace PseudoMoves
    - Namespace MagicNumbers
    - Namespace MCTS
    - Namespace NNUE
    - Namespace UCI
    - Namespace Masks
    - Namespace Zobrist
    - Enum Team, File, Rank, PieceType, GameResult
    - Enum Masks
    - Class Timer
    - Class Move
    - Class TranspositionTable
        - Struct TTEntry
        - Enum NodeType
    - Class History
        - Struct Element
    - Class BoardTile
    - Class Bitboard
        - Class Iterator
    - Class Board
        - Struct Representation
        - Struct MoveCounters
        - Class CastlingRights

Η βιβλιοθήκη NNUE είναι υλοποιημένη σε C για αυτό και οι αντίστοιχοι τύποι είναι στο global namespace

3.4 Αναπαράσταση ταμπλό

Η αναπαράσταση των Bitboards, όπως μελετήθηκε και στην θεωρία, είναι από τις πιο γρήγορες αναπαραστάσεις και χρησιμοποιείται από τις καλύτερες σκακιστικές μηχανές ανεξαρτήτως μεθόδου αναζήτησης κίνησης (πχ η lc0 χρησιμοποιεί Bitboards + UCT).

3.4.1 Θέση ταμπλό

Προτού φτάσουμε στην αναπαράσταση ολόκληρου του ταμπλό πρέπει να αναλύσουμε την ιδέα των πλακιδίων. Η κλάση BoardTile προσομοιώνει ένα σημείο της σκακιέρας. Σαν δομή αποτελείται από έναν 8 bits ακέραιο που περιγράφη την θέση (index) του εκάστοτε τετραγώνου.

```
uint8_t tile_index_ = 0;
```

Για παράδειγμα το 1ο πλακίδιο είναι στην θέση 0 ενώ το τελευταίο στην θέση 64.

Ο Constructor δέχεται 2 συντεταγμένες {x,y} και μεταβάλλει τον ακέραιο βάσει της σχέσης $y * height + width$

```
BoardTile(uint8_t file, uint8_t rank) :
tile_index_(rank * 8 + file){}
```

Αντίστοιχα για να ανακτήσουμε μια από τις 2 συνιστώσες καλούμε την συνάρτηση GetCoords() όπου επιστρέφει ένα tuple βάσει της σχέσης $\{index\ \%\ 8, index\ /\ 8\}$. Άμα θέλουμε αποκλειστικά μια από τις 2 υπάρχουν και οι μεμονωμένες συναρτήσεις GetRank() GetFile(). ($\{file, rank\} -> \{x, y\}$)

```
uint8_t GetRank() const { return tile_index_ / 8; }
uint8_t GetFile() const { return tile_index_ % 8; }

std::tuple<uint8_t, uint8_t> GetCoords() const
{ return {GetFile(), GetRank()}; }
```

Η συνάρτηση Mirror παράγει την αντικατροπισμένη έκδοση του πλακιδίου. Για παράδειγμα άμα βρισκόμαστε στο {4,0} θα μετάβουμε στο {4,8}.

```
void Mirror() { tile_index_ ^= 0b111000; }
```

Η χρησιμότητα της συνάρτησης αυτής θα φανεί αργότερα στην παραγωγή κινήσεων όπου χρησιμοποιούμε ένα μονοχρωματικό ταμπλό.

Οι υπόλοιπες συναρτήσεις αποτελούν υπερφορτωμένους τελεστές για τις πράξεις $\{\&, |, -\}$.

Για παράδειγμα θέλουμε η ένωση 2 τετραγώνων να παράγει ένα νέο ταμπλό με ενεργοποιημένα τα εμπλεκόμενα σημεία.

```
friend Bitboard operator|(const BoardTile &a, const BoardTile &b)
{ return Bitboard(a) | Bitboard(b); }
```

Αντίστοιχα η ένωση ενός τετραγώνου και ενός ταμπλό να παράγει την επαυξημένη έκδοση του ταμπλό.

```
friend Bitboard operator|(const Bitboard &a, const BoardTile &b)
{ return a | Bitboard(b);}
```

Η πρόσθεση ή αφαίρεση ενός τετραγώνου με έναν ακέραιο αλλάζει το δείκτη απευθείας.

```
friend BoardTile operator-(const BoardTile &a, const int &b)
{ return BoardTile(a.tile_index_ - b); }
friend BoardTile operator+(const BoardTile &a, const int &b)
{ return BoardTile(a.tile_index_ + b); }
```

3.4.2 Bitboards

Τα bitboards είναι η κύρια δομή αναπαράστασης πιονιών όπου κάθε bit σηματοδοτεί την ύπαρξη ή μη ενός χαρακτηριστικού σε μια θέση του ταμπλό. Η πλήρης εξήγηση μερικών ιδιοτήτων έχει ήδη γίνει στην ενότητα 2.6. Ως εκ τούτου σε αυτή την υποενότητα θα γίνει πολύ συνοπτική ανάλυση.

Εσωτερικά στην κλάση Bitboard αποθηκεύεται ένας unsigned int των 64 bit.

```
uint64_t data_ = 0;
```

Ένα bitboard μπορεί να δημιουργηθεί είτε με χρήση ενός BoardTile, με χρήση συντεταγμένων ή απευθείας με έναν ακέραιο.

```
explicit Bitboard(uint64_t value) : data_(value) {}
explicit Bitboard(uint8_t file, uint8_t rank);
explicit Bitboard(BoardTile tile);
```

Με αντίστοιχο τρόπο αρκετές συναρτήσεις παρέχουν υπερφορτωμένες εκδοχές για ευκολότερη χρήση ανάλογα την περίσταση.

Η συνάρτηση Set ενεργοποιεί το ζητούμενο bit χρησιμοποιώντας τους τελεστές του δυαδικού and και της ολίσθησης. (Υπενθυμίζουμε ότι η τομή υλοποιείται με το $\&\,(and)$ και η ενωση με το $|\,(or)$)

```
void Bitboard::Set(uint8_t index){
    data_ |= std::uint64_t(1) << index;
}
```

Η συνάρτηση Get επιστρέφει true αν το ζητούμενο bit είναι ενεργό, αλλιώς false. Ο έλεγχος γίνεται με παρόμοιο τρόπο με την συνάρτηση Set αλλά αυτή την φορά χρησιμοποιείται ο δυαδικός τελεστής or.

```
bool Bitboard::Get(uint8_t index) const{
    return data_ & (std::uint64_t(1) << index);
}
```

Η συνάρτηση Reset μηδενίζει το ζητούμενο Bit. Η ποσότητα $\sim(1 \ll index)$ περιέχει κάθε bit εκτός από αυτό στην θέση Index.

```
void Bitboard::Reset(uint8_t index){
    data_ &= ~(std::uint64_t(1) << index);
}
```

Όπως και στην BoardTile η μέθοδος Mirror αντικατοπτρίζει το bitboard εναλλάσσοντας με τον πιο αποδοτικό τρόπο όλα τα rank. (Για αποφυγή πράξεων μεταφέρουμε πολλαπλές σειρές εφαρμόζοντας κατάλληλες μάσκες και ολισθήσεις κάθε φορά)

```
void Bitboard::Mirror(){
    data_ = (data_ & 0x00000000FFFFFFFF) << 32 | (data_ & 0xFFFFFFFF00000000)
>> 32;
    data_ = (data_ & 0x0000FFFF0000FFFF) << 16 | (data_ & 0xFFFF0000FFFF0000)
>> 16;
    data_ = (data_ & 0x00FF00FF00FF00FF) << 8 | (data_ & 0xFF00FF00FF00FF00)
>> 8;
}
```

Για παράδειγμα η σειρά 1 θα γίνει η σειρά 8, η σειρά 2 θα γίνει η σειρά 7 και ούτω καθεξής.

Οι συναρτήσεις ShiftX μας βοηθούν να κουνήσουμε ένα ή παραπάνω πιόνια προς μια κατεύθυνση. Η παραγωγή κινήσεων είναι κρίσιμο να είναι όσο πιο γρήγορη γίνεται, για αυτό και συγκεκριμένες κατευθύνσεις είναι υλοποιημένες σε μεμονωμένες συναρτήσεις.

```
Bitboard ShiftDown2() const
{ return Bitboard(data_ >> (2 * 8)); }
Bitboard ShiftUp1() const
{ return Bitboard(data_ << (1 * 8)); }
Bitboard ShiftUp1Right1() const
{ return Bitboard(data_ << (1 * 8 + 1)); }
Bitboard ShiftUp1Left1() const
{ return Bitboard(data_ << (1 * 8 - 1)); }
```

Για περιπτώσεις που η ταχύτητα δεν είναι πρώτιστης σημασίας (πχ σε διαδικασίες αρχικοποίησης) υπάρχει η γενικευμένη συνάρτηση ShiftTowards. Όπως και προηγουμένως υπολογίζουμε το offset ολίσθησης και η φορά κρίνεται από ένα branch. (Ο τελεστής ολίσθησης υποχρεωτικά παίρνει μόνο θετικές τιμές άρα δεν μπορεί να αποφευχθεί)

```
Bitboard Bitboard::ShiftTowards(std::tuple<int8_t, int8_t> direction) const {
    // index_curr = y * 8 + x
    // index_new = (y+y_off)*8 + (x+x_off)
    // index_new - index_curr =  8*y_off + x_off.
    Bitboard temp = *this;
    auto[x_offset, y_offset] = direction;
    int8_t offset = 8 * y_offset + x_offset;
    if(offset > 0)
        temp.data_ <<= offset;
    else
        temp.data_ >>= abs(offset);
    return temp;
}
```

Η συνάρτηση Count υλοποιεί τον αλγόριθμο PopCount του Kernighan και μετράει το πλήθος των ενεργών bits εντός της λέξης. (Διαισθητικά μετράμε το πλήθος ενεργών πιονιών)

```
uint8_t Bitboard::Count() const {
    uint64_t temp = data_;
    uint8_t count = 0;
    while(temp){
        count++;
        temp &= (temp - 1);
    }
    return count;
}
```

Η πράξη $temp \,\&= \,(temp - 1)$ αφαιρεί κάθε φορά το lsb bit, άρα επαναληπτικά όταν φτάσουμε την μηδενική τιμή θα έχουμε μετρήσει κάθε Bit.

Σε αντιστοιχία η συνάρτηση BitScanForward επιστρέφει την θέση (τετράγωνο) του lsb bit χρησιμοποιώντας ένα προκαθορισμένο πίνακα αναζήτησης και τον αριθμό debruijn64.

```
BoardTile Bitboard::BitScanForward() { /* De Bruijn bitscan algorithm */
    const uint64_t debruijn64 = 0x03f79d71b4cb0a89;
    const uint8_t index64[64] = {…};
    return BoardTile(index64[((data_ & ~(data_-1)) * debruijn64) >> 58]);
}
```

Η εύρεση του lsb bit είναι αρκετά χρήσιμη διαδικασία. Για παράδειγμα άμα έχουμε ένα Bitboard με όλες τις πιθανές κινήσεις να αναπαρίστανται με ενεργά bits, θα θέλαμε να διατρέξουμε κάθε πιθανό τετράγωνο για να εξάγουμε την πληροφορία θέσης. Παίρνοντας το τελευταίο ενεργό bit κάθε φορά γλιτώνουμε την επίσκεψη όλων των 64 θέσεων.

Πιο συγκεκριμένα μπορούμε να εκτελέσουμε τον ακόλουθο κώδικα

```
Bitboard temp = …something…;
while(temp){
    BoardTile tile = temp.BitScanForward();
    …use tile…
    temp &= (temp - 1);
}
```

Στο τέλος της επανάληψης αρκεί να αφαιρέσουμε το lsb bit (αυτό σημαίνει ότι αλλοιώνουμε την πληροφορία και για αυτό πρέπει να κρατηθεί αντίγραφο πριν αρχίσει η διαδικασία).

Στην πράξη καθώς η ίδια προεργασία γίνεται σε πολλά σημεία του κώδικα, χρησιμοποιούνται οι iterator της C++ για πιο εύκολη σύνταξη και χρήση.

```
class Iterator{
public:
    explicit Iterator(uint64_t data): data_(data) {}
    Iterator& operator++() // Remove lsb.
        { data_ &= (data_ - 1); return *this; }
    Iterator operator++(int) // Remove lsb.
        { Iterator temp = *this; data_ &= (data_ - 1); return temp; }
    BoardTile operator*() const; // Gets LSB tile.

    friend bool operator!= (const Iterator& a, const Iterator& b)
        { return a.data_ != b.data_; };
    friend bool operator== (const Iterator& a, const Iterator& b)
        { return a.data_ == b.data_; };
private:
    uint64_t data_;
};
```

Οι παραπάνω συναρτήσεις υλοποιούν ακριβώς την ίδια διαδικασία αλλά ο κώδικας απλοποιείται σημαντικά.

```
Bitboard temp = …something…;
for(auto tile : temp){
    …use tile…
}
```

Ο τελεστής (-) μεταξύ 2 bitboards λειτουργεί ως αφαίρεση συνόλων και υλοποιείται παίρνοντας την τομή του συμπληρώματος $(A \,\&\, {\sim}B)$.

```
friend Bitboard operator-(const Bitboard& a, const Bitboard& b)
{ return Bitboard(a.data_ & ~b.data_); }
```

Οι υπόλοιπες overloaded συναρτήσεις λειτουργούν με τον αναμενόμενο τρόπο και εφαρμόζουν τον αντίστοιχο τελεστή στην εσωτερική αναπαράσταση $data\_$ (δεν θα αναλυθούν μεμονωμένα γιατί είναι αυτονόητη η λειτουργία τους).

Ένα ταμπλό θα λέμε ότι είναι άδειο άμα έχει την τιμή 0 (δηλαδή δεν έχει κανένα πιόνι).

```
bool IsEmpty() const { return data_ == 0; }
```

3.4.3 Βασικές μάσκες

Σε διάφορα σημεία του κώδικα χρησιμοποιούνται μάσκες σε μορφή boardTiles ή bitboards. Οι περισσότερες έχουν αυτονόητο περιεχόμενο βάσει ονοματολογίας. Όπου φανεί αναγκαίο θα αναλυθεί η χρήση τους στα πλαίσια κώδικα. Σε αυτή την ενότητα γίνεται μια απλή αναφορά.

```
namespace Masks{
    // Files.
    const Bitboard file_A = Bitboard(0x0101010101010101ULL);
    const Bitboard file_B = file_A.ShiftTowards({1,0});
    const Bitboard file_G = file_A.ShiftTowards({6,0});
    const Bitboard file_H = file_A.ShiftTowards({7,0});

    // Not files.
    const Bitboard not_file_A = ~file_A;
    const Bitboard not_file_B = ~file_B;
    const Bitboard not_file_G = ~file_G;
    const Bitboard not_file_H = ~file_H;
    const Bitboard not_file_AB = ~(file_A | file_B);
    const Bitboard not_file_GH = ~(file_H | file_G);

    // Ranks.
    const Bitboard rank_1 = Bitboard(0xff);
    const Bitboard rank_2 = rank_1.ShiftTowards({0,1});
    const Bitboard rank_3 = rank_1.ShiftTowards({0,2});
    const Bitboard rank_4 = rank_1.ShiftTowards({0,3});
    const Bitboard rank_6 = rank_1.ShiftTowards({0,5});
    const Bitboard rank_7 = rank_1.ShiftTowards({0,6});
    const Bitboard rank_8 = rank_1.ShiftTowards({0,7});
    const Bitboard rank_1_8 = rank_1 | rank_8;

    const Bitboard outer_tiles = rank_1 | rank_8  | file_A | file_H;
    const Bitboard inner_tiles = ~outer_tiles;
    const Bitboard empty = Bitboard(0);

    // Corners.
    const BoardTile a1_tile = BoardTile(0,0);
    const BoardTile a8_tile = BoardTile(0,7);
    const BoardTile h1_tile = BoardTile(7,0);
    const BoardTile h8_tile = BoardTile(7,7);
    const Bitboard corner_tiles = a1_tile | a8_tile | h1_tile | h8_tile;

    // Castling.
    const BoardTile king_default = BoardTile(4, 0);
    const BoardTile queen_rook = BoardTile(0,0);
    const BoardTile king_rook = BoardTile(7,0);
    const Bitboard queen_castling_tiles = BoardTile(1,0) |
                                          BoardTile(2,0) |
                                          BoardTile(3,0);
    const Bitboard king_castling_tiles = BoardTile(6,0) | BoardTile(5,0);

    const Bitboard light_squares(0x55AA55AA55AA55AAULL);
    const Bitboard dark_squares(0xAA55AA55AA55AA55ULL);
}
```

3.4.4 Αναπαράσταση

Μέχρι τώρα είδαμε πώς μπορούμε να περιγράψουμε την ύπαρξη πιονιών με δυαδικό τρόπο. Στο παιχνίδι του σκακιού όμως έχουμε παραπάνω από ένα είδος πιονιού και δεν μας αρκεί μόνο ένα bitboard για ολόκληρο το ταμπλό. Η υλοποίηση που ακολουθήθηκε είναι επηρεασμένη από την αναπαράσταση της σκακιστικής μηχανής lc0 και εστιάζει στην ελαχιστοποίηση του απαιτούμενου χώρου χωρίς να επιβαρύνει σημαντικά την αναζήτηση θέσεων. Αντί να έχουμε ένα Bitboard ανά είδος προσπαθούμε να εντάξουμε την πληροφορία παραπάνω πιονιών με έμμεσο τρόπο.

Πιο συγκεκριμένα έχουμε το struct representation.

```
struct Representation{
    // Queens share the bitboards of rooks and bishops
    // Pawns can include the en passant square on ranks 1-8.
    Bitboard own_pieces = Bitboard(0);
    Bitboard enemy_pieces = Bitboard(0);
    Bitboard rook_queens = Bitboard(0);
    Bitboard bishop_queens = Bitboard(0);
    Bitboard pawns_enPassant = Bitboard(0);
    BoardTile own_king = BoardTile(0);
    BoardTile enemy_king = BoardTile(0);

    Bitboard Rooks() const { return rook_queens - bishop_queens; }
    Bitboard Queens() const { return rook_queens & bishop_queens; }
    Bitboard Bishops() const { return bishop_queens - rook_queens; }
    Bitboard Pawns() const { return pawns_enPassant - Masks::rank_1_8; }
    Bitboard EnPassant() const { return pawns_enPassant & Masks::rank_1_8; }
    Bitboard Kings() const { return own_king | enemy_king; }
    Bitboard Knights() const {
        return (own_pieces | enemy_pieces)
               - rook_queens - bishop_queens - Pawns()
               - own_king - enemy_king;
    }

    // Mirrors the representation vertically
    void Mirror();
};
```

Οι μεταβλητές $own\_pieces, enemy\_pieces$ αποτελούν occupancy bitboards και περιγράφουν την ύπαρξη πιονιών ανεξαρτήτως είδους.

Οι βασίλισσες αποθηκεύονται δύο φορές μαζί με τους πύργους και τους αξιωματικούς και μπορούν να εξαχθούν παίρνοντας την τομή των δύο συνόλων.

$$queens = rook\_queens \,\&\, bishop\_queens$$

Αντίστοιχα άμα θέλουμε να πάρουμε μόνο τους πύργους αρκεί να αφαιρέσουμε τα bits των βασιλισσών. Γνωρίζουμε ότι το σύνολο $bishop\_queens$ περιέχει την επιθυμητή πληροφορία καθώς οι αξιωματικοί βρίσκονται σε διαφορετικές θέσεις από τους πύργους.

$$rooks = rook\_queens - bishop\_queens$$

Με τον ίδιο τρόπο μπορούμε να πάρουμε μόνο τους αξιωματικούς αφαιρώντας την μεταβλητή $rook\_queens$.

$$bishops = bishop\_queens - rook\_queens$$

Το en passant flag αποθηκεύεται στην 1η και 8η σειρά των pawns bitboard. Οι θέσεις αυτές είναι μη αποδέκτες για τα υπόλοιπα πιόνια καθώς α) δεν μπορούν να πάνε προς τα πίσω β) στην τελευταία σειρά γίνονται προαγωγή.

$$pawns = pawns\_enPassant - rank\_1\_8$$

$$enPassant = pawns\_enPassant \,\&\, rank\_1\_8$$

Η μάσκα $rank\_1\_8$ έχει ενεργά όλα τα bits των σειρών 1,8.

Τα αλόγα αναπαρίστανται με έμμεσο τρόπο και μπορούν να ανακτηθούν αφαιρώντας από τα Occupancy bitboards κάθε άλλο είδος πιονιού.

$$knights = (own \mid enemy) - pawns - bishops - queens - rooks$$

Οι παραπάνω μεταβλητές αναφέρονται σε όλα τα πιόνια ανεξαρτήτως ομάδας. Η συγκεκριμένη πληροφορία μπορεί να γίνει “filter” με χρήση του δυαδικού & και του επιθυμητού occupancy.

Οι βασιλιάδες αποθηκεύονται σε ξεχωριστές μεταβλητές BoardTiles καθώς δεν μπορούν ποτέ να αιχμαλωτιστούν ή να αυξηθούν σε πλήθος.

Άμα είχαμε 1 bitboard ανά piece type θα είχαμε $6 * 2 * bitboards = 12 * 64 = 768\, b$. Η συγκεκριμενη αναπαρασταση απαιτει $5\, bitboards + 2\, tiles = 5 * 64 + 2 * 8 = 336\, b$.

Παρατηρούμε ότι δεν έχουμε διαχωρισμό άσπρων και μαύρων πιονιών αλλά «δικών» μας και «αντίπαλων». Κάθε φορά που αλλάζει το turn θέλουμε οι ρόλοι αυτοί να αντιστρέφονται. Την διαδικασία αυτή κάνει η συνάρτηση Mirror.

```
void Board::Representation::Mirror() {
    rook_queens.Mirror();
    bishop_queens.Mirror();
    pawns_enPassant.Mirror();
    own_king.Mirror();
    enemy_king.Mirror();

    own_pieces.Mirror();
    enemy_pieces.Mirror();

    std::swap(own_pieces, enemy_pieces);
    std::swap(own_king, enemy_king);
}
```

Αφού ανακλαστεί κάθε bitboard αρκεί να ανταλλάξουμε τα occupancies και τους βασιλιάδες. Με αυτόν τον τρόπο το ταμπλό είναι μονοχρωματικό, δηλαδή το βλέπουμε πάντα από την σκοπιά των άσπρων.

3.4.5 Μετρητές κινήσεων

Το struct MoveCounters είναι πολύ απλό στην δομή του και η κύρια χρησιμότητα του είναι η καταμέτρηση διάφορων ιδιοτήτων όπως το πλήθος των κινήσεων που έχουν γίνει, το πλήθος επαναλήψεων της συγκεκριμένης θέσης, πλήθος κινήσεων για την ισοπαλία των 50 κινήσεων. Η μεταβλητή $ply\_counters$ είναι χρήσιμη και για τον προσδιορισμό της θέσης στον πίνακα NNUE.

```
struct MoveCounters{
    // 50 move rule counter.
    uint8_t half_moves = 0;
    // 3 fold repetition.
    uint8_t repetitions = 0;
    // Game's plys.
    uint16_t full_moves = 0;
    uint16_t ply_counter = 0;
};
```

3.4.6 Δικαιώματα ροκέ

Πέρα από την θέση των πιονιών θέλουμε να κρατάμε και πληροφορίες που δεν έχουν οπτικό χαρακτήρα, όπως την ικανότητα για ροκέ. Διακρίνουμε 4 σενάρια:

- άσπρα ροκέ προς την βασίλισσα.
- μαύρα ροκέ προς την βασίλισσα.
- άσπρα ροκέ προς τον βασιλιά.
- μαύρα ροκέ προς τον βασιλιά.

Η πληροφορία αυτή μπορεί να αποθηκευτεί αποδοτικά εντός 4 bits. Η μοντελοποίηση γίνεται με αναμενόμενο τρόπο εντός της class CastlingRights. Η σειρά των bits δεν έχει ιδιαίτερη σημασία αρκεί να είναι σταθερή κατά την εκτέλεση.

Ο έλεγχος ενός δικαιώματος γίνεται με χρήση της αντίστοιχης μάσκας θέσης

```
bool CanOwnQueenSide() const { return data_ & 1; }
bool CanOwnKingSide() const { return data_ & 2; }
bool CanEnemyQueenSide() const { return data_ & 4; }
bool CanEnemyKingSide() const { return data_ & 8; }
```

Με ίδιο τρόπο μπορούμε να αλλάξουμε ένα δικαίωμα

```
void ResetOwnQueenSide() { data_ &= ~1; }
void ResetOwnKingSide() { data_ &= ~2; }
void ResetEnemyQueenSide() { data_ &= ~4; }
void ResetEnemyKingSide() { data_ &= ~8; }
```

Στην συγκεκριμένη υλοποίηση κάθε bit έχει την ακόλουθη αξία.

```
// Bit 1 -> own queen.
// Bit 2 -> own king.
// Bit 3 -> enemy queen.
// Bit 4 -> enemy king.
// By default all are disabled.
uint8_t data_ = 0;
```

Όπως και προηγουμένως έχουμε μια συνάρτηση Mirror που ανταλλάσσει τα rights μεταξύ των 2 ομάδων. Στην πράξη πρέπει να ανταλλάξουμε τις δυάδες από bits.

```
// Swaps the castling rights between own and enemy.
// We dont use bitfields so we can swap the bits ourselves.
void Mirror() { data_ = ((data_ & 0b11) << 2) + ((data_ & 0b1100) >> 2); }
```

Δεν χρησιμοποιήθηκαν Bitfields ώστε να μπορεί να γίνει η ανταλλαγή πιο αποδοτικά.

3.4.7 Ταμπλό

Η κλάση Board περιέχει τα αντικείμενα των παραπάνω struct / classes ώστε να μπορεί να μοντελοποιηθεί ιεραρχικά πλήρως ένα ταμπλό.

```
Representation representation_;
CastlingRights castling_rights_;
MoveCounters move_counters_;

// default state corresponds to white.
bool is_flipped_ = false;
uint64_t zobrist_key_;
```

Επιπρόσθετα έχουμε την μεταβλητή is_flipped_ ώστε να ξέρουμε ποια είναι η πραγματική όψη του μονοχρωματικού ταμπλό όταν αυτό είναι απαραίτητο.

Η μεταβλητή Zobrist_key χρησιμοποιείται για τον κατακερματισμό του ταμπλό και ανανεώνεται βηματικά κατά το παίξιμο μιας κίνησης.

Όπως και προηγουμένως έχουμε μια συνάρτηση Mirror (η πιο ιεραρχικά υψηλή) που καλεί τις αντίστοιχες Mirror κάθε πεδίου.

```
void Board::Mirror() {
    representation_.Mirror();
    castling_rights_.Mirror();
    is_flipped_ = !is_flipped_;

    // Incremental update on zobrist key due to black's turn.
    zobrist_key_ ^= Zobrist::GetSideKey();
}
```

Οι συναρτήσεις GetPieceInfoAt επιστρέφουν ένα tuple με το χρώμα και είδος του πιονιού στην ζητούμενη θέση

```
PieceInfo GetPieceInfoAt(uint8_t file, uint8_t rank) const;
PieceInfo GetPieceInfoAt(BoardTile tile) const;
PieceType GetPieceTypeAt(uint8_t file, uint8_t rank) const;
```

Η παρούσα κλάση περιέχει πολλές ακόμα συναρτήσεις που αξιοποιούνται σε μετέπειτα στάδια, όπως η PlayMove και η GetLegalMoves. Οι αντίστοιχες αναλύσεις θα γίνουν στα εκάστοτε κεφάλαια.

3.5 Πίνακες επιθέσεων

Η παραγωγή κινήσεων γίνεται σε δύο βήματα με τον ίδιο τρόπο που αναφέρθηκε στο θεωρητικό σκέλος. Οι ψεύδο κινήσεις βρίσκονται κυρίως με χρήση πινάκων αναζήτησης που κατασκευάζονται στην αρχή του προγράμματος. Σε αυτή την ενότητα θα περιγραφεί η κατασκευή των πινάκων αυτών. Υπενθυμίζουμε ότι δεν μας ενδιαφέρει η ταχύτητα γιατί οι συναρτήσεις που θα ακολουθήσουν καλούνται μόνο στην φάση αρχικοποίησης.

3.5.1 Leaper επιθέσεις

Ως Leaper πιόνια εννοούμε τον βασιλιά τα αλόγα και τα πιόνια (δηλαδή πιόνια που κάνουν μεμονωμένα άλματα).

Η παραγωγή κινήσεων για τον βασιλιά είναι αρκετά τετριμμένη. Αξιοποιείται η συνάρτηση shiftTowards ώστε να πάρουμε νέες θέσεις βάσει του δοθέντος ταμπλό. Κάθε φορά επαυξάνουμε το σύνολο moves. Στο τέλος της συνάρτησης έχουμε ένα Bitboard με ενεργά bits τις επιτρεπτές θέσεις.

```
Bitboard GetKingAttacks(Bitboard board) {
    Bitboard moves{};

    moves |= board.ShiftTowards({+0, +1}); // Up.
    moves |= board.ShiftTowards({+0, -1}); // Down.

    moves |= board.ShiftTowards({-1, +0}) & Masks::not_file_H; // Left.
    moves |= board.ShiftTowards({-1, +1}) & Masks::not_file_H; // Left Up.
    moves |= board.ShiftTowards({-1, -1}) & Masks::not_file_H; // Left Down.

    moves |= board.ShiftTowards({+1, +0}) & Masks::not_file_A; // Right.
    moves |= board.ShiftTowards({+1, +1}) & Masks::not_file_A; // Right Up.
    moves |= board.ShiftTowards({+1, -1}) & Masks::not_file_A; // Right Down.

    return moves;
}
```

Οι μάσκες not_file_x είναι χρήσιμες για την απαλοιφή λανθασμένων bits λόγω φαινομένων overflowing / underflowing των σειρών (η πιο αναλυτική εξήγηση έχει ήδη γίνει στην θεωρία).

Με ακριβώς ίδιο τρόπο αλλά διαφορετικές μάσκες και κατευθύνσεις παράγουμε τις κινήσεις των αλόγων (knights).

```
Bitboard GetKnightAttacks(Bitboard board) {
    Bitboard moves{};

    moves |= board.ShiftTowards({-1, +2}) & Masks::not_file_H; //Up left.
    moves |= board.ShiftTowards({+1, +2}) & Masks::not_file_A; //Up right.
    moves |= board.ShiftTowards({-2, +1}) & Masks::not_file_GH; //Left up.
    moves |= board.ShiftTowards({+2, +1}) & Masks::not_file_AB; //Right up.

    moves |= board.ShiftTowards({-1, -2}) & Masks::not_file_H; //Down left.
    moves |= board.ShiftTowards({+1, -2}) & Masks::not_file_A; //Down right.
    moves |= board.ShiftTowards({-2, -1}) & Masks::not_file_GH; //Left down.
    moves |= board.ShiftTowards({+2, -1}) & Masks::not_file_AB; //Right down.

    return moves;
}
```

Οι μεταβλητές Board θα μπορούσαν να είναι οποιαδήποτε συστοιχία πιονιών. Σε αυτή την υλοποίηση η χρήση τους περιορίζεται για την αρχικοποίηση πινάκων και περιέχουν πάντα μόνο ένα ενεργό Bit.

Οι προϋπολογισμένες επιθέσεις πιονιών χρησιμοποιούνται κυρίως για τον έλεγχο επίθεσης βασιλιά και όχι για την παραγωγή κινήσεων. Η διαδικασία παραμένει ίδια.

```
Bitboard GetPawnAttacks(Bitboard board) {
    Bitboard left_attack, right_attack;
    left_attack = board.ShiftTowards({-1, 1}) & Masks::not_file_H;
    right_attack = board.ShiftTowards({1, 1}) & Masks::not_file_A;
    return left_attack | right_attack;
}
```

3.5.2 Μαγικοί αριθμοί

Οι μαγικοί αριθμοί που χρησιμοποιούνται για τον κατακερματισμό slider πιονιών είναι υπολογισμένοι με τέτοιο τρόπο ώστε να έχουμε καλή εξοικονόμηση χώρου (άρα και λιγότερων αστοχιών στην cache). Βρέθηκαν στο forum talkchess.com και δεν υπολογίστηκαν από το μηδέν για καλύτερες επιδόσεις.

http://www.talkchess.com/forum3/viewtopic.php?p=727500#p727500

Οι συναρτήσεις BishopMagicHash RookMagicHash χρησιμοποιούνται για την εύρεση του μαγικού αριθμού για μια συστοιχία πιονιών (occupancies) και μιας ζητούμενης θέσης.

```
uint64_t BishopMagicHash(Bitboard board, uint8_t square_index){
    auto square_info = bishopMagics[square_index];
    return ((board.AsInt() * square_info.magic_number) >> (64 -
bishopBitOffset)) + square_info.main_table_offset;
}

uint64_t RookMagicHash(Bitboard board, uint8_t square_index){
    auto square_info = rookMagics[square_index];
    return ((board.AsInt() * square_info.magic_number) >> (64 -
rook_bit_offset)) + square_info.main_table_offset;
}
```

Η μόνη διάφορα από την υλοποίησή της θεωρίας είναι η εισαγωγή των table_offsets για τον προαναφερόμενο λόγο.

3.5.3 Slider επιθέσεις

Ως slider piece εννοούμε τους αξιωματικούς, τους πύργους και τις βασίλισσες (δηλαδή πιόνια που κάνουν “slide” σε πολλαπλά τετράγωνα). Συγκριτικά με τα Leaper πιόνια, δεν μπορούμε να εξάγουμε τις θέσεις βάσει μόνο ενός διανύσματος κατεύθυνσης γιατί πρέπει να λάβουμε υπόψη και το occupancy map. Επίσης μια επαναληπτική προσέγγιση θα ήταν αρκετά χρονοβόρα κατά την κύρια εκτέλεση του προγράμματος.

Καθώς τώρα δεν αναφερόμαστε σε μεμονωμένα άλματα, χρειαζόμαστε μια βοηθητική συνάρτηση που να παράγει ακτίνες ως προς διάφορες κατευθύνσεις.

Η συνάρτηση GetRay δέχεται την αρχική θέση, ένα διάνυσμα κατεύθυνσης και την συστοιχία πιονιών σε μορφή Occupancy map.

```
Bitboard GetRay(BoardTile from,
                std::tuple<int8_t, int8_t> direction,
                Bitboard occupancies)
{
    auto[x, y] = from.GetCoords();
    auto[x_off, y_off] = direction;

    x += x_off;
    y += y_off;
    Bitboard ray{};
    while (x >= 0 && x < 8 && y >= 0 && y < 8) {
        ray |= Bitboard(x, y);
        if (occupancies.Get(x, y))
            break;
        x += x_off;
        y += y_off;
    }

    return ray;
}
```

Ξεκινώντας από την αρχική θέση ενεργοποιούμε κάθε bit σε μια τοπική μεταβλητή ray μέχρι να πετύχουμε ένα εμπόδιο (δηλαδή ένα bit στο occupancies). Το τελικό σύνολο αποτελείται από όλες τις κενές θέσεις ως προς την ζητούμενη ακτίνα, την αρχική θέση και το τελικό εμπόδιο.

Έτσι η παραγωγή κινήσεων των αξιωματικών και των πύργων υλοποιείται με τον ακόλουθο τρόπο.

```
Bitboard GetRookRays(BoardTile from, Bitboard occupancies = Masks::empty) {
    return GetRay(from, {0, 1}, occupancies) |
           GetRay(from, {0, -1}, occupancies) |
           GetRay(from, {1, 0}, occupancies) |
           GetRay(from, {-1, 0}, occupancies);
}

Bitboard GetBishopRays(BoardTile from, Bitboard occupancies = Masks::empty) {
    return GetRay(from, {1, 1}, occupancies) |
           GetRay(from, {-1, 1}, occupancies) |
           GetRay(from, {-1, -1}, occupancies) |
           GetRay(from, {1, -1}, occupancies);
}
```

Η βασική δομή του κώδικα δεν διαφέρει από αυτή των Leaper πιονιών πέρα από την προσαρμογή της συνάρτησης GetRay.

3.5.4 Συνδυασμοί Occupancies

Ο τελικός στόχος είναι να μπορούμε να εξάγουμε τις κινήσεις για τα slider πιόνια χωρίς επιπρόσθετες επαναλήψεις. Για να γίνει αυτό απαιτείται η ύπαρξη ενός πίνακα με αποθηκευμένες τις κινήσεις για κάθε «χρήσιμο» συνδυασμό από occupancy bits. Η πιο αναλυτική εξήγηση της μεθοδολογίας και απλοποίησης του χώρου γίνεται στην ενότητα Απλοποίηση χώρου 2.8.2.

Ο αλγόριθμος που υλοποιείται είναι ο ίδιος που αναφέρεται και στο θεωρητικό σκέλος.

```
template <typename Set>
void ProduceSubSets(Bitboard set, Set process) {
    uint64_t occupants_permutation = 0;
    do {
        process(Bitboard(occupants_permutation));
        occupants_permutation = (occupants_permutation - set.AsInt()) &
        set.AsInt();
    } while (occupants_permutation != 0);
}
```

Η μόνη διάφορα είναι ότι προσθέτουμε την μεταβλητή process που περιέχει την αναγκαία συνάρτηση χειρισμού του εκάστοτε αποτελέσματος. (Με αυτό τον τρόπο δεν χρειαζόμαστε δύο συναρτήσεις για τους αξιωματικούς και πύργους.)

3.5.5 Παραγωγή πινάκων

Οι τελικοί πίνακες αρχικοποιούνται στην αρχή του προγράμματος με την συνάρτηση InitMoveTables.

Η βασική ιδέα είναι ότι διατρέχουμε όλο το ταμπλό και παράγουμε όλες τις απαραίτητες πληροφορίες για την κάθε θέση.

```
for (uint8_t rank = 0; rank < 8; rank++) {
    for (uint8_t file = 0; file < 8; file++) {
        BoardTile tile = BoardTile(file, rank);
        Bitboard tile_board = Bitboard(tile);
        uint8_t tile_index = tile.GetIndex();

        …populate arrays for tile…
    }
}
```

Τα leaper πιόνια ανανεώνονται με τον ακόλουθο τρόπο.

```
// Leaper pieces.
pawn_attacks[tile_index] = GetPawnAttacks(tile_board);
king_attacks[tile_index] = GetKingAttacks(tile_board);
knight_attacks[tile_index] = GetKnightAttacks(tile_board);
```

Βρίσκουμε με τις προηγούμενες συναρτήσεις τις κινήσεις για την τρέχουσα θέση και τις αποθηκεύουμε εντός των πινάκων pawn,king,knight attacks με την μορφή Bitboards.

Για τα slider πιόνια βρίσκουμε ολόκληρες τις ακτίνες (δηλαδή τις ακτίνες αν θεωρήσουμε ότι το ταμπλό είναι άδειο) και αφαιρούμε τις θέσεις του εξωτερικού περιγράμματος.

```
// Relevant slider pieces rays. Meaning the ray in an empty board,excluding
outer edges.
bishop_relevant_rays[tile_index] = GetBishopRays(tile) - Masks::outer_tiles;
Bitboard halting_mask = RookHaltingMask(tile); // For cases where the rook is
on an edge.
rook_relevant_rays[tile_index] = GetRookRays(tile) - halting_mask;
```

Για την ειδική περίπτωση που μελετάμε πύργους σε εξωτερικές θέσεις (δηλαδή $file/rank = \{1,8\}$) δεν θέλουμε να αφαιρέσουμε την πλευρά στην οποία βρισκόμαστε. Το πρόβλημα αυτό λύνει η μάσκα halting_mask που παράγεται με τον ακόλουθο τρόπο

```
Bitboard RookHaltingMask(BoardTile tile){
    // If tile is an outer edge we must include said side.
    // If we simply used the outer tiles mask
    // we would get 0 moves which is not the wanted outcome.
    // This is achieved by creating an outer tile mask
    // without the active side. We also include the corners.
    Bitboard mask = Masks::outer_tiles;
    if(Masks::file_A.Get(tile)){
        mask -= Masks::file_A;
    }
    if(Masks::file_H.Get(tile)){
        mask -= Masks::file_H;
    }
    if(Masks::rank_1.Get(tile)){
        mask -= Masks::rank_1;
    }
    if(Masks::rank_8.Get(tile)){
        mask -= Masks::rank_8;
    }
    return mask | Masks::corner_tiles;
}
```

Μελετάμε τις τέσσερις περιπτώσεις, αν βρισκόμαστε σε μια από αυτές αφαιρούμε την ζητούμενη πλευρά. Στο τέλος προσθέτουμε τις γωνίες. (Η μεταβλητή outer_tiles έχει ενεργά όλα τα Bits του εξωτερικού περιγράμματος).

Παρατηρούμε ότι η παραπάνω πληροφορία αποθηκεύεται σε βοηθητικούς πίνακες. Οι πραγματικοί συνδυασμοί από occupancies γίνονται με τις κλήσεις ProduceSubSets

```
auto set_rooks = [=](Bitboard permutation){
    auto key = MagicNumbers::RookMagicHash(permutation, tile_index);
    auto value = GetRookRays(tile, permutation);
    sliding_pieces_attacks[key] = value;
};
auto set_bishops = [=](Bitboard permutation){
    auto key = MagicNumbers::BishopMagicHash(permutation, tile_index);
    auto value = GetBishopRays(tile, permutation);
    sliding_pieces_attacks[key] = value;
};
// Populates slider pieces attack table based on magic numbers for every
permutation.
ProduceSubSets(rook_relevant_rays[tile_index], set_rooks);
ProduceSubSets(bishop_relevant_rays[tile_index], set_bishops);
```

Οι συναρτήσεις lambda set_rooks / set_bishops ευθύνονται για την παραγωγή και αποθήκευση των κινήσεων για τον εκάστοτε συνδυασμό με χρήση των μαγικών αριθμών.

Στο τέλος της όλης διαδικασίας έχουμε την απαιτούμενη πληροφορία εντός των πινάκων.

```
Bitboard pawn_attacks[64];
Bitboard king_attacks[64];
Bitboard knight_attacks[64];

// Does not include outer tiles.
Bitboard bishop_relevant_rays[64];
Bitboard rook_relevant_rays[64];

Bitboard sliding_pieces_attacks[MagicNumbers::permutations] = {};
```

Για αποφυγή global variables και ευκολότερη χρήση παρέχονται public συναρτήσεις

```
Bitboard PawnsAttacks(uint8_t tile_index){
    return pawn_attacks[tile_index];
}

Bitboard KnightAttacks(uint8_t tile_index){
    return knight_attacks[tile_index];
}

Bitboard KingAttacks(uint8_t tile_index){
    return king_attacks[tile_index];
}

Bitboard RookAttacks(uint8_t tile_index, Bitboard occupancies){
    Bitboard rays = rook_relevant_rays[tile_index];
    auto key = MagicNumbers::RookMagicHash(rays & occupancies, tile_index);
    return sliding_pieces_attacks[key];
}

Bitboard BishopAttacks(uint8_t tile_index, Bitboard occupancies){
    Bitboard rays = bishop_relevant_rays[tile_index];
    auto key = MagicNumbers::BishopMagicHash(rays & occupancies, tile_index);
    return sliding_pieces_attacks[key];
}

Bitboard QueenAttacks(uint8_t tile_index, Bitboard occupancies){
    return RookAttacks(tile_index, occupancies) |
           BishopAttacks(tile_index, occupancies);
}
```

Οι κινήσεις των πύργων και των αξιωματικών ανακτώνται με τον συνδυασμό των relevant_rays και των ανάλογων μαγικών αριθμών (2 loads). Οι κινήσεις των βασιλισσών ανακτώνται με την ένωση των κινήσεων του πύργου και του αξιωματικού (4 loads).

Ένα παράδειγμα της παραπάνω διαδικασίας αναλύεται στην ενότητα 2.7.3.

3.6 Παραγωγή κινήσεων

Σε αυτή την ενότητα θα δούμε πως μπορούμε να αξιοποιήσουμε τα Attack tables για την εξαγωγή κινήσεων σε μορφή λίστας. Η διαδικασία χωρίζεται στην παραγωγή ήρεμων κινήσεων και κινήσεων αιχμαλωσίας για ευκολότερη χρήση εντός της συνάρτησης αναζήτησης βέλτιστης κίνησης.

3.6.1 Αναπαράσταση κίνησης

Οι κινήσεις αναπαρίστανται μέσω της κλάσης Move. Η δομή έχει παρόμοια λογική με την κλάση CastlingRights. Αναθέτουμε σε ένα σύνολο από bits τα χαρακτηριστικά που μας ενδιαφέρουν ώστε να έχουμε αποτελεσματική αποθήκευση με μικρή απώλεια ταχύτητας.

Παρατηρούμε ότι κάθε τετράγωνο μπορεί να περιγραφεί από 6 bits. Τα πιόνια μπορούν να κάνουν προαγωγή σε βασίλισσες, αξιωματικούς, άλογα, πύργους άρα θέλουμε το πολύ 3 bits. Γνωρίζοντας την αρχική και την τελική θέση μπορούμε να εξάγουμε κάθε άλλη πληροφορία ελέγχοντας το ταμπλό. Καταλήγουμε στο ότι θέλουμε $2 * 6 + 3 = 15\ bits$ ανά κίνηση.

```
// 6 bits : from
// 6 bits : to
// 4 bits : promotion
uint16_t data_ = 0;
```

Καθώς δεν μπορούμε να έχουμε 15 bits αρκούμαστε με έναν ακέραιο των 16. Μπορούμε να πάρουμε την ζητούμενη πληροφορία με βοηθητικές μάσκες

```
enum Masks{
    From = 0b111111,
    To = 0b111111 << 6,
    Promotion = 0b1111 << 12
};
```

Έτσι καταλήγουμε στις ακόλουθες βασικές συναρτήσεις

```
BoardTile GetFrom() const {return BoardTile(data_ & Masks::From); }
BoardTile GetTo() const {return BoardTile((data_ & Masks::To) >> 6); }
PieceType GetPromotion() const {return static_cast<PieceType>((data_ &
Masks::Promotion) >> 12); }
```

Η συνάρτηση GetPromotion επιστρέφει τον τύπο PieceType

```
enum PieceType{
    None, King, Queen, Bishop, Knight, Rook, Pawn
};
```

Η τιμή None χρησιμοποιείται για κινήσεις που δεν οδηγούν σε προαγωγές. Οι υπόλοιπες συναρτήσεις κωδικοποιούν την πληροφορία εντός ενός BoardTile.

Η λίστα κινήσεων μοντελοποιήται ως ένα vector από Moves.

```
using MoveList = std::vector<Move>;
```

3.6.2 Πιόνια

Τα πιόνια είναι το μόνο είδος υλικού που δεν αξιοποιεί τους πίνακες επιθέσεων στην παραγωγή κινήσεων. Μπορούμε να εξάγουμε όλες τις κινήσεις «παράλληλα» με χρήση ολισθήσεων. Βασιζόμαστε στην ιδέα ότι γνωρίζουμε την θέση προέλευσης αν εφαρμόζουμε μεμονωμένα κάθε κατεύθυνση (Για παράδειγμα αν πάμε μία θέση πάνω ξέρουμε ότι όλα τα πιόνια ήρθαν από μία θέση κάτω).

Η συνάρτηση GetPawnQuietMoves επιστρέφει τις κινήσεις μίας και δύο θέσεων προς τα πάνω συμπεριλαμβάνοντας και τις περιπτώσεις προαγωγής. Παρατηρούμε ότι καθώς το ταμπλό είναι μονοχρωματικό δεν απαιτείται ειδικός χειρισμός ανάλογα το χρώμα.

```
void GetPawnQuietMoves(const Board::Representation& rep, MoveList& move_list)
{
    constexpr int8_t one_back_offset = -8;

    Bitboard enemy = rep.enemy_pieces;

    Bitboard pawns = rep.Pawns() & rep.own_pieces;;
    Bitboard all = rep.own_pieces | enemy;
    Bitboard pawns_up = pawns.ShiftUp1() - all;
    Bitboard promotions = pawns_up & Masks::rank_8;
    Bitboard quiet = pawns_up - promotions;

    // Quiet single push.
    HandleAttacks(one_back_offset, quiet, move_list);
    // Push promotions.
    HandlePromotionAttacks(one_back_offset, promotions, move_list);
    // Double push.
    Bitboard double_push = (pawns_up & Masks::rank_3).ShiftUp1() - all;
    HandleAttacks(2 * one_back_offset, double_push, move_list);
}
```

Αφαιρώντας από την τιμή $pawns.shiftUp1$ την μεταβλητή all απορρίπτουμε κινήσεις πιονιών που είναι μπλοκαρισμένα. Το βήμα αυτό είναι σημαντικό και για τα double pushes ώστε να μην προσπεράσουμε ένα πιόνι με ένα διπλό άλμα (παράνομη κίνηση). Για να δούμε ποια πιόνια είναι ικανά για δύο κινήσεις ή προαγωγή χρησιμοποιούμε τις μάσκες $rank\_3, rank\_8$.

Η συνάρτηση HandleAttacks εντάσσει τις πραγματικές κινήσεις στο move list.

```
void HandleAttacks(uint8_t from_offset, Bitboard moves,  MoveList&
move_list){
    for(auto to : moves){
        BoardTile from = BoardTile(to.GetIndex() + from_offset);
        Move = Move(from, to, PieceType::None);
        move_list.push_back(move);
    }
}
```

Εξάγουμε κάθε bit από το σύνολο κινήσεων του bitboard και κατασκευάζουμε την κίνηση με θέση προέλευσης την τιμή $to + offset$. Στην συγκεκριμένη περίπτωση για να παμε μία θέση πίσω θέλουμε offset -8, ενώ για δύο θέσεις $-8 * 2$.

Η συνάρτηση HandlePromotionAttacks κάνει ακριβώς την ίδια διαδικασία αλλά εντάσσει στην λίστα μία κίνηση ανά είδος προαγωγής (δηλαδή η προαγωγή θεωρείται τέσσερις κινήσεις).

```
auto HandlePromotionAttacks(uint8_t from_offset, Bitboard moves, MoveList&
move_list){
    for(auto to : moves){
        BoardTile from = to + from_offset;
        move_list.push_back(Move(from, to, PieceType::Rook));
        move_list.push_back(Move(from, to, PieceType::Bishop));
        move_list.push_back(Move(from, to, PieceType::Queen));
        move_list.push_back(Move(from, to, PieceType::Knight));
    }
};
```

Η συνάρτηση GetPawnCaptures ευθύνεται για την παραγωγή κινήσεων που κατακτούν αντίπαλα πιόνια (ειδικές περιπτώσεις το enPassant, προαγωγές με αιχμαλώτιση).

```
void GetPawnCaptures(const Board::Representation& rep, MoveList& move_list) {
    constexpr int8_t one_back_offset = -8;
    constexpr int8_t one_right_offset = 1;
    constexpr int8_t one_left_offset = -1;

    // En passant is in ranks 1 and 8.
    Bitboard enemy = rep.enemy_pieces | rep.EnPassant().ShiftDown2();
    Bitboard pawns = rep.Pawns() & rep.own_pieces;

    Bitboard captures_left = pawns.ShiftUp1Left1() & enemy &
                             Masks::not_file_H;
    Bitboard captures_right = pawns.ShiftUp1Right1() & enemy &
                              Masks::not_file_A;
    Bitboard capture_promotion_left = captures_left & Masks::rank_8;
    Bitboard capture_promotion_right = captures_right & Masks::rank_8;
    Bitboard capture_simple_left = captures_left - capture_promotion_left;
    Bitboard capture_simple_right = captures_right - capture_promotion_right;

    // Capture simple left.
    HandleAttacks(one_back_offset + one_right_offset,
                  capture_simple_left,
                  move_list);
    // Capture simple right.
    HandleAttacks(one_back_offset + one_left_offset,
                  capture_simple_right,
                  move_list);
    // Capture promotion left.
    HandlePromotionAttacks(one_back_offset + one_right_offset,
                           capture_promotion_left,
                           move_list);
    // Capture promotion right.
    HandlePromotionAttacks(one_back_offset + one_left_offset,
                           capture_promotion_right,
                           move_list);
}
```

Η όλη διαδικασία γίνεται με παρόμοιο τρόπο, μόνο που αυτή την φορά εξετάζουμε μεμονωμένα τις διαγώνιες κατευθύνσεις με τα αντίστοιχα offsets. Για τον υπολογισμό της αιχμαλωσίας enPassant θεωρούμε το enPassant tile ενεργό αντίπαλο πιόνι. Υπενθυμίζουμε ότι το flag αποθηκεύεται στις σειρές $\{1,8\}$. Από την στιγμή που το ταμπλό ανακλάται σε κάθε γύρο, ξέρουμε ότι μας αφορά πάντα η σειρά 8 και ότι η πραγματική θέση του βρίσκεται 2 θέσεις πιο κάτω.

3.6.3 Ροκέ

Ο βασιλιάς πέρα από τις κανονικές κινήσεις έχει και τις κινήσεις ροκέ. Πριν κάνουμε οτιδήποτε ελέγχουμε αν οι αντίστοιχες προϋποθέσεις τηρούνται. Πιο συγκεκριμένα μας νοιάζει ο βασιλιάς να βρίσκεται στην αρχική του θέση, τα δικαιώματα να είναι μη απενεργοποιημένα και τα ενδιάμεσα τετράγωνα που μεσολαβούν μέχρι και την τελική θέση να είναι κενά.

```
// Castling. King should be at the default position.
// The castling flags will be checked later.
if(king != Masks::king_default)
    return;

// Check if rooks exist at correct tiles and if in between tiles are empty.
Bitboard all = rep.own_pieces | rep.enemy_pieces;
BoardTile from = Masks::king_default;
if(rights.CanOwnQueenSide() &&
(all & Masks::queen_castling_tiles).IsEmpty()){
    BoardTile to = Masks::queen_rook + 2; // 2 tiles right from rook.
    move_list.push_back(Move(from, to, PieceType::None));
}
if(rights.CanOwnKingSide() &&
(all & Masks::king_castling_tiles).IsEmpty()){
    BoardTile to = Masks::king_rook - 1; // Left of rook.
    move_list.push_back(Move(from, to, PieceType::None));
}
```

Οι αντίστοιχες συνθήκες ελέγχονται μέσω της κλάσης CastlingRights και των μασκών queen_castling_tiles / king_castling_tiles. Η τελική θέση περιγράφεται από τα προκαθορισμένα tiles king_rook / queen_took.

Υπενθυμίζουμε ότι σε αυτό το στάδιο δεν μας αφορούν απειλές βασιλιά γιατί παράγουμε ψευδο-κινήσεις.

3.6.4 Υπόλοιπες κινήσεις

Για την εξαγωγή των υπολοίπων κινήσεων χρησιμοποιούμε τους πίνακες επιθέσεων και τις 2 ακόλουθες συναρτήσεις.

```
// Used for basic movement of kings and knights. Does not work with pawns.
template<typename GetAttacks>
void GetLeaperMoves(Bitboard piece, Bitboard exclude, const
Board::Representation& rep, MoveList& move_list, GetAttacks get) {
    for(auto from : piece){
        Bitboard attacks = get(from.GetIndex());
        HandleAttacks(from, attacks - exclude, move_list);
    }
}

// Used for basic movement of slider pieces.
template<typename GetAttacks>
void GetSliderMoves(Bitboard piece, Bitboard exclude, const
Board::Representation& rep, MoveList& move_list, GetAttacks get) {
    Bitboard all = rep.own_pieces | rep.enemy_pieces;
    for(auto from : piece){
        Bitboard attacks = get(from.GetIndex(), all);
        HandleAttacks(from, attacks - exclude, move_list);
    }
}
```

Διατρέχουμε κάθε πιόνι από το δοθέν bitboard set piece, βρίσκουμε το αντίστοιχο bitboard επιθέσεων με χρήση της συνάρτησης get και εισάγουμε τις κινήσεις εντός της λίστας με την συνάρτηση HandleAttacks (Σε σύγκριση με την handle attacks στα πιόνια, τώρα παρέχουμε απευθείας το τετράγωνο και όχι ένα offset). Από τις κινήσεις αφαιρούμε

το σύνολο exclude ώστε να αγνοήσουμε λανθασμένες περιπτώσεις όπως αιχμαλωσίες των δικών μας πιονιών ή και για καλύτερο διαχωρισμό ανά κατηγορία.

Για παράδειγμα στον πύργο οι ήρεμες κινήσεις είναι όλες οι θέσεις εκτός από τα occupancy bits.

```
Bitboard rooks = rep.Rooks() & rep.own_pieces;
GetSliderMoves(rooks, rep.own_pieces | rep.enemy_pieces, rep, move_list,
AttackTables::RookAttacks);
```

Ενώ οι κινήσεις αιχμαλωσίας είναι μόνο τα occupancy bits του αντίπαλου.

```
Bitboard rooks = rep.Rooks() & rep.own_pieces;
GetSliderMoves(rooks, ~rep.enemy_pieces, rep, move_list,
AttackTables::RookAttacks);
```

(Στην συνάρτηση περνάμε το ανάποδο από αυτό που θέλουμε)

Με αντίστοιχο τρόπο λειτουργούν και οι συναρτήσεις για τα υπόλοιπα πιόνια με κύριες διάφορες στις συναρτήσεις AttackTables::Type (στην περίπτωση leaper πιονιών χρησιμοποιείται η GetLeaperMoves).

Για την παραγωγή όλων των κινήσεων για όλα τα είδη υπάρχουν οι συναρτήσεις GetQuietMoves, GetCaptureMoves.

```
void GetQuietMoves(const Board::Representation& representation,
Board::CastlingRights rights, MoveList& move_list){
    GetPawnQuietMoves(representation, move_list);
    GetKnightQuietMoves(representation, move_list);
    GetBishopQuietMoves(representation, move_list);
    GetRookQuietMoves(representation, move_list);
    GetQueenQuietMoves(representation, move_list);
    GetKingQuietMoves(representation, rights, move_list);
}

void GetCaptures(const Board::Representation& representation, MoveList&
move_list){
    GetPawnCaptures(representation, move_list);
    GetKnightCaptures(representation, move_list);
    GetBishopCaptures(representation, move_list);
    GetRookCaptures(representation, move_list);
    GetQueenCaptures(representation, move_list);
    GetKingCaptures(representation, move_list);
}
```

Ο λόγος που προτιμάμε την διαφοροποίηση μεταξύ ήρεμων κινήσεων – κινήσεων αιχμαλωσίας είναι για ευκολότερη ταξινόμηση στον αλγόριθμο PVS.

3.6.5 Έλεγχος εγκυρότητας

Η συνάρτηση IsLegalMove δέχεται μια κίνηση move και επιστρέφει true ή false αναλόγως αν είναι νόμιμη. Με αυτόν τον τρόπο μπορούμε να διατρέξουμε όλες τις ψευδο-κινήσεις και να φιλτράρουμε μόνο αυτές που μας ενδιαφέρουν. Ο πιο απλός έλεγχος μπορεί να γίνει με το παίξιμο της κάθε κίνησης. Αν οδηγηθούμε σε μια κατάσταση που ο βασιλιάς μας βρίσκεται υπό απειλή τότε η κίνηση είναι παράνομη και μπορεί να απορριφθεί. Σε αυτή την ενότητα θα θεωρήσουμε την συνάρτηση παιξίματος κίνησης γνωστή, η περαιτέρω ανάλυση θα γίνει σε ερχόμενο κεφάλαιο.

Για αρχή ας δούμε πώς μπορούμε να ελέγξουμε αν είμαστε υπό απειλή check.

```
bool Board::IsUnderAttack(BoardTile tile) const{
    // Consider the tile a piece with all the possible moves.
    Bitboard own = representation_.own_pieces;
    Bitboard enemy = representation_.enemy_pieces;
    Bitboard all = own | enemy;
    uint8_t tile_index = tile.GetIndex();

    Bitboard king_attacks = AttackTables::KingAttacks(tile_index) &
                            Bitboard(representation_.enemy_king);
    if(!king_attacks.IsEmpty())
        return true;
    Bitboard rook_attacks = AttackTables::RookAttacks(tile_index, all) &
                            enemy & representation_.rook_queens;
    if(!rook_attacks.IsEmpty())
        return true;
    Bitboard bishop_attacks = AttackTables::BishopAttacks(tile_index, all) &
                              enemy & representation_.bishop_queens;
    if(!bishop_attacks.IsEmpty())
        return true;
    Bitboard knight_attacks = AttackTables::KnightAttacks(tile_index) &
                              enemy & representation_.Knights();
    if(!knight_attacks.IsEmpty())
        return true;
    Bitboard pawn_attacks = AttackTables::PawnsAttacks(tile_index) & enemy &
                            representation_.pawns_enPassant;
    if(!pawn_attacks.IsEmpty())
        return true;

    return false;
}
```

Η συνάρτηση IsUnderAttack ακολουθεί την συλλογιστική που περιεγράφηκε στην θεωρία. Αν θεωρήσουμε την εξεταζόμενη θέση ένα πιόνι που μπορεί να πράξει κάθε είδος κίνησης, αρκεί να ελέγξουμε αν επιτιθόμαστε στα αντίστοιχα αντίπαλα είδη πιονιών. Για παράδειγμα αν μπορούμε να επιτεθούμε σε έναν αντίπαλο πύργο, τότε και ο αντίπαλος πύργος επιτίθεται σε εμάς, αν μπορούμε να επιτεθούμε σε ένα αντίπαλο πιόνι τότε και το αντίπαλο πιόνι επιτίθεται εμάς (εδώ χρησιμοποιούνται και οι πίνακες επιθέσεων των απλών πιονιών όπως είχε αναφερθεί).

Στην περίπτωση του check αρκεί να δούμε αν η θέση του βασιλιά είναι υπό επίθεση

```
bool Board::IsInCheck() const{
    return IsUnderAttack(representation_.own_king);
}
```

Η όλη διαδικασία ελέγχου όμως είναι αρκετά χρονοβόρα γιατί χρειάζεται να αντιγράψουμε πρώτα το ταμπλό σε μια ενδιάμεση μεταβλητή. Η αντιγραφή θα μπορούσε να αποφευχθεί με χρήση μιας συνάρτησης UnplayMove αλλά στην συγκεκριμένη υλοποίηση, η δόμηση της κλάσης Move καθιστά την διαδικασία πιο ακριβή (παραπάνω αντιγραφές ακέραιων).

```
auto try_move = [=]() {
    Board temp = *this;
    temp.PlayMove(move);
    return !temp.IsInCheck();
};
```

Σε συγκεκριμένες περιπτώσεις μπορούμε να κάνουμε πιο έξυπνους ελέγχους και η συνάρτηση try_move να χρησιμοποιείται ευλαβικά.

Η βασική ιδέα είναι ότι αν βρισκόμαστε σε μονό check η κίνηση πρέπει να είναι εντός των γραμμών επίθεσης που προκάλεσαν το check (συμπεριλαμβάνοντας και το ίδιο το πιόνι). Στην πράξη ο έλεγχος αυτό δεν προσέφερε μεγάλη βελτίωση για αυτό και για λόγους απλότητας αφαιρέθηκε.

Αν ένα πιόνι είναι «pinned», δηλαδή είναι καρφιτσωμένο στην θέση του γιατί διαφορετικά θα προκαλούσε check, μπορεί να κουνηθεί μόνο προς την κατεύθυνση της επίθεσης. Για τον υπολογισμό του ελέγχου αυτού παράγουμε ένα bitboard με όλα τα πιθανά Pins. Η διαδικασία αυτή γίνεται μία φορά για όλες τις κινήσεις άρα δεν έχουμε μεγάλη απώλεια χρόνου.

```
Bitboard Board::GetPins() const{
    Bitboard pins(0);

    Bitboard own = representation_.own_pieces;
    Bitboard enemy = representation_.enemy_pieces;
    Bitboard bishops = representation_.bishop_queens;
    Bitboard rooks = representation_.rook_queens;
    Bitboard all = own | enemy;

    uint8_t king_index = representation_.own_king.GetIndex();
    Bitboard rook_pins = AttackTables::RookAttacks(king_index, all) & own;
    Bitboard rook_rays = AttackTables::RookAttacks(king_index);
    Bitboard bishop_pins = AttackTables::BishopAttacks(king_index, all)
                           & own;
    Bitboard bishop_rays = AttackTables::BishopAttacks(king_index);

    for(auto piece : rook_pins){
        uint8_t pinned_piece_index = piece.GetIndex();
        auto mask = rook_rays & enemy & rooks;
        Bitboard rook_attacks =
        AttackTables::RookAttacks(pinned_piece_index, all) & mask;
        pins.SetIf(piece, !rook_attacks.IsEmpty());
    }

    for(auto piece : bishop_pins){
        uint8_t pinned_piece_index = piece.GetIndex();
        auto mask = bishop_rays & enemy & bishops;
        Bitboard rook_attacks =
        AttackTables::BishopAttacks(pinned_piece_index, all) & mask;
        pins.SetIf(piece, !rook_attacks.IsEmpty());
    }

    return pins;
}
```

Με την ίδια συλλογιστική του IsUnderAttack παράγουμε τις επιθέσεις του βασιλιά για τις κινήσεις του αξιωματικού και του πύργου. Αν επιτιθόμαστε σε δικό μας πιόνι τότε αρκεί να ελέγξουμε αν αυτό επιτίθεται από την ίδια ακτίνα επίθεσης αντίπαλου πιονιού. Επιπρόσθετα οι ακτίνες πρέπει να είναι συνευθειακές. Για τον λόγο αυτό χρησιμοποιούνται οι μάσκες rook_rays & enemy & rooks, bishop_rays & enemy & bishop.

Η διαδικασία πρέπει να γίνει μεμονωμένα για κάθε είδος κίνησης γιατί διαφορετικά ο έλεγχος διαγώνιων και κάθετων κινήσεων οδηγεί σε λανθασμένους ελέγχους.

Έτσι σε περιπτώσεις που δεν βρισκόμαστε σε check και το πιόνι που εξετάζουμε είναι Pinned, αρκεί να ελέγξουμε αν το διάνυσμα κίνησης $\{from, to\}$ είναι συνευθειακό με το διάνυσμα $\{king, to\}$ ή $\{king, from\}$. Ο έλεγχος πραγματοποιείται με σύγκριση των κλίσεων.

```
if(pins.Get(from)) {
    uint8_t from_rank = from.GetRank();
    uint8_t to_rank = to.GetRank();
    auto[king_file, king_rank] = representation_.own_king.GetCoords();

    const int dx_from = from_file - king_file;
    const int dy_from = from_rank - king_rank;
    const int dx_to = to_file - king_file;
    const int dy_to = to_rank - king_rank;

    // The move is legal only if the vector {from-king} is codirectional with
    // the vector {to-king}. This is checked by comparing the slopes.
    // dy_from / dx_from = dy_to / dx_to.
    if (dx_from == 0 || dx_to == 0) {
        return (dx_from == dx_to);
    } else {
        return (dx_from * dy_to == dx_to * dy_from);
    }
}
```

Με αυτόν τον τρόπο γλιτώνουνε αρκετούς άσκοπους ελέγχους καθώς τις περισσότερες φορές τα πιόνια που εξετάζουμε δεν είναι Pinned και δεν είμαστε σε κατάσταση check άρα μπορούμε απευθείας να επιστρέψουμε την τιμή true μετά τον έλεγχο την συνθήκης pins.Get(from).

```
if(is_enPassant)
    return try_move();

if(is_in_check){
    if(is_castling)
        return false;
    return try_move();
}

if(is_king){
    // Check in between tile.
    if(is_castling && IsUnderAttack(from + (to_file - from_file) / 2))
        return false;
    return try_move();
}
```

Για τις υπόλοιπες περιπτώσεις αν έχουμε check και προσπαθούμε να κάνουμε ροκέ τότε επιστρέφουμε απευθείας false. Αν προσπαθούμε να κουνήσουμε τον βασιλιά, δεν είμαστε σε check και προσπαθούμε να κάνουμε ροκέ ελέγχουμε ότι δεν είναι υπό επίθεση η ενδιάμεση θέση. Διαφορετικά ελέγχουμε αναλυτικά την κίνηση μέσω της try_move().

Η λίστα κινήσεων μπορεί να παραχθεί μέσω των κλήσεων GetLegalQuietMoves, GetLegalCapturesMoves.

```
MoveList Board::GetLegalQuietMoves(Bitboard pins, bool is_in_check) const {
    MoveList moves;
    moves.reserve(60);
    PseudoMoves::GetQuietMoves(representation_, castling_rights_, moves);

    auto is_illegal = [=](const Move &move)
    { return !IsLegalMove(move, pins, is_in_check); };
    moves.erase(std::remove_if(moves.begin(), moves.end(), is_illegal),
                moves.end());

    return moves;
}
```

(Τα Pins παράγονται ένα επίπεδο πιο πάνω ώστε να παρέχονται και στις δύο συναρτήσεις)

3.6.6 Υπολογισμός τιμών Perft

Η συνάρτηση Perft είναι αρκετά τετριμμένη καθώς υλοποιεί σχεδόν ένα προς ένα τον βασικό ψευδο-κώδικα.

```
int Perft(const Board& board, int depth) {
    int nodes = 0;

    if (depth == 0)
        return 1ULL;

    Bitboard pins = board.GetPins();
    bool is_in_check = board.IsInCheck();
    MoveList moves = board.GetLegalCaptures(pins, is_in_check);
    MoveList quiet_moves = board.GetLegalQuietMoves(pins, is_in_check);
    moves.insert( moves.end(), quiet_moves.begin(), quiet_moves.end());
    for (const Move& move : moves) {
        Board temp = board;
        temp.PlayMove(move);
        temp.Mirror();

        nodes += Perft(temp, depth - 1);
    }

    return nodes;
}
```

Χρησιμοποιήθηκε εκτενώς για τον έλεγχο ορθότητας των συναρτήσεων παραγωγής κινήσεων με χρήση των αποτελεσμάτων perft.

https://www.chessprogramming.org/Perft_Results

3.7 Παίξιμο κίνησης

Το παίξιμο μιας κίνησης αποτελεί μια συνθέτη διαδικασία. Σε αυτή την ενότητα θα γίνει μια συνοπτική αναφορά των κυρίων διεργασιών και όχι πλήρης ανάλυση του κώδικα.

3.7.1 Κλειδιά Zobrist και ιστορικό κινήσεων

Για την παραγωγή των κλειδιών Zobrist αρχικοποιούμε στην αρχή του προγράμματος ένα σύνολο από πίνακες

```
void InitZobristKeysArrays(){
    std::random_device rd;
    std::mt19937_64 e2(rd());
    std::uniform_int_distribution<long long int> dist(0, UINT64_MAX);
    auto rand64 = [&] () { return dist(e2); };

    for (int type = 0; type < 12; type++) {
        for (int square = 0; square < 64; square++) {
            piece_square_key[type][square] = rand64();
        }
    }

    for (int enPassant = 0; enPassant < 8; enPassant++) {
        enPassant_key[enPassant] = rand64();
    }

    for (int castling = 0; castling < 16; castling++) {
        castling_key[castling] = rand64();
    }

    black_side_key = rand64();
}
```

Καθώς η συνάρτηση rand() δεν παράγει τυχαίους αριθμούς των 64 bit, χρησιμοποιούμε την πιο συνθέτη δομή uniform_int_distribution.

Για την αρχική αρχικοποίηση του κλειδιού χρησιμοποιείται η συνάρτηση GetZobristKey που δέχεται ένα στιγμιότυπο της δομής Board.

```
uint64_t key = 0;
auto update_key = [&] (Bitboard piece_board, PieceType type){
    for (auto piece : piece_board) {
        bool is_white = rep.own_pieces.Get(piece);
        key ^= GetPieceSquareKey(type, is_white, piece.GetIndex());
    }
};

update_key(rep.Pawns(), PieceType::Pawn);
update_key(rep.Knights(), PieceType::Knight);
update_key(rep.Bishops(), PieceType::Bishop);
update_key(rep.Rooks(), PieceType::Rook);
update_key(rep.Queens(), PieceType::Queen);
update_key(rep.Kings(), PieceType::King);
```

Η διαδικασία είναι αντίστοιχη με αυτή που περιγράφεται στην θεωρία. Για κάθε χαρακτηριστικό εφαρμόζουμε τον τελεστή xor με τον εκάστοτε τυχαίο αριθμό. (Στον παραπάνω κώδικα φαίνεται το κομμάτι που αφορά μόνο τις θέσεις πιονιών).

Στην συνέχεια του προγράμματος μεταβάλλουμε το κλειδί με επιπρόσθετες πράξεις xor με τους παραγόμενους τυχαίους αριθμούς εντός των πινάκων.

```
static uint64_t piece_square_key[16][64];
static uint64_t enPassant_key[8];
static uint64_t castling_key[16];
static uint64_t black_side_key;
```

Κάθε φορά που παίζουμε μια κίνηση, αποθηκεύουμε το κλειδί του ταμπλό μαζί με την πληροφορία Progress_made σε έναν πίνακα εντός της κλάσης History. Η μεταβλητή αυτή είναι χρήσιμη για τον αποτελεσματικό έλεγχο του κανόνα επαναλαμβανομένης θέσης. Η πληροφορία ανακτάται βάση της μετρικής ply.

```
class History {
public:
    struct Element {
        uint64_t key; // Zobrist key
        bool progress_made; // If capture or pawn move.
    };

    static History& Instance(){
        static History;
        return history;
    }

    void AddState(uint8_t ply, Element element){
        data_[ply] = element;
    }

    Element GetState(uint8_t ply){
        return data_[ply];
    }

private:
    History() = default;
    std::array<Element, HISTORY_SIZE> data_;
};
```

3.7.2 Δομή Dirty piece

Η βιβλιοθήκη NNUE δέχεται μια μεταβλητή τύπου DirtyPiece όπου περιέχει την απαραίτητη πληροφορία για την μεταβολή των βαρών του 1ου Layer του δικτύου. Όπως για τα κλειδιά Zobrist ενημερώνουμε το κλειδί βάσει της επιλεγόμενης κίνησης έτσι και για τα dirty pieces μεταβάλλουμε την εσωτερική πληροφορία του αντίστοιχου struct.

```
typedef struct DirtyPiece {
  int dirtyNum;
  int pc[3];
  int from[3];
  int to[3];
} DirtyPiece;
```

Η μεταβλητή dirtyNum αναφέρεται στο μέγιστο πλήθος αλλαγών (μπορούμε να έχουμε το πολύ 4). Ο πίνακας pc αναφέρεται στα είδη πιονιών που μεταλλάχθηκε η κατάσταση τους στην τελευταία κίνηση. Ο πίνακας from χαρακτηρίζει την αρχική τους θέση ενώ ο πίνακάς to την τελική.

Όταν ένα πιόνι διαγράφεται θεωρούμε ότι η τελική του θέση είναι η καθολική σταθερά REMOVED_SQUARE. Αντίστοιχα όταν ένα πιόνι γίνεται promote θεωρούμε ότι έχουμε δύο κινήσεις, μια διαγραφή και μια εμφάνιση με from = REMOVED_SQUARED. Οι κωδικοποιήσεις θέσεων και είδους πιονιών είναι διαφορετικές από αυτές που ακολουθεί η δικιά μας σκακιστική μηχανή για αυτό και παρέχονται «wrapper» συναρτήσεις για τις αντίστοιχες μετατροπές.

```
// Converts engine's types to the required input of the NNUE probe lib.
static int GetSideEncoding(const Team& team);
static int GetSideEncoding(bool is_flipped);
static int GetPieceEncoding(const PieceType& type, const Team& team);
static int GetPieceEncoding(PieceInfo pieceInfo);
static int GetPieceEncoding(const PieceType& type, bool is_flipped);
static int GetSquareEncoding(const BoardTile& tile);
static int GetSquareEncoding(BoardTile tile, bool is_flipped);
```

Τα dirty pieces variables αποθηκεύονται εντός ενός πίνακα NNUEData με βάση τον τρέχοντα αριθμό κίνησης (ply count).

3.7.3 Προσαρμογή ταμπλό

Το ταμπλό ανανεώνεται με την συνάρτηση PlayMove, όπου χειρίζεται την αλλαγή των μετρητών, αποθήκευση των dirty piece, μεταβολή της bitboards αναπαράστασης, του κλειδιού Zobrist και των δικαιωμάτων ροκέ. Καθώς η κλάση Move δεν παρέχει πληροφορίες για το είδος πιονιού που αναφερόμαστε ή και το είδος της κίνησης, πρέπει με αναλυτικό τρόπο να ελέγξουμε το ήδη υπάρχον ταμπλό για την εξαγωγή επιπλέον ιδιοτήτων.

Για παράδειγμα άμα στην θέση *to* υπάρχει αντίπαλο πιόνι, τότε επιχειρούμε να κάνουμε αιχμαλώτιση.

```
if(representation_.enemy_pieces.Get(to)){
    dirty_piece->dirtyNum = 2;
    auto[enemy_piece_type, enemy_team] = GetPieceInfoAt(to);
    dirty_piece->pc[1] = NNUE::GetPieceEncoding(enemy_piece_type,
                                                enemy_color);
    dirty_piece->from[1] = to_normalised_index;
    dirty_piece->to[1] = REMOVED_SQUARE;

    // Incremental update on zobrist key when capturing enemy piece.
    zobrist_key_ ^= Zobrist::GetPieceSquareKey(enemy_piece_type, is_flipped_,
                                               to_normalised_index);
}
```

Ανανεώνουμε το dirty piece για το αιχμαλωτισμένο πιόνι και αφαιρούμε από το κλειδί την κωδικοποίηση του.

Άμα η κίνηση προέρχεται από τις θέσεις των πύργων, ξέρουμε ότι ένα από τα δικαιώματα ροκέ μπορεί να απενεργοποιηθεί. (Διπλότυπες απενεργοποιήσεις δεν μας πειράζουν)

```
// A Castling right is reset if a rook moves. We can reset the rights
regardless of the piece type.
else if(from == Masks::queen_rook) {
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
    castling_rights_.ResetOwnQueenSide();
    // Incremental update on zobrist key when castling rights change.
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
} else if(from == Masks::king_rook) {
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
    castling_rights_.ResetOwnKingSide();
    // Incremental update on zobrist key when castling rights change.
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
}
```

Αντίστοιχα άμα η τελική θέση είναι μια από τις αρχικές θέσεις αντίπαλων πύργων, απενεργοποιούμε τα δικαιώματα του αντίπαλου.

```
// If an enemy rook is captured, reset castling rights.
const uint8_t enemy_rooks_offset = 7 * 8;
if(to == Masks::queen_rook + enemy_rooks_offset) {
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
    castling_rights_.ResetEnemyQueenSide();
    // Incremental update on zobrist key when enemy rook is captured due to
castling rights change.
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
} else if(to == Masks::king_rook + enemy_rooks_offset) {
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
    castling_rights_.ResetEnemyKingSide();
    // Incremental update on zobrist key when enemy rook is captured due to
castling rights change.
    zobrist_key_ ^= Zobrist::GetCastlingKey(castling_rights_, is_flipped_);
}
```

Άμα προσπαθούμε να κουνήσουμε τον βασιλιά δύο θέσεις μακριά από την αρχική του θέση τότε προσπαθούμε να κάνουμε ροκέ. Το ροκέ αποτελείται από δύο μεταβολές πιονιών καθώς συμπεριλαμβάνει και την κίνηση του πύργου.

```
bool is_castling = abs(from_file - to_file) == 2;
bool is_queen_side = is_castling && to == (Masks::queen_rook + 2);
bool is_king_side = is_castling && to == (Masks::king_rook - 1);
if(is_queen_side)
    castling(Masks::queen_rook, Masks::queen_rook + 3);
else if(is_king_side)
    castling(Masks::king_rook,  Masks::king_rook - 2);
```

Η ανανέωση των bitboards γίνεται με τον ακόλουθο τρόπο. Πρώτα αφαιρούμε τα πιόνια από την θέση to. (Ακόμα και όταν δεν υπάρχουν)

```
// Reset captured pieces.
representation_.rook_queens.Reset(to);
representation_.bishop_queens.Reset(to);
representation_.pawns_enPassant.Reset(to);
```

Κάνουμε Set / Reset τα global occupancies και τις θέσεις προέλευσης.

```
// Reset, set global occupancies.
representation_.own_pieces.Reset(from);
representation_.own_pieces.Set(to);
representation_.enemy_pieces.Reset(to);
...
// Reset from.
representation_.rook_queens.Reset(from);
representation_.bishop_queens.Reset(from);
representation_.pawns_enPassant.Reset(from);
```

(Για αποφυγή branches κάνουμε reset όλα τα bitboards χωρίς να παράγουμε λανθασμένο αποτέλεσμα)

Τέλος ανανεώνουμε τις θέσεις προορισμού με την συνάρτηση setif.

```
// Set bitboard if it is of type [from].
bool is_promo = promotion != PieceType::None;
bool is_rook_queen = representation_.rook_queens.Get(from);
bool is_bishop_queen = representation_.bishop_queens.Get(from);
bool is_pawn = representation_.pawns_enPassant.Get(from);
representation_.rook_queens.SetIf(to, is_rook_queen);
representation_.bishop_queens.SetIf(to, is_bishop_queen);
representation_.pawns_enPassant.SetIf(to, is_pawn && !is_promo);
```

Η συνάρτηση setIf μεταβάλει το αποτέλεσμα μόνο αν η συνθήκη είναι αληθής.

```
void Bitboard::SetIf(BoardTile tile, bool cond) {
    data_ |= std::uint64_t(cond) << tile.GetIndex();
}
```

Άμα δεν έχει γίνει κάποια κίνηση απλού πιονιού ή αιχμαλώτιση αυξάνουμε τον μετρητή των 50 κινήσεων.

```
// Increments only if there were no captures or pawn moves.
if(reset_50_move_rule)
    move_counters_.half_moves = 0;
else
    move_counters_.half_moves++;
```

Η παραπάνω διαδικασία γίνεται με αυστηρή σειρά εκτέλεσης. Για παράδειγμα άμα διαγράψουμε το bit προέλευσης δεν μπορούμε να ξέρουμε το είδος του πιονιού άμα δεν έχει πραγματοποιηθεί ήδη ο έλεγχος.

Για λόγους απλότητας δεν αναλύθηκε κάθε τεχνική λεπτομέρεια ούτε ειδικές περιπτώσεις όπως οι κινήσεις enPassant / προαγωγή. Παρ'ολ'αυτά ο υπόλοιπος κώδικας συμπεριφέρεται με παρόμοιο τρόπο όσο αναφορά τον χειρισμό των τιμών bitboards / dirty piece / κλειδιού Zobrist / ιστορικό / μετρητές.

Η συνάρτηση PlayNullMove επίσης έχει παραπλήσια απλοποιημένη δομή καθώς χειρίζεται μόνο τους μετρητές και το enPassant flag.

3.8 Τερματισμός παιχνιδιού

Το παιχνίδι τερματίζει είτε λόγω ισοπαλίας είτε λόγο checkmate. Ο έλεγχος γίνεται μετά από κάθε κίνηση με την συνάρτηση Result που επιστρέφει μια τιμή τύπου Result.

```
enum class GameResult{
    WhiteWon, BlackWon, Draw, Playing
};
```

Άμα δεν έχουμε νόμιμες κινήσεις και είμαστε σε check τότε έγινε checkmate, αλλιώς stalemate.

```
if(moves.empty()){
    if(IsInCheck()){
        // checkmate.
        return (is_flipped_) ? GameResult::WhiteWon : GameResult::BlackWon;
    }else{
        // stalemate.
        return GameResult::Draw;
    }
}
```

Μπορούμε να βρούμε τον νικητή βάσει της μεταβλητής is_flipped (εξετάζουμε το ταμπλό πάντα από την πλευρά του ηττημένου μιας και έχει γίνει mirrored).

Άμα δεν υπάρχουν αρκετά πιόνια για να είναι δυνατόν να γίνει checkmate τότε έχουμε ισοπαλία.

```
bool Board::InsufficientMaterial() const{
    // Pawns can promote.
    if(!representation_.pawns_enPassant.IsEmpty())
        return false;
    // Rook or queens can lead to a checkmate.
    if(!representation_.rook_queens.IsEmpty())
        return false;

    // Can not check with 1 minor piece.
    if((representation_.own_pieces | representation_.enemy_pieces).Count()
        < 4)
        return true;

    // More than 3 pieces exist that are not rook or queens.
    if(!representation_.Knights().IsEmpty())
        return false;

    // Only bishops remain. If they are same color it's a draw.
    Bitboard light_bishops = representation_.bishop_queens &
                             Masks::light_squares;
    Bitboard dark_bishops = representation_.bishop_queens &
                            Masks::dark_squares;
    return light_bishops.IsEmpty() || dark_bishops.IsEmpty();
}
```

Στην περίπτωση των αξιωματικών μας ενδιαφέρει και το αν είναι ιδίου χρώματος.

```
 // not enough pieces.
if(InsufficientMaterial())
    return GameResult::Draw;
```

Άμα έχουν γίνει πάνω από 50 κινήσεις χωρίς πρόοδο (δηλαδή όχι κινήσεις πιονιών ή αιχμαλωσίες) είναι ισοπαλία.

```
// 50 moves rule.
if(move_counters_.half_moves >= 100)
    return GameResult::Draw;
```

Στον κώδικα ελέγχουμε με την τιμή 100 μιας και δεν συγκρίνουμε με το πλήθος των «ολόκληρων» κινήσεων.

Άμα η τωρινή θέση έχει επαναληφθεί ήδη δύο φορές τότε είναι ισοπαλία.

```
// 3 move repetition.
// If positions has occurred 2 more times.
if(move_counters_.repetitions >= 2)
    return GameResult::Draw;
```

Η μεταβλητή repetitions υπολογίζεται με τον ακόλουθο τρόπο.

```
int Board::StateRepetitions(uint64_t zobrist_key, uint8_t ply) const{
    // If currently made progress, no repetition.
    // Ply should be equal to the index of the last played move.
    if(History::Instance().GetState(ply).progress_made)
        return 0;

    int repetitions = 0;
    for (int i = ply - 2; i >= 0; i-=2) {
        if(History::Instance().GetState(i+1).progress_made)
            break;
        auto state = History::Instance().GetState(i);
        if(state.progress_made)
            break;
        if(state.key == zobrist_key) {
            repetitions++;
        }
    }

    return repetitions;
}
```

Ελέγχουμε κάθε κλειδί από το ιστορικό με το κλειδί της τωρινής θέσης. Πάντα πάμε δύο θέσεις προς τα πίσω γιατί στην σειρά του αντίπαλου σίγουρα δεν έχουμε επαναλαμβανομένη θέση. Άμα σε οποιαδήποτε σημείο του πίνακα πετύχουμε την μεταβλητή progress_made = true τότε σταματάμε την αναζήτηση. Σε αυτό το σημείο πρέπει να αναφερθεί ότι η κλάση History υλοποιεί το singleton pattern για αυτό και απαιτείται η κλήση Instance.

Άμα μια από τις παραπάνω συνθήκες δεν ισχύει το παιχνίδι συνεχίζεται κανονικά.

```
return GameResult::Playing;
```

3.9 Εύρεση βέλτιστης κίνησης

Για την εύρεση βέλτιστης κίνησης χρησιμοποιείται ο αλγόριθμος PVS σε συνδυασμό με ένα δίκτυο NNUE για αξιολόγηση φύλλων κόμβων.

Επιλέχθηκε αυτή η προσέγγιση καθώς:

- Δεν απαιτείται η χρήση της GPU κατά την εκτέλεση του αλγορίθμου.
- Υπάρχουν έτοιμες βιβλιοθήκες για χρήση NNUE μοντέλων.
- Υπάρχουν pre-trained NNUE μοντέλα.
- Οι καλύτερες σκακιστικές μηχανές ακολουθούν μεθοδολογίες τύπου DFS (stockfish).

Αντιθέτως για την προσέγγιση PUCT δεν υπάρχουν βιβλιοθήκες για χρήση των μοντέλων, παρόλο που παρέχονται τα βάρη των δικτύων. Αυτό θα σήμαινε ότι θα έπρεπε να προσαρμοστεί ένα μεγάλο κομμάτι open source κώδικα από το lc0 που δεν είναι φτιαγμένο για χρήση εντός άλλων σκακιστικών μηχανών (Τα μοντέλα δεν χρησιμοποιούν γνωστά APIs αλλά δίκες τους cuda υλοποιήσεις για πιο γρήγορο inference).

Η εκπαίδευση ενός δικτύου από το μηδέν επίσης δεν ήταν εφικτή λύση λόγω περιορισμένων υπολογιστικών πόρων (η σκακιστική μηχανή lc0 χρειάστηκε ~14 εκατομμύρια παιχνίδια ενώ το stockfish παραπλήσιο πλήθος θέσεων).

Παρ'ολ'αυτα και οι δύο μέθοδοι παράγουν πολύ ισχυρές σκακιστικές μηχανές. Το lc0 και το stockfish ανταγωνίζονται συνεχώς για την πρώτη θέση στο ετήσιο τουρνουά TCEC.

3.9.1 Επαναληπτική εμβάθυνση

Η συνάρτηση GetBestMove υλοποιεί τον αλγόριθμο επαναληπτικής εμβάθυνσης με την προσαρμογή των aspiration windows.

```
Move GetBestMove(const Board& board, int depth, int& eval_result){
    // Iterative deepening.
    int a = 2 * INT16_MIN;
    int b = 2 * INT16_MAX;
    Move best_move;
    for (int current_depth = 1; current_depth <= depth; current_depth++) {
        // Using 16 bits because 32 overflows.
        int eval = PVSearch(board, current_depth, 0, a, b, best_move, true);
        eval_result = eval;

        // Aspiration search
        if(eval <= a || eval >= b) {
            a = 2 * INT16_MIN;
            b = 2 * INT16_MAX;
            current_depth--;
            continue;
        }

        a = eval - 100;
        b = eval + 100;
    }

    return best_move;
}
```

Οι τιμές $INT16\_MIN, INT16\_MAX$ προσομοίων τις ποσότητες $-\infty, +\infty$. Η τιμη depth σηματοδοτεί το μέγιστο πλήθος βάθους εξερεύνησης. Η συνάρτηση επιστρέφει την βέλτιστη κίνηση και την αντίστοιχη αξιολόγηση μέσω της μεταβλητής eval_result. Κάθε φορά που η αναζήτηση PVS αποτυγχάνει, επαναλαμβάνουμε την διαδικασία με πλήρες μέγεθος παραθύρου ανανεώνοντας τις τιμές a,b και current_depth.

Στην τελική υλοποίηση η aspiration αναζήτηση έχει απενεργοποιηθεί γιατί έδινε χειροτέρους χρόνους.

3.9.2 PVS

Η συνάρτηση PVSearch ευθύνεται για την αναζήτηση του δέντρου καταστάσεων. Η συνθήκη τερματισμού της αναδρομής είναι η τιμή depth να γίνει μηδενική. Σε αυτή την περίπτωση καλούμε την συνάρτηση QSearch για πιο σταθερή αξιολόγηση.

```
if (depth <= 0) {
    return QSearch(board, a, b);
}
```

Διαφορετικά αρχίζουμε την παραγωγή κινήσεων για την δοθείσα θέση, μέσω της μεταβλητής Board.

```
// Move ordering.
MoveList moves = board.GetLegalCaptures(pins, is_in_check);
SortMoves(board, moves);
MoveList quiet_moves = board.GetLegalQuietMoves(pins, is_in_check);
moves.insert(moves.end(), quiet_moves.begin(), quiet_moves.end());
```

Οι κινήσεις διαχωρίζονται σε ήρεμες / αιχμαλωσίες ώστε να μπορέσουμε να ταξινομήσουμε μόνο το υποσύνολο που μας αφορά.

```
void SortMoves(const Board& board, MoveList& moves){
    auto move_score = [=] (const Move& move){
        PieceType type_own, type_enemy;
        {
            auto[file_from, rank_from] = move.GetFrom().GetCoords();
            type_own = board.GetPieceTypeAt(file_from, rank_from);
            auto[file_to, rank_to] = move.GetTo().GetCoords();
            type_enemy = board.GetPieceTypeAt(file_to, rank_to);
        }
        return GetMVVScore(type_own, type_enemy);
    };

    std::stable_sort(moves.begin(), moves.end(),
                     [=](const Move& mv1, const Move& mv2){
        return move_score(mv1) > move_score(mv2);
    });
}
```

Η ταξινόμηση γίνεται με κριτήριο το MVV score, το οποίο υλοποιείται με απλά switch cases και hardcoded τιμές. Στο τέλος οι δύο λίστες συγχωνεύονται σε μία.

Προτού συνεχίσουμε την διαδικασία αναζήτησης πρέπει να δούμε αν το παιχνίδι έχει ήδη τελειώσει. Ο έλεγχος αυτός προϋποθέτει την ύπαρξη της λίστας κινήσεων για τις συνθήκες stalemate / checkmate.

```
// Draw / Checkmate detection.
GameResult game_result = board.Result(moves);
static constexpr int checkmate_score = INT16_MAX;
switch(game_result){
    // Terminal node.
    case GameResult::WhiteWon:
    case GameResult::BlackWon:
        // The board is flipped. If the game is over
        // the current side lost. We return relative to
        // the current side hence the score is negative.
        return -(checkmate_score + depth);
    case GameResult::Draw:
        return 0;
    case GameResult::Playing:
        break;
}
```

Στην περίπτωση ισοπαλίας, επιστρέφουμε μηδενική τιμή. Διαφορετικά θέλουμε να αξιολογήσουμε την θέση πιο ψηλά από κάθε άλλη κίνηση. Επειδή πάντα κοιτάμε το ταμπλό από την σκοπιά του ηττημένου, το μέγεθος αυτό πρέπει να είναι αρνητικό. Για να βρίσκουμε το γρηγορότερο checkmate, αθροίζουμε και την τιμή depth.

Αν το παιχνίδι δεν έχει τελειώσει συνεχίζουμε με μια σειρά από βελτιστοποιήσεις.

Εφαρμόζουμε το static null move pruning μόνο αν είμαστε σε pv κόμβο, δεν βρισκόμαστε σε κατάσταση check και η τιμή b είναι μικρότερη από το checkmate score.

Γνωρίζουμε ότι βρισκόμαστε σε pv κόμβο αν το παράθυρο a-b είναι μήκους 1.

```
bool is_pv_node = b - a != 1;

// Static Null move pruning.
if(!is_in_check && !is_pv_node && abs(b) < checkmate_score){
    static int static_null_move_pruning_base_margin = 120;
    int static_score = NNUE::Instance().EvaluateIncremental(board);
    int score_margin = static_null_move_pruning_base_margin * depth;
    if(static_score - score_margin >= b){
        return b;
    }
}
```

Αν το τωρινό score μείον μια ποσότητα margin εξακολουθεί να είναι μεγαλύτερο από το b, τότε σταματάμε την περαιτέρω αναζήτηση του κόμβου. (Οι τιμές των margins είναι εμπειρικές)

Αν ο έλεγχος δεν πραγματοποιήθηκε ή ήταν ανεπιτυχής συνεχίζουμε με το null move pruning.

```
// Null move pruning.
if(do_null && !is_in_check && !is_pv_node && depth >= 3){
    int R = 2;
    Board new_board = Board(board);
    new_board.PlayNullMove();
    new_board.Mirror();
    int score = -PVSearch(new_board, depth - R - 1, ply + 1, -b, -b +1,
                          best_move, false);
    if(score >= b && abs(score) < checkmate_score){
        return b;
    }
}
```

Πραγματοποιούμε μια κενή κίνηση και ελέγχουμε αν το score μιας μικρότερης αναζήτησης θα οδηγούσε σε αλλαγή της βέλτιστης κίνησης. Η ποσότητα R ευθύνεται για την μείωση του βάθους. Η μεταβλητή do_null γίνεται false στην επόμενη αναδρομική κλήση ώστε να αποφύγουμε εφαρμογές πολλαπλών συνεχομένων null moves. Όπως και προηγουμένως πρέπει να προσέξουμε η τωρινή κατάσταση να μην βρίσκεται σε check και να μην είναι pv κόμβος. Ο έλεγχος πραγματοποιείται μόνο για βάθη πάνω από τρία.

Το futility pruning εφαρμόζεται αν ισχύουν πάλι παρόμοιες προϋποθέσεις και το βάθος είναι μικρότερο του 8.

```
// Futility pruning.
bool can_futility_prune = false;
if(depth <= 8 && !is_pv_node && !is_in_check && a < checkmate_score) {
    static int futility_margins[] =
    {0, 100, 160, 220, 280, 340, 400, 460, 520};
    int static_score = NNUE::Instance().EvaluateIncremental(board);
    if(static_score + futility_margins[depth] <= a){
        can_futility_prune = true;
    }
}
```

Τα αντίστοιχα margins αυξάνονται με βάση το βάθος αναζήτησης. Αυτή την φορά δεν επιστρέφουμε κατευθείαν κάποια τιμή αλλά μπορούμε να αποκόψουμε ένα μεγάλο πλήθος κινήσεων στην συνέχεια.

Η κύρια επανάληψη της συνάρτησης διατρέχει κάθε παραγόμενη θέση με την ταξινομημένη σειρά και την εφαρμόζει σε ένα αντίγραφο της τωρινής κατάστασης.

```
Board new_board = Board(board);
new_board.PlayMove(move);
new_board.Mirror();
```

Σε κάθε επόμενη αναδρομική κλήση παρέχουμε την μεταβλητή new_board σαν νέα κατάσταση.

Κινήσεις που τηρούν τις προηγούμενες προϋποθέσεις και δεν αποτελούν εμφανείς τακτικές μπορούν να αγνοηθούν με την θεώρηση ότι η ταξινόμηση των κινήσεων μας είναι καλή.

```
// Late move pruning.
static int late_move_pruning_margins[] = {0, 8, 12, 24};
if(depth <= 3 && !is_pv_node && !is_in_check && moves_played >
late_move_pruning_margins[depth]){
    bool tactical = new_board.IsInCheck() || move.GetPromotion() != None;
    if(!tactical){
        continue;
    }
}

// Futility pruning.
if(can_futility_prune && moves_played > 1){
    bool tactical = new_board.IsInCheck() || move.GetPromotion() != None ||
            board.GetRepresentation().enemy_pieces.Get(move.GetTo());
    if(!tactical){
        continue;
    }
}
```

Και στις 2 περιπτώσεις ελέγχουμε ότι έχει εξεταστεί ήδη ένα σταθερό πλήθος κινήσεων. (δηλαδή δεν κλαδεύονται οι πρώτες κινήσεις). Το futility pruning πραγματοποιείται μόνο

άμα η μεταβλητή can_futility_prune είχε ενεργοποιηθεί από τον έλεγχο που εξετάστηκε προηγουμένως εκτός της επανάληψης.

Σε οποιαδήποτε άλλη περίπτωση πρέπει να κάνουμε αναλυτικό έλεγχο της εξεταζόμενης κίνησης.

```
int score;
if (pv_search) {
    score =
    -PVSearch(new_board, depth - 1, ply + 1, -b, -a, best_move, true);
} else {
    score =
    -PVSearch(new_board, depth - 1, ply + 1, -a - 1, -a, best_move, true);
    if (score > a && score < b) {
        score =
        -PVSearch(new_board, depth - 1, ply + 1, -b, -a, best_move, true);
    }
}
```

Αν η πρώτη κίνηση έχει ήδη εξεταστεί, τότε οι επόμενες αναδρομικές κλήσεις γίνονται με μειωμένο παράθυρο όπως αναλύθηκε και στην θεωρία.

Ο παρακάτω κώδικας ευθύνεται για την προσαρμογή των τιμών a,b και τις αντίστοιχες αποκοπές της αναζήτησης ab.

```
if(score > best_score){
    best_score = score;
    current_best_move = move;
}
if(score >= b) {
    break;
}
if(score > a) {
    pv_search = false;
    a = score;

    if(is_root)
        best_move = move;
}
```

Στην περίπτωση που βρισκόμαστε στην αρχική κλήση, θέλουμε να ανανεώνουμε και την μεταβλητή best_move ώστε να ανταποκρίνεται στην κίνηση με το μεγαλύτερο score. Στην περίπτωση που εξετάσουμε κάθε κίνηση και δεν έχει προκύψει κάποια αποκοπή τότε επιστρέφουμε το best_score.

3.9.3 Πίνακας μεταθέσεων

Η προσθήκη του πίνακα μεταθέσεων επηρεάζει την παραπάνω διαδικασία με τον ακόλουθο τρόπο.

Μετά την παραγωγή κινήσεων εξετάζουμε αν η δοθείσα θέση έχει ήδη εξερευνηθεί στο παρελθόν με τιμή μεγαλύτερου βάθους.

```
// TT probing.
TranspositionTable::TTEntry entry_result;
bool entry_found = transposition_table.GetEntry(zobrist_key, entry_result);
if(entry_found && !is_root && entry_result.depth >= depth){
    if(entry_result.type == TranspositionTable::NodeType::Exact){
        return entry_result.evaluation;
    }else if(entry_result.type == TranspositionTable::NodeType::Alpha &&
             entry_result.evaluation <= a){
        return a;
    }else if(entry_result.type == TranspositionTable::NodeType::Beta &&
             entry_result.evaluation >= b){
        return b;
    }
}
```

Ο πίνακας TT μας επιστρέφει έναν τύπο NodeType που περιέχει ένα flag type, την αξιολόγηση του κόμβου, το βάθος της αξιολόγησης και την βέλτιστη κίνηση για την κατάσταση αυτή.

```
struct TTEntry{
    NodeType type; // Determines if we check a,b or just return.
    int evaluation; // Position evaluation.
    uint8_t depth; // depth of search's iteration.
    Move best_move; // Picked move on said search's node.

    TTEntry(uint8_t depth, int evaluation, NodeType, Move best_move);
    TTEntry() = default;
};
```

Το flag παίρνει τις τιμές Alpha, Beta, Exact αναλόγως το είδος του cut-off που προέκυψε κατά την διαδικασία της αναζήτησης.

```
enum class NodeType{
    Alpha, Beta, Exact
};
```

Βάσει αυτού κρίνουμε αν πρέπει να επιστρέψουμε την τιμή a,b ή το γνωστό evaluation.

Στην περίπτωση που η εγγραφή βρέθηκε αλλά περιέχει αποτέλεσμα προηγουμένης αναζήτησης μικρότερου βάθους αξιοποιούμε μόνο την βέλτιστη κίνηση. Με αυτόν τον τρόπο μπορούμε να αξιοποιήσουμε προηγούμενες επαναλήψεις της επαναληπτικής εμβάθυνσης για καλύτερη ταξινόμηση των κινήσεων. Η κίνηση TT τοποθετείται πάντα πρώτη.

```
// Order TT move first.
if(entry_found) {
    Move tt_move = entry_result.best_move;
    auto pivot = std::find(moves.begin(), moves.end(), tt_move);
    if (pivot != moves.end()) {
        std::rotate(moves.begin(), pivot, pivot + 1);
    }
}
```

Στο τέλος της συνάρτησης εντάσσουμε την νέα πληροφορία εντός του πίνακα αν βρέθηκε νέο μέγιστο score.

```
// Add entry to TT.
auto entry = TranspositionTable::TTEntry(depth, best_score, node_type,
current_best_move);
transposition_table.AddEntry(zobrist_key, entry);
```

Η μεταβλητή node_type ανανεώνεται εντός των ελέγχων a-b και αρχικοποιείται με την τιμή Alpha.

```
if(score >= b) {
    node_type = TranspositionTable::NodeType::Beta;
    . . .
}
if(score > a) {
    node_type = TranspositionTable::NodeType::Exact;
    . . .
}
```

Ο πίνακας TT αποτελεί μια δομή unordered map, δεν παρατηρήθηκε υψηλή χρήση μνήμης για βάθη ≤ 8 για αυτό και δεν έχει υλοποιηθεί κάποιο replacement policy. Στην περίπτωση που ένας κόμβος υπάρχει ήδη, τότε γίνεται replace με την ανανεωμένη τιμή.

```
void TranspositionTable::AddEntry(uint64_t zobrist_key, const TTEntry&
entry){
    table_[zobrist_key] = entry;
}
```

Δεν απαιτείται έλεγχος της τιμής βάθους καθώς ξέρουμε ότι μικρότερες τιμές έχουν ήδη «κλαδευτεί» και άρα δεν έχουν φτάσει στο σημείο εισαγωγής στοιχείου.

```
class TranspositionTable{
public:

    . . .

    void Clear() { table_.clear(); }
    void Reserve(int size) { table_.reserve(size); }
    void AddEntry(uint64_t zobrist_key, const TTEntry& entry);
    bool GetEntry(uint64_t zobrist_key, TTEntry& result) const;
    int GetSize() const;

private:
    // <zobrist key, entry>
    std::unordered_map<uint64_t, TTEntry> table_;
};
```

3.9.4 Quiescence αναζήτηση

Η συνάρτηση QSearch υλοποιεί την Quiescence αναζήτηση.

```
int QSearch(const Board& board, int a, int b) {
    search_nodes++;

    int best_score = NNUE::Instance().EvaluateIncremental(board);
    if(best_score >= b)
        return b;
    if(best_score > a)
        a = best_score;

    Bitboard pins = board.GetPins();
    bool is_in_check = board.IsInCheck();
    MoveList moves = board.GetLegalCaptures(pins, is_in_check);
    SortMoves(board, moves);

    for (const auto& move : moves) {
        // Only capture moves.
        Board new_board = Board(board);
        new_board.PlayMove(move);
        new_board.Mirror();

        int score = -QSearch(new_board, -b, -a);
        if(score > best_score)
            best_score = score;
        if (score >= b)
            return b;
        if (score > a)
            a = score;
    }

    return best_score;
}
```

Παράγουμε μόνο τις κινήσεις αιχμαλωσίας και εφαρμόζουμε ab cut-offs με παρόμοιο τρόπο με τον αλγόριθμο PVS. Η αξιολόγηση της τελικής θέσης γίνεται με χρήση του NNUE μοντέλου. Οι κινήσεις ταξινομούνται πάλι με το κριτήριο MVV.

3.9.5 NNUE

Για την αξιολόγηση θέσης χρησιμοποιείται ένα δίκτιο NNUE μέσω της βιβλιοθήκης NNUE-probe https://github.com/dshawul/nnue-probe. Τα βάρη του δικτύου είναι διαθέσιμα στην σελίδα https://tests.stockfishchess.org/nns? σε μορφή .nnue αρχείων. Πιο πρόσφατες εκδόσεις των μοντέλων απαιτούν διαφορετική αναπαράσταση εισόδου από αυτή που υποστηρίζεται από την βιβλιοθήκη, για αυτό αρκούμαστε σε αρχεία του 2020.

Η βιβλιοθήκη μας δίνει την δυνατότητα για:

Φόρτωμα των βαρών από nnue αρχεία, αξιολόγηση θέσης μέσω συμβολοσειρών fen, αξιολόγηση θέσης μέσω αναπαράστασης ταμπλό, αξιολόγηση θέσης βηματικά.

Καθώς είναι γραμμένη στην γλώσσα C, υλοποιήθηκε μια «wrapper» κλάση για ευκολότερο χειρισμό.

```
class NNUE{
public:
    static NNUE& Instance() {
        static NNUE instance;
        return instance;
    }

    static void InitModel(char* file_name);

    static int Evaluate(const Board& board);
    int EvaluateIncremental(const Board& board);

    void InitAccumulator(int ply);
    void CopyToNextAccumulator(int ply);
    DirtyPiece* GetDirtyPiece(int ply);

    // Converts engine's types to the required input of the NNUE probe lib.
    static int GetSideEncoding(const Team& team);
    static int GetSideEncoding(bool is_flipped);
    static int GetPieceEncoding(const PieceType& type, const Team& team);
    static int GetPieceEncoding(PieceInfo pieceInfo);
    static int GetPieceEncoding(const PieceType& type, bool is_flipped);
    static int GetSquareEncoding(const BoardTile& tile);
    static int GetSquareEncoding(BoardTile tile, bool is_flipped);

private:
    NNUE() { AlignedReserve<NNUEdata>(nnue_data_arr, MAX_HSTACK); }
    NNUEdata* nnue_data_arr;
};
```

Αξιοποιείται το singleton pattern μέσω της συνάρτησης Instance όπως έγινε και για την κλάση History. Οι συναρτήσεις GetSide / GetPiece / GetSquare αφορούν την μετατροπή της κωδικοποίησης από την δικιά μας σκακιστική μηχανή σε αυτήν που χρησιμοποιείται εσωτερικά στην βιβλιοθήκη.

Ο πίνακας nnue_data_arr ανανεώνεται με στοιχεία από dirty piece για τον αποδοτικό υπολογισμό των βαρών (όπως είδαμε και στην συνάρτηση PlayMove). Η συνάρτηση AlignedReserve επιβάλλει την ανάθεση ευθυγραμμισμένης μνήμης για γρηγορότερη προσπέλαση.

```
// Allocate aligned memory.
template<typename T, int ALIGNMENT = CACHE_LINE_SIZE, bool large_pages =
false>
void AlignedReserve(T*& mem,const size_t& size) {
#ifdef __ANDROID__
    mem = (T*) memalign(ALIGNMENT,size * sizeof(T));
#elif defined(_WIN32)
    mem = (T*)_aligned_malloc(size * sizeof(T),ALIGNMENT);
#else
    posix_memalign((void**)&mem,ALIGNMENT,size * sizeof(T));
    #if defined(MADV_HUGEPAGE)
        if(large_pages)
            madvise(mem,size * sizeof(T),MADV_HUGEPAGE);
    #endif
#endif
}
```

Η συνάρτηση EvaluateIncremental υπολογίζει την αξία της δοθείσας θέσης με βάση τις προηγούμενες καταστάσεις (Η απλή Evaluate δεν χρησιμοποιείται στον κώδικα πάρα μόνο για λογούς debugging.)

```
int NNUE::EvaluateIncremental(const Board& board){
    const int max_pieces = 16 * 2;
    static int pieces[max_pieces + 1];
    static int squares[max_pieces];

    Board::Representation representation = board.GetRepresentation();
    bool is_flipped = board.IsFlipped();

    InitInput(representation, is_flipped, pieces, squares);
    int side = NNUE::GetSideEncoding(is_flipped);

    uint8_t current_ply = board.GetPlyCounter() - 1;
    NNUEdata* nnue_latest_data[3] = {0, 0, 0};
    for(int i = 0; i < 3 && current_ply >= i; i++)
        nnue_latest_data[i] = &nnue_data_arr[current_ply - i];

    return nnue_evaluate_incremental(side, pieces, squares,
                                     &nnue_latest_data[0]);
}
```

Οι πίνακες pieces, squares παρέχουν τις αντίστοιχες αναπαραστάσεις και ανανεώνονται μέσω της InitInput.

```
if(is_flipped)
    representation.Mirror();

int index = 0;
auto update_arrays = [&] (Bitboard piece_board, PieceType type) {
    // White first for kings to work.
    for(auto tile : piece_board & representation.own_pieces){
        squares[index] = NNUE::GetSquareEncoding(tile);
        pieces[index++] = NNUE::GetPieceEncoding(type, White);
    }
    for(auto tile : piece_board & representation.enemy_pieces){
        squares[index] = NNUE::GetSquareEncoding(tile);
        pieces[index++] = NNUE::GetPieceEncoding(type, Black);
    }
};

update_arrays(representation.Kings(), King);
update_arrays(representation.Pawns(), Pawn);
update_arrays(representation.Knights(), Knight);
update_arrays(representation.Queens(), Queen);
update_arrays(representation.Rooks(), Rook);
update_arrays(representation.Bishops(), Bishop);

pieces[index] = 0;
```

Τα βάρη του 1ου layer ονομάζονται “accumulators” και η ανανέωση τους γίνεται εντός της συνάρτησης transform. Στην πιο απλοποιημένη μορφή έχουμε τον παρακάτω κώδικα

```
// Difference calculation for the deactivated features
for (unsigned k = 0; k < removed_indices[c].size; k++) {
  unsigned index = removed_indices[c].values[k];
  const unsigned offset = kHalfDimensions * index;

  for (unsigned j = 0; j < kHalfDimensions; j++)
    accumulator->accumulation[c][j] -= ft_weights[offset + j];
}

// Difference calculation for the activated features
for (unsigned k = 0; k < added_indices[c].size; k++) {
  unsigned index = added_indices[c].values[k];
  const unsigned offset = kHalfDimensions * index;

  for (unsigned j = 0; j < kHalfDimensions; j++)
    accumulator->accumulation[c][j] += ft_weights[offset + j];
}
```

Η λογική είναι αντίστοιχη αυτής που περιεγράφηκε στο θεωρητικό σκέλος. Οι accumulators προσαρμόζονται ανάλογα με τις προσθήκες και αφαιρέσεις της νέας κίνησης.

Ο κώδικας είναι γραμμένος με τέτοιο τρόπο ώστε να αξιοποιούνται SIMD εντολές σε κβαντισμένες τιμές, εφόσον έχουν ενεργοποιηθεί κατά την μεταγλώττιση τα απαραίτητα flags.

```
USE_SSE41
USE_SSE3
USE_SSE2
USE_SSE
```

Ο παραπάνω κώδικας με χρήση εντολών SSE2 μετατρέπεται στο ακόλουθο

```
// Difference calculation for the deactivated features
for (unsigned k = 0; k < removed_indices[c].size; k++) {
  unsigned index = removed_indices[c].values[k];
  const unsigned offset = kHalfDimensions * index + i * TILE_HEIGHT;

  vec16_t *column = (vec16_t *)&ft_weights[offset];
  for (unsigned j = 0; j < NUM_REGS; j++)
    acc[j] = vec_sub_16(acc[j], column[j]);
}

// Difference calculation for the activated features
for (unsigned k = 0; k < added_indices[c].size; k++) {
  unsigned index = added_indices[c].values[k];
  const unsigned offset = kHalfDimensions * index + i * TILE_HEIGHT;

  vec16_t *column = (vec16_t *)&ft_weights[offset];
  for (unsigned j = 0; j < NUM_REGS; j++)
    acc[j] = vec_add_16(acc[j], column[j]);
}
```

Όπου vec_add_16, vec_sub_16 τα macros :

```
#define vec_add_16(a,b) _mm_add_epi16(a,b)
#define vec_sub_16(a,b) _mm_sub_epi16(a,b)
```

Με παρόμοιο τρόπο αξιοποιούνται και τα υπόλοιπα Intrinsic για τον υπολογισμό των απαραίτητων πράξεων για το inference του δικτύου. Διαφορετικά χρησιμοποιούνται απλοί τελεστές που είναι πολύ πιο αργοί.

3.10 Πρωτόκολλο UCI

Το UCI (Universal chess protocol) είναι το πιο διαδεδομένο πρωτόκολλο επικοινωνίας μεταξύ σκακιστικών μηχανών και γραφικών περιβάλλοντων GUI. Υλοποιήθηκαν οι πιο βασικές λειτουργίες του και ελέγχθηκε η ορθότητα του εντός του προγράμματος arena.

Περισσότερες πληροφορίες μπορούν να βρεθούν στον παρακάτω σύνδεσμο

http://wbec-ridderkerk.nl/html/UCIProtocol.html

Η βασική ιδέα είναι ότι το GUI ανοίγει την σκακιστική μηχανή σαν διεργασία παιδί και κάνει ανακατεύθυνση τις αντίστοιχες ροές. Έτσι με απλές εντολές εισόδου / εξόδου ανταλλάσσουμε πληροφορίες για την κατάσταση του ταμπλό.

Η σκακιστική μηχανή δέχεται συνεχώς εντολές μέχρι να δοθεί η λέξη “quit” όπου και το πρόγραμμα τερματίζει.

Η αρχική επικοινωνία εδραιώνεται με την εντολή uci. Η σκακιστική μηχανή μπορεί να στείλει διάφορες επιλογές (options) ακολουθούμενες από ένα “uci ok” για την ένδειξη επιτυχούς αρχικοποίησης.

```
void CommandUCI() {
    static std::string name = "a";
    static std::string author = "b";

    // ID.
    std::cout << "id name " << name << std::endl;
    std::cout << "id author " << author << std::endl;

    // Done.
    std::cout << "uciok" << std::endl;
}
```

Για να ορίσουμε μια θέση χρησιμοποιείται η εντολή position [startpos / fen] moves […]. Δηλαδή δίνουμε μια συμβολοσειρά fen (η την λέξη startpos για την αρχική θέση) και το σύνολο κινήσεων που έχουν ήδη παιχθεί.

```
Board CommandPosition(const std::vector<std::string> &words) {
    . . . string parsing code . . .

    ChessEngine::Board::BoardInfo info = {};
    ChessEngine::ParseFenString(fen, info);
    ChessEngine::Board board(info);

    // Play moves if any.
    if (iter != words.end()) {
        // Skip moves word.
        iter++;
        for (; iter != words.end(); iter++) {
            Move move(*iter, board.IsFlipped());
            board.PlayMove(move);
            board.Mirror();
        }
    }

    return board;
}
```

Οι κινήσεις δίνονται σε απλή αλγεβρική μορφή. Ο constructor της κλάσης Move έχει υλοποιηθεί ώστε να μπορεί να δέχεται τις συμβολοσειρές απευθείας.

```
std::string from_string = std::string(&algebraic_notation[0],
&algebraic_notation[2]);
std::string to_string = std::string(&algebraic_notation[2],
&algebraic_notation[4]);

std::tuple<uint8_t, uint8_t> coords_from;
NotationToCoords(from_string, coords_from);
auto [from_file, from_rank] = coords_from;
BoardTile from(from_file, from_rank);

std::tuple<uint8_t, uint8_t> coords_to;
NotationToCoords(to_string, coords_to);
auto [to_file, to_rank] = coords_to;
BoardTile to(to_file, to_rank);

if(is_flipped){from.Mirror(); to.Mirror();}

if(algebraic_notation.size() > 4){
    // Add promotion.
    PieceInfo info;
    CharToPieceInfo(algebraic_notation[4], info);
    auto [piece_type, team] = info;
    *this = Move(from, to, piece_type);
}else{
    *this = Move(from, to, PieceType::None);
}
```

Για να βρούμε την βέλτιστη κίνηση για την δοθείσα θέση χρησιμοποιούμε την εντολή go depth n, όπου n το μέγεθος του επιθυμητού βάθους. Στην περίπτωση που δεν δοθεί κάποια τιμή, αρχικοποιείται με την τιμή 8.

```
void CommandGo(const std::vector<std::string> &words, const Board &board) {
    int depth = 8;
    int index;
    if(FindWord(words, "depth", index)){
        depth = atoi(words[index + 1].c_str());
    }

    int eval;
    Move best_move = GetBestMove(board, depth, eval);
    std::cout << "info depth " << depth << " score cp " << eval << std::endl;
    std::cout << "bestmove " <<
    best_move.AlgebraicNotation(board.IsFlipped()) << std::endl;
}
```

Η κίνηση επιστρέφεται με το μήνυμα bestmove [move]. Άμα θέλουμε να προσφέρουμε επιπλέον πληροφορίες, το κάνουμε μέσω της εντολής info. Στην συγκεκριμένη υλοποίηση τυπώνεται και το score αξιολόγησης.

Το πρωτόκολλο UCI παρέχει επιπλέον εντολές για υπολογισμό κίνησης βάσει χρόνου. Οι περιπτώσεις αυτές δεν υποστηρίζονται για λογούς απλότητας.

4. ΣΥΜΠΕΡΑΣΜΑΤΑ

Η υλοποίηση σκακιστικών μηχανών με χρήση bitboards είναι μια σύνθετη αλλά ιδιαίτερη διαδικασία που προσφέρει γρήγορη παραγωγή κινήσεων με χρήση ειδικών τεχνασμάτων για αναζητήσεις σταθερού χρόνου. Η ιδέα μπορεί να εφαρμοστεί σε οποιαδήποτε πρόβλημα δυαδικών συνόλων που επωφελείται από την παράλληλη επεξεργασία λέξεων μέσω δυαδικών τελεστών.

Τα μοντέλα NNUE είναι αρκετά ρηχά δίκτυα που υπερβαίνουν τις ικανότητες πρόβλεψης ανθρώπινων συναρτήσεων αξιολόγησης, χωρίς ανάγκη για ειδική γνώση για το συγκεκριμένο πρόβλημα προς λύση. Ακόμα και με μικρό μέγεθος δικτύου, χρήση βηματικών ενημερώσεων και εντολών SIMD, ο δείκτης nps μειώνεται σημαντικά (80%). Παρ'ολ'αυτα οδηγούμαστε σε εύρεση καλυτέρων κινήσεων ακόμα και με μειωμένο βάθος αναζήτησης. Μια υβριδική λύση για τον υπολογισμό μη-ήρεμων θέσεων, ενδεχομένως να οδηγούσε σε γρηγορότερους υπολογισμούς με μικρές απώλειες ακρίβειας.

Η εισαγωγή νευρωνικών δικτύων στην διαδικασία αναζήτησης συντελεί στην καλύτερη γενίκευση των αλγορίθμων και την ευκολότερη εφαρμογή αυτών σε λιγότερο μελετημένα παιχνίδια. Παρ'ολ'αυτα ένα σημαντικό βήμα βελτιστοποίησης των παραλλαγών ab, είναι η σειρά ταξινόμησης των κινήσεων που εξακολουθεί να γίνεται με ευρετικό τρόπο. Αντίστοιχα άλλες τεχνικές αποκοπής, όπως το futility pruning, βασίζονται στην χρήση εμπειρικών margins. Ένας ιδανικός αλγόριθμος θα αξιοποιούσε επιπρόσθετα δίκτυα χωρίς την εισαγωγή μεγάλου χρονικού κόστους.

Πιο σύνθετες αρχιτεκτονικές όπως αυτή του Alpha zero επιφέρουν καλύτερη αξιολόγηση θέσης αλλά οδηγούν σε μικρότερη εξερεύνηση του χώρου καταστάσεων. Αντιθέτως απλούστερες αρχιτεκτονικές όπως αυτή του NNUE είναι λιγότερο αξιόπιστες μεμονωμένα αλλά οδηγούν σε ισάξια αποτελέσματα λόγω αυξημένου πλήθους επισκεπτόμενων κόμβων.

Η σκακιστική μηχανή που υλοποιήθηκε για αυτή την πτυχιακή παίζει σε λογικούς χρόνους ($< 5s$ ανα κινηση) για βάθη ≤ 8 ενώ μεγαλύτερες τιμές όπως οι $\{9, 10\}$ προσεγγίζουν τα $\sim 5m$ σε ισόπαλες θέσεις.

Το τελικό επίπεδο παιξίματος δεν μπορεί να ποσοτικοποιηθεί με ακρίβεια, αλλά μετά από ένα σύνολο παιχνιδιών ενάντιων άλλων σκακιστικών μηχανών ή κανονικών παιχτών καταλήγουμε στο ότι είναι ικανή να κερδίσει παίχτες ≤ 2000 elo (σε παιχνίδια περιορισμένου χρόνου) ενώ συχνά καταλήγει σε ισοπαλίες με μηχανές ~ 2600 elo (με σταθερό βάθος και απεριόριστο χρόνο). Για τις περισσότερες ήττες συνθέτων παιχνιδιών ευθύνονται ανακριβείς κινήσεις που δημιουργούν μελλοντικές ανισορροπίες λόγω περιορισμένου βάθους αναζήτησης.

Για την περαιτέρω βελτίωση της μηχανής συνίσταται, η παραλληλοποίηση του αλγορίθμου αναζήτησης με χρήση Lazy SMP και η καλύτερη ταξινόμηση κινήσεων είτε με χρήση άλλων στατικών τεχνικών είτε με προσαρμογή επιπλέον ρηχών δικτύων.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Perfect information	Τέλεια πληροφόρηση
Imperfect information	Ατελής πληροφόρηση
Sequential game	Διαδοχικό παιχνίδι
Zero sum game	Παιχνίδι μηδενικού αθροίσματος
Infinite play	Άπειρο παιχνίδι
State	Κατάσταση
Board	Ταμπλό
Brute force	Ωμή βία
Node	Κόμβος
Perfect play	Ιδανικό παίξιμο
Conversion	Μετατροπή
Game engine	Μηχανή παιχνιδιού
Castling	Ροκέ
Tile	Πλακάκι ταμπλό
Feature representation	Αναπαράσταση χαρακτηριστικών
Register	Καταχωρητής
Chunk	Κομμάτι
Undefined	Μη ορισμένο
Portable	Φορητή
Pseudo legal	Ψευδο-νόμιμη
Self-wrapping	Αυτό τύλιγμα
Stall	Χαμένος κύκλος
Pipeline	Διοχέτευση
Rays	Ακτίνες
Principal variation	Ακολουθία βέλτιστων κινήσεων
Prune	Κλαδεύω (αποκόπτω)
Transposition	Μετάθεση
Evaluation	Αξιολόγηση
Classification	Κατηγοριοποίηση
Data augmentation	Επαύξηση δεδομένων
Blocker pieces	Πιόνια εμπόδια
PopCount	Αριθμός ενεργών bits
Minimal	Ελάχιστη συνάρτηση κατακερματισμού
Collision	Συγκρούσεις
Quiet moves	Ήρεμες κινήσεις
Futility pruning	Κλάδεμα ματαιότητας
Activation function	Συνάρτηση ενεργοποίησης
Rotated bitboards	Περιστρεφόμενα Bitboards
Tree parallelization	Παραλληλοποίηση δέντρου
Leaves parallelization	Παραλληλοποίηση φύλλων
Root parallelization	Παραλληλοποίηση ρίζας
Horizon effect	Φαινόμενο του ορίζοντα

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

NPS	Nodes per second
AI	Artificial intelligence
CCR	Compact chess representations
MSB	Most significant bit
LSB	Least significant bit
MVV-LVA	Most valuable victim least valuable Aggressor
PVS	Principal variation search
YBWC	Young brothers wait concept
DFS	Depth first search
BFS	Best first search
MCTS	Monte Carlo tree search
UCB	Upper confidence bound
NNUE	Efficiently updatable neural network
TPU	Tensor processing unit
SIMD	Single instruction multiple data
Lc0	Leela chess zero
MSE	Mean squared error
CNN	Convolutional neural networks
ResNet	Residual networks
UCI	Universal chess protocol
SEE	Static exchange evaluation

ΠΑΡΑΡΤΗΜΑ

[Event "Computer chess game"]
[Site "chessclub.com"]
[White "MyChessEngine"]
[Black "anonymous"]
[BlackElo "2064"]
[TimeControl "120+6"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Bxc6 dxc6 5. Nc3 Bc5 6. Nxe5 Bxf2+ 7. Kxf2 Qd4+ 8. Ke1 Qxe5 9. d4 Qe7 10. Be3 Nf6 11. Qf3 h6 12. Qg3 Nxe4 13. Qxg7 Qf6 14. Qxf6 Nxf6 15. Rf1 Nd5 16. Nxd5 cxd5 17. Bf4 c6 18. Kd2 Be6 19. Rf2 Kd7 20. Raf1 Rag8 21. Be5 Rh7 22. b3 h5 23. a4 Rg4 24. Kc3 Re4 25. a5 c5 26. Rf4 cxd4+ 27. Bxd4 f5 28. Kd3 h4 29. R1f2 h3 30. g3 Kc6 31. R2f3 Kb5 32. Bb6 Rh8 33. Bd4 Rc8 34. Rh4 Kxa5 35. Rxh3 f4 36. Rh5 fxg3 37. Rxg3 Rf8 38. Be3 Rf1 39. c4 Bf5 40. Ke2 Bg4+ 41. Rxg4 Rxg4 42. Kxf1 Kb4 43. cxd5 Kxb3 44. d6 {Black resigns} 1-0

- -

[Event "Computer chess game"]
[White "Chess.Com bot"]
[Black "MyChessEngine"]
[WhiteElo "2600"]

1. Nf3 d5 2. c4 d4 3. e3 Nc6 4. exd4 Nxd4 5. Nxd4 Qxd4 6. Nc3 Nf6 7. d3 e5 8. Be3 Qd7 9. d4 exd4 10. Qxd4 Qxd4 11. Bxd4 Bf5 12. O-O-O O-O-O 13. Be2 Bd6 14. g3 Be7 15. Rhe1 Be6 16. f4 Rhe8 17. Bxa7 b6 18. Bf3 Bxc4 19. Rxd8+ Bxd8 20. Rxe8 Nxe8 21. b3 Be6 22. Kc2 Nf6 23. a4 Ng4 24. h3 Ne3+ 25. Kd3 Nf1 26. g4 Bxb3 27. Bc6 Ng3 28. a5 bxa5 29. Bd4 Bf6 30. Bxf6 gxf6 31. Ke3 f5 32. Kf3 Ne4 33. Nxe4 fxe4+ 34. Kxe4 a4 35. Kd3 a3 36. Kc3 Be6 37. f5 Bd7 38. Be4 f6 39. Kb3 c6 40. Kxa3 Kc7 41. Kb4 Kd6 42. Kc4 h6 43. Kd4 c5+ 44. Kc4 Bc8 45. Kb5 Bd7+ 46. Kc4 Bc8 47. Kb5 Bd7+ 48. Kc4 {3-fold repetition} 1/2-1/2

ΑΝΑΦΟΡΕΣ

[1] C. Shannon, "Programming a Computer for Playing Chess," *Computer Chess Compendium,* pp. 2-13, 1988.

[2] H. J. v. d. Herik, "Games solved: Now and in the future," *Artificial Intelligence,* vol. 134, no. 1-2, pp. 277-311, 2002.

[3] D. Freeman, "Chess statistics," [Online]. Available: https://www.chessgames.com/chessstats.html. [Accessed 6 10 2021].

[4] M. Lai, "Giraffe: Using Deep Reinforcement Learning to Play Chess," *CoRR,* vol. abs/1509.01549, 2015.

[5] N. U. M. Sigit Kariagil Bimonugroho, "A Hybrid Approach to Representing Chessboard using Bitboard and Compact Chessboard Representation," 2020.

[6] F. Reul, "New Architectures in Computer Chess," Gildeprint, TICC Dissertation Series 6, 2009.

[7] V. Vuckovic, "The compact chessboard representation," *ICGA Journal,* vol. 31, no. 3, pp. 157-164, 01 09 2008.

[8] C. Browne, "Bitboard methods for games," *ICGA Journal,* vol. 37, no. 2, pp. 67-84, 2014.

[9] R. Grimbergen, "Using bitboards for move generation in shogi," *ICGA Journal,* vol. 30, no. 1, pp. 25-34, 2007.

[10] "Bitboard Board-Definition - Chessprogramming wiki," [Online]. Available: https://www.chessprogramming.org/Bitboard_Board-Definition. [Accessed 16 1 2022].

[11] W. Peter, "A technique for counting ones in a binary computer.," Cambridge, Massachusetts , 1960.

[12] D. Knuth, The Art of Computer Programming, Boston, MA: Addison-Wesley, 2009.

[13] A. J. Bik1, "Computing deep perft and divide numbers for Checkers," *ICGA Journal,* vol. 35, no. 4, pp. 206-213, 2012.

[14] "Magic Bitboards - Chessprogramming wiki," [Online]. Available: https://www.chessprogramming.org/Magic_Bitboards. [Accessed 20 1 2022].

[15] M. v. Kervinck, "Traversing Subsets of a Set - Chessprogramming wiki," [Online]. Available: https://www.chessprogramming.org/Traversing_Subsets_of_a_Set. [Accessed 16 1 2022].

[16] K. Hoki, "Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move," *Entertainment Computing,* vol. 3, no. 3, pp. 51-57, 2012.

[17] O. David-Tabibi, "Extended Null-Move Reductions," *Computers and Games,* pp. 205-216, 2008.

[18] A. A. Elnaggar, "A Comparative Study of Game Tree Searching," *(IJACSA) International Journal of Advanced Computer Science and Applications,* vol. 5, no. 5, 2014.

[19] K. Hazama, "Branch and bound algorithm for parallel many-core architecture," *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW),* 2018.

[20] L. Huizhan, "Hash Table in Chinese Chess," *Chinese Control and Decision Conference (CCDC),* 2012.

[21] J. Ros-Giralt, "Lockless Hash Tables with Low False Negatives," *2014 IEEE High Performance Extreme Computing Conference (HPEC),* 2014.

[22] D. Perez, "Knowledge-based Fast Evolutionary MCTS for general video game playing," *2014 IEEE Conference on Computational Intelligence and Games,* 2014.

[23] L. Kocsis, Bandit based Monte-Carlo Planning, Budapest: Springer Berlin Heidelberg, 2006, pp. 282-293.

[24] H. Baier, "Monte-Carlo Tree Search and Minimax Hybrids," *2013 IEEE Conference on Computational Inteligence in Games (CIG),* 2013.

[25] H. Baier, "A Rollout-Based Search Algorithm," *Communications in Computer and Information Science,* pp. 57-70, 2017.

[26] G. M.J-B, Parallel Monte-Carlo Tree Search, Maastricht: Springer Berlin Heidelberg, pp. 60-71.

[27] Y. Nasu, "Efficiently updatable neural network based evaluation function for computer shogi," Ziosoft Computer Shogi Club, 2018.

[28] D. Klein, Neural networks for chess, Independently published, 2021.

[29] T. Sobczyk, "NNUE-pytorch," Stockfish, 2021. [Online]. Available: https://github.com/glinscott/nnue-pytorch/blob/master/docs/nnue.md#feature-factorization. [Accessed 16 1 2022].

[30] D. Silver, "Mastering the game of Go with deep neural networks and tree search," *Nature,* vol. 529, no. 7587, pp. 484-489, 2016.

[31] D. Silver, "Mastering the game of Go without human knowledge," *Nature,* vol. 550, no. 7676, pp. 354-359, 2017.

[32] D. Silver, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *Science,* vol. 362, no. 6419, pp. 1140-1144, 2018.

[33] J. Schrittwieser, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature,* vol. 588, no. 7839, pp. 604-609, 2020.