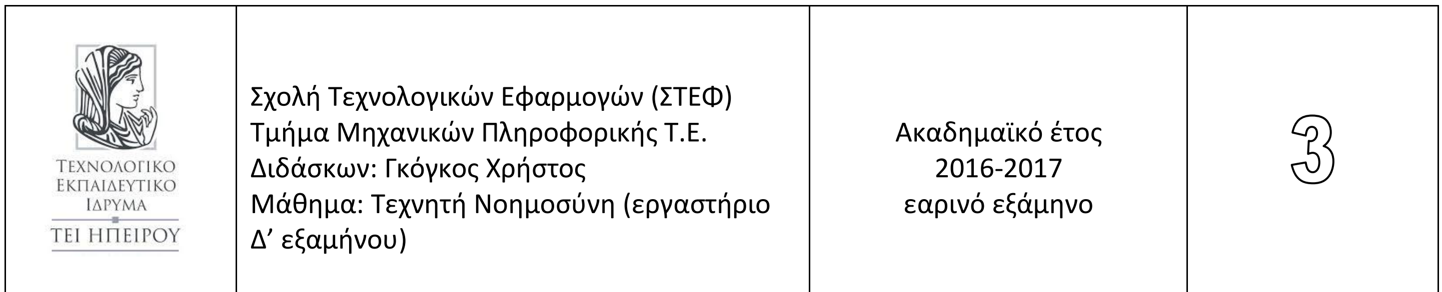


Αναζήτηση με αντιπαλότητα (Adversarial Search)

Η αναζήτηση με αντιπαλότητα αφορά προβλήματα στα οποία επιχειρείται η σχεδίαση ενός πλάνου ενεργειών σε καταστάσεις στις οποίες παίκτες σχεδιάζουν ενέργειες ο ένας εναντίον του άλλου. Περιβάλλοντα στα οποία δημιουργούνται προβλήματα αναζήτησης με αντιπαλότητα είναι γνωστά ως παιχνίδια ή παίγνια.

Παιχνίδια (Games)

Τα παιχνίδια με τα οποία θα ασχοληθούμε ανήκουν στην κατηγορία των ντετερμινιστικών, παρατηρήσιμων, και μηδενικού αθροίσματος παιχνιδιών. Οι όροι που αναφέρθηκαν σημαίνουν:

- **Ντετερμινιστικό (deterministic)**: Κάθε ενέργεια εφόσον επαναληφθεί ξεκινώντας από την ίδια κατάσταση κάθε φορά δίνει το ίδιο αποτέλεσμα. Για παράδειγμα το σκάκι είναι ντετερμινιστικό παιχνίδι αλλά το τάβλι δεν είναι.
- **Παρατηρήσιμο (observable)**: Οι πλήρεις πληροφορίες είναι διαθέσιμες στους παίκτες. Η τρίλιζα είναι παρατηρήσιμη ενώ το πόκερ δεν είναι καθώς στο πόκερ ο ένας παίκτης δεν έχει πρόσβαση στα χαρτιά του άλλου παίκτη.
- **Μηδενικού αθροίσματος (zero sum game)**: Κάθε κέρδος για έναν παίκτη σημαίνει απώλεια ίσης αξίας για τον άλλο παίκτη.

Παραδείγματα παιχνιδιών αυτού του είδους που παίζονται από δύο παίκτες είναι η τρίλιζα, το οθέλλο[1], το σκάκι, το γκο[2] και άλλα. Ένα παιχνίδι μπορεί να οριστεί ως ένα πρόβλημα αναζήτησης με συστατικά στοιχεία τα ακόλουθα:

- Την αρχική κατάσταση.
- Ένα χώρο καταστάσεων με όλες τις επιτρεπτές καταστάσεις στις οποίες μπορεί να μεταβεί το παιχνίδι μέσω των τελεστών μετάβασης (των επιτρεπτών κινήσεων).
- Ένα σύνολο τελικών καταστάσεων που υποδηλώνουν καταστάσεις στις οποίες το παιχνίδι τερματίζεται με νίκη του ενός από τους δύο αντίπαλους ή με ισοπαλία. Εναλλακτικά, μια τελική κατάσταση μπορεί να χαρακτηρίζεται από μια αριθμητική τιμή η οποία να υποδηλώνει το σκορ της νίκης του κάθε παίκτη.

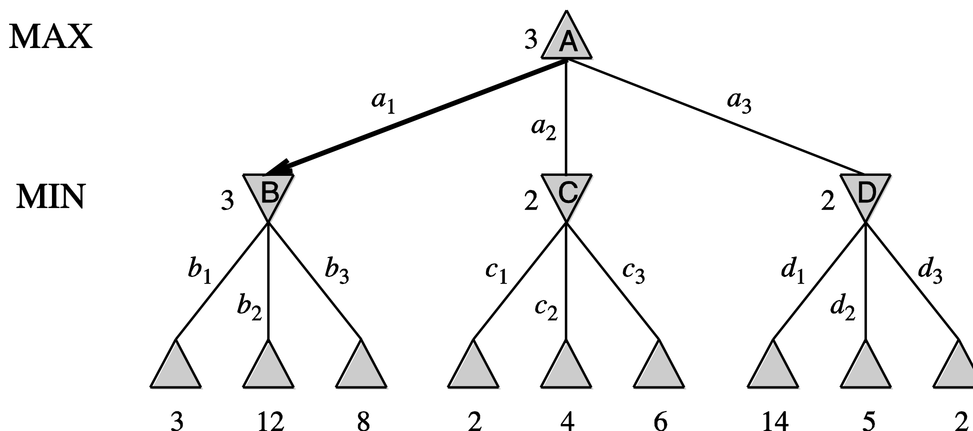
Ο αλγόριθμος minimax

Έχει καθιερωθεί σε προβλήματα αναζήτησης με αντιπαλότητα δύο παικτών ο ένας παίκτης να αναφέρεται ως MAX και ο άλλος ως MIN. Κάθε παίκτης θα πρέπει να καταστρώσει μια βέλτιστη στρατηγική η οποία θα προσδιορίζει τις κινήσεις που θα τον οδηγήσουν στη νίκη λαμβάνοντας υπόψη τις καλύτερες κινήσεις που θα μπορούσε να κάνει ο αντίπαλός του. Για να γίνει αυτό δημιουργείται ένα δένδρο παιχνιδιού το οποίο έχει ως ρίζα την αρχική κατάσταση και κάθε νέο επίπεδο του δένδρου δημιουργείται εναλλάξ από τις έγκυρες κινήσεις των δύο παικτών.

Για παράδειγμα στο ακόλουθο δένδρο παιχνιδιού ο παίκτης που παίζει πρώτος είναι ο MAX. Η επιλογή της κίνησης που θα κάνει εξαρτάται από τις τιμές οι οποίες υποδηλώνουν τη χρησιμότητα (utility) κάθε τερματικού κόμβου για τον MAX. Σε κάθε κόμβο, πλην των τερματικών, υπολογίζεται η χρησιμότητα που είναι η μέγιστη τιμή από τα παιδιά του αν είναι κόμβος MAX ή η ελάχιστη τιμή από τα παιδιά του αν είναι κόμβος MIN. Οι τιμές συμπληρώνονται στο

[1] http://www.othelloonline.org/
[2] https://en.wikipedia.org/wiki/Go_(game)

δένδρο από κάτω προς τα πάνω. Η κίνηση που θα πρέπει να κάνει ο MAX είναι εκείνη που δίνει την υψηλότερη τιμή χρησιμότητας στη ρίζα του δένδρου. Στο παράδειγμα της εικόνας 1 αυτό συμβαίνει με την κίνηση a1 που οδηγεί στην κατάσταση B που έχει μεγαλύτερη χρησιμότητα έναντι των καταστάσεων C και D. Η κίνηση a1 ακολουθείται από την κίνηση b1 που είναι η καλύτερη δυνατή απάντηση του MIN δεδομένης της κατάστασης B.

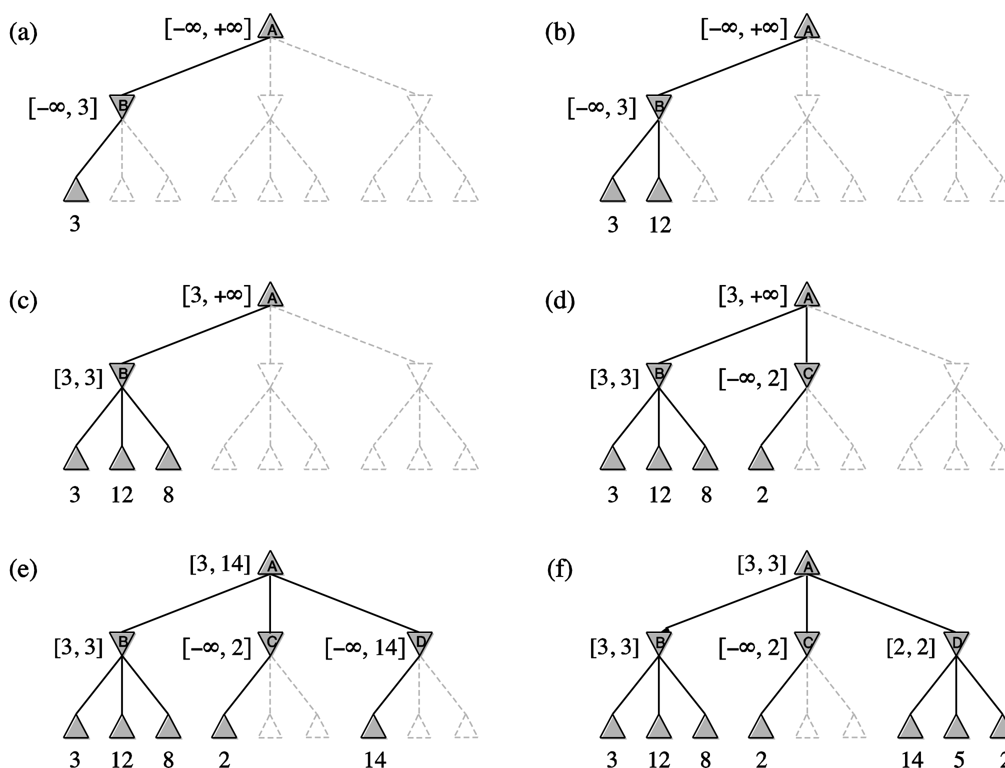


Εικόνα 1. Δένδρο παιχνιδιού

Ο αλγόριθμος minimax είναι μια μορφή αναζήτησης πρώτα κατά βάθος η οποία μπορεί να υλοποιηθεί αναδρομικά. Ο αριθμός των κόμβων που εξετάζονται είναι εκθετικός ως προς των αριθμό των κινήσεων στο παιχνίδι. Στην ορολογία των παιχνιδιών λέγεται ότι μια κίνηση αποτελείται από δύο μισές κινήσεις (στρώσεις-plies), μια στρώση (ply) από τον παίκτη MAX ακολουθούμενη από μια στρώση από τον παίκτη MIN.

Κλάδεμα Άλφα-Βήτα

Η τεχνική του κλαδέματος άλφα-βήτα επιτρέπει στον αλγόριθμο άλφα-βήτα να είναι αποδοτικότερος έναντι του αλγορίθμου minimax καθώς εξαλείφει τμήματα του δένδρου αναζήτησης στα οποία δεν πρόκειται να εντοπιστεί η ακολουθία των βέλτιστων κινήσεων. Ο αλγόριθμος άλφα-βήτα επιστρέφει πάντα το ίδιο αποτέλεσμα με τον αλγόριθμο minimax εξετάζοντας όμως στην συντριπτική πλειοψηφία των περιπτώσεων σημαντικά λιγότερους κόμβους.



Εικόνα 2. Κλάδεμα α-β

Ο αλγόριθμος άλφα-βήτα λαμβάνει το όνομά του από τις παραμέτρους α και β που προσδιορίζουν δύο φράγματα:

- α = τιμή της καλύτερης επιλογής που έχει βρεθεί για τον MAX κατά μήκος της διαδρομής που έχει διανυθεί. Η τιμή του α είναι αρχικά -∞.
- β = τιμή της καλύτερης επιλογής που έχει βρεθεί για τον MIN κατά μήκος της διαδρομής που έχει διανυθεί Η τιμή του β είναι αρχικά +∞.

Καθώς η αναζήτηση προχωρά, οι τιμές των α και β ενημερώνονται και τα υποδένδρα ενός κόμβου κλαδεύονται μόλις διαπιστωθεί ότι η τιμή του κόμβου είναι χειρότερη από την τρέχουσα τιμή του α ή του β. Στο παράδειγμα της εικόνας 2 το σκεπτικό με το οποίο πραγματοποιείται κλάδεμα του δένδρου είναι το ακόλουθο:

- Στο βήμα (a) και εξετάζοντας **μόνο** το πλέον αριστερό παιδί της κατάστασης B που έχει τιμή 3 διαπιστώνουμε ότι ο MIN μπορεί να επιτύχει τιμή μικρότερη ή ίση του 3.
- Στο βήμα (c) έχοντας εξετάσει όλα τα παιδιά της κατάστασης B μπορούμε να συμπεράνουμε ότι η ρίζα (δηλαδή η κατάσταση A) θα έχει τιμή μεγαλύτερη ή ίση του 3.
- Στο βήμα (d) εξετάζοντας το πλέον αριστερό παιδί της κατάστασης C που έχει τιμή 2 διαπιστώνουμε ότι η κατάσταση C θα έχει τιμή μικρότερη ή ίση του 2. Γνωρίζοντας ότι η ρίζα έχει ήδη τιμή μεγαλύτερη ή ίση του 3, αυτό σημαίνει ότι η κατάσταση C δεν πρόκειται ποτέ να επιλεγεί από τον MAX και συνεπώς τα υπόλοιπα παιδιά της κατάστασης C μπορούν να κλαδευτούν.
- Στα βήματα (e) και (f) φαίνεται ότι δεν μπορεί να ληφθεί αντίστοιχη απόφαση με το βήμα (d) καθώς οι τιμές κόμβων με τη σειρά που εξετάζονται δεν επιτρέπουν το κλάδεμα. Οι δύο πρώτες (14 και 5) είναι μεγαλύτερες από το 3 και η τρίτη (το 2) όταν εξετάζεται δεν έχει άλλες τιμές δεξιότερα για να κλαδέψει.

Στο παράδειγμα της εικόνας 2 δε χρειάστηκε να ασχοληθούμε καθόλου με δύο από τους κόμβους φύλλα. Στη γενική περίπτωση η αποδοτικότητα του αλγορίθμου άλφα-βήτα εξαρτάται από τη σειρά με την οποία εξετάζονται τα παιδιά των κόμβων. Στην καλύτερη περίπτωση ο δραστικός παράγοντας διακλάδωσης γίνεται $\sqrt{b}$ έναντι του b και συνεπώς ο αλγόριθμος άλφα-βήτα μπορεί να βλέπει δύο φορές πιο μπροστά σε σχέση με τον minimax.

Στη συνέχεια παρουσιάζεται μια αλγοριθμική περιγραφή του τρόπου με τον οποίο τα όρια α και β λαμβάνουν τιμές κατά τη διάρκεια της αναζήτησης οδηγώντας σε κλάδεμα τμημάτων του δένδρου αναζήτησης.

Αλγοριθμική περιγραφή κλαδέματος άλφα-βήτα
1. Ορίζονται για κάθε κόμβο του δένδρου, πλην των τερματικών, τα όρια α και β. a. Στη ρίζα του δένδρου δίνονται οι τιμές α=-∞ και β=+∞. b. Η αναζήτηση προχωρά «πρώτα κατά βάθος (DFS=Depth First Search)» και οι τιμές των α και β αντιγράφονται σε κάθε κόμβο που συναντά για πρώτη φορά η DFS από τον γονέα του (δηλαδή από πάνω προς τα κάτω). 2. Όταν λάβει τιμή ένας κόμβος x είτε διότι είναι τερματικός είτε διότι έχουν εξεταστεί τα παιδιά του τότε: a. Αν ο γονέας του x είναι κόμβος max τότε αλλάζει το όριο α για το γονέα εφόσον η τιμή του x είναι μεγαλύτερη από το α οπότε και το α γίνεται η τιμή του x. b. Αν ο γονέας του x είναι κόμβος min τότε αλλάζει το όριο β για το γονέα εφόσον η τιμή του x είναι μικρότερη από το β οπότε και το β γίνεται η τιμή του x. c. Αν σε κάποιο κόμβο γίνει το α>=β τότε τα υπόλοιπα κλαδιά του κόμβου που δεν έχουν ακόμα εξερευνηθεί από την αναζήτηση DFS κλαδεύονται.

Πρακτική χρήση των αλγορίθμων minimax και άλφα-βήτα

Στην πράξη λόγω του μεγέθους του δένδρου του παιχνιδιού δεν είναι δυνατόν οι αλγόριθμοι minimax και άλφα-βήτα να φτάσουν μέχρι τις τερματικές καταστάσεις έτσι ώστε να υπολογίσουν εκεί τη χρησιμότητα κάθε κατάστασης. Άρα ξεκινώντας από την τρέχουσα κατάσταση ορίζεται ένα τεχνητό βάθος που ορίζει μέχρι ποια στρώση θα αναπτύσσεται το δένδρο του παιχνιδιού. Η δε χρησιμότητα, στους κόμβους που παίζουν τον ρόλο των τερματικών κόμβων (χωρίς υποχρεωτικά να είναι) υπολογίζεται με μια ευρετική συνάρτηση στην οποία γίνεται προσπάθεια να αποδοθούν

διαβαθμισμένα υψηλές τιμές για καταστάσεις που είναι υπέρ του παίκτη MAX και αντίστοιχα διαβαθμισμένα χαμηλές τιμές για καταστάσεις που είναι υπέρ του παίκτη MIN.

Ένα πρόβλημα της ανωτέρω προσέγγισης είναι το φαινόμενο του ορίζοντα (horizon effect) σύμφωνα με το οποίο μια κακή κίνηση θεωρείται καλή λόγω του ότι το αποτέλεσμά της αποκαλύπτεται μετά το βάθος στο οποίο είναι σε θέση να «δει» ο αλγόριθμος. Υπάρχουν διάφορες προσεγγίσεις για την επίλυση του συγκεκριμένου προβλήματος (π.χ. scout algorithms, singular extensions, quiescence search κ.α.).

Το παιχνίδι Οθέλλο

Το παιχνίδι Οθέλλο (γνωστό και ως Reversi) είναι ένα παιχνίδι δύο ατόμων που παίζεται σε ένα ταμπλό 8 X 8 χρησιμοποιώντας πούλια που από τη μια πλευρά είναι λευκά και από την άλλη μαύρα. Στην υλοποίηση που θα ακολουθήσει τα λευκά πούλια αναπαρίστανται με τον χαρακτήρα ‘O’ και τα μαύρα με τον χαρακτήρα ‘X’. Η αρχική διαμόρφωση του παιχνιδιού είναι η ακόλουθη:

```
  0 1 2 3 4 5 6 7
0| | | | | | | | |
1| | | | | | | | |
2| | | | | | | | |
3| | | |X|O| | | |
4| | | |O|X| | | |
5| | | | | | | | |
6| | | | | | | | |
7| | | | | | | | |
```

Οι παίκτες παίζουν εναλλάξ τοποθετώντας ένα νέο πούλι του χρώματός τους σε κάποια από τις θέσεις οι οποίες θα προκαλέσουν την αναστροφή χρώματος σε ένα τουλάχιστον πούλι του αντιπάλου. Για να συμβεί αυτό θα πρέπει ανάμεσα στο πούλι που πρόκειται να τοποθετήσει ο παίκτης και σε ένα άλλο πούλι του ίδιου χρώματος να υπάρχουν πούλια του αντιπάλου τα οποία θα αλλάζουν χρώμα μόλις τοποθετήσει ο παίκτης το πούλι του. Συνεπώς, για να μπορεί να παίξει ένας παίκτης θα πρέπει να έχει την δυνατότητα να τοποθετήσει ένα πούλι προκαλώντας αναστροφή. Αν δεν έχει αυτή τη δυνατότητα, παραχωρεί τη σειρά του στον αντίπαλο. Το παιχνίδι τελειώνει όταν είτε το ταμπλό έχει γεμίσει είτε όταν και οι δύο παίκτες δεν έχουν επιτρεπτές κινήσεις. Ο στόχος για κάθε παίκτη είναι στο τέλος του παιχνιδιού να έχει μεγαλύτερο αριθμό από πούλια του χρώματός του σε σχέση με τον αριθμό από πούλια του αντιπάλου του.

Ο χώρος αναζήτησης του Οθέλλο είναι μικρότερος σε σχέση με άλλα παιχνίδια όπως το σκάκι και το γκο καθώς οι νόμιμες κινήσεις ανά πάσα στιγμή είναι συνήθως από 5 μέχρι 15.

Υλοποίηση

Το ταμπλό (`board`) του παιχνιδιού αναπαρίσταται ως ένας δισδιάστατος πίνακας χαρακτήρων 8 X 8 και αρχικοποιείται με τη συνάρτηση `char **initialize_board().`

Οι βασικές συναρτήσεις οι οποίες χρησιμοποιούνται στην υλοποίηση των επιλυτών του προβλήματος είναι οι συναρτήσεις που εμφανίζονται στο αρχείο lab03_board.hpp.

Η συνάρτηση `void update_board(char **aboard, int r, int c, char disk)` αναλαμβάνει την ενημέρωση του ταμπλό έτσι ώστε να αναπαριστά το νέο ταμπλό που θα προκύψει όταν τοποθετηθεί ένα πούλι τύπου disk (‘X’ ή ‘O’) στη γραμμή r και στη στήλη c.

Η συνάρτηση `list<char **> get_successor_boards(char **aboard, char disk)` επιστρέφει μια λίστα με όλα τα πιθανά ταμπλό που μπορούν να προκύψουν αν τοποθετηθεί ένα πούλι τύπου disk σε οποιαδήποτε από τις πιθανές έγκυρες θέσεις στις οποίες θα μπορούσε να τοποθετηθεί.

Συναρτήσεις για την εναλλαγή χρώματος στα πούλια

Η συνάρτηση `list<pair<int, int>> all_directions_disks_to_flip(char **aboard, int r, int c, char disk)` επιστρέφει μια λίστα με ζεύγη ακεραίων που αναπαριστούν όλα τα πούλια που πρόκειται να αντιστραφούν σε περίπτωση που ο παίκτης για τον οποίο είναι η σειρά του να παίξει τοποθετήσει ένα πούλι στη γραμμή r και στη στήλη c.

Η συνάρτηση `list<pair<int, int>> disks_to_flip(char **aboard, int r, int c, char disk, int xdir, int ydir)` επιστρέφει μια λίστα από πούλια που πρόκειται να αντιστραφούν αν ο παίκτης κινηθεί στην κατεύθυνση που υποδηλώνεται από τα xdir, ydir και η οποία μπορεί να είναι μια από οκτώ πιθανά ζεύγη τιμών. Για παράδειγμα το ζεύγος 1,1 υποδηλώνει κίνηση προς τη διαγώνιο κάτω δεξιά σε σχέση με τη τρέχουσα θέση. Οι οκτώ κατευθύνσεις φαίνονται στον ακόλουθο πίνακα.

-1, -1	-1, 0	-1, 1
0, -1	τρέχουσα θέση	0, 1
1, -1	1, 0	1, 1

Η συνάρτηση `disks_to_flip` εξετάζει την κίνηση προς την κατεύθυνση που υποδηλώνεται από τα xdir, ydir εφόσον το πούλι που συναντά είναι του αντιπάλου. Αν συναντήσει πούλι του χρώματος του παίκτη που πραγματοποιεί την κίνηση επιστρέφει εισάγοντας σε μια λίστα τα πούλια που πρέπει να αντιστραφούν. Η συνάρτηση `all_directions_disks_to_flip` καλεί τη συνάρτηση `disks_to_flip` για όλες τις (8) πιθανές κατευθύνσεις.

```
#ifndef LAB03_BOARD_HPP_INCLUDED
#define LAB03_BOARD_HPP_INCLUDED

#include <cstdio>
#include <iostream>
#include <utility>
#include <list>
#include <vector>
#include <algorithm>
#include <climits>
#include <iomanip>

using namespace std;

const int ROWS{8};
const int COLS{8};

// αρχικοποίηση ταμπλό παιχνιδιού
char **initialize_board();

// σχεδίαση του ταμπλό στην οθόνη και απεικόνιση του σκορ
void draw_board(char **aboard);

// διάνυσμα με έγκυρες θέσεις στις οποίες μπροούν να τοποθετηθούν πούλια
vector<pair<int, int>> get_valid_positions(char **aboard, char disk);

// ενημέρωση του ταμπλό
void update_board(char **aboard, int r, int c, char disk);

// διαγραφή της μνήμης που καταλαμβάνει ένα ταμπλό
void delete_board(char **aboard);
```

```
// λίστα νέων ταμπλό που προκύπτουν αν τοποθετηθεί ένα πούλι τύπου (disk)
list<char **> get_successor_boards(char **aboard, char disk);

// αντιγραφή ενός ταμπλό
char **copy_board(char **aboard);

// πλήθος από πούλια τύπου disk
int get_disks_with_color(char **aboard, char disk);

#endif
```
lab03_board.hpp

```
#include "lab03_board.hpp"

// αντιγραφή ενός ταμπλό
char **copy_board(char **aboard) {
  char **duplicated_board = new char *[ROWS];
  for (int i = 0; i < ROWS; i++)
    duplicated_board[i] = new char[COLS];
  for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
      duplicated_board[i][j] = aboard[i][j];
  return duplicated_board;
}

// είναι οι συντεταγμένες r, c (γραμμή και στήλη) εντός των ορίων του ταμπλό;
bool in_bounds(int r, int c) {
  return (r >= 0 && r < ROWS && c >= 0 && c < COLS);
}

// λίστα με πούλια που θα άλλάξουν χρώμα στην κατεύθυνση που υποδηλώνεται από τα
// xdir, ydir
list<pair<int, int>> disks_to_flip(char **aboard, int r, int c, char disk,
                                   int xdir, int ydir) {
  aboard[r][c] = disk;
  char opposite_disk;
  if (disk == 'X')
    opposite_disk = 'O';
  else
    opposite_disk = 'X';
  int x = r, y = c;
  x += xdir;
  y += ydir;
  list<pair<int, int>> disks_to_flip{};
  if (in_bounds(x, y) && aboard[x][y] == opposite_disk) {
    x += xdir;
    y += ydir;
    if (in_bounds(x, y)) {
      while (aboard[x][y] == opposite_disk) {
        x += xdir;
        y += ydir;
```

```
        if (!in_bounds(x, y)) {
          break;
        }
      }
      if (in_bounds(x, y))
        if (aboard[x][y] == disk)
          while (true) {
            x -= xdir;
            y -= ydir;
            if ((x == r) && (y == c))
              break;
            disks_to_flip.push_back(make_pair(x, y));
          }
    }
  }
  aboard[r][c] = ' ';
  return disks_to_flip;
}

// λίστα με πούλια που θα αλλάξουν χρώμα σε όλες τις κατευθύνσεις
list<pair<int, int>> all_directions_disks_to_flip(char **aboard, int r, int c,
                                                   char disk) {
  list<pair<int, int>> list_of_disks_to_flip{};
  int directions[8][2] = {{0, 1},  {1, 1},   {1, 0},  {1, -1},
                          {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}};
  for (int i = 0; i < 8; i++) {
    list<pair<int, int>> l =
        disks_to_flip(aboard, r, c, disk, directions[i][0], directions[i][1]);
    list_of_disks_to_flip.insert(list_of_disks_to_flip.end(), l.begin(),
                                 l.end());
  }
  return list_of_disks_to_flip;
}

char **initialize_board() {
  char **board = new char *[ROWS] {};
  for (int i = 0; i < ROWS; i++) {
    board[i] = new char[COLS]{};
    for (int j = 0; j < COLS; j++)
      board[i][j] = ' ';
  }
  board[ROWS / 2 - 1][COLS / 2 - 1] = 'X';
  board[ROWS / 2 - 1][COLS / 2] = 'O';
  board[ROWS / 2][COLS / 2] = 'X';
  board[ROWS / 2][COLS / 2 - 1] = 'O';
  return board;
}

int get_disks_with_color(char **aboard, char disk) {
  int c{0};
  for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
      if (aboard[i][j] == disk)
```

```
        c++;
  return c;
}

void display_score(char **aboard) {
  cout << "Score X=" << get_disks_with_color(aboard, 'X')
       << " O=" << get_disks_with_color(aboard, 'O') << endl;
}

void draw_board(char **aboard) {
  cout << "   ";
  for (int j = 0; j < COLS; j++)
    cout << setw(2) << right << j;
  cout << endl;
  for (int i = 0; i < ROWS; i++) {
    cout << setw(2) << right << i << "|";
    for (int j = 0; j < ROWS; j++)
      cout << setw(1) << aboard[i][j] << "|";
    cout << endl;
  }
  display_score(aboard);
}

vector<pair<int, int>> get_valid_positions(char **aboard, char disk) {
  vector<pair<int, int>> valid_moves{};
  for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
      if (aboard[i][j] == ' ') {
        if (!all_directions_disks_to_flip(aboard, i, j, disk).empty())
          valid_moves.push_back(make_pair(i, j));
      }
  return valid_moves;
}

void update_board(char **aboard, int r, int c, char disk) {
  for (pair<int, int> p : all_directions_disks_to_flip(aboard, r, c, disk)) {
    aboard[p.first][p.second] = disk;
  }
  aboard[r][c] = disk;
}

void delete_board(char **aboard) {
  for (int i = 0; i < ROWS; i++)
    delete[] aboard[i];
  delete[] aboard;
}

list<char **> get_successor_boards(char **aboard, char disk) {
  list<char **> successors{};
  for (pair<int, int> p : get_valid_positions(aboard, disk)) {
    char **b = copy_board(aboard);
    update_board(b, p.first, p.second, disk);
    successors.push_back(b);
```

```
  }
  return successors;
}
```
lab03_board.cpp

Απεικόνιση του ταμπλό και χρήση των βασικών συναρτήσεων χειρισμού του ταμπλό

Στη συνέχεια παρουσιάζεται ένα απλό πρόγραμμα (lab03_01.cpp) το οποίο αρχικοποιεί ένα παιχνίδι, επιλέγει με τυχαίο τρόπο μια έγκυρη κίνηση για τον παίκτη X και εμφανίζει όλα τα πιθανά ταμπλό που μπορούν να προκύψουν για τον παίκτη O.

```
#include "lab03_board.hpp"

int main() {
  default_random_engine gen;
  gen.seed(time(NULL));
  char **board = initialize_board();
  draw_board(board);
  vector<pair<int, int>> valid_pairs_x = get_valid_positions(board, 'X');
  cout << "Valid moves for X: ";
  for (pair<int, int> p : valid_pairs_x)
    cout << "(" << p.first << "," << p.second << ") ";
  shuffle(valid_pairs_x.begin(), valid_pairs_x.end(), gen);
  pair<int, int> p = valid_pairs_x.front();
  cout << "Move selected: (" << p.first << "," << p.second << ")" << endl;
  update_board(board, p.first, p.second, 'X');
  draw_board(board);
  int c = 0;
  for (char **b : get_successor_boards(board, 'O')) {
    cout << "\nPossible next board " << ++c << endl;
    draw_board(b);
    delete_board(b);
  }
  delete_board(board);
}
```
lab03_01.cpp

Μεταγλώττιση και εκτέλεση του κώδικα

```
g++ lab03_board.cpp lab03_01.cpp -o lab03_01 -std=c++11
./lab03_01
```

```
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|O| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=2 O=2
Valid moves for X: (2,4) (3,5) (4,2) (5,3) Move
selected: (3,5)
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|X|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=4 O=1

Possible next board 1
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | |O| | | | |
 3| | | |O|X|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3
```

```
Possible next board 2
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | |O| | |
 3| | | |X|O|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3

Possible next board 3
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|X|X| | |
 4| | | |O|O|O| | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3
```

Παρτίδα παιχνιδιού ανάμεσα σε 2 παίκτες

Εδώ, θα παρουσιαστεί η υλοποίηση του κώδικα που θα επιτρέπει σε δύο παίκτες (ανθρώπους) να παίξουν ένα παιχνίδι Οθέλλο.

```cpp
#include "lab03_board.hpp"

void human_move(char** board, char disk);
```

lab03_human.hpp

```cpp
#include "lab03_human.hpp"

void human_move(char** board, char disk) {
    if (disk == 'X')
        cout << "Player X turn, valid moves=";
    else
        cout << "Player O turn, valid moves=";
    vector<pair<int, int>> valid_pairs = get_valid_positions(board, disk);
    for (pair<int, int> x : valid_pairs)
        cout << "(" << x.first << " " << x.second << ")";
    int r { }, c { };
    do {
```

```
        cout << endl << "Enter move:";
        cin >> r >> c;
    } while (find(valid_pairs.begin(), valid_pairs.end(), make_pair(r, c))
            == valid_pairs.end());
    update_board(board, r, c, disk);
    draw_board(board);
}
```

lab03_human.cpp

```
#include "lab03_board.hpp"
#include "lab03_human.hpp"

int main() {
  char** board = initialize_board();
  bool flag{false};
  int moves{0};
  draw_board(board);
  while (!flag) {
    bool player1_moved{false};
    bool player2_moved{false};
    if (!(get_valid_positions(board, 'X').empty())) {
      human_move(board, 'X');
      player1_moved = true;
      moves++;
    }
    if (!(get_valid_positions(board, 'O').empty())) {
      human_move(board, 'O');
      player2_moved = true;
      moves++;
    }
    if ((moves == ROWS * COLS - 4) || (!player1_moved && !player2_moved))
      flag = true;
  }
  delete_board(board);
}
```

lab03_02.cpp

Μεταγλώττιση και εκτέλεση του κώδικα

```
g++ lab03_board.cpp lab03_human.cpp lab03_02.cpp -o lab03_02 -std=c++11
./lab03_02
```

```
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|O| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=2 O=2
Player X turn, valid moves=(2 4)(3 5)(4 2)(5
3)
Enter move:2 4
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | |X| | | |
 3| | | |X|X| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=4 O=1
```

```
Player O turn, valid moves=(2 3)(2 5)(4 5)
Enter move:2 3
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | |O|X| | | |
 3| | | |O|X| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3
Player X turn, valid moves=(1 2)(2 2)(3 2)(4
2)(5 2)
Enter move: ...
...
```

Αυτόματοι επιλυτές

Στη συνέχεια θα παρουσιαστούν 3 αυτόματοι επιλυτές: ένας ευρετικός επιλυτής, ο επιλυτής minimax και ο επιλυτής άλφα-βήτα. Ο κώδικας και για τους 3 επιλυτές βρίσκεται στα αρχεία lab03_solvers.hpp και lab03_solvers.cpp.

```
#include "lab03_board.hpp"

// η κίνηση που θα γίνει αποφασίζεται από μια απλή ευρετική συνάρτηση που
// μετρά για κάθε έγκυρη θέση τον αριθμό από πούλια που θα προκύψει
// και επιλέγει την κίνηση με την μεγαλύτερη τιμή
void computer_move_using_simple_heuristic(char **board, char disk);

void computer_move_using_min_max(char **board, int plies, char disk);

void computer_move_using_alpha_beta(char **board, int plies, char disk);
```

lab03_solvers.hpp

```
#include "lab03_solvers.hpp"

void computer_move_using_simple_heuristic(char **board, char disk) {
  cout << "Computer using heuristic plays with disk " << disk << endl;
  int best_r{-1};
  int best_c{-1};
  int best_value{-1};
  for (pair<int, int> x : get_valid_positions(board, disk)) {
    char **duplicated_board = copy_board(board);
    int r{x.first};
    int c{x.second};
    update_board(duplicated_board, r, c, disk);
    int value = get_disks_with_color(duplicated_board, disk);
    if (value > best_value) {
```

```
      best_r = r;
      best_c = c;
      best_value = value;
    }
    cout << "(" << x.first << " " << x.second << ")->" << value << " ";
    delete_board(duplicated_board);
  }
  cout << endl;
  update_board(board, best_r, best_c, disk);
  draw_board(board);
}

// MINIMAX & ALPHA-BETA

// ευρετική συνάρτηση για την αριθμητική σχέση στα πούλια των δύο αντιπάλων
int parity(char **aboard) {
  int max_player_disks = get_disks_with_color(aboard, 'O');
  int min_player_disks = get_disks_with_color(aboard, 'X');
  return 100 * (max_player_disks - min_player_disks) /
         (max_player_disks + min_player_disks);
}

// ευρετική συνάρτηση για τη σχέση κινητικότητας ανάμεσα στους δύο αντιπάλους
int mobility(char **aboard) {
  int max_player_moves = get_valid_positions(aboard, 'O').size();
  int min_player_moves = get_valid_positions(aboard, 'X').size();
  if (max_player_moves + min_player_moves == 0)
    return 0;
  else
    return 100 * (max_player_moves - min_player_moves) /
           (max_player_moves + min_player_moves);
}

// αριθμός από γωνίες στις οποίες βρίσκεται πούλι τύπου disk
int player_corners(char **aboard, char disk) {
  int c{0};
  if (aboard[0][0] == disk)
    c++;
  if (aboard[0][COLS - 1] == disk)
    c++;
  if (aboard[ROWS - 1][0] == disk)
    c++;
  if (aboard[ROWS - 1][COLS - 1] == disk)
    c++;
  return c;
}

// ευρετική συνάρτηση για τη σχέση κατειλημμένων γωνιών από τους δύο αντιπάλους
int corners(char **aboard) {
  int max_player_corners = player_corners(aboard, 'O');
  int min_player_corners = player_corners(aboard, 'X');
  if (max_player_corners + min_player_corners == 0)
    return 0;
```

```
  else
    return 100 * (max_player_corners - min_player_corners) /
           (max_player_corners + min_player_corners);
}

// συνάρτηση χρησιμότητας
int utility(char **aboard) {
  return parity(aboard) + mobility(aboard) + corners(aboard);
}

bool is_complete(char **aboard) {
  int c{0};
  for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
      if ((aboard[i][j] == 'X') || (aboard[i][j] == 'O'))
        c++;
  if (c == ROWS * COLS)
    return true;
  else
    return false;
}

int min_value(char **aboard, int plies);

int max_value(char **aboard, int plies) {
  if (is_complete(aboard) || plies == 0 ||
      get_valid_positions(aboard, 'O').empty())
    return utility(aboard);
  list<char **> succ_boards = get_successor_boards(aboard, 'O');
  int v = INT_MIN;
  for (char **b : succ_boards) {
    int v2 = min_value(b, plies - 1);
    if (v2 > v)
      v = v2;
    delete_board(b);
  }
  return v;
}

int min_value(char **aboard, int plies) {
  if (is_complete(aboard) || plies == 0 ||
      get_valid_positions(aboard, 'X').empty())
    return utility(aboard);
  list<char **> succ_boards = get_successor_boards(aboard, 'X');
  int v = INT_MAX;
  for (char **b : succ_boards) {
    int v2 = max_value(b, plies - 1);
    if (v2 < v)
      v = v2;
    delete_board(b);
  }
  return v;
}
```

```
pair<int, int> minimax_decision(char **aboard, int plies, char disk) {
  if (disk == 'O') {
    int max = INT_MIN;
    pair<int, int> move;
    for (pair<int, int> p : get_valid_positions(aboard, disk)) {
      char **b = copy_board(aboard);
      update_board(b, p.first, p.second, 'O');
      int v = max_value(b, plies - 1);
      if (v > max) {
        max = v;
        move.first = p.first;
        move.second = p.second;
      }
      delete_board(b);
    }
    return move;
  } else {
    int min = INT_MAX;
    pair<int, int> move;
    for (pair<int, int> p : get_valid_positions(aboard, disk)) {
      char **b = copy_board(aboard);
      update_board(b, p.first, p.second, 'X');
      int v = min_value(b, plies - 1);
      if (v < min) {
        min = v;
        move.first = p.first;
        move.second = p.second;
      }
      delete_board(b);
    }
    return move;
  }
}

void computer_move_using_min_max(char **board, int plies, char disk) {
  cout << "Computer using min-max plays with disk " << disk << endl;
  pair<int, int> p = minimax_decision(board, plies, disk);
  update_board(board, p.first, p.second, disk);
  draw_board(board);
}

int ab_min_value(char **aboard, int alpha, int beta, int plies);

int ab_max_value(char **aboard, int alpha, int beta, int plies) {
  if (is_complete(aboard) || plies == 0 ||
      get_valid_positions(aboard, 'O').empty())
    return utility(aboard);
  list<char **> succ_boards = get_successor_boards(aboard, 'O');
  int v = INT_MIN;
  for (char **b : succ_boards) {
    int v2 = ab_min_value(b, alpha, beta, plies - 1);
    if (v2 > v)
```

```
      v = v2;
    if (v >= beta) {
      delete_board(b);
      return v;
    }
    if (v > alpha)
      alpha = v;
    delete_board(b);
  }
  return v;
}

int ab_min_value(char **aboard, int alpha, int beta, int plies) {
  if (is_complete(aboard) || plies == 0 ||
      get_valid_positions(aboard, 'X').empty())
    return utility(aboard);
  list<char **> succ_boards = get_successor_boards(aboard, 'X');
  int v = INT_MAX;
  for (char **b : succ_boards) {
    int v2 = ab_max_value(b, alpha, beta, plies - 1);
    if (v2 < v)
      v = v2;
    if (v <= alpha) {
      delete_board(b);
      return v;
    }
    if (v < beta)
      beta = v;
    delete_board(b);
  }
  return v;
}

pair<int, int> ab_minimax_decision(char **aboard, int plies, char disk) {
  if (disk == 'O') {
    int max = INT_MIN;
    pair<int, int> move;
    for (pair<int, int> p : get_valid_positions(aboard, disk)) {
      char **b = copy_board(aboard);
      update_board(b, p.first, p.second, 'O');
      int v = ab_max_value(b, INT_MIN, INT_MAX, plies - 1);
      if (v > max) {
        max = v;
        move.first = p.first;
        move.second = p.second;
      }
      delete_board(b);
    }
    return move;
  } else {
    int min = INT_MAX;
    pair<int, int> move;
    for (pair<int, int> p : get_valid_positions(aboard, disk)) {
```

```
      char **b = copy_board(aboard);
      update_board(b, p.first, p.second, 'X');
      int v = ab_min_value(b, INT_MIN, INT_MAX, plies - 1);
      if (v < min) {
        min = v;
        move.first = p.first;
        move.second = p.second;
      }
      delete_board(b);
    }
    return move;
  }
}

void computer_move_using_alpha_beta(char **board, int plies, char disk) {
  cout << "Computer using alpha-beta plays with disk " << disk << endl;
  pair<int, int> p = ab_minimax_decision(board, plies, disk);
  update_board(board, p.first, p.second, disk);
  draw_board(board);
}
```
lab03_solvers.cpp

Παρτίδα παιχνιδιού με παίκτη (άνθρωπο) και αντίπαλο τον ευρετικό επιλυτή

Ο αυτόματος ευρετικός επιλυτής προκειμένου να αποφασίσει την κίνηση που πρόκειται να κάνει εξετάζει όλες τις πιθανές κινήσεις και επιλέγει την κίνηση η οποία θα μετατρέψει περισσότερα πούλια στο δικό του χρώμα.

```
#include "lab03_board.hpp"
#include "lab03_human.hpp"
#include "lab03_solvers.hpp"

int main() {
  char **board = initialize_board();
  bool flag{false};
  int moves{0};
  draw_board(board);
  while (!flag) {
    bool player1_moved{false};
    bool player2_moved{false};
    if (!(get_valid_positions(board, 'X').empty())) {
      human_move(board, 'X');
      player1_moved = true;
      moves++;
    }
    if (!(get_valid_positions(board, 'O').empty())) {
      computer_move_using_simple_heuristic(board, 'O');
      player2_moved = true;
      moves++;
    }
    if ((moves == ROWS * COLS - 4) || (!player1_moved && !player2_moved))
      flag = true;
  }
  delete_board(board);
}
```

```
lab03_03.cpp
```

Μεταγλώττιση και εκτέλεση του κώδικα

```
g++ lab03_board.cpp lab03_human.cpp lab03_solvers.cpp lab03_03.cpp -o lab03_03 -std=c++11
./lab03_03
```

```
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|O| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=2 O=2
Player X turn, valid moves=(2 4)(3 5)(4 2)(5
3)
Enter move:3 5
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|X|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=4 O=1
Computer using heuristic plays with disk O
(2 3)->3 (2 5)->3 (4 5)->3
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | |O| | | | |
 3| | | |O|X|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3
```

```
Player X turn, valid moves=(1 2)(2 2)(3 2)(4
2)(5 2)
Enter move:3 2
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | |O| | | | |
 3| | |X|X|X|X| | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=5 O=2
Computer using heuristic plays with disk O
(2 1)->4 (2 5)->4 (4 1)->4 (4 5)->5
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | |O| | | | |
 3| | |X|X|O|X| | |
 4| | | |O|O|O| | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=5
Player X turn, valid moves=(1 3)(1 4)(5 3)(5
4)(5 5)
Enter move: ...
...
```

Ο αλγόριθμος minimax και ο αλγόριθμος alpha-beta στο παιχνίδι Οθέλλο

Η συνάρτηση χρησιμότητας `int utility(char **aboard)` επιστρέφει μια ακέραια τιμή για κάθε διαμόρφωση του ταμπλό. Τιμές με υψηλότερες θετικές τιμές είναι προτιμότερες για τον παίκτη ‘O’ ενώ συμμετρικά μικρότερες αρνητικές τιμές είναι προτιμότερες για τον παίκτη ‘X’. Η συνάρτηση χρησιμότητας καλεί τρεις ευρετικές συναρτήσεις προκειμένου να διαμορφώσει την τιμή της. Οι ευρετικές συναρτήσεις αφορούν:

1. τον αριθμό από πούλια που έχει στο ταμπλό ο κάθε παίκτης
2. τον αριθμό των κινήσεων που είναι σε θέση να κάνει ο κάθε παίκτης
3. τον αριθμό από γωνίες που έχει καταλάβει ο κάθε παίκτης

Εναλλακτικά, θα μπορούσε να χρησιμοποιηθεί κάποια άλλη συνάρτηση χρησιμότητας.

Παρτίδα παιχνιδιού με παίκτη και αντίπαλο τον αλγόριθμο minimax

```
#include "lab03_board.hpp"
#include "lab03_human.hpp"
#include "lab03_solvers.hpp"

int main(int argc, char *argv[]) {
  if (argc != 3)
    exit(-1);
```

```
  string solver = argv[1];
  int plies = atoi(argv[2]);
  char **board = initialize_board();
  bool flag{false};
  int moves{0};
  draw_board(board);
  while (!flag) {
    bool player1_moved{false};
    bool player2_moved{false};
    if (!(get_valid_positions(board, 'X').empty())) {
      human_move(board, 'X');
      player1_moved = true;
      moves++;
    }
    if (!(get_valid_positions(board, 'O').empty())) {
      if (solver.compare("MINIMAX")==0)
        computer_move_using_min_max(board, plies, 'O');
      else
        computer_move_using_alpha_beta(board, plies, 'O');
      player2_moved = true;
      moves++;
    }
    if ((moves == ROWS * COLS - 4) || (!player1_moved && !player2_moved))
      flag = true;
  }
  delete_board(board);
}
```
lab03_04.cpp

Μεταγλώττιση και εκτέλεση του κώδικα

```
g++ lab03_board.cpp lab03_human.cpp lab03_solvers.cpp lab03_04.cpp -o lab03_04 -std=c++11
./lab03_04 MINIMAX 5
```

```
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|O| | | |
 4| | | |O|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=2 O=2
Player X turn, valid moves=(2 4)(3 5)(4 2)(5
3)
Enter move:4 2
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | | | | |
 3| | | |X|O| | | |
 4| | |X|X|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=4 O=1
Computer using min-max plays with disk O
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
```

```
Computer using min-max plays with disk O
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2|O| | | | |X| | |
 3| |O|O|O|X|O| | |
 4| | |O|O|O| | | |
 5| |O|O|O|O| | | |
 6| | |X| | | | | |
 7|X|X|X| | | | | |
Score X=6 O=12
Player X turn, valid moves=(2 2)(2 6)(3 0)(3
6)(4 0)(4 5)(6 1)(6 4)
Enter move:6 1
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2|O| | | | |X| | |
 3| |O|O|O|X|O| | |
 4| | |O|X|O| | | |
 5| |O|X|O|O| | | |
 6| |X|X| | | | | |
 7|X|X|X| | | | | |
Score X=9 O=10
Computer using min-max plays with disk O
   0 1 2 3 4 5 6 7
```

```
 1| | | | | | | | |
 2| | | | | | | | |
 3| | |O|O|O| | | |
 4| | |X|X|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=3 O=3
Player X turn, valid moves=(2 1)(2 2)(2 3)(2
4)(2 5)
Enter move:2 5
   0 1 2 3 4 5 6 7
 0| | | | | | | | |
 1| | | | | | | | |
 2| | | | | |X| | |
 3| | |O|O|X| | | |
 4| | |X|X|X| | | |
 5| | | | | | | | |
 6| | | | | | | | |
 7| | | | | | | | |
Score X=5 O=2
...
```

```
 0| | | | | | | | |
 1| | | | | | | | |
 2|O| | | |O|X| | |
 3| |O|O|O|O|O| | |
 4| | |O|X|O| | | |
 5| |O|X|O|O| | | |
 6| |X|X| | | | | |
 7|X|X|X| | | | | |
Score X=8 O=12
Player X turn, valid moves=(2 1)(2 2)(2 3)(2
6)(4 0)(4 1)(4 5)(5 0)(5 5)(6 3)(6 5)
Enter move: ...
...
```

Εκτέλεση του κώδικα για τον άλφα-βήτα με 5 στρώσεις

```
./lab03_04 ALPHABETA 5
```

Τα ίδια αποτελέσματα με το MINIMAX αλλά ταχύτερα

Αναφορές

1. Τεχνητή Νοημοσύνη – μια σύγχρονη προσέγγιση, Β’ έκδοση, Stuart Russell, Peter Norvig, Εκδόσεις Κλειδάριθμος, 2003.
2. Τεχνητή Νοημοσύνη, Γ’ έκδοση, Ι. Βλαχαβάς, Π. Κεφαλάς, Ν. Βασιλειάδης, Φ. Κόκορας , Η. Σακελλαρίου, Γκιούρδας Εκδοτική, 2006.
3. Introduction to Artificial Intelligence, Wolfgang Ertel, Springer, 2011.
4. https://inventwithpython.com/chapter15.html
5. https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/
6. http://www.cc.gatech.edu/~riedl/classes/2014/cs3600/homeworks/minimax-prob.pdf
7. https://medium.freecodecamp.com/simple-chess-ai-step-by-step-1d55a9266977