# ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ
# Candy Crush


## Εργασία C:

# MinMax Algorithm

**Ράπτης Δημήτριος** **Α.Ε.Μ.: 8467** **e-mail: dmraptis@auth.gr**

**Παπαγεωργίου Θωμάς** **Α.Ε.Μ.: 8577** **e-mail: tompap@live.com**

# ΕΙΣΑΓΩΓΗ

Σε αυτή τη δεύτερη καλούμαστε να υλοποιήσουμε τον **αλγόριθμο MinMax** για να βελτιστοποιήσουμε τον παίχτη μας. Δημιουργήθηκε ένα δέντρο βάθους 2 (λόγω πίεσης χρόνου..) που αξιολογεί τις κινήσεις μας και τις κινήσεις του αντιπάλου μας και επιλέγει την βέλτιστη για εμάς κίνηση.

Επίσης βελτιστοποιήθηκε ο κώδικας του Heuristic Player καθώς προστέθηκε και η αξιολόγηση για τα chain moves.

Παρατηρήθηκε στατιστικά πως η ύπαρξη της συνάρτηση `sameColorInProximity()` έκανε τον MinMax Player λιγότερο αποδοτικό, οπότε απενεργοποιήθηκε η χρήση της στην κλάση Heuristic Player. Αυτό συμβαίνει καθώς νέα πλακίδια προσθέτονται στο ταμπλό με τυχαίο χρώμα οπότε η αξιολόγηση αυτή έβγαζε διαφορετικό αποτέλεσμα από ότι στον πραγματική ροή. Έτσι γινόταν λάθος επιλογή κίνησης.

Μερικά γενικά σχόλια που θα βοηθήσουν στην κατανόηση του κώδικα:

- Έχουν χρησιμοποιηθεί αρκετές `println` για λόγους debugging οι οποίες εμφανίζονται με την μορφή σχολίων στον κώδικα για δύο λόγους. Για την περίπτωση μελλοντική αναδιαμόρφωση του κώδικα ή για την πιθανή ανάγκη ελέγχου ορισμένων μεταβλητών από τον διορθωτή της.
- Έχουν δημιουργηθεί ορισμένες παραπάνω συναρτήσεις για καλύτερη κατανόηση του κώδικα που γράφτηκε για τον αλγόριθμο AB Pruning στην `chooseMove()`.

# CLASS NODE84678577

Δημιουργήθηκαν οι Getters και οι Setters της κλάσης, οι οποίοι είναι εύκολοι στην κατανόηση οπότε δεν θα σχολιαστούν.

Όσον αφορά τους Constructors δημιουργήθηκαν δύο:

- `public Node84678577 (int nodeDepth, Board nodeBoard)`

  Ο Constructor αυτός είναι για την ρίζα του δέντρου, διότι χρειάζεται μόνο το ταμπλό της αρχικής κατάστασης και το βάθος του κόμβου για να προσδιοριστεί.

- `public Node84678577 (Node84678577 parent, int nodeDepth, int [] nodeMove, Board nodeBoard)`

  Ο Constructor αυτός είναι για τους υπόλοιπους κόμβους που χρειάζονται πλήρη προσδιορισμό.

*Αξίζει να σημειωθεί πως η εντολή :

```
this.children = new ArrayList<Node84678577>();
```

στους Constructors αρχικοποιεί το `ArrayList` των παιδιών του κόμβου.

Τέλος τα παιδιά θα προσθέτονται ένα-ένα μετά την δημιουργία τους οπότε απαιτείται μια ξεχωριστή συνάρτηση για αυτή την λειτουργία.

```
public void addChild (Node84678577 child)
```

Η συνάρτηση αυτή είναι η παραπάνω και απλώς προσθέτει ένα παιδί στην `ArrayList` με όλα τα παιδιά του κόμβου:

# CLASS HEYRISTIC PLAYER

Οι τροποποιήσεις που έγιναν είναι οι εξής δύο:

- Όπως αναφέραμε προηγουμένως αφαιρέθηκε η συνάρτηση `sameColorInProximity()` για να λειτουργήσει ορθότερα ο MinMax παίκτης μας.
- Προστέθηκε η συνάρτηση «`int chainMoves (Board board)`» για τον υπολογισμό των ζαχαρωτών που θα διαγραφούν λόγω των chain moves.

`int chainMoves (Board board);`

Η συνάρτηση αυτή επιστρέφει μια `int` τιμή, που ουσιαστικά είναι ο αριθμός των διαγραμμένων ζαχαρωτών.

ΥΛΟΠΟΙΗΣΗ:

Δημιουργεί ένα ταμπλό με διαγραμμένα τα ζαχαρωτά που συμμετείχαν στην κίνηση και τις κενές θέσεις στην κορυφή του ταμπλό γεμάτες με νέα τυχαία ζαχαρωτά.

Έπειτα ελέγχει όλων τον πίνακα για τυχόν νέες κατακόρυφες ή οριζόντιες ν-άδες .

Για τις κατακόρυφες ν-άδες χρησιμοποιείται η συνάρτηση:

`deletedCandiesAtColumn()`

ενώ για τις οριζόντιες η:

`deletedCandiesAtRow()`

# CLASS MINMAX PLAYER

```
private void createMySubTree (Node84678577 parent, int depth)
```

Δημιουργεί το υποδέντρο του παίχτη μας.
Τα παιδιά του κόμβου `parent` που δημιουργούνται θα έχουν βάθος ίσο με `depth.`

ΥΛΟΠΟΙΗΣΗ:

Αρχικά βρίσκουμε τις διαθέσιμες κινήσεις για τον κόμβο `parent` δίνοντας ως όρισμα το ταμπλό του στην συνάρτηση `CrushUtilities.getAvailableMoves` .

Στην συνέχεια για κάθε διαθέσιμη κίνηση βρίσκουμε το ταμπλό μετά την εφαρμογή της συγκεκριμένης κίνησης και αρχικοποιούμε το παιδί του parent με τις εξής παραμέτρους.

- Πατέρας: `parent`
- Βάθος: `depth`
- Κίνηση: την διαθέσιμη κίνηση που εξετάζουμε
- Ταμπλό: το ταμπλό που βρήκαμε προηγουμένως

Για την αξιολόγηση του κόμβου ελέγχουμε στην αρχή αν ο παίκτης νικάει (Εάν το σκορ του παίχτη υπερβαίνει την τιμή 300).
Αν ναι τότε ορίζουμε ως αξιολόγηση του κόμβου το θετικό άπειρο.
Αλλιώς δημιουργούμε ένα αντικείμενο της κλάσης `HeuristicPlayer` για να χρησιμοποιήσουμε την συνάρτηση αξιολόγησης `moveEvaluation()` και να αξιολογήσουμε τον κόμβο.

Προσθέτουμε το παιδί στον πατέρα και καλούμε την συνάρτηση
`createOpponentSubTree (tempNode, depth + 1)`
με ορίσματα τον κόμβο που μόλις δημιουργήσαμε και βάθος μεγαλύτερο κατά ένα.

Έτσι δημιουργείται και το υποδέντρο του παιδιού και τελικά, όταν επαναληφθεί αυτή η διαδικασία για όλες τις κινήσεις, όλο το δέντρο.

```
private void createOpponentSubTree (Node84678577 parent, int depth)
```

Δημιουργεί το υποδέντρο του αντιπάλου μας.
Τα παιδιά του κόμβου `parent` που δημιουργούνται θα έχουν βάθος ίσο με `depth`.

ΥΛΟΠΟΙΗΣΗ:

Αρχικά δημιουργούμε τον πατέρα του κόμβου parent που δίνεται ως όρισμα για να πάρουμε το αρχικό ταμπλό `initialBoard` (το ταμπλό της ρίζας του δέντρου). Στο ταμπλό αυτό εφαρμόζουμε την *`boardAfterFullMove`*`()` και παίρνουμε το ταμπλό μετά την εφαρμογή της κίνησης του parent και αφού διαγραφούν ΟΛΕΣ οι ν-αδες, το `fullBoard`. Βρίσκουμε μετά τις διαθέσιμες κινήσεις του `fullBoard`.

Για κάθε διαθέσιμη κίνηση ακολουθείται η διαδικασία που εφαρμόστηκε στην *`boardAfterFullMove`* με την μόνη διαφορά πως στην αξιολόγηση του κόμβου βάζουμε το αρνητικό άπειρο (αν νικάει ο αντίπαλος) ή την αρνητική αξιολόγηση που μας δίνει η συνάρτηση `moveEvaluation()` της κλάσης `heuristicPlayer` .

Τέλος προσθέτουμε τον κάθε κόμβο-παιδί στον πατέρα.

```
private void addEvaluationToChildren (Node84678577 root)
```

Δεχόμαστε σαν όρισμα έναν κόμβο και προσθέτουμε την αξιολόγησή του στα παιδιά του.

ΥΛΟΠΟΙΗΣΗ:

Αρχικά παίρνουμε τα παιδιά του `parent` και για κάθε παιδί ως αξιολόγησή του ορίζουμε αυτήν που έχει ήδη συν την αξιολόγηση του `parent`.

```
private double findMinEvaluationOfChildren (Node84678577 root)
```

Δεχόμαστε σαν όρισμα έναν κόμβο και βρίσκουμε την ελάχιστη τιμή της αξιολόγησης των παιδιών του. Επιστρέφεται αυτή η τιμή.

ΥΛΟΠΟΙΗΣΗ:

Αρχικά παίρνουμε τα παιδιά του `parent` και αρχικοποιούμε την μεταβλητή `min` με την μεγαλύτερη πιθανή αξιολόγηση, δηλαδή το θετικό άπειρο.

Για κάθε παιδί ελέγχουμε εάν η αξιολόγηση του είναι μικρότερη από την μεταβλητή `min` και αν είναι θέτουμε αυτή ως `min`. Έτσι βρίσκουμε την ελάχιστη αξιολόγηση.
Αφού ελεγχθούν όλα τα παιδιά επιστρέφουμε την ελάχιστη αξιολόγηση που βρήκαμε.

```
private double findMaxEvaluationOfChildren (Node84678577 root)
```

Όμοια συνάρτηση με την `findMinEvaluationOfChildren()`,απλώς επιστρέφεται η μέγιστη τιμή των παιδιών του parent.

```
private int findIndexOfEvaluation (Node84678577 root, double evaluation)
```

Δεχόμαστε σαν όρισμα έναν κόμβο και μια τιμή αξιολόγησης και βρίσκουμε τον δείκτη του παιδιού με αυτή την αξιολόγηση.
Εάν βρεθεί επιστρέφεται ο δείκτης, αλλιώς η τιμή `-1`.

ΥΛΟΠΟΙΗΣΗ:

Αρχικά παίρνουμε τα παιδιά του `parent` και για κάθε παιδί ελέγχω εάν η αξιολόγηση του είναι ίδια με την μεταβλητή `evaluation`. Εάν είναι επιστρέφω τον δείκτη του παιδιού. Στην περίπτωση που ελεγχτούν όλα τα παιδιά και δεν βρεθεί ταύτιση επιστρέφεται η τιμή `-1`.

```
private int chooseMove (Node84678577 root)
```

Δέχεται σαν όρισμα την ρίζα του δέντρου μας και επιστρέφει τον δείκτη της βέλτιστης κίνησης για τον παίχτη μας σύμφωνα με τον αλγόριθμο **MinMax**. Δουλεύει για δέντρο βάθους 2 η συγκεκριμένη συνάρτηση. Για μεγαλύτερο βάθος χρειάζονται τροποποιήσεις.

ΥΛΟΠΟΙΗΣΗ:

Αρχικά παίρνουμε τα παιδιά της ρίζας `root` .

Για κάθε παιδί προσθέτω την αξιολόγηση του στα εγγόνια της ρίζας (`grandchildren` της `root` σε βάθος 2) καλώντας την `addEvaluationToChildren()` και όρισμα το παιδί.

Έπειτα μηδενίζω την αξιολόγηση του παιδιού (είναι περιττό βήμα).

Βρίσκω την ελάχιστη αξιολόγηση των εγγονιών και την τοποθετώ ως αξιολόγηση του παιδιού, με την εντολή:

```
children.get(i).setNodeEvaluation( findMinEvaluationOfChildren
                    (children.get(i)) );
```

Ολοκληρώνω την διαδικασία για όλα τα παιδιά και στην συνέχεια βρίσκω την μέγιστη αξιολόγηση των παιδιών της ρίζας και ορίζω την αξιολόγηση της ρίζας με αυτήν την τιμή που βρήκα. Χρησιμοποιείται η αντίστοιχη συνάρτηση με όρισμα την ρίζα του δέντρου:

```
root.setNodeEvaluation (findMaxEvaluationOfChildren (root));
```

Τέλος ελέγχω τα παιδιά και βρίσκω τον δείκτη αυτού που έχει την ίδια αξιολόγηση με την ρίζα με την βοήθεια της:
`findIndexOfEvaluation(root, root.getNodeEvaluation()).`

Πρέπει να τονιστεί πως η παραπάνω εντολή δεν γίνεται να επιστρέψει `-1` καθώς ξέρουμε πως κάποιο παιδί θα έχει αξιολόγηση ίση με την αξιολόγηση της ρίζας.

Προφανώς επιστρέφεται ο δείκτης αυτός.

```
public int[] getNextMove (ArrayList<int[]> availableMoves, Board board)
```

Η συνάρτηση αυτή επιστρέφει ένας `int[4]` πίνακα, που είναι η καλύτερη κίνηση του παίχτη μας.
Η επιστρεφόμενη τιμή είναι στην μορφή που επιστρέφει η συνάρτηση `CrushUtilities.calculateNextMove()`, δηλαδή `[x1 y1 x2 y2]`.

ΥΛΟΠΟΙΗΣΗ:

Δημιουργεί ένα αντίγραφο του ταμπλό της τωρινής κατάστασης.

Στη συνέχεια αρχικοποιείται η ρίζα του δέντρο μας, με βάθος 0 και ως ταμπλό το αντίγραφο που δημιουργήσαμε.

Αρχικοποιείται το δέντρο μας με την συνάρτηση:

```
createMySubTree (root, 1);
```

Και ουσιαστικά με την παραπάνω εντολή έχουμε δημιουργήσει ολόκληρο το δέντρο μας, βάθους 2.

Μετά υπολογίζουμε τον δείκτη της καλύτερης κίνησής μας με την:

```
int indexBest = chooseMove (root);
```

Βρίσκουμε την καλύτερη κίνηση (ξέροντας τον δείκτη της) και την επιστρέφουμε ως όρισμα στην `CrushUtilities.calculateNextMove()`.

```
public void showTree (Node84678577 root)
```

Η δημιουργία αυτής της συνάρτησης αποσκοπεί απλώς στην διευκόλυνση του debugging. Εμφανίζει στην κονσόλα το δέντρο με ρίζα τον κόμβο `root`.
Μόνοι περιορισμοί: το δέντρο να είναι βάθους 2 και ως όρισμα να δοθεί η ρίζα του.