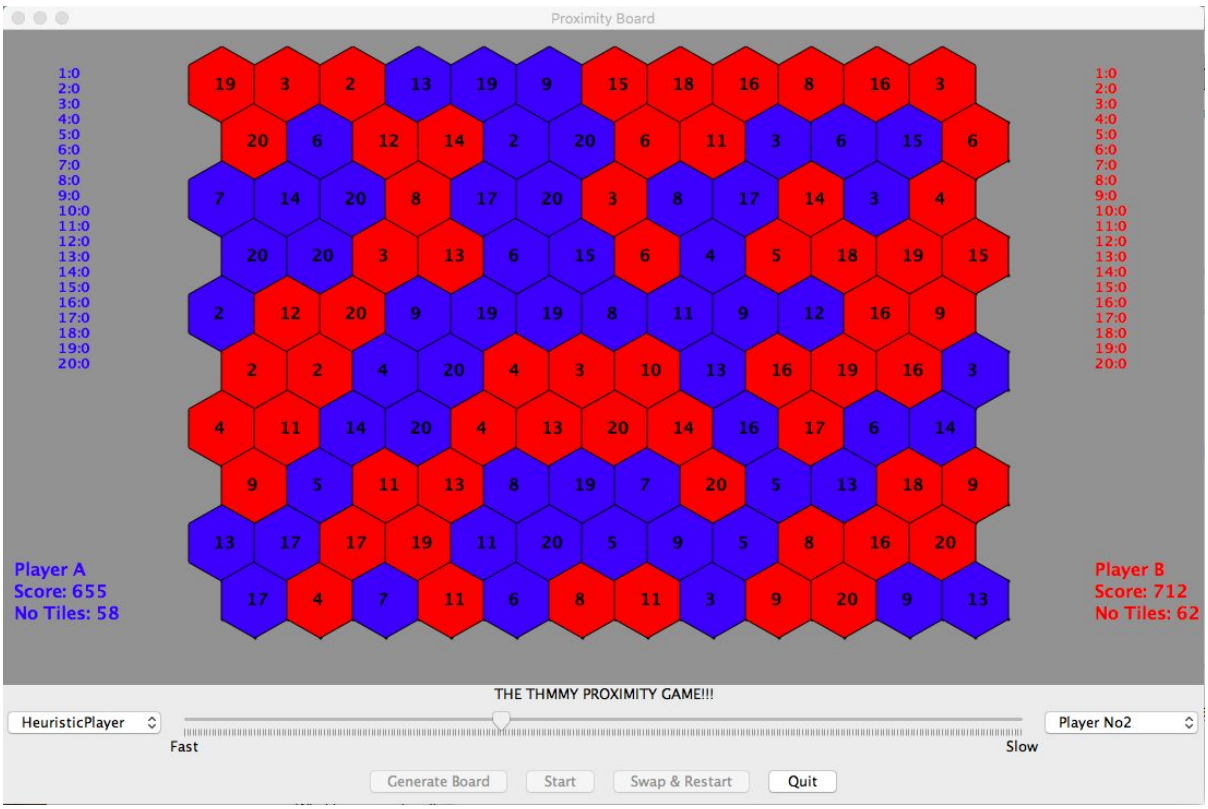
Εργαστήριο Επεξεργασίας Πληροφορίας και Υπολογισμών

Τομέας Ηλεκτρονικής και Υπολογιστών

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Α.Π.Θ



DS Proximity

Task 2 Report

Mamalakis Evangelos | 8191
Evangelou Alexandros | 8309

Το πρόβλημα

Ζητείται η δημιουργία ενός λειτουργικού παίκτη του ds-proximity ο οποίος θα χρησιμοποιεί ένα heuristic αλγόριθμο για να επιλέξει την επόμενη κίνηση του. Ο παίκτης θα λαμβάνει υπόψιν του τα δεδομένα που έχει διαθέσιμα την κάθε στιγμή.

Υλοποίηση της HeuristicPlayer.getNextMove()

Η συνάρτηση getNextMove() επιλέγει την επόμενη κίνηση με βάσει τις αξιολογήσεις που παίρνει από την συνάρτηση getEvaluation(). Αρχικά προσδιορίζει τις πιθανές κινήσεις, δηλαδή τα κενά κελιά και τα αποθηκεύει σε μία λίστα ArrayList, μαζί με τις αξιολογήσεις τους.

Για την εύκολη αποθήκευση τους μετατρέπει τις συντεταγμένες x, y σε δεκαδικές τιμές(double), ώστε να ταιριάζουν με την αξιολόγηση και στη συνέχεια τις αποθηκεύει όλες μαζί σε ένα array:

```
        Tile t = board.getTile(x,y);
        Double e = getEvaluation(board, randomNumber, t);
        possibleMoves.add( new double[] { x, y, e } );
```

Στη συνέχεια επιλέγει την κίνηση με την καλύτερη αξιολόγηση και την επιστρέφει. Για να το πετύχει αυτό χρησιμοποιεί την συνάρτηση Collections.max από την αντίστοιχη βιβλιοθήκη, ορίζοντας ταυτόχρονα τον κατάλληλο Comparator που συγκρίνει τις αξιολογήσεις τις κάθε κίνησης:

```
double[] bestMove = Collections.max(possibleMoves, new Comparator<double[]>() {
        public int compare(double[] move1, double[] move2) {
                return (int)(move1[2] - move2[2]);
        }
});
```

Υλοποίηση της HeuristicPlayer.isTaken()

```
        private boolean isTaken(Board board, int x, int y) {
                return board.getTile(x, y).getColor() != 0;
        }
```

Η συνάρτηση αυτή αναλαμβάνει να προσδιορίσει αν το δοσμένο εξάγωνο με συντεταγμένες x, y έχει ήδη καταληφθεί από κάποιον παίκτη. Για το σκοπό αυτό αξιοποιείται η ιδιότητα color της κλάσης Tile. Όταν και μόνο όταν το εξάγωνο είναι κενό η color έχει την τιμή μηδέν.

Υλοποίηση της HeuristicPlayer.getEvaluation()

Η συνάρτηση getEvaluation() υλοποιεί ένα κατά το δυνατόν από ευριστικό αλγόριθμο για να υπολογίσει την αξιολόγηση μιας κίνησης. Συγκεκριμένα κίνηση αξιολογείται με βάση δύο παραμέτρους: την scoreGain και την coveredGain. Κάθε γειτονικό πλακίδιο εξετάζεται ξεχωριστά, και ανάλογα με τον τύπο του (κενό, εκτός του ταμπλό, κατειλημμένο από τον παίκτη, κατειλημμένο από τον αντίπαλο) μεταβάλλονται οι τιμές scoreGain και coveredGain:

```
Tile[] neighbors = ProximityUtilities.getNeighbors(tile.getX(), tile.getY(), board);
for (Tile neighbor:neighbors) { ... }
```

Στο τέλος επιστρέφουμε το άθροισμα των scoreGain και coveredGain.

Η παράμετρος scoreGain

Η scoreGain μετράει απλά πόσο θα αυξηθεί το σκορ του παίκτη με την επιλεγμένη κίνηση. *Δεν υπολογίζει το σκορ του παιζόμενου πλακιδίου(randomNumber) αφού αυτό είναι ίδιο για όλες τις κινήσεις.*

Για κάθε γείτονα που ανήκει στον παίκτη το scoreGain αυξάνεται κατά ενα:

```
scoreGain += 1;
```

Για κάθε εχθρικό γείτονα έχουμε δύο περιπτώσεις:
Αν ο γείτονας έχει μικρότερο σκορ από αυτό του παιζόμενου πλακιδίου, τότε καταλαμβάνεται Έτσι το σκορ του παίκτη αυξάνεται και του αντιπάλου μειώνεται ισόποσα. Συνολικά η scoreGain αυξάνεται κατά το διπλάσιο σκορ του καταλαμβανόμενου πλακιδίου:

```
scoreGain += 2 * neighbor.getScore();
```

Αν ο γείτονας έχει μεγαλύτερο σκόρ τότε η scoreGain δε μεταβάλλεται.

Η παράμετρος coveredGain

Η coveredGain μετράει το πόσο αυξάνεται η κάλυψη των πλακιδίων του παίκτη με την επιλεγμένη κίνηση. Είναι δηλαδή ένα μέτρο του πόσο προστατεύονται τα πλακίδια του παίκτη από κατάληψη σε επόμενες κινήσεις. Θεωρούμε ότι η επικινδυνότητα εξαρτάται από το σκορ του πλακιδίου και απο τον αριθμό των γειτόνων του. Για να υπολογίσουμε πόσο εκτεθειμένο είναι ένα πλακίδιο στον αντίπαλο ορίζουμε την calcVulnerability η οποία περιγράφεται παρακάτω.

Στην coveredGain προστίθεται αρχικά η ασφάλεια της επιλεγμένης θέσεις όπου ασφάλεια = μέγιστη επικινδυνότητα - επικινδυνότητα:

```
coveredGain += randomNumber - calcVulnerability(randomNumber,
countEmptyNeighbours(board, tile));
```

Στη συνέχεια για κάθε φιλικό γείτονα προσθέτουμε στην coveredGain την διαφορά που προκύπτει στην ασφάλεια του γείτονα αυτού με την επιλεγμένη κίνηση. Αυτή οφείλεται και

στην αύξηση του σκορ του γείτονα, αλλά και στην μείωση των ελεύθερων γειτόνων του γείτονα. Κάνουμε χρήση της συνάρτησης getVulnerabilityDiff() που δημιουργήθηκε για το σκοπό αυτό:

```
coveredGain += getVulnerabilityDiff(board, neighbor, 1, 1);
```

Τέλος για κάθε εχθρικό γείτονα που κυριευτεί προσθέτουμε στην coveredGain την ασφάλεια του κυριευμένου πλακιδίου, όπως κάναμε και για το δικό μας πλακίδιο:

```
coveredGain += randomNumber - calcVulnerability(neighbor.getScore(),
countEmptyNeighbours(board, neighbor));
```

Υλοποίηση της HeuristicPlayer.getVulnerabilityDiff()

Η συνάρτηση αυτή αναλαμβάνει να υπολογίσει τις διαφορές που προκύπτουν στην ασφάλεια ενός πλακιδίου με την προσθήκη σε αυτό ενός πόντου ή και ενός γείτονα.

Υλοποίηση της HeuristicPlayer.calcVulnerability()

Η συνάρτηση αυτή αναλαμβάνει να υπολογίσει πόσο ευάλωτο είναι ενα πλακίδιο στις μελλοντικές κινήσεις του αντιπάλου. Συγκεκριμένα υπολογίζει δύο συντελεστές:

Ο συντελεστής enemyHigherCoef

Ο συντελεστής enemyHigherCoef υπολογίζεται με βάση το σκορ του πλακιδίου και μετράει την πιθανότητα να παίξει ο αντίπαλος ένα πλακίδιο με μεγαλύτερο σκορ και πιθανώς να το καταλάβει(αν το παίξει σε γειτονικό εξάγωνο).

```
double enemyHigherCoef = (20 - score) / 20;
```

Ο συντελεστής emptyNeighboursCoef

Ο συντελεστής enemyHigherCoef υπολογίζεται με βάση τους κενούς γείτονες του πλακιδίου. Αντιστοιχεί στις θέσεις στις οποίες αν παιχτεί εχθρικό πλακίδιο με μεγαλύτερο σκορ, θα καταλάβει το φιλικό πλακίδιο. Η συνάρτηση της ρίζας pow(..., 0.4) επιλέχτηκε για να δώσει μεγαλύτερη σχετικά επικινδυνότητα σε μικρό αριθμό κενών γειτόνων, αφού στην ουσία ο αντίπαλος μπορεί να καταλάβει εύκολα ένα πλακίδιο και με ένα κενό γείτονα. Για μεγάλους αριθμούς κενών γειτώνων οι επικινδυνότητα μεταβάλλεται πολύ λίγο.

```
double emptyNeighboursCoef = Math.pow(emptyNeighbours / 6, 0.4);
```

Το σκορ του πλακιδίου πολλαπλασιάζεται με τους παραπάνω συντελεστές και επιστρέφεται

Υλοποίηση της HeuristicPlayer.countEmptyNeighbours()

Η συνάρτηση countEmptyNeighbours αναλαμβάνει να μετρήσει τους γείτονες ενός πλακιδίου που είναι κενοί:

```
else if ( neighbor.getPlayerId() ==  0 ) { // empty neighbor
        emptyNeighbours++;
}
```

Πηγές και χρήσιμοι σύνδεσμοι

http://jayisgames.com/review/proximity.php

https://en.wikipedia.org/wiki/Heuristic_(computer_science)