ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

ΕΡΓΑΣΙΑ ΕΞΑΜΗΝΟΥ

Μέλη ομάδας:

Αθανασόπουλος Αντύρας Φίλιππος | 5113

Νίκος Κωνσταντινίδης | 5155

Νίκος Παπαδόπουλος | 4938

Όλες οι μετρήσεις έγιναν στο παρακάτω μηχανήματα

Όνομα	#cores	Μνήμη
Dell G15	16	16 GB
		32 GB

Οδηγίες εκτέλεσης στο τέλος της αναφοράς

Άσκηση 1

Στην 1η άσκηση δίνεται ένα σύνολο κύβων , ο καθένας με έναν αριθμό , τοποθετημένοι σε 3 γραμμές η μια πάνω από την άλλη. Σκοπός είναι η αύξουσα ταξινόμηση των κύβων κάθε σειράς , καθώς και η αύξουσα ταξινόμηση των σειρών η μια πάνω στην άλλη.

Υλοποιήσαμε δυο λύσεις , η 1η με τον αλγόριθμο αναζήτησης ομοιόμορφου κόστους **UCS** , και η 2η με τον αλγόριθμο **A*.**

Υλοποίηση κύβων , γραμμών και Cube matrix:

Η αρχιτεκτονική του παιχνιδιού μας είναι η εξής :

Cube : κάθε κύβος έχει έναν αριθμό και συντεταγμένες x,y στο Cube matrix. Αν ο κύβος έχει τον αριθμό 0 τότε θεωρείται κενή θέση.

Cube Line : κάθε Cube Line έχει μια λίστα όπου διατηρεί τους κύβους πάνω σε αυτήν.

Cube Matrix : Το Cube Matrix έχει 3 Cube Lines και διαθέτει μεθόδους για την μετακίνηση των κύβων.

Υλοποίηση κόμβων (Node):

Κάθε Node αποτελεί μια κατάσταση. Συγκεκριμένα για κάθε κατάσταση αποθηκεύουμε :

- Τον γονέα του
- Τα παιδιά του
- Το κόστος για να την μετάβαση από τον γονιό του κόμβου προς τον κόμβο.
- Το συνολικό κόστος ξεκινώντας από τον αρχικό κόμβο root
- Το την Cube matrix της συγκεκριμένης κατάστασης
- Το ευρετικό κόστος , δηλαδή μια εκτίμηση της απόστασης από την τελική κατάσταση.

Σημαντικότερες μέθοδοι :

1. printPathFromRoot() , επιστρέφει μια λίστα με τους προγονούς του Node , δηλαδή τις κινήσεις που οδηγήσαν σε αυτή τη κατάσταση και την τυπώνει.

Υλοποίηση AI (Botaki):

Το Botaki δέχεται ένα Cube Matrix και μπορεί να εφαρμόσει σε αυτό τις μεθόδους UCS και A*.

Σημαντικότερες μέθοδοι :

1. calculateHeuristicCost() : δέχεται ως όρισμα ένα CubeMatrix .Επιστρέφει μια εκτίμηση της απόστασης από την τελική κατάσταση.
2. calculateAllPossibleMoves() : δέχεται δέχεται ως όρισμα ένα CubeMatrix και ένα Cube. Επιστρέφει όλες τις δυνατές μετακινήσεις του κύβου.

   Παρατήρηση: από τις μετακινήσεις στις επιπλέον θέσεις της 1^{ης} γραμμής κρατάμε μόνο μια κίνηση .Έτσι παραλείπουμε περιττές καταστάσεις χωρίς να επηρεάζεται η λύση του προβλήματος.

3. expandNode() : δέχεται μια κατάσταση και την επεκτείνει προσθέτοντας ως παιδιά τα αποτελέσματα των πιθανών μετακινήσεων στο στιγμιότυπο αυτό.

Δεν θα αναπτυχθεί εξήγηση των κλάσεων Table και CubeManager καθώς δεν αποτελούν κομμάτι της αλγοριθμικής λύσης του προβλήματος

Η μέθοδος UCS

Η μέθοδος UCS παίρνει ως το αρχικό Cube Matrix (root) και αναζητεί την βέλτιστη λύση με τον εξής τρόπο.

1) Βγάλε από την λίστα προτεραιότητας τον κόμβο με το μικρότερο συνολικό κόστος
2) Αν το Cube Matrix βρίσκεται σε τελική κατάσταση , επέστρεψε την διαδρομή που ακολουθήθηκε για να βρεθεί αυτή η τελική κατάσταση και τερμάτισε.
3) Αλλιώς επέκτεινε τον κόμβο προσθέτοντας ως παιδιά όλες τις δυνατές κινήσεις στο συγκεκριμένο Cube Matrix.
4) Πρόσθεσε τα νέα παιδιά στην ουρά προτεραιότητας
5) Επανέλαβε το βήμα 1

```java
public void UCS(Node root){
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(node -> node.getTotalCost()));
    int numberOfNodesExpanded = 0;
    queue.add(root);
    Node result = null;
    search :{
        while(true){
            // System.out.println("Number of nodes expanded : " + numberOfNodesExpanded + "\r");
            System.out.print("Current total cost : " + queue.peek().getTotalCost()  + "\r");
            //if the node with the smallest cost is in order then we have found the solution
            if(queue.peek().getCubeMatrix().isInOrder()){
                result = queue.peek();
                break search;
            }
            //if not export the min value element from queue and expand it
            else{
                ArrayList<Node> newNodesForQueue = expandNode(queue.poll());
                queue.addAll(newNodesForQueue);
                numberOfNodesExpanded ++;
                newNodesForQueue.clear();
            }
        }
    }
    result.printPathFromRoot();
    System.out.println(ANSI_GREEN + "Total cost is :" + result.getTotalCost() + ANSI_RESET);
    System.out.println("Number of nodes expanded : " + numberOfNodesExpanded);
}
```

Η μέθοδος A*

Η μέθοδος A* ακολουθεί τα ιδιά βήματα με την UCS , με την διαφορά ότι κάθε φορά επεκτείνουμε τον κόμβο με τον μικρότερο e(n) = g(n) + h(n) , όπου g(n) το πραγματικό συνολικό κόστος και h(n) η εκτίμηση της απόστασης από την τελική κατάσταση.

Για την εκτίμηση την απόστασης από την τελική κατάσταση χρησιμοποιούμε την εξής ευρετική συνάρτηση :

h(n) = α + β όπου :

α : 2 * Αριθμός κύβων που δεν είναι σωστά στοιβαγμένοι

β : 2 * Πλήθος κύβων που μπλοκάρουν κύβους που δεν είναι στην τελική τους θέση.

Η λογική της ευρετικής είναι η εξής. Αν σε μια νέα κατάσταση υπάρχουν περισσότεροι ορθά στοιβαγμένοι κύβοι ή/και ταυτόχρονα οι κύβοι που πρέπει να στοιβαχτούν ξεμπλοκάρονται , τότε πλησιάζουμε στην τελική κατάσταση.

Επιπλέον , αν υποθέσουμε ότι μπορούμε να μειώσουμε το **α** η το **β κατά έναν κύβο** με μόνο μια κίνηση , θεωρούμε ότι αυτή η κίνηση θα έχει το χειρότερο δυνατό κόστος που στο παιχνίδι είναι 2. Για τον λόγο αυτό πολλαπλασιάζουμε την κάθε τιμή με 2.

Στην τελική κατάσταση ισχύει ότι

α = 0 , καθώς όλοι οι κύβοι είναι σωστά στοιβαγμένοι

β = 0 , καθώς όλοι οι κύβοι είναι στην τελική τους θέση

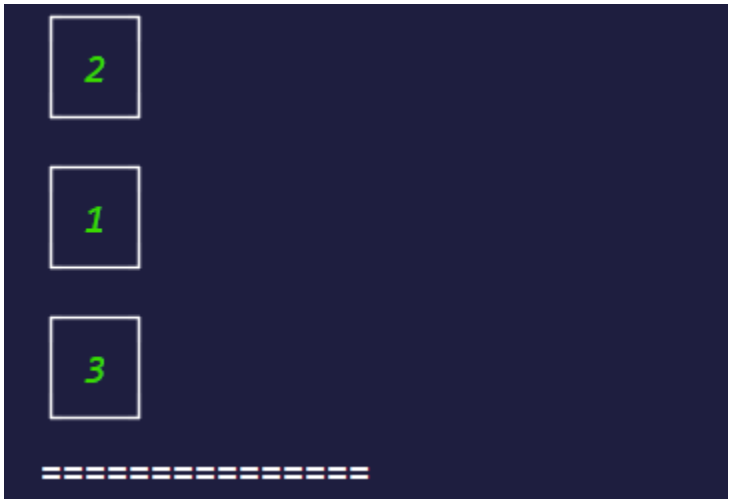
Αρά **h(n) = 0** , επομένως η ευρετική συνάρτηση είναι αποδεκτή

```
public void AStar(Node root){
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(node -> node.getTotalCost() + node.getHeuristicCost()));
    int numberOfNodesExpanded = 0;
    //ask user to press ENTER
    System.out.println(x:"Press enter to sort the cube matrix");
    try{
        System.in.read();
    }
    catch(Exception e){
        System.out.println(e);
    }
    //add root to queue
    queue.add(root);
    Node result = null;
    search :{
        while(true){
            System.out.print("Number of nodes expanded : " + numberOfNodesExpanded + "\r");
            if(queue.peek().getCubeMatrix().isInOrder()){
                result = queue.peek();
                break search;
            }
            else{
                ArrayList<Node> newNodesForQueue = expandNode(queue.poll());
                queue.addAll(newNodesForQueue);
                numberOfNodesExpanded ++;
                newNodesForQueue.clear();
            }
        }
    }
    result.printPathFromRoot();
    System.out.println(ANSI_GREEN + "Total cost is :" + result.getTotalCost() + ANSI_RESET);
    System.out.println("Number of nodes expanded : " + numberOfNodesExpanded);
}
```

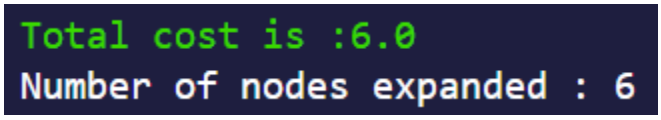
<u>Σύγκριση UCS και A*:</u>

Στο πρόγραμμα μας , αφού κατασκευάσουμε ένα Cube matrix εφαρμόζουμε πρώτα την μέθοδο A* και έπειτα την μέθοδο UCS.

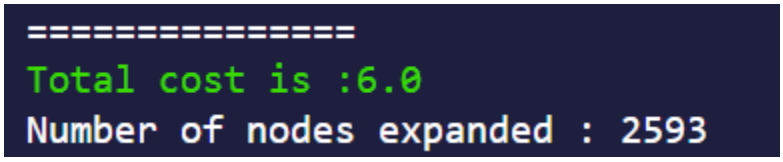
Έστω για κ = 1 η παρακάτω αρχική κατάσταση :



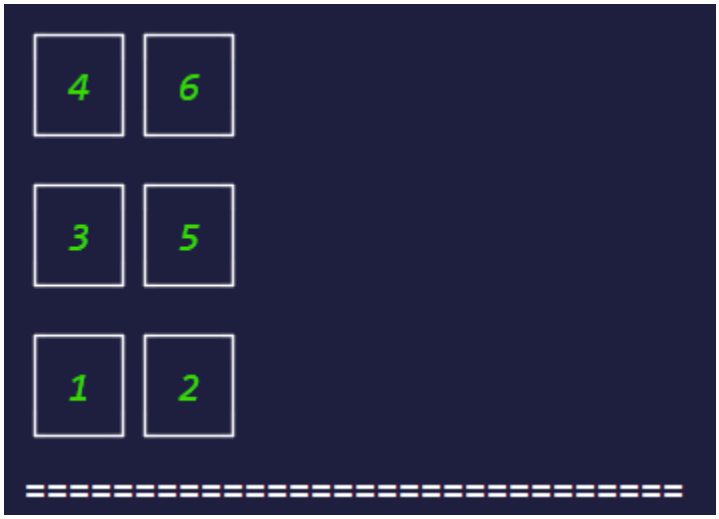
Ο αλγόριθμος **A*** λύνει το πρόβλημα με συνολικό κόστος **6.0** και **6** επεκτάσεις.



Ο αλγόριθμος UCS επιβεβαιώνει ότι το ελάχιστο κόστος είναι **6.0** , ωστόσο απαιτεί **2593** επεκτάσεις.



Δοκιμάζοντας για **κ = 2** , μια σχετικά εύκολη περίπτωση προκειμένου να λάβουμε αποτέλεσμα από την UCS σε εύλογο χρόνο, έχουμε την παρακάτω αρχική κατάσταση :



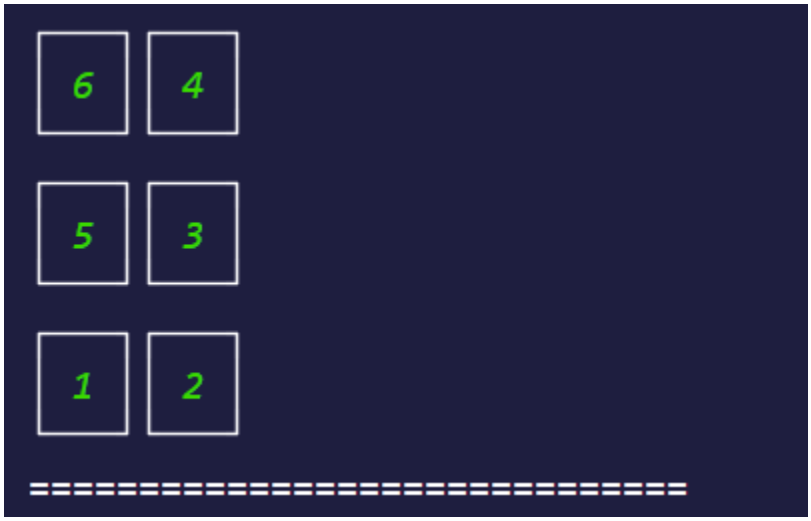
Ο αλγόριθμος **A*** λύνει το πρόβλημα με συνολικό κόστος **6.0** και **10** επεκτάσεις.

```
Total cost is :6.0
Number of nodes expanded : 10
```

Ο αλγόριθμος UCS επιβεβαιώνει ότι το ελάχιστο κόστος είναι **6.0** , ωστόσο απαιτεί **245.154** επεκτάσεις.

```
Total cost is :6.0
Number of nodes expanded : 245154
```

Αν στο ίδιο Cube matrix βάλουμε αρχικά σε λάθος θέση έναν παραπάνω κύβο (π.χ. τον 3) τότε :



A* : κόστος **6.75** με **16** επεκτάσεις

UCS :

<u>Συμπεράσματα και σχόλια:</u>

Παρατηρούμε ότι με την εύρεση μιας καλής ευρετικής συνάρτησης μπορούμε να αποφύγουμε πολλές καταστάσεις του παιχνιδιού , μειώνοντας την πολυπλοκότητα χρόνου και χώρου του προγράμματος.

Στην διάρκεια της κατασκευής των αλγορίθμων , δοκιμάστηκαν και άλλες ευρετικές συναρτήσεις όπως άθροισμα αποστάσεων

Manhattan , άθροισμα αποστάσεων Manhattan με βάρη, πλήθος κύβων που δεν είναι στην τελική τους θέση κ.α.

Η ευρετική συνάρτηση που διαλέξαμε στο τέλος αποδείχθηκε η πιο αποδοτική.

Οδηγίες εκτέλεσης :

1. Στον φάκελο με όλες τις κλάσεις εκτελέστε **javac *.java**
2. Τρέξτε το Table εκτελώντας **java Table**

Αν διαθέτετε καλό σύστημα μπορείτε να τρέξετε το πρόγραμμα για μεγάλο K (π.χ. 10) σε σύντομο χρόνο.