**N Puzzle**

με τη βοήθεια του αλγορίθμου **A\* Search**

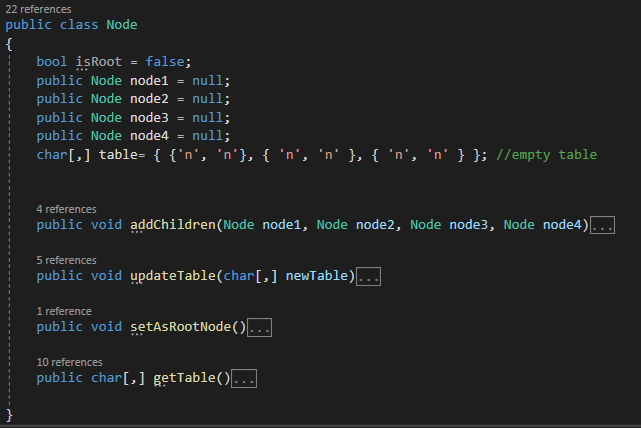
(Χρήση Manhattan Distance)

Καστανάς Αχιλλεύς

Π18066

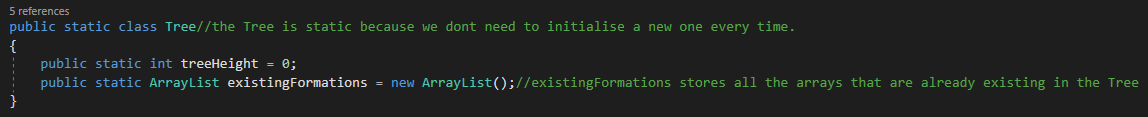
**Οι κλάσεις:**

public **Node**:



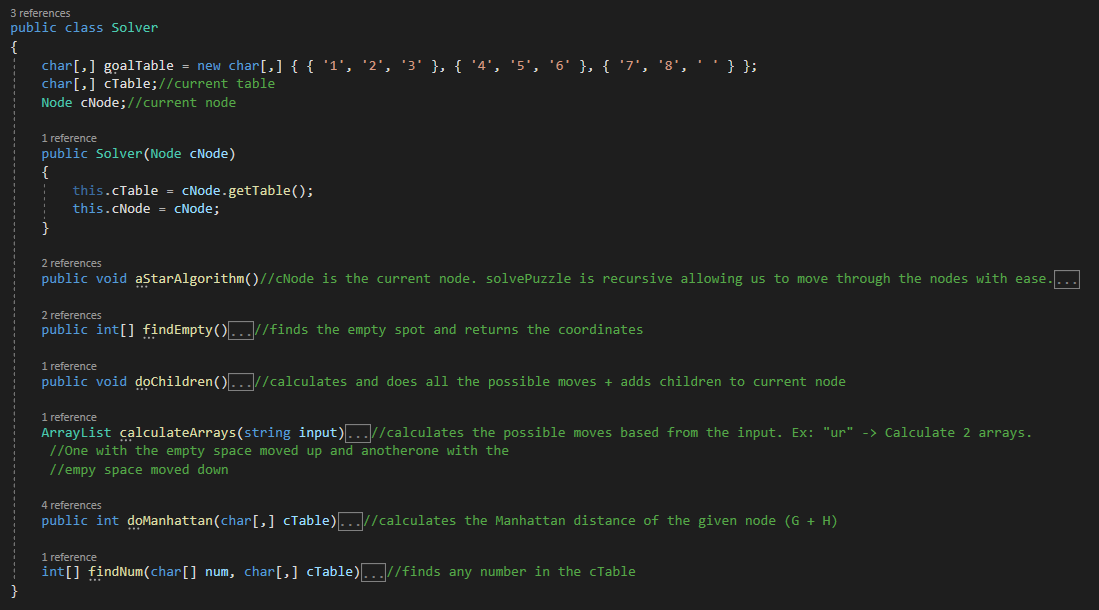
Η κλάση Node αντιπροσωπεύει τον κάθε κόμβο του δέντρου ξεχωριστά.  Ο κάθε Κόμβος έχει τέσσερα παιδιά και αποθηκεύει έναν δισδιάστατο πίνακα χαρακτήρων.  Απευθύνεται στο κάθε παιδί με τη χρήση των μεταβλητών node1-4  τύπου Node

public static **Tree**:



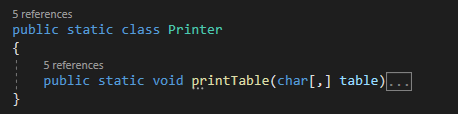
Η κλάση Tree απευθύνεται στο ένα και μοναδικό δέντρο.  Περιέχει το ύψος του δέντρου καθώς και όλους τους συνδυασμούς πινάκων του n puzzle τους οποίους έχουν ήδη συναντήσει.  Οι συνδυασμοί αυτοί αποθηκεύονται σε μία ArrayList  ονόματι existingFormations.  Η κλάση αυτή είναι static διότι δεν χρειάζονται περισσότερα από ένα instances της  και μας διευκολύνει στην πρόσβαση του ίδιου δέντρου καθόλη τη διάρκεια του προγράμματος.

public **Solver(Node cNode)**:



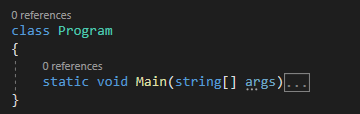
Η κλάση solver είναι η βασικότερη του προγράμματος.  Μέσα σε αυτήν γίνονται όλες οι λειτουργίες. Ο constructor της δέχεται έναν node  και τον εκχωρεί ως τρέχον.

public static **Printer**:



Η κλάση printer αναλαμβάνει την εκτύπωση του δισδιάστατου πίνακα χαρακτήρων που της δόθηκε

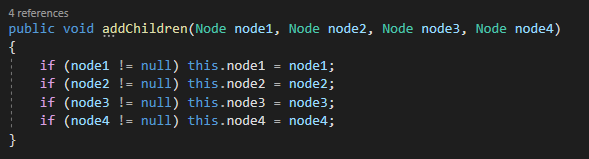
**Program**:



Η program περιέχει την main από την οποία ξεκινάει η εκτέλεση του προγράμματος

**Οι συναρτήσεις της Node:**

* public void addChildren(Node node1, Node node2, Node node3, Node node4):



Αναλαμβάνει την εισαγωγή των παιδιών στον τρέχον κόμβο.  Άμα κάποια από τις εισόδους είναι null,  τότε παραλείπεται δίνοντας μας Έτσι την ικανότητα να την καλέσουμε πολλές φορές εισάγοντας τα παιδιά ένα την φορά

* public void updateTable(char[,] newTable):

Ενημερώνει τον πίνακα του τρέχον node

* public void setAsRootNode():

Θέτει τον τρέχον κόμβο ως root node

* public char[,] getTable():

getter που επιστρέφει τον πίνακα του τρέχων κόμβου

**Οι συναρτήσεις της Solver:**

* public void aStarAlgorithm():

Η συνάρτηση είναι αναδρομική. Κατά την κλήση της αυξάνει το ύψος του δέντρου κατά ένα και εκτυπώνει τον τρέχον node με την βοήθεια της Printer.printTable().

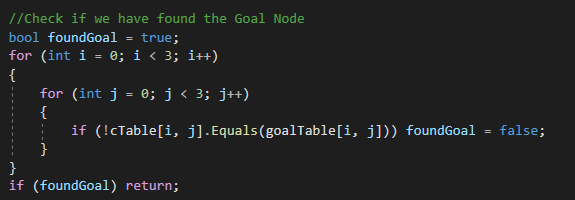
Στη συνέχεια γίνεται έλεγχος αν ο τρέχων Κόμβος αντιστοιχεί στον κόμβο που προσπαθούμε να φτάσουμε. Αν ναι, γίνεται λήξη του προγράμματος και αν όχι συνεχίζει.

Γίνεται κλήση της συναρτήσεως doChildren(), και σύμφωνα με το πόσα παιδιά μας έχει επιστρέψει (με μέγιστο αριθμό το 4) ελέγχονται το κάθε ένα διαδοχικά με τη βοήθεια της συνάρτησης doManhattan().

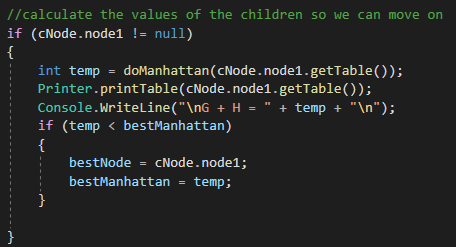
Τα παιδιά βαθμολογούνται ένα-ένα και το παιδί με το μικρότερο αποτέλεσμα θέτεται ως τρέχον κόμβος και η συνάρτηση ξανά καλείται από την αρχή.

Ακολουθούν αποσπάσματα κώδικα:

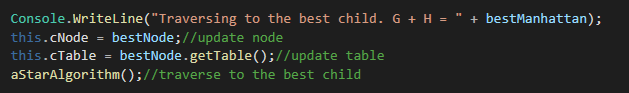
Έλεγχος goal:



Manhattan στο παιδί 1, και έλεγχος της βαθμολογίας που έλαβε σε σχέση με τα λοιπά παιδιά:



Αντικατάσταση του τρέχον κόμβου με το παιδί με τη βέλτιστη βαθμολογία:



* public int[] findEmpty():

Η findEmpty() αναζητά με την βοήθεια εμφωλευμένων loop τον χαρακτήρα ‘ ‘ που αντιπροσωπεύει το κενό.

* public void doChildren():

Η doChildren() αρχικά υπολογίζει ποιες κινήσεις επιτρέπονται να συμβούν στο παζλ έτσι ώστε να μην βγούμε ποτέ out of bounds. Στη συνέχεια καλεί την calculateArrays(results) της δίνει τα αποτελέσματα των πράξεων.

Οι πίνακες που λαμβάνει από την calculateArrays(results), αρχικά ελέγχονται σχετικά με το αν έχουνε ξαναεμφανιστεί ποτέ στο δέντρο και έπειτα υποθέτοντας ότι δεν έχουν, προστίθονται.

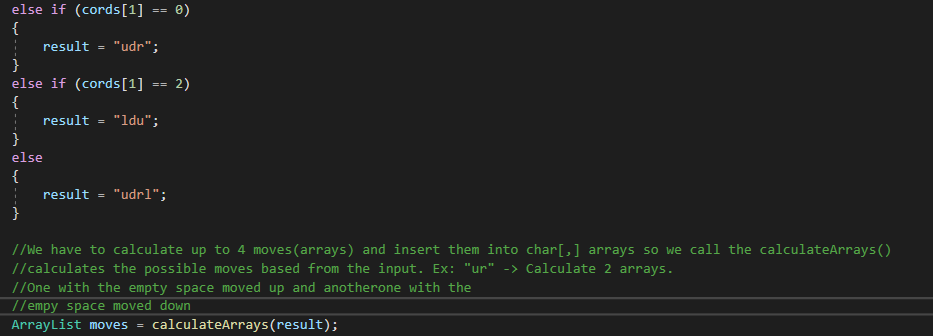
Ένας νέος κόμβος δημιουργείται ως παιδί του τρέχον κόμβου και εισάγεται ο νέος πίνακας που αντιπροσωπεύει την καινούργια κίνηση.

Σημείωση: για να ελεγχθούν αν οι κινήσεις που μόλις "σκέφτηκε" η doChildren() έχουν ήδη εμφανιστεί, αναγκαία είναι η υλοποίησή τους σε πίνακα χαρακτήρων έτσι ώστε να είναι δυνατή η σύγκριση τους με όλους τους κόμβους του δέντρου εξού και η ανάγκη ύπαρξης της calculateArrays(results).

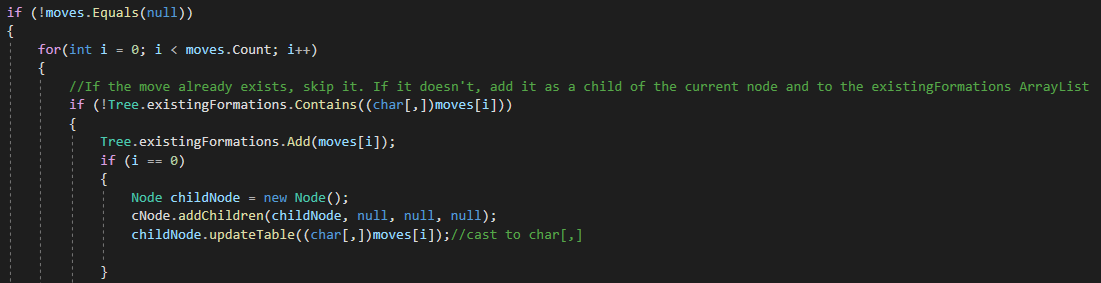
Ακολουθούν αποσπάσματα κώδικα:

Περιορισμός των περιπτώσεων out of bounds:

(Στα σχόλια εξηγείται η λειτουργεία του string result για την calculateArrays(result))



Έλεγχος του πρώτου πιθανού παιδιού. Αφού δεν εμφανίζεται στο δέντρο (if (!Tree.existingFormations.Contains((char[,])moves[i]))) προστίθεται και δημιουργείται ο ανάλογος κόμβος παιδί.

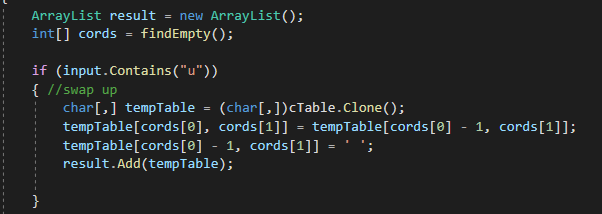


* ArrayList calculateArrays(string input)

Η calculateArrays(string input) σύμφωνα με το input δημιουργεί τους ανάλογους πίνακες.

Ακολουθούν αποσπάσματα κώδικα:

swap up με τη βοήθεια ενός tempTable

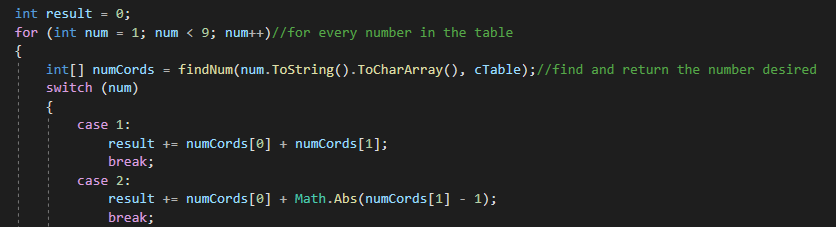


* public int doManhattan(char[,] cTable):

Η doManhattan(char[,] cTable) δέχεται έναν δισδιάστατο πίνακα χαρακτήρων και υπολογίζει την απόσταση του κάθε χαρακτήρα από την τελική του θέση ξεχωριστά.

Ακολουθούν αποσπάσματα κώδικα:

Υπολογισμός αποστάσεων των αριθμών 1 και 2 από την τελική τους θέση



* int[] findNum(char[] num, char[,] cTable):

Η findNum(char[] num, char[,] cTable) είναι μία παραλλαγή της findEmpty(), στην οποία αντί για το κενό ψάχνουμε έναν συγκεκριμένο αριθμό και επιστρέφουμε τις συντεταγμένες του.

**Οι συναρτήσεις της Printer:**

* public static void printTable(char[,] table):

Επιστρέφει τις συντεταγμένες της κενής θέσης του παζλ σε μορφή πίνακα int. Η εύρεση στις κενές θέσεις γίνεται με μία απλή εμφωλευμένη loop.

**Οι συναρτήσεις της Program:**

* static void Main(string[] args):

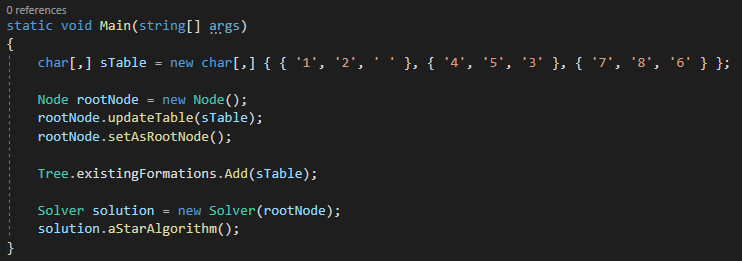
Μέσα στη Main δίνουμε έναν τυχαίο πίνακα.

Για παράδειγμα έναν πίνακα 3x3:



Στη συνέχεια δημιουργείται ένας κόμβος, εκχωρείται στο δέντρο και καλείται η Solver(Node newNode)με τον Node που μόλις δημιουργήθηκε.

Ακολουθεί η κλήση της aStarAlgorithm() που ξεκινά την επίλυση του 3x3 puzzle.



**Παραδείγματα εκτέλεσης:**

