ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Σύγκριση των αλγορίθμων αναζήτησης IDA* και A* στο 15-puzzle

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΑΝΩΛΙΑΔΗ ΓΕΩΡΓΙΟΥ (ΑΜ 55)

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΣΤΕΡΓΙΟΥ ΚΩΝΣΤΑΝΤΙΝΟΣ

ΚΟΖΑΝΗ

ΙΟΥΝΙΟΣ 2012

Περίληψη

Το αντικείμενο αυτής της Διπλωματικής Εργασίας είναι η μελέτη του IDA* και η σύγκριση του με τον αλγόριθμο A*. Οι δυο αυτοί αλγόριθμοι είναι γενικοί αλγόριθμοι αναζήτησης στην Τεχνητή Νοημοσύνη και συγκρίνονται ως προς την απόδοση τους στο γνωστό πρόβλημα αναζήτησης 15-puzzle με την βοήθεια της ευρετικής συνάρτησης Manhattan Distance. Η πειραματική διαδικασία αφορά τυχαία puzzle που λύνονται από την εφαρμογή εκτυπώνοντας στην οθόνη τα βήματα της λύσης και τον χρόνο που χρειάστηκαν. Μελετούμε τους χρόνους που κάνει ο κάθε αλγόριθμος γνωρίζοντας ότι από τη φύση τους βρίσκουν το βέλτιστο μονοπάτι από το δοθέν πρόβλημα προς την κατάσταση στόχο. Για την υλοποίηση χρησιμοποιείται η γλώσσα προγραμματισμού C++.

Λέξεις-κλειδιά

IDA*, A*, Manhattan Distance, 15-puzzle, C++, Αλγόριθμοι Αναζήτησης, Τεχνητή Νοημοσύνη

Abstract

The subject of this thesis is the study of IDA* and the comparison with the A* algorithm. These two algorithms are generic search algorithms in the field of Artificial Intelligence and are compared in terms of their performance to the known search problem 15-puzzle using the heuristic function Manhattan Distance. In the experimental procedure random puzzles are generated and solved. The application prints the solving path and the time used by the algorithms. Both algorithms are made to find the optimal path from the state given to the final state. For the implementation C++ programming language was used.

Λέξεις-κλειδιά

IDA*, A*, Manhattan Distance, 15-puzzle, C++, Pathfinding Algorithms, Artificial Intelligence

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή 7

1.1 Τεχνητή Νοημοσύνη 7

1.2 Ιστορία τεχνητής νοημοσύνης 7

1.3 Χρονικό Εξέλιξης 9

2. Η Γλώσσα προγραμματισμού C++ **12**

2.1 C++ 12

2.2 Η Φιλοσοφία της C++ 12

2.3 Τελεστές και υπερφόρτωση τελεστών 13

3. Αλγόριθμοι και Τρόπος Επίλυσης Προβλημάτων **14**

3.1 Αλγόριθμοι Αναζήτησης 14

3.2 Αλγόριθμοι Τυφλής Αναζήτησης 14

3.3 Αλγόριθμοι Πληροφορημένης αναζήτησης 15

3.4 Ευρετική συνάρτηση 18

3.5 Manhattan Distance 18

3.6 Ο αλγόριθμος A* 19

3.7 Ψευδοκώδικας του A* 20

3.8 Πολυπλοκότητα του A* 20

3.9 Ο αλγόριθμος IDA* (Iterative deepening A star) 21

3.10 Ψευδοκώδικας του IDA* 22

3.11 15-puzzle 23

3.12 Επιλυσιμότητα του 15-puzzle 25

3.13 Αναδρομή 26

4. Υλοποίηση **27**

5. Συναρτήσεις που χρησιμοποιήθηκαν 29

5.1 Η συνάρτηση Ramdom_Array() 29

5.2 Ο πίνακας target 30

5.3 Η δομή New_Array 30

5.4 Η συνάρτηση min_deq 31

5.5 Η συνάρτηση Find_Solution_Path 32

5.6 Η συνάρτηση Manhattan 33

5.7 Η συνάρτηση PrintBoard 34

5.8 Η συνάρτηση Move 35

5.9 Η συνάρτηση A_Star 37

5.10 Η συνάρτηση ida 39

6. Αποτελέσματα 41

7. Συμπεράσματα και Μελλοντικές Επεκτάσεις 44

Παράρτημα Α: Πηγαίος Κώδικας 45

Βιβλιογραφία 55

1.Εισαγωγή

1.1 Τεχνητή Νοημοσύνη

Η Τεχνητή Νοημοσύνη είναι κλάδος της Επιστήμης των Υπολογιστών που ασχολείται με τις διεργασίες σκέψης, την συλλογιστική και την συμπεριφορά. Ο όρος τεχνητή νοημοσύνη (ΤΝ, εκ του Artificial Intelligence) αναφέρεται στον κλάδο της επιστήμης υπολογιστών ο οποίος ασχολείται με τη σχεδίαση και την υλοποίηση υπολογιστικών συστημάτων που μιμούνται στοιχεία της ανθρώπινης συμπεριφοράς τα οποία υπονοούν έστω και στοιχειώδη ευφυΐα: μάθηση, προσαρμοστικότητα, εξαγωγή συμπερασμάτων, κατανόηση από συμφραζόμενα, επίλυση προβλημάτων κλπ. Ο Τζον Μακάρθι όρισε τον τομέα αυτόν ως «επιστήμη και μεθοδολογία της δημιουργίας νοούντων μηχανών». Η ΤΝ αποτελεί σημείο τομής μεταξύ πολλών πεδίων όπως της επιστήμης υπολογιστών, της ψυχολογίας, της φιλοσοφίας, της νευρολογίας, της γλωσσολογίας και της επιστήμης μηχανικών, με στόχο τη σύνθεση ευφυούς συμπεριφοράς, με στοιχεία συλλογιστικής, μάθησης και προσαρμογής στο περιβάλλον, ενώ συνήθως εφαρμόζεται σε μηχανές ή υπολογιστές ειδικής κατασκευής.

1.2 Η Ιστορία της Τεχνητής Νοημοσύνης

Κατά τη δεκαετία του 1940 εμφανίστηκε η πρώτη μαθηματική περιγραφή τεχνητού νευρωνικού δικτύου, με πολύ περιορισμένες δυνατότητες επίλυσης αριθμητικών προβλημάτων. Καθώς ήταν εμφανές ότι οι ηλεκτρονικές υπολογιστικές συσκευές που κατασκευάστηκαν μετά τον Β' Παγκόσμιο Πόλεμο ήταν ένα τελείως διαφορετικό είδος μηχανής από ό,τι προηγήθηκε, η συζήτηση για την πιθανότητα εμφάνισης μηχανών με νόηση ήταν στην ακμή της. Το 1950 ο μαθηματικός Άλαν Τούρινγκ, πατέρας της θεωρίας υπολογισμού και προπάτορας της τεχνητής νοημοσύνης, πρότεινε τη δοκιμή Τούρινγκ, μία απλή δοκιμασία που θα μπορούσε να εξακριβώσει

αν μία μηχανή διαθέτει ευφυΐα. Η τεχνητή νοημοσύνη θεμελιώθηκε τυπικά ως πεδίο στη συνάντηση ορισμένων επιφανών Αμερικανών επιστημόνων του τομέα το 1956 (Τζον Μακάρθι, Μάρβιν Μίνσκυ, Κλοντ Σάνον κλπ). Τη χρονιά αυτή παρουσιάστηκε για πρώτη φορά και το Logic Theorist, ένα πρόγραμμα το οποίο στηριζόταν σε συμπερασματικούς κανόνες τυπικής λογικής και σε ευρετικούς αλγορίθμους αναζήτησης για να αποδεικνύει μαθηματικά θεωρήματα. Επόμενοι σημαντικοί σταθμοί ήταν η ανάπτυξη της γλώσσας προγραμματισμού LISP το 1958 από τον Μακάρθι, δηλαδή της πρώτης γλώσσας συναρτησιακού προγραμματισμού η οποία έπαιξε πολύ σημαντικό ρόλο στη δημιουργία εφαρμογών ΤΝ κατά τις επόμενες δεκαετίες, η εμφάνιση των γενετικών αλγορίθμων την ίδια χρονιά από τον Φρίντμπεργκ και η παρουσίαση του βελτιωμένου νευρωνικού δικτύου perceptron το '62 από τον Ρόσενμπλατ. Κατά τα τέλη της δεκαετίας του '60 όμως άρχισε ο χειμώνας της ΤΝ, μία εποχή κριτικής, απογοήτευσης και υποχρηματοδότησης των ερευνητικών προγραμμάτων καθώς όλα τα μέχρι τότε εργαλεία του χώρου ήταν κατάλληλα μόνο για την επίλυση εξαιρετικά απλών προβλημάτων. Στα μέσα του '70 ωστόσο προέκυψε μία αναθέρμανση του ενδιαφέροντος για τον τομέα λόγω των εμπορικών εφαρμογών που απέκτησαν τα έμπειρα συστήματα, μηχανές ΤΝ με αποθηκευμένη γνώση για έναν εξειδικευμένο τομέα και δυνατότητα ταχείας εξαγωγής λογικών συμπερασμάτων, τα οποία συμπεριφέρονται όπως ένας άνθρωπος ειδικός στον αντίστοιχο τομέα. Παράλληλα έκανε την εμφάνισή της η γλώσσα λογικού προγραμματισμού Prolog η οποία έδωσε νέα ώθηση στη συμβολική ΤΝ, ενώ στις αρχές της δεκαετίας του '80 άρχισαν να υλοποιούνται πολύ πιο ισχυρά και με περισσότερες εφαρμογές νευρωνικά δίκτυα, όπως τα πολυεπίπεδα perceptron και τα δίκτυα Hopfield. Ταυτόχρονα οι γενετικοί αλγόριθμοι και άλλες συναφείς μεθοδολογίες αναπτύσσονταν πλέον από κοινού, κάτω από την ομπρέλα του εξελικτικού υπολογισμού. Κατά τη δεκαετία του '90, με την αυξανόμενη σημασία του Internet, ανάπτυξη γνώρισαν οι ευφυείς πράκτορες, αυτόνομο λογισμικό ΤΝ τοποθετημένο σε κάποιο περιβάλλον με το οποίο αλληλεπιδρά, οι οποίοι βρήκαν μεγάλο πεδίο εφαρμογών λόγω της εξάπλωσης του Διαδικτύου. Οι πράκτορες στοχεύουν συνήθως στην παροχή βοήθειας στους χρήστες

τους, στη συλλογή ή ανάλυση γιγάντιων συνόλων δεδομένων ή στην αυτοματοποίηση επαναλαμβανόμενων εργασιών (π.χ. βλέπε διαδικτυακό ρομπότ), ενώ στους τρόπους κατασκευής και λειτουργίας τους συνοψίζουν όλες τις γνωστές μεθοδολογίες ΤΝ που αναπτύχθηκαν με το πέρασμα του χρόνου. Έτσι σήμερα, όχι σπάνια, η ΤΝ ορίζεται ως η επιστήμη που μελετά τη σχεδίαση και υλοποίηση ευφυών πρακτόρων. Επίσης τη δεκαετία του '90 η ΤΝ, κυρίως η μηχανική μάθηση και η ανακάλυψη γνώσης, άρχισε να επηρεάζεται πολύ από τη θεωρία πιθανοτήτων και τη στατιστική. Τα μπεϋζιανά δίκτυα είναι η εστίαση αυτής της νέας μετακίνησης που παρέχει τις συνδέσεις με τα πιο σχολαστικά θέματα της στατιστικής και της επιστήμης μηχανικών, όπως τα πρότυπα Markov και τα φίλτρα Kalman. Αυτή η νέα πιθανοκρατική προσέγγιση έχει αυστηρά υποσυμβολικό χαρακτήρα, όπως και οι τρεις μεθοδολογίες οι οποίες κατηγοριοποιούνται κάτω από την ετικέτα της υπολογιστικής νοημοσύνης: τα νευρωνικά δίκτυα, ο εξελικτικός υπολογισμός και η ασαφής λογική.

1.3 Χρονικό Εξέλιξης

1950 Ο Άλαν Τούρινγκ περιγράφει τη δοκιμή Τούρινγκ, που επιδιώκει να εξετάσει την ικανότητα μιας μηχανής να συμμετάσχει απρόσκοπτα σε μια ανθρώπινη συνομιλία.

1951 Τα πρώτα προγράμματα ΤΝ γράφονται για τον υπολογιστή Ferranti Mark I στο Πανεπιστήμιο του Μάντσεστερ: ένα πρόγραμμα που παίζει ντάμα από τον Κρίστοφερ Στράκλι και ένα που παίζει σκάκι από τον Ντίτριχ Πρίνζ.

1956 Ο Τζον Μακάρθι πλάθει τον όρο «Τεχνητή Νοημοσύνη» ως κύριο θέμα της διάσκεψης του Ντάρτμουθ.

1958 Ο Τζον Μακάρθι εφευρίσκει τη γλώσσα προγραμματισμού Lisp.

1965 Ο Έντουαρτ Φάιγκενμπαουμ ξεκινά το Dendral, μια δεκαετή προσπάθεια ανάπτυξης λογισμικού που θα συμπεράνει τη μοριακή δομή οργανικών ενώσεων χρησιμοποιώντας ενδείξεις επιστημονικών οργάνων. Ήταν το πρώτο έμπειρο σύστημα (expert system).

1966 Ιδρύεται το Εργαστήριο Μηχανικής Νοημοσύνης στο Εδιμβούργο – το πρώτο από μια σημαντική σειρά εγκαταστάσεων που οργανώνονται από τον Ντόναλντ Μίτσι και άλλους.

1970 Αναπτύσσεται το Planner και χρησιμοποιείται στο SHRDLU, μια εντυπωσιακή επίδειξη αλληλεπίδρασης μεταξύ ανθρώπου και υπολογιστή.

1971 Ξεκινά η εργασία πάνω στο σύστημα αυτόματης απόδειξης θεωρημάτων BoyerMoore στο Εδιμβούργο.

1972 Η γλώσσα προγραμματισμού Prolog αναπτύσσεται από τον Αλάν Κολμεροέρ.

1973 Ρομπότ συναρμολόγησης «Φρέντι» στο Εδιμβούργο: ένα ευπροσάρμοστο σύστημα συναρμολόγησης που ελέγχεται από υπολογιστές.

1974 Ο Τέντ Σόρτλιφ γράφει τη διατριβή του για το πρόγραμμα MYCIN (Στάνφορντ), το οποίο κατέδειξε μια πολύ πρακτική προσέγγιση στην ιατρική διάγνωση που βασίζεται σε κανόνες, ενώ λειτουργεί ακόμα και με παρουσία αβεβαιότητας. Αν και δανείστηκε από το DENDRAL, οι δικές του συνεισφορές επηρέασαν έντονα το μέλλον των έμπειρων συστημάτων, ένα μέλλον με πολλαπλές εμπορικές εφαρμογές.

1991 Η εφαρμογή σχεδίασης ενεργειών DART χρησιμοποιείται αποτελεσματικά στον Α' Πόλεμο του Κόλπου και ανταμείβει 30 χρόνια έρευνας στην ΤΝ του Αμερικανικού Στρατού.

1994 Ντίκμαννς και Ντάιμλερ-Μπενζ οδηγούν περισσότερο από 1000 km σε μια εθνική οδό του Παρισιού υπό συνθήκες βαρείας κυκλοφορίας και σε ταχύτητες ως και 130 km/ώρα. Επιδεικνύουν αυτόνομη οδήγηση σε ελεύθερες παρόδους, οδήγηση σε συνοδεία, αλλαγή παρόδων και αυτόματη προσπέραση άλλων οχημάτων.

1997 Ο υπολογιστής Deep Blue της IBM κερδίζει των παγκόσμιο πρωταθλητή σκακιού Γκάρι Κασπάροφ.

1998 Κυκλοφορεί ο Φέρμπι της Tiger Electronics και γίνεται η πρώτη επιτυχημένη εμφάνιση ΤΝ σε οικιακό περιβάλλον.

1999 Η Sony λανσάρει το AIBO, που είναι ένα από τα πρώτα αυτόνομα κατοικίδια.

2004 Η DARPA ξεκινά το πρόγραμμα DARPA Grand Challenge («Μεγάλη Πρόκληση DARPA»), που προκαλεί τους συμμετέχοντες να δημιουργήσουν αυτόνομα οχήματα για ένα χρηματικό βραβείο.

2.Η Γλώσσα Προγραμματισμού C++

2.1 C++

Η C++ (C Plus Plus) είναι μια γενικού σκοπού γλώσσα προγραμματισμού Η/Υ. Θεωρείται μέσου επιπέδου γλώσσα, καθώς περιλαμβάνει έναν συνδυασμό χαρακτηριστικών από γλώσσες υψηλού και χαμηλού επιπέδου. Είναι μια πολλαπλών παραδειγμάτων, μεταφράσιμη γλώσσα όπου η μετάφρασή της (compilation) δημιουργεί κώδικα μηχανής για ένα συγκεκριμένο τύπο υλικού. Υποστηρίζει δομημένο, αντικειμενοστραφή και γενικό προγραμματισμό. Η γλώσσα αναπτύχθηκε από τον Bjarne Stroustrup το 1979 στα εργαστήρια Bell της AT&T, ως βελτίωση της ήδη υπάρχουσας γλώσσας προγραμματισμού C, και αρχικά ονομάστηκε "C with Classes", δηλαδή C με Κλάσεις. Μετονομάστηκε σε C++ το 1983. Οι βελτιώσεις ξεκίνησαν με την προσθήκη κλάσεων, και ακολούθησαν, μεταξύ άλλων, εικονικές συναρτήσεις, υπερφόρτωση τελεστών, πολλαπλή κληρονομικότητα, πρότυπα κ.α. Η γλώσσα ορίστηκε παγκοσμίως, το 1998, με το πρότυπο ISO/IEC 14882:1998. Η πιο γνωστή και ευρέως διαδεδομένη έκδοση αυτού του προτύπου είναι αυτή του 2003, η ISO/IEC 14882:2003. Η τελευταία εκδοση του προτύπου είναι η *ISO/IEC 14882:2011, γνωστή και ως* C++11, πολλά στοιχεία του οποίου χρησιμοποιήθηκαν στην συγκεκριμένη εργασία.

2.2 Φιλοσοφία C++

Στο βιβλίο του "The Design and Evolution of C++ (1994), ο Bjarne Stroustrup περιγράφει κάποιους κανόνες που χρησιμοποιεί για το σχεδιασμό της C++:

• η C++ είναι σχεδιασμένη ως μια γενικής χρήσης γλώσσα με στατικούς τύπους που είναι όσο αποτελεσματική και φορητή, όσο η C.

• η C++ είναι σχεδιασμένη να υποστηρίζει άμεσα και σφαιρικά πολλά είδη

προγραμματισμού (δομημένος προγραμματισμός, αντικειμενοστραφής προγραμματισμός, γενικός προγραμματισμός).

• η C++ είναι σχεδιασμένη να δίνει επιλογές στον προγραμματιστή, ακόμα κι αν του επιτρέπει να επιλέξει λανθασμένα.

• η C++ είναι σχεδιασμένη να είναι όσο το δυνατόν συμβατή με τη C, ώστε να διευκολύνει τη μετάβαση από τη C.

• η C++ αποφεύγει χαρακτηριστικά που αναφέρονται σε συγκεκριμένες πλατφόρμες ή δεν είναι γενικής χρήσης.

• η C++ δεν έχει κόστος για χαρακτηριστικά της γλώσσας που δεν χρησιμοποιούνται.

• η C++ είναι σχεδιασμένη να λειτουργεί χωρίς κάποιο εξελιγμένο προγραμματιστικό περιβάλλον.

Το βιβλίο Inside the C++ Object Model (Lippman, 1996) περιγράφει πως οι μεταγλωττιστές μπορούν να μετατρέψουν εντολές ενός προγράμματος C++ σε μια διάταξη στη μνήμη. Παρ' όλα αυτά, οι συγγραφείς μεταγλωττιστών είναι γενικά ελεύθεροι να υλοποιήσουν το πρότυπο με δικό τους τρόπο.

2.3 Τελεστές και υπερφόρτωση τελεστών

Η C++ παρέχει περισσότερους από 30 τελεστές, που καλύπτουν τη βασική αριθμητική, το χειρισμό bit, αναφορά δεικτών, συγκρίσεις, λογικές πράξεις κ.α. Σχεδόν όλοι οι τελεστές μπορούν να υπερφορτωθούν για τύπους ορισμένους από το χρήστη, με λίγες εξαιρέσεις όπως πρόσβαση μέλους (. και .*). Το πλούσιο σύνολο από υπερφορτώσιμους τελεστές είναι βασικό για τη χρήση της C++ ως γλώσσα ειδικού πεδίου (domain specific language).

Οι υπερφορτώσιμοι τελεστές είναι ακόμα βασικό μέρος πολλών προχωρημένων τεχνικών προγραμματισμού της C++, όπως οι έξυπνοι δείκτες. Η υπερφόρτωση ενός τελεστή δεν αλλάζει την προτεραιότητα των υπολογισμών όπου χρησιμοποιείται, ούτε τον αριθμό των τελεστέων που χρησιμοποιεί ο τελεστής (αν και οποιοσδήποτε τελεστέος μπορεί απλά να αγνοείται).

3. Αλγόριθμοι και τρόπος επίλυσης προβλημάτων

3.1 Αλγόριθμοι Αναζήτησης

Οι αλγόριθμοι αναζήτησης αφορούν τον σχεδιασμό κατάλληλων ενεργειών με στόχο την άφιξη ενός ελέγξιμου συστήματος σε μία αποδεκτή τελική κατάσταση, εκκινώντας από κάποια προκαθορισμένη αρχική κατάσταση. Οι αλγόριθμοι λαμβάνουν ως είσοδο το δοθέν πρόβλημα και επιστρέφουν ως έξοδο μία λύση σε αυτό, αφού αξιολογήσουν πρώτα μία ομάδα υποψηφίων λύσεων. Πρόκειται για ένα θεμελιώδες γνωστικό πεδίο της Τεχνητής Νοημοσύνης το οποίο γνώρισε μεγάλη ανάπτυξη από τη δεκαετία του 1950. Αποτέλεσε μία προσπάθεια αλγοριθμικής εξομοίωσης της διαδικασίας της σκέψης, ενώ με τον καιρό ενσωμάτωσε μεθοδολογίες από τη θεωρία βελτιστοποίησης και τη θεωρία γράφων.

3.2 Αλγόριθμοι Τυφλής Αναζήτησης

Οι αλγόριθμοι τυφλής αναζήτησης (blind search algorithms) εφαρμόζονται σε προβλήματα στα οποία δεν υπάρχει πληροφορία που να επιτρέπει την αξιολόγηση των καταστάσεων του χώρου αναζήτησης. Έτσι οι αλγόριθμοι αυτοί αντιμετωπίζουν με τον ίδιο ακριβώς τρόπο οποιοδήποτε πρόβλημα καλούνται να λύσουν. Για τους αλγόριθμους τυφλής αναζήτησης, το τι απεικονίζει κάθε κατάσταση του προβλήματος είναι παντελώς αδιάφορο. Σημασία έχει η χρονική σειρά με την οποία παράγονται οι καταστάσεις από το μηχανισμό επέκτασης.

3.3 Αλγόριθμοι Πληροφορημένης αναζήτησης

Οι αλγόριθμοι τυφλής αναζήτησης προχωρούν σε πλάτος ή βάθος χωρίς να έχουν καμία απολύτως πληροφορία για το αν το μονοπάτι που ακολουθούν οδηγεί σε τερματική κατάσταση. Συνεπώς, η τυφλή αναζήτηση δεν επαρκεί για μεγάλους χώρους καταστάσεων που συνήθως εμφανίζονται σε πραγματικά προβλήματα. Ο ρυθμός με τον οποίο αναπτύσσεται ένας χώρος αναζήτησης είναι ταχύτατος με αποτέλεσμα να παρατηρείται το φαινόμενο της συνδυαστικής έκρηξης. Σε τέτοιους χώρους, η τυφλή αναζήτηση διαρκεί τόσο πολύ που πρακτικά η λύση δε βρίσκεται ποτέ. Σε τέτοιες καταστάσεις χρησιμοποιούνται αλγόριθμοι πληροφορημένης αναζήτησης. Σκοπός λοιπόν είναι να μειωθεί ο χρόνος αναζήτησης, δηλαδή ουσιαστικά να μειωθεί ο αριθμός των καταστάσεων που εξετάζει ένας αλγόριθμο. Για να επιτευχθεί κάτι τέτοιο είναι απαραίτητη η ύπαρξη κάποιας πληροφορίας για την αξιολόγηση των καταστάσεων η οποία θα είναι ικανή να καθοδηγήσει την αναζήτηση σε καταστάσεις που οδηγούν σε μια λύση και ίσως να βοηθήσει στο κλάδεμα ορισμένων καταστάσεων που δεν οδηγούν πουθενά. Προκειμένου να μειωθεί ο γιγάντιος για ρεαλιστικά προβλήματα, χώρος αναζήτησης και ο απαιτούμενος για την εύρεση της λύσης χρόνος, μπορούν να χρησιμοποιηθούν αλγόριθμοι που εκμεταλλεύονται ευρετικούς μηχανισμούς, δηλαδή στρατηγικές (συνήθως συναρτήσεις που εξαρτώνται από το εκάστοτε πρόβλημα) οι οποίες αξιολογούν προσεγγιστικά τις ενδιάμεσες καταστάσεις ως προς την εκτιμώμενη απόσταση τους από μία τελική κατάσταση, επεκτείνουν πρώτα αυτές με τη βέλτιστη ευρετική τιμή (οι οποίες αναμένεται να οδηγήσουν συντομότερα σε λύση) ή/και κλαδεύουν τις υπόλοιπες. Οι ευρετικοί μηχανισμοί δεν είναι αντικειμενικοί και, παρόλο που κωδικοποιούνται αλγοριθμικά υπό τη μορφή της ευρετικής συνάρτησης, δεν μπορούν να θεωρηθούν αλγόριθμοι. Αυτό γιατί, προκειμένου να μειώσουν το χώρο αναζήτησης ή να επιταχύνουν την εύρεση της λύσης, λειτουργούν προσεγγιστικά και «διαισθητικά» (περίπου όπως οι άνθρωποι), ενώ οι αλγόριθμοι είναι ακριβείς και λειτουργούν πάντα ορθά. Στην πλειονότητα των περιπτώσεων

πάντως οι ευρετικές στρατηγικές οδηγούν σε πολύ καλά αποτελέσματα (αναλόγως βέβαια του προβλήματος), ωστόσο απέχουν πολύ από το να προσομοιώνουν τους μηχανισμούς της ανθρώπινης σκέψης: η τελευταία χρησιμοποιεί επίσης ευρετικές μεθόδους οι οποίες όμως είναι ποιοτικές, όχι ποσοτικές / αριθμητικές όπως η ευρετική συνάρτηση, και φαίνεται να αποδίδουν καλύτερα.

Ένας βασικός ευρετικός αλγόριθμος είναι ο HC (Hill Climbing ή αναρρίχηση λόφων), ο οποίος μοιάζει με τον DFS αλλά σε κάθε επανάληψη κλαδεύει όλες τις καταστάσεις που προκύπτουν από μία επέκταση εκτός από την ευρετικά βέλτιστη (δηλαδή κάθε στιγμή το Μ.Α. έχει μία κατάσταση) και μεταβαίνει στην τελευταία μόνο αν έχει καλύτερη ευρετική τιμή από το γονέα της· διαφορετικά τερματίζει έχοντας βρει μία τοπικά βέλτιστη λύση. Προφανώς ο HC δεν είναι πλήρης αλλά είναι πολύ γρήγορος και καθόλου μνημοβόρος. Υπάρχουν διάφορες παραλλαγές του που θυσιάζουν λίγη από την ταχύτητα του προκειμένου να αυξήσουν την πιθανότητα του να βρει λύση. Μία παραλλαγή είναι ο EHC (Enforced Hill Climbing ή εξαναγκασμένη αναρρίχηση λόφων), στον οποίον διατηρούνται στο Μ.Α. τα αδέρφια του τρέχοντος κόμβου και, αν η επέκταση του τελευταίου δεν οδηγήσει σε μετάβαση, αντί ο αλγόριθμος να τερματίσει εκτελεί μία αναζήτηση κατά πλάτος στα αδέρφια του μέχρι να βρεθεί μία καλύτερη κατάσταση οπότε και συνεχίζεται η αναρρίχηση από εκεί. Επίσης δημοφιλής είναι και ο SA (Simulated Annealing ή προσομοιωμένη ανόπτηση), ο οποίος δίνει μία πιθανότητα μετάβασης σε χειρότερες καταστάσεις (p), αφήνοντας έτσι περιθώριο στην αναζήτηση να ξεφύγει από τοπικά βέλτιστα. Αν η πιθανότητα p τείνει στο 0 ο SA λειτουργεί όπως ο HC. Επίσης υπάρχει ο TS (Taboo Search ή αναζήτηση με απαγορεύσεις), όπου σε κάθε επέκταση γίνεται πάντα μετάβαση στο καλύτερο παιδί, ακόμα και αν είναι χειρότερη κατάσταση από την τρέχουσα, και η αναζήτηση συμβουλεύεται μία λίστα απαγορευμένων καταστάσεων (παρόμοιας λειτουργικότητας με το Κλειστό Σύνολο αλλά σταθερού μεγέθους). Ο BS (Beam Search ή ακτινωτή αναζήτηση), όπου ένας σταθερός αριθμός εκ των καλύτερων καταστάσεων παραμένει στο Μ.Α. δίνοντας τη δυνατότητα

οπισθοδρόμησης αν χρειαστεί, είναι μία ακόμα επέκταση του κεντρικού αλγορίθμου αναρρίχησης λόφων.

Όλοι αυτοί οι ευρετικοί αλγόριθμοι κατάγονται από τη θεωρία μαθηματικής βελτιστοποίησης, όπου αναπτύχθηκαν για να εντοπίζουν το ελάχιστο ή το μέγιστο μίας πραγματικής συνάρτησης διακριτής μεταβλητής. Στην επίλυση προβλημάτων τον ρόλο της τελευταίας προφανώς τον παίζει η ευρετική συνάρτηση και ο χώρος των λύσεων οπτικοποιείται ως ένα γεωγραφικό «τοπίο»: όσο περισσότερο δύο λύσεις διαφέρουν τόσο απέχουν μεταξύ τους σε αυτό το τοπίο, ενώ όσο καλύτερη ευρετική τιμή έχει μία λύση τόσο υψηλότερα από το επίπεδο του «εδάφους» τοποθετείται σε αυτό το τοπίο. Το τελευταίο, καθώς οι υποψήφιες καταστάσεις είναι διακριτές μεταξύ τους, ουσιαστικά είναι ένας γράφος με κορυφές τις καταστάσεις και ακμές τους τελεστές μετάβασης. Η παραλλαγή της αναρρίχησης λόφων σε συνεχή χώρο, με στόχο την εύρεση ακρότατου μιας συνάρτησης συνεχούς μεταβλητής, ονομάζεται άνοδος κλίσης (gradient ascent, αν η συνάρτηση εκφράζει βελτιστότητα και αναζητείται το μέγιστό της) ή κάθοδος κλίσης (gradient descent, αν η συνάρτηση εκφράζει σφάλμα / απόκλιση από το βέλτιστο και αναζητείται το ελάχιστό της) και υλοποιείται με μεθόδους του απειροστικού λογισμού.

Άλλος δημοφιλής ευρετικός αλγόριθμος είναι ο BestFS (αναζήτηση πρώτα στο καλύτερο) ο οποίος κρατά όλες τις καταστάσεις στο Μ.Α. και μοιάζει με τον BFS, μόνο που σε κάθε επέκταση εφαρμόζει τον ευρετικό μηχανισμό και στην επόμενη επανάληψη μεταβαίνει στο ευρετικά βέλτιστο παιδί. Είναι πλήρης, μνημοβόρος και δεν εγγυάται ότι θα βρει τη βέλτιστη λύση αφού εξαρτάται απόλυτα από την εγκυρότητα των εκτιμήσεων της ευρετικής συνάρτησης. Τροποποίηση του BestFS αποτελεί ο πλήρης και βέλτιστος αλγόριθμος A*, στον οποίον η ευρετική τιμή που αντιστοιχίζεται σε κάθε νέα κατάσταση Κ για να την αξιολογήσει ο μηχανισμός δεν είναι μόνο μία εκτίμηση Α της απόστασης της από μία τελική κατάσταση, αλλά το άθροισμα Α συν την ακριβή απόσταση της Κ από τη ρίζα. Ο A* εγγυάται ότι θα βρει τη βέλτιστη λύση αρκεί η ευρετική συνάρτηση να είναι πάντα υποεκτίμηση της πραγματικής απόστασης από τη λύση και ποτέ υπερεκτίμηση («αποδεκτή

συνάρτηση»). Σε περίπτωση που είναι σπουδαιότερη η ταχύτητα παρά η βελτιστότητα δε χρειάζεται η ευρετική συνάρτηση να είναι αποδεκτή.

3.4 Ευρετική συνάρτηση

Ευρετική συνάρτηση ή ευρετικός μηχανισμός λέγεται μια στρατηγική βασισμένη στη γνώση για το συγκεκριμένο πρόβλημα, η οποία χρησιμοποιείται σαν βοήθημα στη γρήγορη επίλυση του. Ο ευρετικός μηχανισμός υλοποιείται με ευρετική συνάρτηση, που έχει πεδίο ορισμού το σύνολο των καταστάσεων ενός προβλήματος και πεδίο τιμών το σύνολο τιμών που αντιστοιχεί σε αυτές. Ευρετική τιμή είναι η τιμή της ευρετικής συνάρτησης και εκφράζει το πόσο κοντά βρίσκεται μία κατάσταση σε μία τελική. Η ευρετική τιμή δεν είναι η πραγματική τιμή της απόστασης από μία τερματική κατάσταση, αλλά μία εκτίμηση που πολλές φορές μπορεί να είναι και λανθασμένη Η ευρετική συνάρτηση εξαρτάται από την φύση του προβλήματος. Για συγκεκριμένα προβλήματα χρησιμοποιούνται ευρετικές συναρτήσεις προσαρμοζόμενες στις ανάγκες της αναζήτησης.

3.5 Manhattan Distance

Το Manhattan Distance είναι μια ευρετική συνάρτηση για την λύση περίπλοκων προβλημάτων. Θεωρείται ότι ανακαλύφθηκε από τον Hermann Minkowski τον 19° αιώνα. Η απόσταση μεταξύ 2 σημείων υπολογίζεται ως η απόλυτη διαφορά των συντεταγμένων των σημείων. Ονομάζεται και taxicad geometry, απόσταση οικοδομικών τετραγώνων (city bock distance). Η αναφορά στο Manhattan γίνεται για την δομή πλέγματος των περισσότερων οδών του νησιού Manhattan. Στο 15-puzzle η απόσταση Manhattan είναι το άθροισμα των αποστάσεων των πλακιδίων από τις θέσεις προορισμού τους. Επειδή τα πλακίδια δεν μπορούν να μετακινούνται διαγώνια,

η απόσταση που θα μετράμε είναι το άθροισμα των οριζόντιων και κάθετων αποστάσεων. Αυτή η ευρετική συνάρτηση είναι παραδεκτή, δηλαδή δεν υπερεκτιμά τις πραγματικές αποστάσεις. Στο παρακάτω παράδειγμα ο πίνακας έχει Manhattan Distance 19. Ο μηχανισμός δεν υπερεκτιμά το πραγματικό κόστος λύσης που είναι 23.

1	**3**	**6**	**4**
10	**9**	**8**	**12**
5	**14**	**2**	**0**
13	**15**	**11**	**7**

3.6 Ο αλγόριθμος Α*

Ο Α* είναι από τους πιο γνωστούς και ευρέως χρησιμοποιημένους αλγόριθμους της Τεχνητής Νοημοσύνης που χρησιμοποιείται για την αναζήτηση μονοπατιού και για την διάσχιση γράφου. Ανήκει στην κατηγορία των αλγορίθμων πληροφορημένης ή ευρετικής αναζήτησης. Οι αλγόριθμοι αυτοί χρησιμοποιούν ευρετική συνάρτηση με τη βοήθεια της οποίας αξιολογούν τις επόμενες πιθανές καταστάσεις και επιλέγουν την καλύτερη από αυτές. Σκοπός του αλγορίθμου είναι η εύρεση της βέλτιστης λύσης δηλαδή το συντομότερο μονοπάτι με το λιγότερο κόστος μεταξύ 2 σημείων που ονομάζονται κόμβοι. Ο αλγόριθμος βρίσκει ευρεία χρήση λόγω της ακρίβειας και των επιδόσεων του. Αρχικά παρουσιάστηκε το 1968 από τον Peter Hart, τον Nils Nilsson και τον Bertram Rapheal του Ερευνητικού Ινστιτούτου Stanford. Είναι αποτέλεσμα συνδυασμού της άπληστης αναζήτησης με την αναζήτηση με βάση το κόστος. Πραγματοποιεί μια ταξινόμηση των καταστάσεων(κόμβων) με βάση την συνάρτηση f(k)= g(k) + h(k). Η συνάρτηση g αντιπροσωπεύει το κόστος για να φθάσουμε από το αρχική κατάσταση στην τρέχουσα κατάσταση, ενώ η συνάρτηση h είναι μια ευρετική

συνάρτηση που εκτιμά την απόσταση από την παρούσα κατάσταση στην κατάσταση στόχο

3.7 Ψευδοκώδικας του A*

Δημιούργησε στοίβα S για τα δεδομένα προς επεξεργασία

Όσο (Η στοίβα δεν είναι άδεια ή δεν έχει βρεθεί ο στόχος)

Βρες το στοιχείο x της στοίβας S με το μικρότερο κόστος

Εκχώρησε x σε προσωρινή μεταβλητή y

Σβήσε από την στοίβα το στοιχείο x

Αν είναι το y είναι λύση

Σταμάτα

Αλλιώς

Δημιούργησε τα παιδιά του y και τοποθέτησε τα στην αρχή της στοίβας S

3.8 Πολυπλοκότητα του A*

Η αναζήτηση A* είναι πλήρης, βέλτιστη και βέλτιστα αποδοτική μεταξύ όλων των αλγορίθμων αυτού του είδους. Δυστυχώς, αυτό δεν σημαίνει ότι ο A* είναι η απάντηση για όλες μας τις ανάγκες αναζήτησης. Το πρόβλημα είναι ότι στα περισσότερα προβλήματα, ο αριθμός των κόμβων μέσα στο χώρο αναζήτησης εξακολουθεί να αυξάνεται εκθετικά με το μήκος της λύσης. Έχει αποδειχθεί ότι η αύξηση θα είναι εκθετική εκτός αν το σφάλμα της ευρεστικής συνάρτησης δεν αυξάνεται γρηγορότερα από το λογάριθμο του πραγματικού κόστους διαδρομής. Σε μαθηματική σημειογραφία, η συνθήκη για αύξηση χαμηλότερη της εκθετικής είναι

$$|h(n)-h^*(n)| \leq O(\log h^*(x))$$

Όπου h* είναι το πραγματικό κόστος της μετάβασης από τον κόμβο n στον στόχο. Για όλους σχεδόν τους ευρεστικούς μηχανισμούς που χρησιμοποιούνται στην πράξη, το σφάλμα είναι τουλάχιστον ανάλογο με το κόστος διαδρομής και η εκθετική αύξηση που προκύπτει τελικά ξεπερνά τις δυνατότητες οποιουδήποτε υπολογιστή. Για αυτό αρκετές φορές είναι άσκοπο να προσπαθούμε να βρούμε βέλτιστη λύση. Ο χρόνος υπολογισμού δεν είναι όμως το κύριο μειονέκτημα της αναζήτησης A*. Επειδή διατηρεί στην μνήμη όλους τους κόμβους που παράγονται, ο αλγόριθμος A* εξαντλεί πολύ συχνά τον χώρο που έχουμε στην διάθεση μας. Για αυτόν τον λόγο ο A* δεν είναι χρήσιμος για πολλά προβλήματα μεγάλης κλίμακας. Για τον λόγω αυτό έχουν επινοηθεί νέοι αλγόριθμοι που ξεπερνούν το πρόβλημα του χώρου χωρίς να θυσιάσουν τη βέλτιστη συμπεριφορά ή την πληρότητα, με μικρό κόστος σε χρόνο εκτέλεσης.

3.9 Ο αλγόριθμος IDA*(Iterative deepening A star)

Παραλλαγή του A* αποτελεί ο IDA* (A* με επαναληπτική εκβάθυνση) ο οποίος αναπτύσσει το δένδρο αναζήτησης κατά βάθος σε διαδοχικές επαναλήψεις, αξιοποιώντας μια ευρετική συνάρτηση για να επιλέξει την επεκτεινόμενη κάθε φορά κατάσταση, αλλά όταν η ευρετική τιμή μίας νέας κατάστασης ξεπερνά το όριο που έχει τεθεί για την τρέχουσα επανάληψη όλο το υποδένδρο το οποίο ξεκινά από αυτήν κλαδεύεται. Στην επόμενη επανάληψη, όπου όπως στον IDS το δένδρο αναζήτησης κατασκευάζεται από την αρχή, το νέο ευρετικό όριο τίθεται στη μικρότερη τιμή που εμφανίστηκε στις καταστάσεις οι οποίες κλαδεύτηκαν κατά την προηγούμενη επανάληψη. Ο IDA* χρησιμοποιεί αναδρομική κλήση του αλγόριθμου depth-first search(DFS), θέτονται κάθε φορά ένα καινούργιο όριο. Η πολυπλοκότητα χώρου του αλγορίθμου είναι O(c) και η πολυπλοκότητα χώρου είναι $O(b^{c/e})$. Κύριο χαρακτηριστικό του IDA* είναι η περιορισμένη χρήση μνήμης. Πιο αναλυτικά ο αλγόριθμος δεν χρειάζεται να αποθηκεύει τα δεδομένα που επισκέφτηκε στην προηγούμενη επανάληψη αλλά τα διαγράφει και δημιουργεί καινούργιο δέντρο και

κάνει καινούργια κλαδέματα σε κάθε αύξηση του ορίου. Σε κάθε επανάληψη οι λίστες που φυλάσσουν τα δεδομένα προς επεξεργασία και τα δεδομένα που ο αλγόριθμος έχει επισκεφτεί καθαρίζονται ώστε είναι κενές για την επόμενη επανάληψη.

3.10 Ψευδοκώδικας IDA*

Δημιούργησε στοίβα S για τα δεδομένα προς επεξεργασία
Δημιούργησε στοίβα visited για τους κόμβους που θα έχει επισκεφτεί ο αλγόριθμος

```
Διαδικασία DFS με threshold(κόμβος, όριο)
    Όσο ( Η στοίβα δεν είναι άδεια ή δεν έχει βρεθεί λύση )
        Εκχώρησε το πρώτο στοιχείο της στοίβας x σε προσωρινή μεταβλητή y
        Τοποθέτησε το στοιχείο x στην αρχή της στοίβας visited
        Σβήσε από την στοίβα το στοιχείο x
        Αν (όριο ≥ Ευρετική_Συνάρτηση(y) )
            Αν (y είναι λύση)
                Λύση =  Αληθές
                Επέστρεψε λύση, νέο όριο
            Αλλιώς
                Δημιούργησε τα παιδιά του y και τοποθέτησε τα στην
            αρχή της στοίβας S
νέο όριο = min(visited)
Επέστεψε λύση, νέο όριο

Διαδικασία IDA*  (αρχ. κόμβος)
    λύση = ψευδές
    λύση, όριο = Τρέξε Διαδικασία DFS με threshold (αρχ. κόμβος, όριο )
    Όσο (λύση είναι ψευδές)
        λύση, όριο = Τρέξε Διαδικασία DFS με threshold (αρχ. κόμβος, όριο )
```

Αν (λύση είναι αληθές)

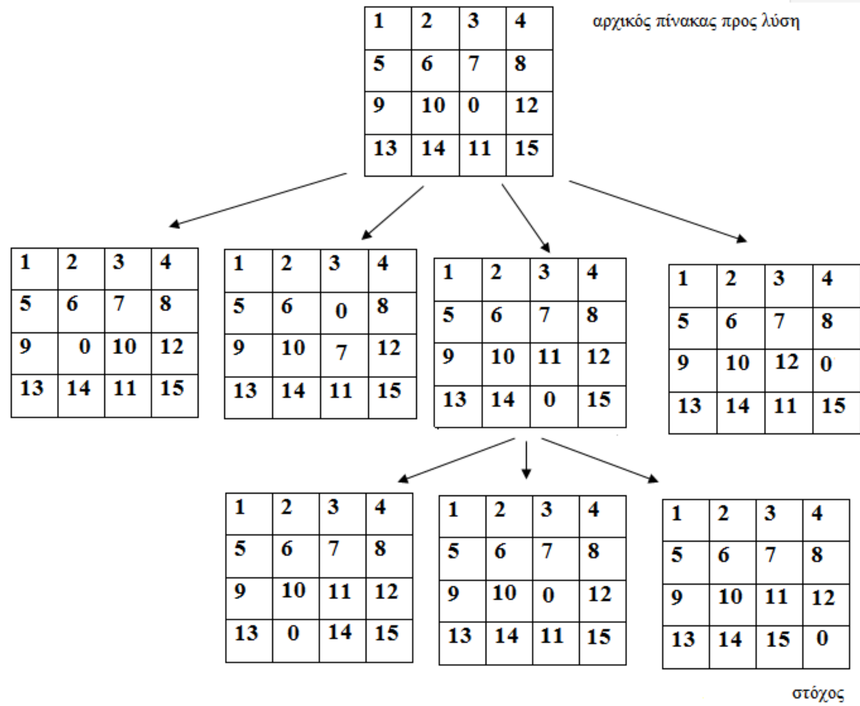
Βρέθηκε λύση

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

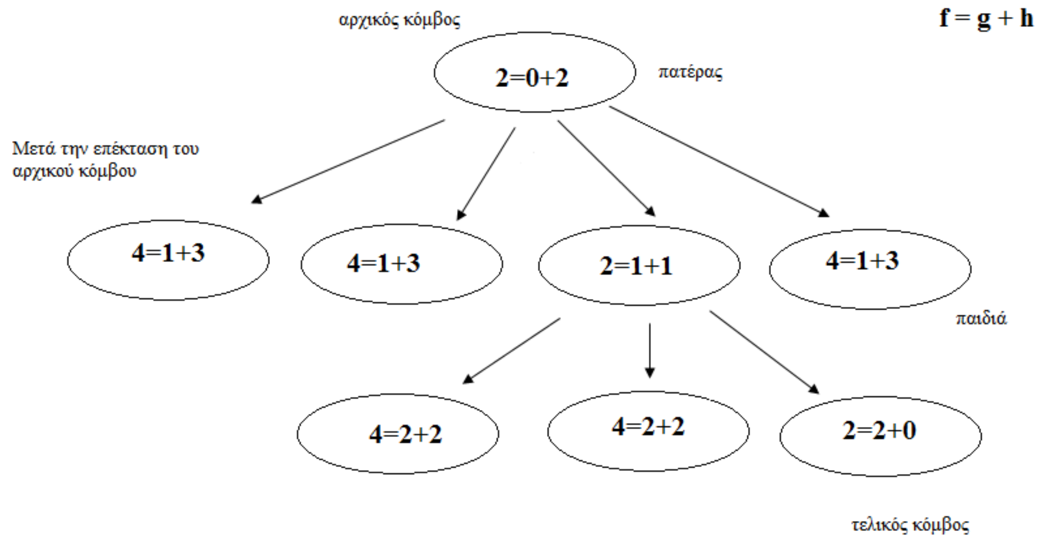
3.11 15-puzzle

Το παζλ αποτελείται από ένα πλαίσιο που περιέχει 15 συρόμενα τετράγωνα και ένα κενό σε διάταξη 4x4. Κάθε τετράγωνο του puzzle είναι αριθμημένο από 1 έως 15. Σκοπός είναι να τοποθετηθούν οι αριθμοί στην σωστή διάταξη (εικόνα) μετακινώντας τα τετράγωνα με την βοήθεια του κενού. Το κενό μπορεί να μετακινηθεί προς τέσσερις κατευθύνσεις, δεξιά, αριστερά, πάνω και κάτω. Το puzzle υπάρχει και σε άλλα μεγέθη όπως το 8-puzzle, δηλαδή διάταξη 3x3, ή σε μεγαλύτερα μεγέθη. Το 15-puzzle εφευρέθηκε από τον Noyes Chapman, στην Καναστότα της Νέας Υόρκης, ο οποίος το δήλωσε σαν πατέντα τον Μάρτιο του 1874. Έγινε ευρέως γνωστό στις ΗΠΑ το 1880. Τα n-puzzle είναι κλασικό πρόβλημα για την μοντελοποίηση ευρετικών αλγορίθμων. Συνήθης ευρετικοί μηχανισμοί για την λύση του συγκεκριμένου προβλήματος είναι ο αριθμός των τετραγώνων που δεν βρίσκονται στην σωστή θέση και η εύρεση της Manhattan απόστασης του κάθε τετραγώνου σε σχέση με την θέση του στην κατάσταση στόχου.

Δέντρο αναζήτησης για ένα 15-puzzle (απεικόνιση πινάκων):



Δέντρο Αναζήτησης για ένα 15-puzzle (απεικόνιση κόμβων) :



3.12 Επιλυσιμότητα του 15-puzzle

Δεν μπορεί να λυθεί ένα οποιοδήποτε 15-puzzle. Τα μισά από τα πιθανά puzzle (τα πιθανά puzzle είναι 16! = 20.922.789.888.000) είναι αδύνατον να λυθούν. O Johnson & Story το (1879) αναφέρουν τον τρόπο για να διαπιστώσει κανείς αν ένα 15-puzzle λύνετε. Ονομάζουμε n τον αριθμό όπου, ένα τετράγωνο περιέχει το αριθμό i εμφανίζετε πριν από n αριθμούς (ξεκινώντας από αριστερά προς δεξιά και από πάνω προς τα κάτω) που είναι μικρότεροι από i. Αθροίζουμε τα n για όλα τα i.

$$N \equiv \sum_{i=1}^{15} n_i = \sum_{i=2}^{15} n_i ,$$

Το άθροισμα είναι το ίδιο είτε το ορίσουμε με αρχή το 1 είτε με 2, αφού αριθμοί μικρότεροι του 1 δεν υπάρχουν στο puzzle. Ορίζουμε e την στήλη όπου βρίσκεται το κενό. Αν N+e είναι άρτιος τότε το puzzle έχει σίγουρα λύση. Αλλιώς, αν N+e είναι περιττός τότε το puzzle δεν έχει λύση.

13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	

Το παραπάνω τυχαίο τετράγωνο έχει N =12+9+9+5+4+4+3+3+0+3+3+2 +1 +1+0. Δηλαδή, N=59. Αφού ο αριθμός αυτός είναι περιττός η παραπάνω διάταξη των αριθμών του puzzle δεν μπορεί να λυθεί.

3.13 Αναδρομή

Με τον όρο αναδρομή εννοείται η δυνατότητα ένας κανόνας να περιέχει στο σώμα του μια κλήση προς τον εαυτό του. Οι κανόνες που χρησιμοποιούν αναδρομή χαρακτηρίζονται σαν αναδρομικοί κανόνες. Η χρήση της αναδρομής οδηγεί σε μικρότερα προγράμματα.

4. Υλοποίηση

Στην εργασία ασχολούμαστε με την μελέτη των δυο αλγορίθμων καθώς και με την υλοποίηση προγράμματος με το όποιο θα συγκριθούν οι αποδόσεις των αλγορίθμων. Σκοπός αυτής της εργασίας είναι η σύγκριση των αλγορίθμων A* και IDA*. Για να γίνει αυτό οι δυο αλγόριθμοι υλοποιήθηκαν στην γλώσσα προγραμματισμού C++. Για την υλοποίηση θα χρησιμοποιήσουμε σύγχρονες βιβλιοθήκες(C++11) όπως η deque, η tuple, η array και η vector. Οι συναρτήσεις που λαμβάνουμε αυτές τις βιβλιοθήκες είναι βασικές οντότητες για τον κώδικα.

Ως αναφορά τους αλγορίθμους που θα υλοποιηθούν, ο IDA star χρησιμοποιεί την αναδρομή ως βασικό του συστατικό. Αρχικά κάνει *αναζήτηση κατά βάθος (Depth First Search)* με όριο το αποτέλεσμα της ευρετικής συνάρτησης του αρχικού puzzle που δίνετε προς λύση. Στο συγκεκριμένο παράδειγμα η ευρετική συνάρτηση που χρησιμοποιούμε είναι το *manhattan distance* όλων των πλακιδίων του puzzle. Αν για παράδειγμα το *manhattan distance* του αρχικού puzzle είναι 15, τότε ο DFS επεκτείνει μόνο τους κόμβους που είναι ίσοι ή μικρότεροι από το συγκεκριμένο threshold. Αν δεν βρεθεί λύση μετά την επέκταση όλων των συγκεκριμένων κόμβων τότε το threshold αυξάνεται και παίρνει την τιμή του μικρότερου από τα ξεπερασμένου threshold. Για παράδειγμα, αν ο DFS βρήκε μόνο κόμβους που είχαν Manhattan distance 17, 19 και 22, θα επαναλάβει την αναζήτηση DFS με threshold το 17 που είναι το μικρότερο από τα ξεπερασμένα thresholds. Η συνάρτηση DFS επιστρέφει στην συνάρτηση IDA* και μια λογική μεταβλητή η οποία φέρει την πληροφορία σχετικά με το άμα έχει ή όχι βρεθεί λύση από την προηγούμενη αναδρομή.

Ο A*(A star) δεν χρησιμοποιεί αναδρομή. Σαν είσοδος του δίνεται ένας πίνακας προς λύση. Ο A* επεκτείνει τον αρχικό κόμβο, δηλαδή δημιουργεί το πολύ 4 παιδιά του κόμβου αυτού και τα προσθέτει στην στοίβα προς επεξεργασία. Έπειτα, βρίσκει το στοιχείο της στοίβας (φύλλο του δέντρου) που έχει το μικρότερο f, ελέγχει αν είναι λύση. Αν είναι, εκτυπώνει πως βρήκε λύση. Διαφορετικά επεκτείνει τον

κόμβο αυτόν και τον τοποθετεί στην στοίβα επεξεργασίας. Τα βήματα αυτά επαναλαμβάνονται μέχρι να βρεθεί λύση.

Στην διαδικασία της υλοποίησης κάνουμε την εξής παραδοχή: οι πίνακες για τους οποίους, ένας τουλάχιστον από τους 2 αλγορίθμους, χρειάζεται μεγάλο χρονικό διάστημα για να βρεθεί λύση δεν λαμβάνονται υπόψη. Χρησιμοποιούμε αυτήν την παραδοχή επειδή γνωρίζουμε πως ο A* αντιμετωπίζει προβλήματα με τα μεγάλα δέντρα αναζήτησης ως αναφορά τον χώρο, διότι πρέπει να αποθηκεύει πολύ μεγάλο όγκο κόμβων. Από την άλλη μεριά ο IDA* δεν αντιμετωπίζει προβλήματα χώρου λόγω της επαναληπτικής του μορφής.

5. Συναρτήσεις και δομές δεδομένων που χρησιμοποιήθηκαν

5.1 Η συνάρτηση Ramdom_Array()

Οι πίνακες προς επεξεργασία, δηλαδή οι πίνακες που είναι είσοδοι στους δύο πίνακες που μελετούμε πρέπει να είναι τυχαίοι. Αυτό συμβαίνει για να αποδείξουμε ότι οι αλγόριθμοι που υλοποιήσαμε δουλεύουν σωστά για κάθε πίνακα. Για την δημιουργία των τυχαίων πινάκων υλοποιούμε την συνάρτηση Random_Array η συνάρτηση αυτή δημιουργεί μια δομή vector, ως μονοδιάστατο πίνακα 1x16. Σε αυτόν τον πίνακα εκχωρούνται οι αριθμοί από 0 έως 15 διαδοχικά. Ύστερα, χρησιμοποιούμε την συνάρτηση random_shuffle της βιβλιοθήκης <algorithm>. Η συνάρτηση αυτή ανακατεύει τυχαία τους αριθμούς που βρίσκονται στο vector. Τέλος, εκχωρούμε όλα τα δεδομένα του vector στον πίνακα της βιβλιοθήκης array ονομαζόμενο arr1 και επιστέφουμε τον πίνακα αυτό. Ο πίνακας που έχουμε επιστρέψει είναι ένας πίνακας 4x4 με τυχαία ανακατεμένους τους αριθμούς από 0 έως 15.

```
array<array<int,4>,4> Random_Array()
{
	array<array<int,4>,4> arr1;
	vector<int> myvector;

	srand((unsigned)time(NULL));
	for (int i=0; i<16; i++) myvector.push_back(i);
	random_shuffle ( myvector.begin(), myvector.end() );

	for(int i=0; i!=myvector.size(); i++)
	{
		arr1[i/4][i%4]=myvector[i];
	}

	return(arr1);
}
```

5.2 Ο πίνακας target

Ο πίνακας target είναι ο πίνακας που ταυτίζετε με την κατάσταση στόχου. Ο πίνακας αυτός στην υλοποίηση είναι καθολικός και σταθερός. Μπορεί δηλαδή να χρησιμοποιηθεί σε όλη την έκταση του προγράμματος και δεν μπορεί να μεταβληθεί. Όταν βρεθεί αυτός ο πίνακας από τους αλγορίθμους σημαίνει ότι έχει βρεθεί η λύση.

```
const array<array<int,4>,4> target = {{
                     {1, 2, 3, 4},
                     {5, 6, 7, 8},
                     {9, 10, 11, 12},
                     {13, 14, 15, 0}
       }};
```

5.3 Η δομή New_Array

Κατασκευάσαμε μια δομή(struct) που περιέχει έναν πίνακα και έναν ακέραιο αριθμό. Ουσιαστικά η δομή αυτή περιέχει τον εκάστοτε κόμβο και το βάθος του. Είναι πολύ σημαντικό να γνωρίζουμε το βάθος του κόμβου καθώς το βάθος g και το manhattan distance (h) μας δίνει το f, το οποίο χρησιμεύει ώστε να αποφασίσουμε αν πρέπει να επεκτείνουμε τον συγκεκριμένο κόμβο.

```
struct New_Array {
       array<array<int,4>,4> pinak;
       int g;
}str_test,Jim;
```

5.4 Η συνάρτηση min_deq

Όπως έχουμε προαναφέρει ο A Star (A*) επεκτείνει από στοίβα δεδομένων προς επεξεργασία τον πίνακα με το μικρότερο f. Ο αλγόριθμος min_deq βρίσκει τον πίνακα με το μικρότερο f και επιστρέφει την θέση που βρίσκεται ο πίνακας στην στοίβα. Σαν όρισμα η min_deq δέχεται μια δομή deque που περιέχει στοιχεία της struct New_Array. Στην προκειμένη περίπτωση είναι η στοίβα δεδομένων προς επεξεργασία του αλγορίθμου A star. Η συνάρτηση min_deq θέτει σαν minimum το πρώτο στοιχείο της στοίβας. Ύστερα, κάνει μια for-loop από την αρχή μέχρι το τέλος της στοίβας και αν κάποιο στοιχείο έχει μικρότερο f από το f του min τότε εκχωρεί το στοιχείο αυτό σαν minimum. Η συνάρτηση δεν επιστέφει το στοιχείο με το μικρότερο f άλλα την θέση του στοιχείου αυτού διότι με αυτόν τον τρόπο μπορούμε και να διαγράψουμε το στοιχείο από την στοίβα δεδομένων επεξεργασία καθώς το επεξεργαζόμαστε.

```
int min_deq(deque<New_Array> deq)
{
	int min=0;
	for(unsigned int i=0; i<deq.size(); i++)
	{
		if (manhattan(deq[i].pinak)+deq[i].g <
manhattan(deq[min].pinak)+deq[min].g)
		{
			min=i;
		}
	}
	return(min);
}
```

5.5 Η συνάρτηση Find_Solution_Path

Σκοπός της συνάρτησης αυτής είναι να εκτυπώνει την λύση του προβλήματος. Η συνάρτηση δέχεται σαν ορίσματα τα δεδομένα της στοίβας visited δηλαδή τα δεδομένα που έχει επισκεφτεί ο αλγόριθμος πριν βρει την λύση και τον ακέραιο bhmata, δηλαδή τον αριθμό των μετακινήσεων του κενού που χρειάστηκαν για να φθάσουμε από το δοθέν puzzle στην κατάσταση στόχο. Η συνάρτηση χρησιμοποιεί 2 for-loops. Το πρώτο έχει όρια από 0 έως τον αριθμό τον βημάτων και το δεύτερο έχει όρια από το τέλος της στοίβας visited έως την αρχή της. Στο τέλος αυτής της διαδικασίας θα έχουν εκτυπωθεί x πίνακες (όπου x ο αριθμός των μετακινήσεων του κενού για να φτάσουμε στην λύση) από τον πίνακα που δόθηκε προς λύση μέχρι τον τελικό. Η συνάρτηση εκτυπώνει το πρώτο στοιχείο που βρίσκεται πιο κοντά στο τέλος της στοίβας και το οποίο έχει g ίσο με το ζητούμενο. Αυτό συμβαίνει διότι ο πίνακας που μας οδήγησε στην λύση θα βρίσκεται πιο κοντά στο τέλος της στοίβας visited αφού αναφερόμαστε αναζήτηση που χρησιμοποιεί τον DFS.

```
void Find_Solution_Path(deque<New_Array> visited, int bhmata)
{

	cout<<"to megethos tou visited einai "<<visited.size()<<endl;
	cout<<"ta bhmata pou kanw einai "<<bhmata<<endl;


	for(int i=0; i<bhmata+1; i++)
	{
		for(int j=visited.size()-1;j>-1;j--)
		{
			if (visited[j].g==i)
			{
				PrintBoard(visited[j]);
				cout<<endl;
				break; }}}}
```

5.6 Η συνάρτηση Manhattan

Η συνάρτηση Manhattan χρησιμοποιείται για να υπολογίσουμε την Manhattan απόσταση (ή αλλιώς την απόσταση οικοδομικών τετραγώνων) ενός πίνακα. Είναι η ευρετική συνάρτηση με την οποία ο αλγόριθμος πληροφορείται αν οδεύει προς την λύση. Κάθε πίνακας έχει διαφορετική απόσταση Manhattan. Η συνάρτηση δέχεται σαν όρισμα έναν πίνακα. Για κάθε τετράγωνο υπολογίζει την απόσταση από την αρχική του θέση. Για παράδειγμα, το 6 αν βρίσκεται στην πάνω αριστερή απέχει μια θέση κάθετα και μια θέση οριζόντια, δηλαδή στο σύνολο 2. Το άθροισμα όλων των τετραγώνων από τις αρχικές τους θέσεις είναι η απόσταση Manhattan του πίνακα. Δεν υπολογίζουμε πόσο απέχει το 0, δηλαδή το κενό, από την αρχική του θέση. Για την υλοποίηση χρησιμοποιούμε 2 for-loop, από 0 έως 3 για να ερευνήσουμε όλον τον 4x4 πίνακα. Για κάθε στοιχείο του πίνακα βρίσκουμε το πηλίκο και το υπόλοιπο του αριθμού με το 4. Τα αποτελέσματα αυτά είναι η θέση της γραμμής και ή θέση της στήλη που θα έπρεπε να βρίσκεται το στοιχείο. Ύστερα, βρίσκουμε την απόλυτη διαφορά του πηλίκου και του υπολοιπου. Αυτά τα στοιχεία αθροιζόμενα μας δίνουν την απόσταση Manhattan του ενός στοιχείου. Άμα εκτελέσουμε την διαδικασία για τα 15 τετράγωνα βρίσκουμε την απόσταση Manhattan του πίνακα.

```
int manhattan(array<array<int,4>,4> something)// Algorithmos Manhattan . Einai h
eurestiki  sunartisi .
{

	int manhatt=0;
	for (int i=0;i<4;i++){
		for (int j=0;j<4;j++)
		{
			int current=something[i][j];
			if (current!=0)
			{
				current=current-1;
```

```
                                int modulo=current % 4;
                                int divided=current / 4;
                                int diffx=abs(modulo-j);
                                int diffy=abs(divided-i);
                                int currentdistance=diffx+diffy;
                                manhatt=manhatt+currentdistance;
                        }
                }
        }
        return(manhatt);
}
```

5.7 Η συνάρτηση PrintBoard

Χρειαζόμαστε μια συνάρτηση η οποία όποτε την καλούμε να εκτυπώνει τον πίνακα που επιθυμούμε. Αυτή η συνάρτηση είναι η PrintBoard. Δέχετε σαν όρισμα ένα στοιχείο δομής New_Array και εκτυπώνει τον πίνακα που περιέχετε στην δομή αυτή. Η συνάρτηση αυτή έχει χρήση στο πρόγραμμα όταν θέλουμε να εκτυπώνουμε την λύση, καθώς και σε στάδια αποσφαλμάτωσης και ότι θέλουμε να επιβεβαιώσουμε ότι το πρόγραμμα λειτουργεί σωστά.

```
void PrintBoard(New_Array tt) {  //ΣΥΝΑΡΤΗΣΗ  ΜΕ ΤΙΝ ΟΠΟΙΑ ΕΚΤΥΠΩΝΟΥΜΕ ΠΙΝΑΚΕΣ
ΣΤΗΝ ΟΘΟΝΗ

        array<array<int,4>,4> giorgos;
        giorgos=tt.pinak;
        for (int iRow = 0; iRow < 4; ++iRow)
        {
                for (int iCol = 0; iCol < 4; ++iCol)
                {
                        cout << giorgos[iRow][iCol];
                        cout << "  ";//auto einai to keno gia na xwrizontai oi
arithmoi metaksu tous
```

```
		}
		cout << endl;
	}
}
```

5.8 Η συνάρτηση Move

Οι αλγόριθμοι IDA* και A* επεκτείνουν κόμβους και τους τοποθετούν στην αρχή της στοίβας δεδομένων προς επεξεργασία. Η διαδικασία αυτή απαιτεί την μετακίνηση του κενού προς μια κατεύθυνση και την αντικατάστασή του κενού από τον αριθμό που βρισκόταν στην θέση που έλαβε το κενό. Για αυτήν την μετακίνηση χρησιμοποιείται η συνάρτηση Move. Σαν ορίσματα η συνάρτηση δέχεται ένα πίνακα και την κατεύθυνση. Η συνάρτηση βρίσκει που βρίσκεται το κενό και αποθηκεύει σε προσωρινές μεταβλητές τις συντεταγμένες της θέσης αυτής. Ύστερα, με εντολή επιλογής switch, αποθηκεύουμε την θέση που θα μετακινηθεί το κενό. Τέλος, έχοντας γνώσει όλων των δεδομένων που χρειαζόμαστε κάνουμε την αλλαγή των δυο τετραγώνων και επιστρέφουμε τον πίνακα.

```
array<array<int,4>,4> Move(array<array<int,4>,4> z, const EMove keMove) {
	int iRowSpace;
	int iColSpace

	for (int iRow = 0; iRow < 4; ++iRow) {
		for (int iCol = 0; iCol < 4; ++iCol) {
			if (z[iRow][iCol] == 0) {
				iRowSpace = iRow;
				iColSpace = iCol;
			}}}
	int iRowMove(iRowSpace);
	int iColMove(iColSpace);
	switch (keMove) {
		case keUp:
```

```
				{
					iRowMove = iRowSpace + 1;
					break;
				}
			case keDown:
				{
					iRowMove = iRowSpace - 1;
					break;
				}
			case keLeft:
				{
					iColMove = iColSpace + 1;
					break;
				}
			case keRight:
				{
					iColMove = iColSpace - 1;
					break;
				}
		}

		if (iRowMove >= 0 && iRowMove < 4 && iColMove >= 0 && iColMove < 4) {
			z[iRowSpace][iColSpace]		= z[iRowMove][iColMove];
			z[iRowMove][iColMove]		= 0;
		}
		return (z);
}
```

5.9 Η συνάρτηση A_Star

Η υλοποίηση του αλγορίθμου Α* αποτελεί κύριο στοιχείο του προγράμματος. Η συνάρτηση A_Star δέχεται σαν όρισμα ένα πίνακα προς λύση και εκτυπώνει τα βήματα και τον χρόνο που χρειάστηκε για να βρεθεί η λύση. Αρχικά, ορίζουμε 2 καθολικές στοίβες. Την στοίβα visited η οποία περιλαμβάνει τα δεδομένα που έχουμε επισκεφτεί και την στοίβα myStack2 η οποία περιέχει τα δεδομένα προς επεξεργασία. Τοποθετούμε το δοθέν αρχικό πίνακα μέσα στην στοίβα MyStack2. Χρησιμοποιούμε μια while-loop ώστε να τρέχει το πρόγραμμα έως η στοίβα αδειάσει. Τα επόμενα βήματα γίνονται επαναλήπτικά μεχρι να αδειάσει η στοίβα ή μέχρι να βρεθεί λύση. Χρησιμοποιούμε την συνάρτηση min_deq για να βρούμε το μικρότερο από τα περιεχόμενα της MyStack2. Επειδή επεξεργαζόμαστε το στοιχείο αυτό, το διαγράφουμε από την στοίβα MyStack2 και το τοποθετούμε στην στοίβα visited. Αν το τρέχον στοιχείο ταυτίζεται με τον πίνακα που αναζητούμε τότε εκτυπώνουμε ότι βρήκαμε την λύση και πόσα βήματα χρειάστηκαν. Επίσης επιβάλουμε στο πρόγραμμα να φύγει από την while-loop καθώς δεν θέλουμε να επεξεργαστούμε άλλους πίνακες. Διαφορετικά, αν δεν είναι λύση ο τρέχων πίνακας, ελέγχουμε αν υπάρχει στην στοίβα visited, δηλαδή αν το έχουμε επεξεργαστεί αυτόν τον κόμβο ή κάποιον άλλο κόμβο που έχει τον ίδιο πίνακα με αυτόν. Αν δεν ισχύει αυτό τότε δημιουργούμε όλα τα παιδιά του τρέχοντος πίνακα-κόμβου, αυξάνοντας ταυτόχρονα το βάθος τους κατά 1 σε σχέση με τον πατέρα τους και τοποθετούμε τα παιδιά του στην αρχή της στοίβας δεδομένων προς επεξεργασία.

```
void A_Star(array<array<int,4>,4> initial)
{

	cout <<endl<< "starting A Star"<< endl;

	PrintArray(initial);
```

```
Jim.pinak=initial;
Jim.h=0;
myStack2.push_front(Jim);

while (myStack2.size()>0)         {

       int minimum=min_deq(myStack2);

       Jim=myStack2[minimum];
       myStack2.erase( myStack2.begin()+minimum  );

       if (Jim.pinak==target)
       {
              visited2.push_back(Jim);
              cout << endl << " vrika apotelesma  " << endl;
              cout<< endl<<"H lush xreiazetai  "<<Jim.h<<" kinhseis "<<
endl;

              break;
       }

else if (find(visited2.begin(), visited2.end(),
Jim.pinak)==visited2.end())

                            {
              x2=Jim.pinak;

              int new_h=Jim.h;
              new_h++;

              visited2.push_back(Jim);

              Jim.pinak=Move(x2,keRight);
              Jim.h=new_h;
              myStack2.push_front(Jim);
```

```
                    Jim.pinak=Move(x2,keUp);
                    Jim.h=new_h;
                    myStack2.push_front(Jim);


                    Jim.pinak=Move(x2,keDown);
                    Jim.h=new_h;
                    myStack2.push_front(Jim);


                    Jim.pinak=Move(x2,keLeft);
                    Jim.h=new_h;
                    myStack2.push_front(Jim);

            }
        }

}
```

5.10 Η συνάρτηση ida

Η συνάρτηση ida υλοποιεί τον αλγόριθμο IDA*. Αρχικά καλούμε την DFS με όριο το manhattan distance του αρχικού puzzle. Αν ο αλγόριθμος DFS βρει λύση τότε θα μας επιστρέψει true την μεταβλητή solution και την στοίβα visited, δηλαδή την στοίβα με τα στοιχεία που προσπέλασε για να βρει την λύση. Η στοίβα αυτή θα χρησιμοποιηθεί για την κλήση της Find_Solution_Path. Αν ο αλγόριθμος DFS δεν βρει λύση τότε επιστέφει το μικρότερο από τα ξεπερασμένα όρια. Το όριο αυτό θα γίνει είσοδος στην επόμενη κλήση του DFS. Τα τελευταία δύο βήματα επαναλαμβάνονται μέχρις ότου βρεθεί λύση.

```
void ida(array<array<int,4>,4> initial)
{

	PrintArray(initial);
	bool solution=false;
	int kainourgio_thres;
std::tie(visited, solution, kainourgio_thres) = dfs ( initial,
manhattan(initial)) ; // η συναρτηση dfs εχει σαν εισοδο 2 ορισματα τον αρχικο
πινακα
// και το threshold που θα χρησιμοποιησει για να κανει dfs
	while (solution==false)
	{
		cout << endl << " IDA*" << endl;
		std::tie(visited, solution, kainourgio_thres)=dfs(initial,
kainourgio_thres);

	}
	if (solution==true)
	{
		cout << endl << " BRHKA LUSH  " << endl;
		Find_Solution_Path(visited,kainourgio_thres);
	}
}
```

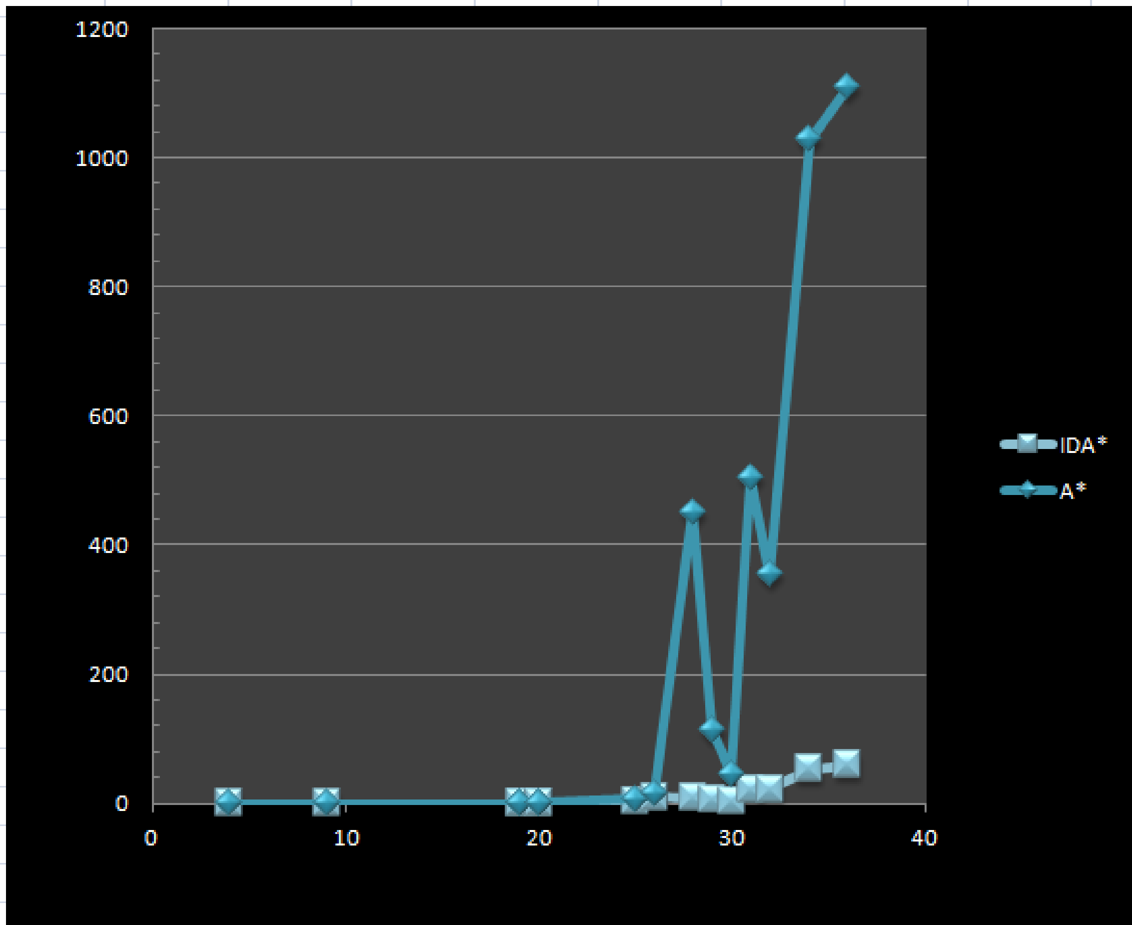
6. Αποτελέσματα

Μετά την ολοκλήρωση της υλοποίησης της εφαρμογής και έχοντας υπόψη το θεωρητικό υπόβαθρο εκτελούμε την εφαρμογή και δοκιμάζουμε τους αλγορίθμους στην επίλυση τυχαίων αρχικών πινάκων. Από τις 50 τυχαίες αρχικές καταστάσεις, η εφαρμογή βρήκε για 20 καταστάσεις λύση στο χρονικό περιθώριο της μισής ώρας που θέσαμε. Κατατάσσουμε τα καταγεγραμμένα αποτελέσματα ως προς f, δηλαδή ως προς τον αριθμό μετακινήσεων του κενού για να βρεθεί λύση. Εμφανίζουμε μόνο 1 αποτέλεσμα για κάθε τιμή του f. Ο παρακάτω πίνακας παρουσιάζει τα αποτελέσματα της εφαρμογής. Πιο συγκεκριμένα, για κάθε puzzle εμφανίζονται η Manhattan απόσταση, το f, ο χρόνος που χρειάστηκε ο IDA* για να βρει λύση και ο χρόνος που χρειάστηκε ο A* για να βρει λύση.

Αρχικό Εκτιμώμενο Κόστος (Manhattan απόσταση)	F / Κινήσεις για λύση puzzle	IDA* (sec)	A* (sec)
4	4	0,121	0,023
9	9	0,094	0,019
17	19	0,273	0,069
11	19	0,985	1,283
16	20	0,3	0,267
15	20	0,795	2,033
17	25	2,75	7,778
16	26	10	15
22	28	9	451
25	29	6	115
22	30	2	47
27	31	22	504

28	32	22	355
30	34	53,187	1028,1
22	36	59	1.110

Γράφημα Χρόνου(sec) / f



Όπως παρατηρούμε και στο γράφημα χρόνου προς f, δηλαδή χρόνος προς τον τελικό αριθμό των βημάτων προς την λύση, οι χρόνοι που χρειάζεται ο IDA* για να βρει μονοπάτι λύσης στα προβλήματα είναι σταθερά μικρός. Για πολύ μικρά f ο A* αποδίδει ανεπαίσθητα καλύτερα από τον IDA*, αλλά για μέτριου προς μεγάλου

μεγέθους f ο IDA* αποδίδει καλύτερα. Να αναφέρουμε επίσης πως όσο αυξάνετε το f αυξάνετε και η δυσκολία του puzzle άρα και ο όγκος των δεδομένων που πρέπει να διαχειριστεί ο αλγόριθμος για την εύρεση λύσης. Καταστάσεις που έχουν λύσεις με f>40 χρειάζονται πάνω από 30 λεπτά.

7. Συμπεράσματα και Μελλοντικές Επεκτάσεις

Το 1986 ο Korf έθεσε τις βάσεις του IDA*, ενός αλγορίθμου αναζήτησης που συνδυάζει τον A* και τον IDS. Παρόλα αυτά, 26 χρόνια μετά, η ερευνητική δραστηριότητα γύρω από αυτόν τον αλγόριθμο είναι περιορισμένη. Στην παρούσα εργασία υλοποιήθηκε με επιτυχία μια εφαρμογή που συγκρίνει στην πράξη τους 2 αλγορίθμους, με την χρήση τυχαίων πινάκων, και αποδεικνύει πως ο IDA* σε σχέση με τον A* βρίσκει λύσεις σε 15-puzzle προβλήματα σε πιο σύντομο χρόνο. με την βοήθεια ευρετικής συνάρτησης Manhattan Distance. Παρατηρούμε ότι οι χρόνοι έχουν εκθετική αύξηση και ότι για f> 25 ο A* αποδίδει πολύ χειρότερα από τον IDA*. Αυτό συμβαίνει διότι η πολυπλοκότητα χώρου του A* είναι πολύ μεγαλύτερη από αυτή του IDA*. Συνεπώς, η αναζήτηση στοιχείων στην στοίβα γίνεται εξαντλητική για τον αλγόριθμο A*.

Μια μελλοντική επέκταση της εφαρμογής θα μπορούσε να συμπεριλάβει την προσθήκη και άλλων αλγορίθμων πληροφορημένης αναζήτησης ώστε να δοκιμαστούν περεταίρω οι δυο αλγόριθμοι. Επίσης, μελλοντική επέκταση θα μπορούσε να είναι η οπτική δενδρική αναπαράσταση της λειτουργίας και των διαφορών των δυο αλγορίθμων.

Παράρτημα Α: Πηγαίος Κώδικας

```
#include <iostream>
#include <deque>
#include <array>
#include <tuple>
#include <time.h>
#include <vector>


using namespace std;

array<array<int,4>,4> myarray;
array<array<int,4>,4> y;

const array<array<int,4>,4> target = {{  // Κατασταση Στοχος. Στο συγκεκριμενο
puzzle ειναι η λυση .
                     {1, 2, 3, 4},
                     {5, 6, 7, 8},
                     {9, 10, 11, 12},
                     {13, 14, 15, 0}
       }};

struct New_Array {                   // Δομη η οποια περιεχει: 1.τον πινακα
(puzzle)
       array<array<int,4>,4> pinak;     //                        2. το βαθος
που βρησκεται ο συγκεκριμενος πινακας
       int h;
}str_test,Jim;                       // αντικειμενα δομής


enum EMove {  keUp = 's',
                            keDown = 'w',
                            keLeft = 'd',
                            keRight = 'a'};


///// Δήλωση Συναρτήσεων///////////////
void InitializeBoard();
void PrintBoard(New_Array tt);
array<array<int,4>,4> Move(array<array<int,4>,4> x, const EMove keMove);
int manhattan(array<array<int,4>,4> something);
std::tuple<deque<New_Array>, bool, int> dfs(array<array<int,4>,4> initial,int
threshold);
void ida(array<array<int,4>,4> initial);
void Find_Solution_Path(deque<New_Array> visited, int bhmata);
void PrintArray(array<array<int,4>,4> parray);
array<array<int,4>,4> Random_Array();
int min_deq(deque<New_Array> deq);
void A_Star(array<array<int,4>,4> initial);
////////////////////////////////////////
```

```
int main()
{
        for (int i=0; i<50; i++)
        {
              array<array<int,4>,4> initial=Random_Array();
              int clo = clock();

              ida(initial);


              int t=clock() - clo;
              float final_time=((float)t)/CLOCKS_PER_SEC;
                     if (final_time<60)
                     {
                            cout<< endl <<"the time spend is: "<<  final_time
<<" seconds"<< endl;
                     }
                     else
                     {
                            cout<< endl <<"the time spend is: "<<
int(final_time/60) <<" minutes and "<< int(final_time)%60<<" seconds"<< endl;
                     }



              int cloc=clock();

              A_Star(Random_Array());


              int t_A=clock() - cloc;
              float final_time_of_Astar=((float)t_A)/CLOCKS_PER_SEC;
                     if (final_time_of_Astar<60)
                     {
                            cout<< endl <<"the time spend is: "<<
final_time_of_Astar <<" seconds"<< endl;
                     }
                     else
                     {
                            cout<< endl <<"the time spend is: "<<
int(final_time_of_Astar/60) <<" minutes and "<< int(final_time_of_Astar)%60<<"
seconds"<< endl;
                     }
        }

       char test_char;
       cin>> test_char;  // αυτο εχει γραφτει ωστε το παραθυρο να παραμενει
ανοιχτο και να μην κλεινει αστραπιαια

       return EXIT_SUCCESS;
}
```

```
array<array<int,4>,4> Random_Array()
{

	array<array<int,4>,4> arr1;
	vector<int> myvector;

	srand((unsigned)time(NULL));
	for (int i=0; i<16; i++) myvector.push_back(i);
	random_shuffle ( myvector.begin(), myvector.end() );

	for(int i=0; i!=myvector.size(); i++)
	{
		arr1[i/4][i%4]=myvector[i];
	}


	return(arr1);
}


deque<New_Array> visited2;
deque<New_Array> myStack2;
array<array<int,4>,4> x2;

void A_Star(array<array<int,4>,4> initial)
{

	cout <<endl<< "starting A Star"<< endl;

	PrintArray(initial);
	Jim.pinak=initial;
	Jim.h=0;
	myStack2.push_front(Jim);

	while (myStack2.size()>0)   // Oso uparxoun stoixeia mesa sthn stoiva
dedomenwn pros epeksergasia
	{
		time_t thistime;
		thistime=time(NULL);
		thistime=thistime+60*30;

		while(time(NULL)<thistime)
		{
		int minimum=min_deq(myStack2);

		Jim=myStack2[minimum];
		myStack2.erase( myStack2.begin()+minimum  );

		if (Jim.pinak==target)
		{
			visited2.push_back(Jim);
			cout << endl << " vrika apotelesma  " << endl;
			cout<< endl<<"H lush xreiazetai  "<<Jim.h<<" kinhseis "<<
endl;
```

```
                        break;
                }

                else if (find(visited2.begin(), visited2.end(),
Jim.pinak)==visited2.end())
                {
                        x2=Jim.pinak;

                        int new_h=Jim.h;
                        new_h++;


                        visited2.push_back(Jim);

                        Jim.pinak=Move(x2,keRight);
                        Jim.h=new_h;
                        myStack2.push_front(Jim);


                        Jim.pinak=Move(x2,keUp);
                        Jim.h=new_h;
                        myStack2.push_front(Jim);


                        Jim.pinak=Move(x2,keDown);
                        Jim.h=new_h;
                        myStack2.push_front(Jim);


                        Jim.pinak=Move(x2,keLeft);
                        Jim.h=new_h;
                        myStack2.push_front(Jim);

                }
        }
        }
}

int min_deq(deque<New_Array> deq)
{
        int min=0;
        for(unsigned int i=0; i<deq.size(); i++)
        {
                if (manhattan(deq[i].pinak)+deq[i].h <
manhattan(deq[min].pinak)+deq[min].h)
                {
                        min=i;
                }
        }
        return(min);
}
```

```
deque<New_Array> visited;//domh dedomenwn pou krataei tous pinakes pou exoume
episkefthei
deque<New_Array> myStack;     // στοιβα δεδομένων προς επεξεργασια
array<array<int,4>,4> x;


std::tuple< deque<New_Array>, bool, int> dfs(array<array<int,4>,4> initial, int
threshold) // η συναρτηση αυτη κανει dfs με threshold .
       {                                                                      //
επιστρεφει 3 τιμες : 1. το solution, αμα εχουμε φτασει δηλαδη στην
                                                                           //
κατασταση στοχος , και 2. την μικροτερη απο τις τιμες που ξεπερνουν το
                                                                              //
threshold 3. Τον πινακα visited για να βρουμε ύστερα το μονοπάτι λυσης
       cout << endl<<  "Starting DFS search"<< endl<<"to orio einai
"<<threshold;
       int new_thres=10000;

       Jim.pinak=initial;
       Jim.h=0;
       myStack.push_front(Jim);// kanw push to initial stin arxi tou stack


       while (myStack.size()>0)   // Oso uparxoun stoixeia mesa sthn stoiva
dedomenwn pros epeksergasia

       {


              Jim=myStack.front();
              myStack.pop_front();


              if (manhattan(Jim.pinak)+Jim.h <= threshold)
              {


                     if (Jim.pinak==target)
                     {
                            visited.push_back(Jim);

                            cout << endl << " vrika apotelesma  " << endl;

                            cout<< endl<<"H lush xreiazetai  "<<Jim.h<<"
kinhseis "<< endl;
                            return std::make_tuple(visited, true, Jim.h);
                            break;
                     }

                     else if (find(visited.begin(), visited.end(),
Jim.pinak)==visited.end())// psaxnei se olo to visited kai ama den to brei

                                                 // tote epistrefei
visited.end()
```

```
                        {
                                x=Jim.pinak;

                                int new_h=Jim.h;
                                new_h++;        // αυξανουμε το βάθος


                                                                                        //  bale ta
paidia tou mesa sto myStack
                                visited.push_back(Jim);

                                Jim.pinak=Move(x,keRight);
                                Jim.h=new_h;
                                myStack.push_front(Jim);


                                Jim.pinak=Move(x,keUp);
                                Jim.h=new_h;
                                myStack.push_front(Jim);


                                Jim.pinak=Move(x,keDown);
                                Jim.h=new_h;
                                myStack.push_front(Jim);


                                Jim.pinak=Move(x,keLeft);
                                Jim.h=new_h;
                                myStack.push_front(Jim);

                        }
                }

                else                            ///             ***** BRHSKOYME TO
MIKROTERO APO TA KSEPERASMENA THRESHOLD ******
                {
                        if (       (threshold<(manhattan(Jim.pinak)+Jim.h) )
&&          ( (manhattan(Jim.pinak)+Jim.h) < new_thres)    )
                        {
                                new_thres=manhattan(Jim.pinak)+Jim.h;
                        }

                }
        }


        cout<<endl<<"to neo orio pou epistrefw einai "<<new_thres;
        cout << endl << " Telos   " << endl;
        return std::make_tuple(NULL, false, new_thres);    //EPISTREFEI TO
MIKROTERO APO TA KSEPERASMENA THRESHOLD
        visited.clear();
        myStack.clear();
}
```

```
void ida(array<array<int,4>,4> initial)
{

	PrintArray(initial);

	bool solution=false;

	int kainourgio_thres;

	time_t thistime;
	thistime=time(NULL);
	thistime=thistime+60*30;

	std::tie(visited, solution, kainourgio_thres) =dfs(initial,

manhattan(initial));// η συναρτηση dfs εχει σαν εισοδο 2 ορισματα τον αρχικο
πινακα

	                                                    // και το threshold που θα
χρησιμοποιησει για να κανει dfs
	while (solution==false || time(NULL)<thistime )
	{
			cout << endl << " IDA*" << endl;
			std::tie(visited, solution, kainourgio_thres)=dfs(initial,
kainourgio_thres);
	}

	if (solution==true)
	{
		cout << endl << " BRHKA LUSH  " << endl;

		Find_Solution_Path(visited,kainourgio_thres);

	}

}

void Find_Solution_Path(deque<New_Array> visited, int bhmata)
{

	cout<<"to megethos tou visited einai "<<visited.size()<<endl;
	cout<<"ta bhmata pou kanw einai "<<bhmata<<endl;


	for(int i=0; i<bhmata+1; i++)
	{
		for(int j=visited.size()-1;j>-1;j--)
		{
			if (visited[j].h==i)
```

```
                        {
                                PrintBoard(visited[j]);
                                cout<<endl;
                                break;
                        }
                }
        }
}



int manhattan(array<array<int,4>,4> something)// Algorithmos Manhattan . Einai h
eurestiki  sunartisi .
{

        int manhatt=0;
        for (int i=0;i<4;i++){

                for (int j=0;j<4;j++)
                {
                        int current=something[i][j];
                        if (current!=0)
                        {

                                current=current-1;
                                int modulo=current % 4;
                                int divided=current / 4;
                                int diffx=abs(modulo-j);
                                int diffy=abs(divided-i);
                                int currentdistance=diffx+diffy;
                                manhatt=manhatt+currentdistance;
                        }
                }
        }

        return(manhatt);

}

void PrintBoard(New_Array tt) {  //ΣΥΝΑΡΤΗΣΗ  ΜΕ ΤΙΝ ΟΠΟΙΑ ΕΚΤΥΠΩΝΟΥΜΕ ΠΙΝΑΚΕΣ
ΣΤΗΝ ΟΘΟΝΗ

        array<array<int,4>,4> giorgos;
        giorgos=tt.pinak;
        for (int iRow = 0; iRow < 4; ++iRow)
        {
                for (int iCol = 0; iCol < 4; ++iCol)
                {

                        cout << giorgos[iRow][iCol];
                        cout << "  ";//auto einai to keno gia na xwrizontai oi
arithmoi metaksu tous
                }
```

```
		cout << endl;
	}
}

void PrintArray(array<array<int,4>,4> parray) {

	cout<< "The array that we are going to solve is: "<<endl;
	for (int iRow = 0; iRow < 4; ++iRow)

	{
		for (int iCol = 0; iCol < 4; ++iCol)
		{
			cout << parray[iRow][iCol];
			cout << "   ";
		}

		cout << endl;
	}
}



array<array<int,4>,4> Move(array<array<int,4>,4> z, const EMove keMove) {   //
ALGORITHMOS POU METAKINEI TO KENO STON PINAKA
	int iRowSpace;
// DEXETAI SAN ORISMATA TON PINAKA PROS EPEKSERGASIA
	int iColSpace;
//  KAI THN KATEYTHUNSH POY THA GINEI H KINHSH

	for (int iRow = 0; iRow < 4; ++iRow) {
		for (int iCol = 0; iCol < 4; ++iCol) {
			if (z[iRow][iCol] == 0) {
				iRowSpace = iRow;
				iColSpace = iCol;
			}}}

	int iRowMove(iRowSpace);
	int iColMove(iColSpace);
	switch (keMove) {
		case keUp:
			{

				iRowMove = iRowSpace + 1;
				break;
			}
		case keDown:
			{

				iRowMove = iRowSpace - 1;
				break;
			}
		case keLeft:
			{
```

```
                        iColMove = iColSpace + 1;
                        break;
                    }
                case keRight:
                    {

                        iColMove = iColSpace - 1;
                        break;
                    }
        }


        if (iRowMove >= 0 && iRowMove < 4 && iColMove >= 0 && iColMove < 4) {
            z[iRowSpace][iColSpace]     = z[iRowMove][iColMove];
            z[iRowMove][iColMove]       = 0;
        }
        return (z);
}


bool operator==
        (const New_Array& n,
         const array<array<int,4>,4>& a)
{
        return (n.pinak == a);
}
```

Βιβλιογραφία

1. Richard E. Korf, Depth-first Iterative-Deepening: An Optimal Admissible Tree search (1985), Artificial Intelligence Volume 27 Issue 1, Sept.1985, 97 - 109
2. Richard E. Korf and A. Felner. Disjoint pattern database heuristics. Artificial Intelligence, Volume 134, 2002, 9-22
3. Judea Pearl, Heuristics: Intelligent search strategies for computer problem solving, Addison-Wesley, 1984
4. Stuart Russel και Peter Norvig, Τεχνητή Νοημοσύνη, μια σύγχρονη προσέγγιση, εκδόσεις Κλειδάριθμος
5. Βλαχάβας, Κεφαλάς, Βασιλειάδης, Κόκκορας, Σακελλαρίου, Τεχνητή Νοημοσύνη - Γ' Έκδοση, Εκδόσεις Γκιούρδας,
6. B. Goertzel, Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms (Frontiers in Artificial Intelligence and Applications)
7. M. Tim Jones, Artificial Intelligence :A Systems Approach
8. Hart, Nilsson, A formal basis for the heuristic determination of minimum cost paths, 1968
9. http://el.wikipedia.org/wiki/Υπολογιστική_νοημοσύνη
10. http://www.daniweb.com/software-development/cpp/threads/351851/thread-safe-timer-for-c-callback
11. http://www.apl.jhu.edu/~hall/AI-Programming/IDA-Star.html
12. http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2
13. http://intelligence.worldofcomputing.net/ai-search/iterative-deepening-a-star.html
14. http://mathworld.wolfram.com/15Puzzle.html