ΜΑΘΗΜΑ: **ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ**

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: **ΡΕΦΑΝΙΔΗΣ ΙΩΑΝΝΗΣ**

Ο ΚΟΣΜΟΣ ΤΩΝ ΚΥΒΩΝ

Ονοματεπώνυμο: Σιαλάκης Ραφαήλ Χρυσοβαλάντης
Αριθμός Μητρώου: ics22006

Περιεχόμενα

Εισαγωγή

Ο κόσμος των κύβων αποτελεί ένα κλασικό πρόβλημα διάταξης στην τεχνητή νοημοσύνη, που χρησιμοποιείται ευρέως για τη μελέτη και αξιολόγηση αλγορίθμων πληροφορημένης και μη πληροφορημένης αναζήτησης. Το πρόβλημα περιλαμβάνει τη μετάβαση από μια αρχική κατάσταση σε μια επιθυμητή τελική κατάσταση, μέσα από μια σειρά ενεργειών, με στόχο την επίλυση του προβλήματος με τον αποδοτικότερο δυνατό τρόπο. Αποτελεί ιδανικό πεδίο για την ανάλυση και σύγκριση αλγορίθμων λόγω της φαινομενικής απλότητας του, που όμως κρύβει έντονη υπολογιστική πολυπλοκότητα καθώς αυξάνονται οι παράμετροι.

Τα προβλήματα του κόσμου των κύβων ορίζονται στη γλώσσα **PDDL** (Planning Domain Definition Language), μια τυπική γλώσσα περιγραφής προβλημάτων και τομέων σχεδιασμού. Η PDDL επιτρέπει τη διατύπωση τόσο του πεδίου (domain), δηλαδή των κανόνων και των περιορισμών του προβλήματος, όσο και της συγκεκριμένης κατάστασης που καλούμαστε να επιλύσουμε. Αυτή η δομή προσφέρει έναν ισχυρό μηχανισμό για τη μοντελοποίηση σύνθετων προβλημάτων σχεδιασμού και επιτρέπει τη χρήση ποικίλων αλγορίθμων και στρατηγικών αναζήτησης.

Έχοντας στη διάθεσή μας ένα αρχείο PDDL, είναι κρίσιμο να εξάγουμε όσο το δυνατόν περισσότερες πληροφορίες από την περιγραφή του προβλήματος. Αυτό περιλαμβάνει την αναγνώριση των αρχικών και τελικών καταστάσεων, τον καθορισμό των επιτρεπτών ενεργειών (operators) και των περιορισμών που διέπουν την εφαρμογή τους. Η σωστή ανάλυση και κατανόηση αυτών των πληροφοριών μας επιτρέπει να μοντελοποιήσουμε το πρόβλημα αποτελεσματικά, δημιουργώντας μια αναπαράσταση κατάλληλη για επίλυση από τους επιλεγμένους αλγορίθμους αναζήτησης.

1. Αλγόριθμοι αναζήτησης

Η αναζήτηση είναι μια θεμελιώδης μέθοδος στην Τεχνητή Νοημοσύνη και τους αλγορίθμους, που χρησιμοποιείται για την επίλυση προβλημάτων σε χώρους καταστάσεων (state spaces). Οι **αλγόριθμοι αναζήτησης** διακρίνονται κυρίως σε **πληροφορημένους** (informed) και **απληροφορητους** (uninformed), ανάλογα με το αν χρησιμοποιούν πρόσθετες πληροφορίες για την επίτευξη του στόχου.

1.1 Απληροφόρητοι Αλγόριθμοι Αναζήτησης

Οι αλγόριθμοι αυτοί δεν έχουν καμία πληροφορία για την επίτευξη του στόχου πέρα από τον ίδιο τον ορισμό του προβλήματος (αρχική κατάσταση, μεταβάσεις και συνθήκη στόχου). Βασίζονται σε "τυφλές" στρατηγικές για να εξερευνήσουν τον χώρο αναζήτησης.

Βασικοί Αλγόριθμοι Απληροφόρητης Αναζήτησης:

1. **Διατεταγμένη Αναζήτηση σε Πλάτος (Breadth-First Search - BFS):**
   Εξερευνά πρώτα όλες τις καταστάσεις στο ίδιο βάθος πριν προχωρήσει στο επόμενο.
   Εγγυάται τη βέλτιστη λύση αν το κόστος μετάβασης είναι το ίδιο για όλες τις ακμές.
2. **Διατεταγμένη Αναζήτηση σε Βάθος (Depth-First Search - DFS):**
   Εξερευνά όσο πιο βαθιά μπορεί πριν επιστρέψει σε ανώτερα επίπεδα για άλλες επιλογές.
   Δεν εγγυάται βέλτιστη λύση, αλλά είναι πιο αποδοτική στη χρήση μνήμης.

1.2 Πληροφορημένοι Αλγόριθμοι Αναζήτησης

Οι αλγόριθμοι αυτοί αξιοποιούν πρόσθετες πληροφορίες για να καθοδηγήσουν την αναζήτηση προς τον στόχο πιο αποδοτικά. Η πληροφορία αυτή παρέχεται συνήθως μέσω μιας **ευρετικής συνάρτησης** (heuristic function), η οποία εκτιμά πόσο "κοντά" βρίσκεται μια κατάσταση στον στόχο.

Βασικοί Αλγόριθμοι Πληροφορημένης Αναζήτησης:

1. **Αναζήτηση με Ευρετική (Greedy Best-First Search):**
   Επιλέγει τις καταστάσεις με βάση την ευρετική συνάρτηση h(n)h(n)h(n), που εκτιμά το κόστος από την τρέχουσα κατάσταση μέχρι τον στόχο.
   Δεν εγγυάται βέλτιστη λύση.
2. **Αναζήτηση Α***:
   Συνδυάζει το πραγματικό κόστος από την αρχή g(n)g(n)g(n) και την ευρετική εκτίμηση h(n)h(n)h(n) για να ελαχιστοποιήσει το συνολικό κόστος f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n).
   Εγγυάται βέλτιστη λύση αν η ευρετική συνάρτηση είναι αποδεκτή (admissible) και συνεπής (consistent).

2. Παρουσίαση λειτουργίας του προγράμματος

2.1 Block

Αρχικά η αφαίρεση του προβλήματος ξεκινάει από την κλάση Block(), η οποία αποτελεί ένα record με δύο μεθόδους, την **equals**() για να συγκρίνουμε δύο μπλόκ μεταξύ τους και η μέθοδος **toString**() για την μετατροπή ενός Block σε αλφαριθμητικό

2.2 Substack

Στη συνέχεια τα blocks τοποθετούνται σε υποστοίβες (substacks), οι οποίες ακολουθούν τις ίδιες αρχές με τη δομή **Stack**() της java.

2.3 State

Η κλάση State περιγράφεται από ένα σύνολο από υποστοίβες, οι οποίες αποθηκεύονται σε μια λίστα. Πέραν αυτού, κάθε κατάσταση περιγράφεται από τις ιδιότητες parent, που είναι ο γονέας μιας κατάστασης, g που είναι το κόστος για να φτάσουμε στη συγκεκριμένη κατάσταση, h που είναι η ευρετική τιμή και f που αποτελεί το άθροισμα τους. Στη κλάση αυτή οι μέθοδοι **generateChildrenUninformed**() και **generateChildrenInformed**() παράγουν τα παιδιά μιας κατάστασης σε αλγόριθμο απληροφόρητης και πληροφορημένης αναζήτησης αντίστοιχα. Επιστρέφουν μια λίστα με όλες τις καταστάσεις που γεννιούνται από την τωρινή κατάσταση που βρισκόμαστε, εκτός αν έχουμε ξαναπεράσει από την κατάσταση αυτή. Αν δεν έχουμε ξαναεπισκεφθεί έναν κόμβο-παιδί, ορίζουμε την κατάσταση που βρισκόμαστε τώρα ως γονιό του, τον προσθέτουμε στη λίστα και ορίζουμε τις ευρετικές του τιμές αν εκτελούμε αλγόριθμο πληροφορημένης αναζήτησης. Με σκοπό την μετάβαση από κατάσταση σε κατάσταση ήταν απαραίτητη η υλοποίηση μιας μεθόδου **deepCopyStacks**, η οποία δημιουργεί ένα αντίγραφο μιας λίστας από υποστοίβες, έτσι ώστε να μην αλλοιωθεί η αρχική μας κατάσταση κατά την δημιουργία των παιδιών, καθώς και ύψιστης σημασίας είναι η συνάρτηση **filterChild**(), ή οποία αφαιρεί όλες τις άδειες υποστοίβες από ένα State, αν υπάρχουν και δημιουργεί μια καινουργία, εξασφαλίζοντας ότι θα αποφευχθούν κινήσεις από το τραπέζι πάλι στο τραπέζι. Επιπλέον η μέθοδος **normalizeState**(), εξασφαλίζει οτι θα αφαιρεθούν όλες οι άδειες υποστοίβες από ένα State και οτι θα ταξινομηθούν οι υποστοίβες του. Μας είναι χρήσιμο κατά τη χρήση της μεθόδου **equals**(), οπού αποφεύγουμε καταστάσεις οι οποίες παράγουν τα ίδια παιδιά. Πιο συγκεκριμενα:
(ON TABLE C) (ON TABLE B (CLEAR C) A ON B
Όπου το C θα είναι στο τραπέζι είτε δεξιά είτε αριστερά της άλλης υποστοίβας.
Χρησιμοποιείται επιπλέον μια μέθοδος **filterState**(), η οποία απλά αφαιρεί όλες τις άδειες υποστοίβες, καθαρά και μόνο για λόγους καλύτερης εμφάνισης της εξόδου.

2.4 Problem

Η κλάση Problem αναπαριστά το πρόβλημα που επιλύουμε, το οποίο περιλαμβάνει την αρχική κατάσταση των blocks (initial state) και την επιθυμητή τελική κατάσταση (goal state). Αποτελεί τον πυρήνα του προβλήματος και λειτουργεί ως γέφυρα μεταξύ των δεδομένων εισόδου και των αλγορίθμων αναζήτησης που εφαρμόζονται. Στην κλάση αυτή έχει υλοποιηθεί η **ευρετική μέθοδος** που θα χρησιμοποιηθεί στους αλγορίθμους πληροφορημένης αναζήτησης.

Η **Heuristic** είναι μια συνάρτηση που εκτιμά το κόστος μετάβασης από την τρέχουσα κατάσταση (currentState) στην τελική (goalState). Χρησιμοποιείται σε πληροφορημένους αλγορίθμους, όπως ο A*, για να καθοδηγήσει την αναζήτηση προς την πιο υποσχόμενη κατεύθυνση, μειώνοντας το πλήθος των καταστάσεων που εξετάζονται.

Η λογική της ευρετικής είναι η εξής:

1. **Λάθος τοποθετημένα blocks**: Εντοπίζονται blocks που δεν βρίσκονται στη σωστή θέση σε σχέση με τον στόχο. Το κόστος αυξάνεται κατά μία μονάδα για κάθε τέτοιο block.
2. **Ποινές για εξαρτήσεις**: Αν ένα block βρίσκεται πάνω από ένα άλλο στην τρέχουσα κατάσταση, αλλά στη στόχευση πρέπει να τοποθετηθεί κάτω από αυτό, προστίθεται ποινή. Αυτό διασφαλίζει ότι η συνάρτηση λαμβάνει υπόψη τη λανθασμένη διάταξη που περιπλέκει τη μετάβαση.
3. **Ποινές για ύψος**: Όσο πιο χαμηλά βρίσκεται ένα λάθος τοποθετημένο block, τόσο μεγαλύτερη η ποινή, επειδή απαιτούνται περισσότερες μετακινήσεις για τη διόρθωσή του.

2.5 Search

Η κλάση **Search** είναι υπεύθυνη για την υλοποίηση και εκτέλεση διαφόρων αλγορίθμων αναζήτησης με σκοπό την επίλυση του προβλήματος μετακίνησης μπλοκ από την αρχική στην τελική κατάσταση. Η βασική της λειτουργία είναι να επιλέγει τον κατάλληλο αλγόριθμο, να βρίσκει τη βέλτιστη διαδρομή λύσης, αν αυτή υπάρχει, και να παρουσιάζει το αποτέλεσμα στον χρήστη.

Η μέθοδος **SelectSearch** είναι το κεντρικό σημείο εισόδου της κλάσης. Δέχεται ως είσοδο ένα αντικείμενο τύπου **Problem**, το οποίο περιγράφει την αρχική και τελική κατάσταση, το όνομα του αλγορίθμου αναζήτησης που θα εκτελεστεί (π.χ., "depth" για αναζήτηση βάθους, "breadth" για αναζήτηση πλάτους, "astar" για A* ή "best" για Best-First), και το όνομα του αρχείου όπου θα αποθηκευτεί η διαδρομή λύσης. Ανάλογα με την επιλογή, καλείται ο αντίστοιχος αλγόριθμος.

- **Αναζήτηση βάθους (DFS)**: Εξετάζει κάθε διαδρομή όσο το δυνατόν βαθύτερα πριν επιστρέψει και δοκιμάσει άλλη διαδρομή.
- **Αναζήτηση πλάτους (BFS)**: Εξετάζει όλες τις διαδρομές στο ίδιο επίπεδο πριν προχωρήσει σε βαθύτερα επίπεδα, διασφαλίζοντας ότι θα βρει τη μικρότερη διαδρομή.
- **A***: Χρησιμοποιεί μια συνάρτηση κόστους που υπολογίζεται ως το άθροισμα του πραγματικού κόστους (g) και της ευρετικής τιμής (h). Εξασφαλίζει ότι η αναζήτηση είναι αποτελεσματική και στοχευμένη.
- **Best-First Search**: Βασίζεται αποκλειστικά στην ευρετική τιμή (h) για να καθοδηγήσει την αναζήτηση προς τον στόχο.

Καθένας από τους παραπάνω αλγόριθμους υλοποιείται ξεχωριστά ως μέθοδος μέσα στην κλάση **Search**, με κατάλληλες δομές δεδομένων, όπως ουρές, στοίβες και προτεραιότητες, ώστε να διαχειρίζεται αποτελεσματικά τις καταστάσεις. Το αποτέλεσμα της αναζήτησης επιστρέφεται ως μια λίστα από καταστάσεις, η οποία αντιπροσωπεύει τη διαδρομή λύσης, και αποθηκεύεται στο καθορισμένο αρχείο εξόδου.

2.6 PDDLParser

Η κλάση **PDDLParser** αναλαμβάνει την ανάλυση (parsing) αρχείων .pddl, τα οποία σχετίζονται με το πρόβλημα του κόσμου των μπλοκ (blocks world problem). Η κλάση αυτή δημιουργεί και επιστρέφει ένα αντικείμενο **Problem**, το οποίο περιέχει την αρχική και την τελική κατάσταση του προβλήματος.

Η διαδικασία ανάλυσης του αρχείου ξεκινά με την παρακάμπτει των δύο πρώτων γραμμών του αρχείου, οι οποίες είναι άσχετες με το πρόβλημα. Στη συνέχεια, η μέθοδος διαβάζει τα αντικείμενα από το αρχείο και τα αποθηκεύει σε μια λίστα, ενώ φιλτράρει τυχόν κενά ή άχρηστες πληροφορίες. Έπειτα, διαβάζει και αναλύει την αρχική κατάσταση, χρησιμοποιώντας τα patterns "CLEAR", "ONTABLE" και "ON". Τα μπλοκ που είναι τοποθετημένα πάνω σε άλλα μπλοκ (ON) τοποθετούνται σε μοναδικά υποστοίβες (substack), ενώ τα μπλοκ που βρίσκονται πάνω στο τραπέζι (ONTABLE) προστίθενται σε μια ξεχωριστή λίστα. Στην περίπτωση των μπλοκ που είναι "CLEAR" και "ONTABLE", η μέθοδος δημιουργεί υποστοίβες από τα κοινά μπλοκ των δύο καταστάσεων.

Στη συνέχεια, η μέθοδος συνεχίζει με την ανάλυση της τελικής κατάστασης, προσδιορίζοντας και αποθηκεύοντας τα μπλοκ που είναι "CLEAR", "ONTABLE" ή "ON" σύμφωνα με την αντίστοιχη μορφή. Η τελική κατάσταση δημιουργείται με την αναστροφή των μπλοκ στις υποστοίβες που δημιουργήθηκαν στην αρχική κατάσταση.

Η μέθοδος **parseOutputFile** είναι υπεύθυνη για την εξαγωγή των κινήσεων μεταξύ των καταστάσεων. Αν η πορεία (path) περιέχει τουλάχιστον δύο καταστάσεις, η μέθοδος γράφει τις κινήσεις που απαιτούνται για τη μετάβαση από την αρχική στην τελική κατάσταση σε ένα αρχείο εξόδου. Αυτό γίνεται με τη βοήθεια της μεθόδου **findMove**, η οποία εντοπίζει τις διαφορές μεταξύ δύο καταστάσεων και επιστρέφει την κίνηση (move) που χρειάζεται για τη μετάβαση.

Η μέθοδος **findMove** εντοπίζει την κίνηση που απαιτείται για να μετατραπεί μια κατάσταση στην επόμενη. Συγκρίνονται οι υποστοίβες των δύο καταστάσεων και προσδιορίζεται ποια μπλοκ πρέπει να μετακινηθούν. Αν οι καταστάσεις έχουν τον ίδιο αριθμό υποστοίβων, η μέθοδος εντοπίζει τη διαφορά και προσδιορίζει το μπλοκ που πρέπει να μετακινηθεί, καθώς και την αρχική και την τελική θέση του.

Η μέθοδος **filterEqualSubstacks** χρησιμοποιείται για να αφαιρέσει τις ίδιες υποστοίβες από δύο καταστάσεις, έτσι ώστε να συγκριθούν μόνο οι υποστοίβες που έχουν αλλάξει. Αυτό βοηθά στη μείωση της πολυπλοκότητας της σύγκρισης και διασφαλίζει ότι μόνο οι ουσιαστικές αλλαγές καταγράφονται.

Τέλος, η μέθοδος **getBlockByName** αναζητά ένα μπλοκ με συγκεκριμένο όνομα σε μια λίστα μπλοκ, και οι μέθοδοι **getIndexOf** και **getElementAtIndex** χρησιμοποιούνται για να βρουν την τοποθεσία ενός μπλοκ σε μια συλλογή.

2.7 Main

Η κλάση **Main** είναι υπεύθυνη για την εκκίνηση και τη διαχείριση της κύριας ροής του προγράμματος για την επίλυση του προβλήματος του κόσμου των μπλοκ. Χρησιμοποιεί το αρχείο εισόδου σε μορφή PDDL για να ορίσει το πρόβλημα και κατόπιν εφαρμόζει έναν αλγόριθμο αναζήτησης για να βρει την καλύτερη λύση. Η λύση αποθηκεύεται σε ένα αρχείο εξόδου, ενώ το πρόγραμμα εμφανίζει πληροφορίες για την αρχική και τελική κατάσταση του προβλήματος καθώς και τα επιμέρους στατιστικά απόδοσης.

1. **main(String[] args)**: Η μέθοδος **main** είναι το σημείο εκκίνησης του προγράμματος. Αν ο χρήστης εισάγει τρία ορίσματα (αλγόριθμος αναζήτησης, αρχείο εισόδου, αρχείο εξόδου), καλείται η μέθοδος **AgentActions** για την ανάλυση του προβλήματος και την επίλυσή του. Αν δεν παρέχονται τα σωστά ορίσματα, η μέθοδος **printInfo** καλείται για να εμφανίσει οδηγίες χρήσης για τον χρήστη.
2. **AgentActions(String[] args)**: Η μέθοδος αυτή αναλαμβάνει την κύρια λογική του προγράμματος. Πρώτα, αναλύει το αρχείο εισόδου και δημιουργεί τα δεδομένα του προβλήματος, όπως την αρχική και τελική κατάσταση. Στη συνέχεια, εμφανίζει τις καταστάσεις αυτές στην κονσόλα και καλεί τον καθορισμένο αλγόριθμο αναζήτησης για την επίλυση του προβλήματος με χρονικό όριο 60 δευτερολέπτων. Μετά την εκτέλεση της αναζήτησης, καταγράφει την επίλυση στο αρχείο εξόδου και εμφανίζει τον χρόνο εκτέλεσης.
3. **printInfo()**: Η μέθοδος **printInfo** εμφανίζει τις οδηγίες χρήσης του προγράμματος στην κονσόλα. Εξηγεί τον τρόπο χρήσης της γραμμής εντολών, τα απαιτούμενα ορίσματα, τις διαθέσιμες επιλογές αλγορίθμων αναζήτησης και παραδείγματα χρήσης. Επίσης, παρέχει πληροφορίες για τη μορφή του αρχείου εισόδου PDDL και τα υποστηριζόμενα χαρακτηριστικά των αλγορίθμων για βελτιστοποιημένη απόδοση.

3. Στιγμιότυπα εκτέλεσης του προγράμματος

Η εκτέλεση του προγράμματος γίνεται με την εντολή:
java -jar blocksworld.jar <αλγόριθμος> <αρχείο εισόδου> <αρχείο εξόδου>

Κατά την εκτέλεση με το που κάνει parse τα δεδομένα από το .pddl αρχείο, εμφανίζει την κατάσταση από όπου ξεκινάμε, καθώς και την κατάσταση στην οποία πρέπει να καταλήξουμε, και εμφανίζει τον χρόνο που του πήρε η διαδικασία του parsing. Στη συνέχεια ξεκινάει ένα thread SIGAlarm, το οποίο θα τερματίσει την εκτέλεση του προγράμματος αν περάσουν 60 δευτερόλεπτα και μετά ξεκινάει η αναζήτηση με τον επιλεγμένο αλγόριθμο. Με το πέρας της αναζήτησης μας εμφανίζει τον χρόνο που πήρε η εύρεση λύσης, και τερματίζει το SIGAlarm επαναφέροντας το στην αρχική του κατάταση. Στο αρχείο εξόδου που ορίστηκε, αποθηκεύεται η ακολουθία κινήσεων του μονοπατιού που βρέθηκε.

3.1 Breadth first search

Ο BFS είναι πολύ γρήγορος σε μικρά και εύκολα προβλήματα και επιστρέφει πάντα την βέλτιστη λύση. Αυτό γίνεται διότι εξερευνεί όλους τους κόμβους κατά πλάτος, οπότε η πρώτη λύση που θα βρεί θα είναι και η καλύτερη. Το πρόβλημα του είναι οι μεγάλες απαιτήσεις σε μνήμη, που τον καθιστούν ως έναν μη βιώσιμο αλγόριθμο για μεγάλα και δύσκολα προβλήματα. Παρακάτω παρατίθεται στιγμιότυπο εκτέλεσης του BFS στο πρόβλημα prodBLOCKS.4.0.pddl, καθώς και το αρχείο εξόδου που παράγεται.

```
/home/rafail/.jdks/openjdk-22.0.1/bin/java -Xss128m -javaagent:/opt/intellij-idea-community/lib/idea_rt.jar=40795:/opt/intellij-idea-community/bin -Dfile.encoding=UTF-8 -Dsun.stdout
  .encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath /home/rafail/Desktop/Computer_Science/Semester_5/AI/blocks-world-java/out/production/blocks-world-java Main breadth
  probBLOCKS-4-0.pddl out.txt
Initial State:
[C]
[A]
[B]
[D]

Final State:
[A, B, C, D]

Problem parsed in: 0.039 seconds.

Path found:
[C]
[A]
[B]
[D]

[C]
[A, B]
[D]

[A, B, C]
[D]

[A, B, C, D]

Breadth First Search needs 4 moves.
Elapsed time: 0.062 seconds.

Process finished with exit code 0
```

Εικόνα 1: Αποτελέσματα εκτέλεσης του BFS στο πρόβλημα

3.2 Depth first search

Ο DFS είναι και αυτός πολύ γρήγορος κατά την εκτέλεση του. Παρόλα αυτά σε αντίθεση με τον BFS, δεν επιστρέφει βέλτιστες λύσεις, αλλά πολύ μεγάλες λύσεις, ειδικά όσο τα προβλήματα μεγαλώνουν. Το πλεονέκτημα του σε αντίθεση με τον BFS είναι ότι δεν έχει τόσο μεγάλες απαιτήσεις σε μνήμη, καθώς είναι ένας αναδρομικός αλγόριθμος, αλλά πρέπει να είμαστε προσεκτικοί με το μέγεθος της στοίβας κλήσης, καθώς σε μεγάλα προβλήματα ο DFS μπορεί να την υπερβεί και να καταλήξουμε σε σφάλμα. Για τον λόγο αυτό κατά την εκτέλεση μεγάλων προβλημάτων με DFS χρησιμοποιούμε το flag στον JVM -Xss128m, έτσι ώστε να αυξήσουμε το μέγεθος της στοίβας κλήσης και συνεπώς το όριο αναδρομής κατά λόγου χάριν 128Mb. Παρακάτω παρατίθεται στιγμιότυπο εκτέλεσης του prodBLOCKS.7.1.pddl:

```
[A, E]
[D, C, G, F, B]

[A]
[D, C, G, F, B, E]

[D, C, G, F, B, E, A]

Depth First Search needs 15583 moves.
Elapsed time: 8.415 seconds.
```

Εικόνα 4: Αποτελέσματα εκτέλεσης του DFS στο πρόβλημα 7.1

3.3 Best first search

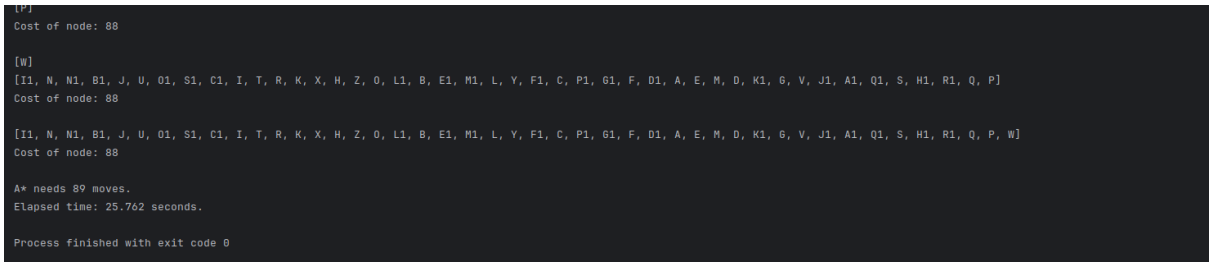
Ο best first search εκτελείται πιο γρήγορα από όλους τους υπόλοιπους αλγορίθμους. Είναι ένας “άπληστος” αλγόριθμος, ο οποίος επεκτείνει τον κόμβο με τη χαμηλότερη ευρετική τιμή. Ο αλγόριθμος αυτός είναι ο μοναδικός ο οποίος κατάφερε να λύσει όλα τα προβλήματα, μέχρι το probBlocks-60-1.pddl, σαφώς χωρίς να βρίσκει πάντοτε τις βέλτιστες λύσεις. Παρακάτω παρατίθεται στιγμιότυπο εκτέλεσης του prodBLOCKS-60-1.pddl

```
[G]
[E2, O, D, V1, Y, K, D1, Z1, S1, A1, Q1, H2, C2, M, Q, U, U1, R, B1, W1, E, L, T, I1, X1, Y1, L1, G2, F, V, H1, H, E1, A, S, G1, P, J1, J, A2, K1, C1, C, F1, N1, P1, T1, O1, M1, B,
 F2, W, I, N, Z, D2, R1, X, B2]
Cost of node: 1

[E2, O, D, V1, Y, K, D1, Z1, S1, A1, Q1, H2, C2, M, Q, U, U1, R, B1, W1, E, L, T, I1, X1, Y1, L1, G2, F, V, H1, H, E1, A, S, G1, P, J1, J, A2, K1, C1, C, F1, N1, P1, T1, O1, M1, B,
 F2, W, I, N, Z, D2, R1, X, B2, G]
Cost of node: 0

Best first needs 115 moves.
Elapsed time: 19.17 seconds.

Process finished with exit code 0
```

Εικόνα 5: Αποτελέσματα εκτέλεσης του Best First Search στο πρόβλημα 60.1

3.4 A* search

Ο A* επίσης είναι μακράν πιο γρήγορος από τους αλγορίθμους απληροφόρητης αναζήτησης, αλλά πολύ πιο αργός από τον best. Σε αντίθεση με τον best, λαμβάνει υπόψη του το κόστος για να φτάσεις σε κάποιον κόμβο, αθροιστικά με την τιμή της ευρετικής συνάρτησης στον κόμβο αυτόν, επιλέγοντας το παιδί με το χαμηλότερο κόστος. Σε πολλά από τα προβλήματα που δοκιμάστηκαν και οι δύο αλγόριθμοι πληροφορημένης αναζήτησης. Ο A* δεν βρίσκει πάντοτε την βέλτιστη λύση, αντιθέτως βρίσκει παρόμοιες και πολύ κοντινές λύσεις με τον best first search. Συμπεραίνουμε λοιπόν ότι η ευρετική συνάρτηση που επιλέχθηκε για την επίλυση του προβλήματος δεν είναι παραδεκτή. Παρακάτω παρατίθεται στιγμιότυπο εκτέλεσης του prodBLOCKS-45-0.pddl



Εικόνα 5: Αποτελέσματα εκτέλεσης του A* στο πρόβλημα 45.0

4. Χρονική απόδοση αλγορίθμων

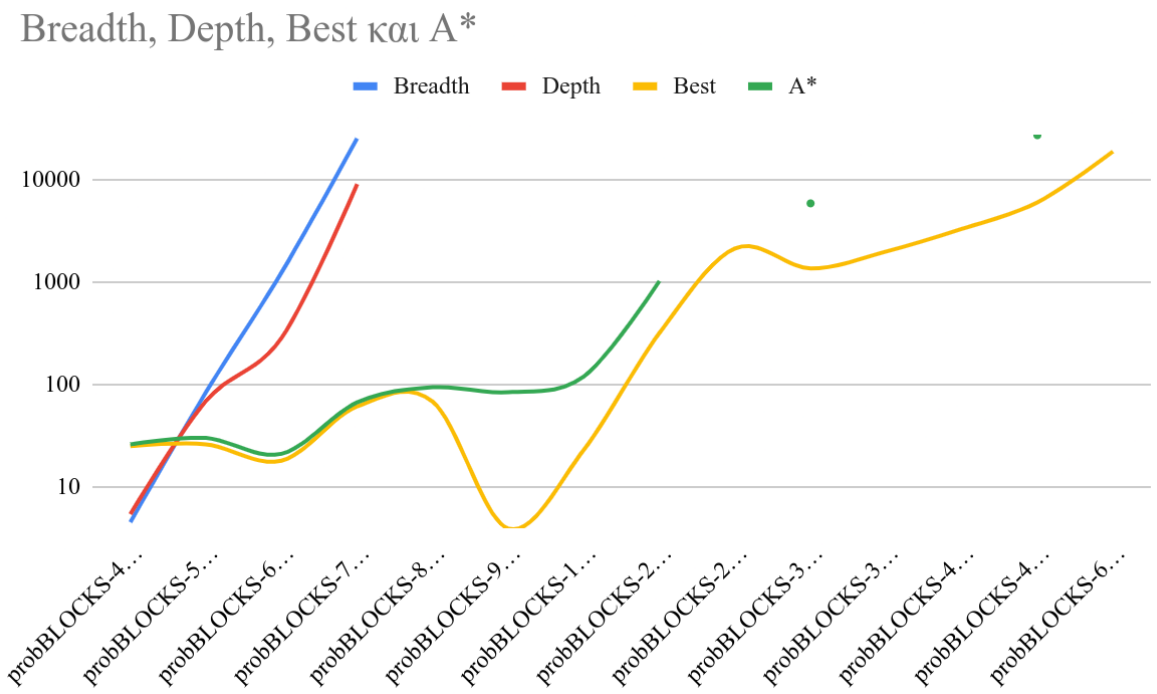
Παρακάτω παρουσιάζεται ένα πινακάκι με τον απαιτούμενο χρόνο εκτέλεσης του προγράμματος, με είσοδο τα .pddl αρχεία:

4.1 Πίνακας απόδοσης

	Depth	Breadth	Best	A*
probBLOCKS-4-0.pddl	0.0045	0.0054	0.025	0.026
probBLOCKS-5-0.pddl	0.084	0.069	0.026	0.03
probBLOCKS-6-0.pddl	1.244	0.286	0.018	0.021
probBLOCKS-7-0.pddl	25.527	9.117	0.061	0.067
probBLOCKS-8-0.pddl	-	-	0.068	0.094
probBLOCKS-9-0.pddl	-	-	0.0039	0.084
probBLOCKS-10-0.pddl	-	-	0.0239	0.12
probBLOCKS-20-0.pddl	-	-	0.3193	1.03
probBLOCKS-25-0.pddl	-	-	2.1451	3.91
probBLOCKS-30-0.pddl	–	-	1.368	5.916
probBLOCKS-35-0.pddl	-	-	2.001	-
probBLOCKS-40-0.pddl	-	-	3.323	-
probBLOCKS-45-0.pddl	-	-	6.044	27.453
probBLOCKS-60-0.pddl	-	-	19.031	-

4.2 Γράφημα απόδοσης

Παρακάτω παρουσιάζεται το γράφημα απόδοσης των 4 αλγορίθμων στους οποίους αναφερθήκαμε παραπάνω:



Συμπεράσματα

Παρατηρούμε ότι στα αρχικά προβλήματα, όπου οι χωρικές και χρονικές απαιτήσεις είναι περιορισμένες, η εύρεση λύσης γίνεται γρήγορα από όλους τους αλγορίθμους. Στα προβλήματα αυτά, το μέγεθος του δέντρου αναζήτησης είναι μικρό, επιτρέποντας ακόμη και στους αλγορίθμους μη πληροφορημένης αναζήτησης, όπως ο BFS (Breadth-First Search) και ο DFS (Depth-First Search), να αποδώσουν αποτελεσματικά. Ωστόσο, καθώς οι μεταβλητές αυξάνονται και τα προβλήματα απαιτούν μεγαλύτερο δέντρο αναζήτησης, οι περιορισμοί αυτών των αλγορίθμων γίνονται εμφανείς.

Ο BFS, αν και εξασφαλίζει την εύρεση βέλτιστης λύσης εφόσον υπάρχει, εξαντλεί γρήγορα τη διαθέσιμη μνήμη λόγω της ευρείας φύσης του δέντρου αναζήτησης. Από την άλλη, ο DFS, αν και μνημονικά αποδοτικότερος, συχνά εγκλωβίζεται σε μη παραγωγικά μονοπάτια, ειδικά σε προβλήματα με μεγάλες ή πολύπλοκες διαδρομές, οδηγώντας σε αποτυχία εύρεσης λύσης.

Ο αλγόριθμος A*, που βασίζεται σε ευρετικές συναρτήσεις, παρουσιάζει καλύτερη απόδοση σε σύγκριση με τους μη πληροφορημένους αλγορίθμους. Με την κατάλληλη ευρετική, ο A* μπορεί να περιορίσει σημαντικά τον αριθμό των καταστάσεων που εξετάζονται και να αυξήσει τις πιθανότητες εύρεσης λύσης. Παρ' όλα αυτά, σε μεγάλα προβλήματα, η επιτυχία του εξαρτάται άμεσα από την αποδοτικότητα και την παραδεκτότητα της ευρετικής συνάρτησης που χρησιμοποιείται. Ελλείψει μιας ισχυρής ευρετικής, ο A* συχνά αποτυγχάνει λόγω των αυξημένων χωρικών και χρονικών απαιτήσεων.

Αντίθετα, ο αλγόριθμος best-first search, που χρησιμοποιεί ευρετική χωρίς να απαιτεί παραδεκτότητα ή βέλτιστες λύσεις, αποδεικνύεται ιδιαίτερα αποδοτικός σε μεγάλα και πολύπλοκα προβλήματα. Παρόλο που δεν εγγυάται βέλτιστες λύσεις, καταφέρνει να βρίσκει έγκυρες λύσεις σε όλες τις περιπτώσεις και μάλιστα σε μικρό χρονικό διάστημα. Αυτό οφείλεται στο ότι η ευρετική του καθοδηγεί την αναζήτηση σε μονοπάτια που, αν και δεν είναι πάντα βέλτιστα, οδηγούν γρήγορα στον στόχο.

Συμπερασματικά, ενώ οι μη πληροφορημένοι αλγόριθμοι είναι κατάλληλοι για μικρά προβλήματα, η χρήση πληροφορημένων αλγορίθμων, όπως ο A* και ο best-first search, είναι απαραίτητη για την επίλυση μεγαλύτερων και πιο απαιτητικών προβλημάτων. Ο A* παραμένει η προτιμότερη επιλογή για βέλτιστες λύσεις, ενώ ο best-first search είναι ιδανικός για γρήγορες και πρακτικές λύσεις, ειδικά σε περιπτώσεις όπου η ταχύτητα είναι πιο σημαντική από τη βέλτιστη απόδοση.