ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΡΗΤΗΣ

Σχολή Τεχνολογικών Εφαρμογών
Τμήμα Εφαρμοσμένης Πληροφορικής και Πολυμέσων

Πτυχιακή Εργασία

Τίτλος : Ευφυές Σύστημα για το παιχνίδι Σκάκι

Ρίζος Γεώργιος (ΑΜ :2240)

Επιβλέπων Καθηγητής : Παπαδάκης Νικόλαος

ΗΡΑΚΛΕΙΟ 2011

Abstract

The purpose of this thesis was to develop an intelligent program of the classic and known to all of us chess game. The user can play chess with the computer. This program is implemented in C programming language and uses artificial intelligence to be able to find the best move of all possible moves each time. The victory goes to the computer if the user loses its king.

Σύνοψη

Σκοπός της πτυχιακής εργασίας ήταν η ανάπτυξη ενός ευφυούς προγράμματος του κλασικού και γνωστού σε όλους μας παιχνιδιού σκάκι. Ο χρήστης θα μπορεί να παίζει σκάκι με τον υπολογιστή. Το πρόγραμμα αυτό είναι υλοποιημένο σε γλώσσα προγραμματισμού C και χρησιμοποιεί τεχνητή νοημοσύνη για να μπορεί να βρίσκει την καλύτερη κίνηση από όλες τις δυνατές κινήσεις κάθε φορά. Νικητής βγαίνει ο υπολογιστής σε περίπτωση που ο χρήστης χάσει το βασιλιά του.

Πίνακας Περιεχομένων

1 Εισαγωγή

1.1 Ιστορία 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.

Ακολουθούν οι πιο σπουδαίες στιγμές στην ιστορία της ΤΝ:

Χρόνος Εξέλιξη

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 Boyer-Moore 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.

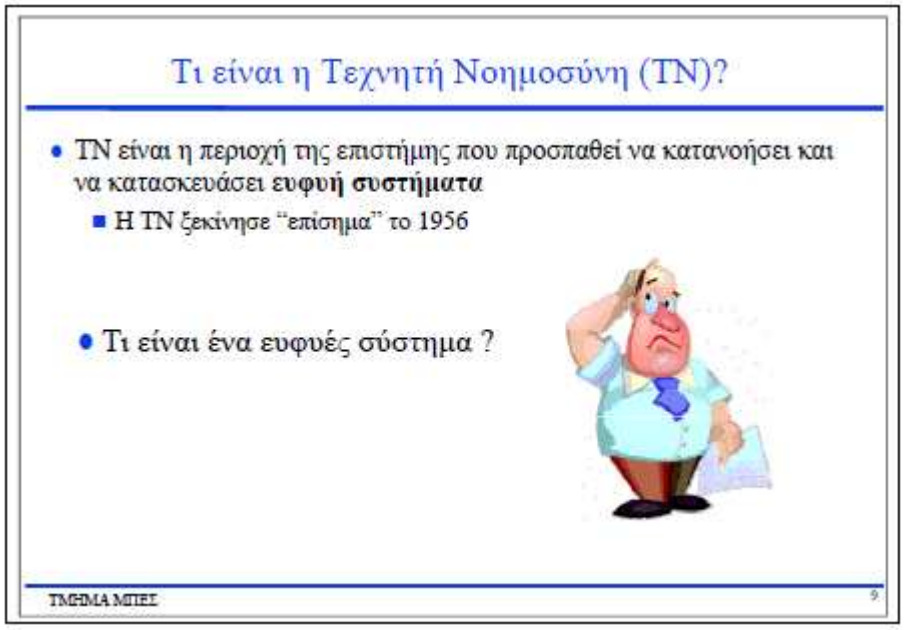
1997 Ο υπολογιστής Deep Blue της IBM κερδίζει των παγκόσμιο πρωταθλητή σκακιού Γκάρι Κασπάροφ.

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 ΤΝ.
2004 Η DARPA ξεκινά το πρόγραμμα DARPA Grand Challenge («Μεγάλη Πρόκληση DARPA»), που προκαλεί τους συμμετέχοντες να δημιουργήσουν αυτόνομα οχήματα για ένα χρηματικό βραβείο.
Το σκάκι είναι ένα από τα πιο παλιά και γνωστά σε όλο τον κόσμο παιχνίδι. Η ιστορία του παιγνιδιού αυτού χάνεται στα βάθη των αιώνων. Παιχνίδια σχετιζόμενα με το σκάκι παίζονταν ήδη από την μακρινή αρχαιότητα, στην περιοχή από την Ελλάδα και την Αίγυπτο ως και την Κίνα. Δεν έχει μέχρι σήμερα καθορισθεί ούτε ο εφευρέτης του, ούτε ο χρόνος της εμφάνισής του. Ήδη χιλιάδες εφαρμογές της ΤΝ κατακλύζουν αθόρυβα τη ζωή μας, κάνοντάς τη πιο εύκολη και ασφαλή. "Από τη ρύθμιση των φαναριών στο δρόμο και τη διαχείριση της εναέριας κυκλοφορίας μέχρι την αποκωδικοποίηση του DNA και την ιατρική, η ΤΝ έχει κυρίαρχο ρόλο" διευκρινίζει ο κ. Παναγιωτόπουλος. Και συνεχίζει: "Ένας ευφυής υπολογιστής κατευθύνει με εξαιρετική ακρίβεια και ασφάλεια εκατοντάδες αεροπλάνα ταυτόχρονα. Κανένας άνθρωπος δεν μπορεί να το κάνει αυτό. Όπου η ΤΝ αναπτύχθηκε με βάση εξειδικευμένες ανάγκες, τα αποτελέσματα που προέκυψαν ήταν θαυμαστά. Σε πολλούς τομείς οι υπολογιστές μάς έχουν ξεπεράσει. Δε διαθέτουν ωστόσο κοινό νου. Φυσικά είναι πολύ δύσκολο να πούμε αν θα τον αποκτήσουν ποτέ. Αλλά και πάλι δεν πρέπει να είναι αυτός ο στόχος μας".

Πάντως το όραμα για τη δημιουργία μιας μηχανής με ανθρώπινη σκέψη παραμένει ζωντανό. Ο Ντάγκλας Λένατ, πρωτοπόρος ερευνητής της ΤΝ, κατασκεύασε τη CYC, μια μηχανή με το μεγαλύτερο αρχείο δεδομένων στον κόσμο, την οποία προγραμμάτισαν γλωσσολόγοι, ψυχολόγοι, μαθηματικοί και άλλοι ειδικοί. Μετά από κοπιαστική εργασία 17 χρόνων η CYC έμαθε ότι τα δέντρα ευδοκιμούν συνήθως στην εξοχή. Τώρα ο Λένατ προσπαθεί να της διδάξει την έννοια της ζωής. Οι περισσότεροι επιστήμονες αμφιβάλλουν για την επιτυχία του εγχειρήματος. Εξάλλου το ερώτημα αυτό ταλανίζει τον άνθρωπο εδώ και χιλιάδες χρόνια. Θα βρει άραγε τη λύση ένας υπολογιστής;

1.2 Τι είναι τεχνητή νοημοσύνη

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.



Διαιρείται στη συμβολική τεχνητή νοημοσύνη, η οποία επιχειρεί να εξομοιώσει την ανθρώπινη νοημοσύνη αλγοριθμικά χρησιμοποιώντας σύμβολα και λογικούς κανόνες υψηλού επιπέδου, και στην υποσυμβολική τεχνητή νοημοσύνη, η οποία προσπαθεί να αναπαράγει την ανθρώπινη ευφυΐα χρησιμοποιώντας στοιχειώδη αριθμητικά μοντέλα που συνθέτουν επαγωγικά νοήμονες συμπεριφορές με τη διαδοχική αυτοοργάνωση απλούστερων δομικών συστατικών («συμπεριφορική τεχνητή νοημοσύνη»), προσομοιώνουν πραγματικές βιολογικές διαδικασίες όπως η εξέλιξη των ειδών και η λειτουργία του εγκεφάλου («υπολογιστική νοημοσύνη»), ή αποτελούν εφαρμογή στατιστικών μεθοδολογιών σε προβλήματα ΤΝ.

Η διάκριση σε συμβολικές και υποσυμβολικές προσεγγίσεις αφορά τον χαρακτήρα των χρησιμοποιούμενων εργαλείων, ενώ δεν είναι σπάνια η σύζευξη πολλαπλών προσεγγίσεων (διαφορετικών συμβολικών, υποσυμβολικών, ή ακόμα συμβολικών και υποσυμβολικών μεθόδων) κατά την προσπάθεια αντιμετώπισης ενός προβλήματος. Με βάση τον επιθυμητό επιστημονικό στόχο η ΤΝ κατηγοριοποιείται σε άλλου τύπου ευρείς τομείς, όπως επίλυση προβλημάτων, μηχανική μάθηση, ανακάλυψη γνώσης, συστήματα γνώσης κλπ. Επίσης υπάρχει επικάλυψη με συναφή επιστημονικά πεδία όπως η μηχανική όραση, η επεξεργασία φυσικής γλώσσας, η ρομποτική κλπ.

Η λογοτεχνία και ο κινηματογράφος επιστημονικής φαντασίας από τη δεκαετία του 1920 μέχρι σήμερα έχουν δώσει στο ευρύ κοινό την αίσθηση ότι η ΤΝ αφορά την προσπάθεια κατασκευής μηχανικών ανδροειδών ή αυτοσυνείδητων προγραμμάτων υπολογιστή (ισχυρή ΤΝ), επηρεάζοντας μάλιστα ακόμα και τους πρώτους ερευνητές του τομέα. Στην πραγματικότητα οι περισσότεροι επιστήμονες της τεχνητής νοημοσύνης προσπαθούν να κατασκευάσουν λογισμικό ή πλήρεις μηχανές οι οποίες να επιλύουν με αποδεκτά αποτελέσματα ρεαλιστικά υπολογιστικά προβλήματα οποιουδήποτε τύπου (ασθενής ΤΝ), αν και πολλοί πιστεύουν ότι η εξομοίωση ή η προσομοίωση της πραγματικής ευφυΐας η ισχυρή ΤΝ, πρέπει να είναι ο τελικός στόχος.

1.3 Γλώσσες προγραμματισμου και τεχνητή νοημοσύνη

1.3.1 Prolog

Η **Prolog** είναι μια γλώσσα λογικού προγραμματισμού γενικής χρήσης που κυρίως χρησιμοποιείται στον τομέα της τεχνητής νοημοσύνης. Δημιουργήθηκε στις αρχές του 1970 από τους Ρόμπερτ Κοβάλσκι και Alain Colmeraue. Το όνομα Prolog το έβγαλε ο συνεργάτης του Kowalski' Philippe Roussel και είναι συντομογραφία του γαλλικού «PROgramation et LOGique» (*«Προγραμματισμός και Λογική»*)

1.3.1.1 Εισαγωγή

Η γλώσσα προγραμματισμού Prolog δημιουργήθηκε στις αρχές της δεκαετίας του 1970 από τον Alain Colmerauer του Πανεπιστημίου της Μασσαλίας και θεωρείται ακόμα και σήμερα μια από τις πιο επιτυχημένες γλώσσες για προγραμματισμό εφαρμογών Τεχνητής Νοημοσύνης. Η Prolog έχει
εντελώς διαφορετική φιλοσοφία από τις γνωστές γλώσσες γενικού σκοπού (Pascal, Basic, C,FORTRAN, κλπ). Στις παραπάνω γλώσσες ο προγραμματισμός είναι ***διαδικαστικός*** (ή***αλγοριθμικός***), δηλαδή λέμε στη γλώσσα να εκτελέσει μια αυστηρά καθορισμένη ακολουθία ενεργειών (εντολών) προκειμένου να βρεί τη λύση στο πρόβλημά μας. Φυσικά, αυτό προϋποθέτει ότι εμείς (οι προγραμματιστές) γνωρίζουμε εκ των προτέρων τον αλγόριθμο, δηλαδή τον τρόπο που λύνεται το πρόβλημα και θα μπορούσαμε ενδεχομένως να το λύσουμε χωρίς υπολογιστή χρησιμοποιώντας μολύβι, χαρτί και πολύ χρόνο. Καταλαβαίνουμε λοιπόν ότι ο αλγοριθμικός
προγραμματισμός χρησιμοποιεί τον υπολογιστή σαν μια γρήγορη αριθμομηχανή μάλλον, παρά σαν ένα "έξυπνο" μηχάνημα. Και αυτή την έννοια έχει η χρησιμότητα του αναλυτή εφαρμογών. Είναι ο άνθρωπος που βρίσκει τους αλγορίθμους, είναι αυτός που λύνει το πρόβλημα και όχι ο υπολογιστής.
Στην προσπάθειά μας να μεταφέρουμε στον υπολογιστή και το φορτίο της ανάλυσης, οφείλεται η ανάπτυξη του ***δηλωτικού*** προγραμματισμού. Για να λύσουμε το πρόβλημα δε χρειάζεται να γνωρίζουμε εκ των προτέρων κάποιον αλγόριθμο. Αρκεί να το ορίσουμε **πλήρως** και ο υπολογιστής αναλαμβάνει να το λύσει.
Η Prolog είναι μια γλώσσα δηλωτικού προγραμματισμού. Από τη στιγμή που θα της πούμε ποιό είναι το πρόβλημα, αναλαμβάνει να κάνει πλήρη διερεύνηση και να βρεί όλες τις δυνατές λύσεις του. Αυτό το πετυχαίνει με τη βοήθεια μιας ενσωματωμένης μηχανής αναζήτησης τύπου depth-first
η οποία ενεργοποιείται αυτόματα κάθε φορά που ρωτάμε κάτι τη γλώσσα. Βέβαια η δυσκολία τώρα έχει μεταφερθεί από την εύρεση του κατάλληλου αλγορίθμου, στη δημιουργία του σωστού και πλήρους ορισμού του προβλήματος. Εκ πρώτης όψεως λοιπόν, δε μπορεί να πεί κανείς ότι

κερδίσαμε κάτι ιδιαίτερα σημαντικό με την εισαγωγή του δηλωτικού προγραμματισμού. Αποκτήσαμε όμως ένα δεύτερο εργαλείο. Τα κλασσικά προβλήματα στα οποία μπορούμε να βρούμε αλγόριθμο επίλυσης θα συνεχίσουμε να τα αντιμετωπίζουμε με τις κλασσικές αλγοριθμικές
γλώσσες. Για τα προβλήματα που είτε δε μπορούμε να βρούμε αλγόριθμο είτε κωδικοποιούνται πιο εύκολα με τον δηλωτικό προγραμματισμό, θα χρησιμοποιούμε γλώσσες σαν την Prolog.
Η Prolog λοιπόν, ως δηλωτική γλώσσα που είναι, χαρακτηρίζεται από μια "αυτενέργεια" που μερικές φορές ξενίζει τον προγραμματιστή που έχει συνηθίσει στον πλήρη έλεγχο που προσφέρουν οι αλγοριθμικές γλώσσες. Χαρακτηριστικό παράδειγμα είναι ότι σε προβλήματα με πολλές λύσεις,
το ποιά λύση θα βρεθεί πρώτη και ποιά δεύτερη είναι κάτι που το αποφασίζει η γλώσσα (ο εσωτερικός μηχανισμός αναζήτησης) κι όχι ο προγραμματιστής. Βέβαια, μόλις ο προγραμματιστής φτάσει στο σημείο να καταλαβαίνει τον τρόπο με τον οποίο "σκέφτεται" η γλώσσα, θα μπορεί να προβλέψει τη σειρά των απαντήσεων και να κατευθύνει την Prolog να επιλέξει τη σειρά που
επιθυμεί ο ίδιος, τροποποιώντας όμως τον ορισμό του προβλήματος και όχι τον μηχανισμό αναζήτησης. Από αυτό το γεγονός φαίνεται και η μεγάλη διαφορά που έχουν οι δηλωτικές γλώσσες σε σύγκριση με τις αλγοριθμικές: ο προγραμματιστής έχει ένα λογικό "εργαλείο" που για κάποιες αρχικές συνθήκες του δίνει κάποιο αποτέλεσμα. Για να πάρει το επιθυμητό αποτέλεσμα δεν επεμβαίνει στο ίδιο το εργαλείο (πρόγραμμα) αλλά αλλάζει τις αρχικές συνθήκες (δηλώσεις του προβλήματος). Το σύνολο των δηλώσεων με τις οποίες ορίζεται ένα πρόβλημα, το λέμε (καταχρηστικά) "πρόγραμμα".
Σύμφωνα με τα όσα είπαμε ως τώρα, το βασικό πρώτο βήμα για να μάθει κανείς Prolog είναι να καταλάβει τον ιδιαίτερο τρόπο με τον οποίο η γλώσσα αυτή "σκέφτεται" και ενεργεί. Αυτό θα το δούμε στη συνέχεια, βήμα-βήμα, βάσει παραδειγμάτων. Τα παραδείγματα είναι βασισμένα στο ISO Standard και έχουν δοκιμαστεί στην SWI-Prolog1. Εδώ πρέπει να εξηγήσουμε ότι για τις εκδόσεις Prolog υπάρχουν δυο standards. Υπάρχει το παλιότερο Edinburgh-standard και το πιο σύγχρονο ISOstandard.Τα δυο standards διαφέρουν κυρίως στα ονόματα των built-in συναρτήσεων που αναγνωρίζει η γλώσσα. Οι νέοι compilers ακολουθούν το ISO-standard, όμως για λόγους συμβατότητας πολλοί αναγνωρίζουν και το Edinburgh.

1.3.1.2 Αναδρομή και αναδρομικές δομές

Η αναδρομή ως τεχνική προγραμματισμού
Η *αναδρομή* (*recursion*) αποτελεί βασικό στοιχείο στον προγραμματισμό με PROLOG. 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.

Άρνηση ως αποτυχία
Μία από τις σημαντικότερες διαφορές της PROLOG από την κατηγορηματική λογική είναι η σημασία της *άρνησης* (*negation*). Στην κατηγορηματική λογική μπορεί να υπάρξουν αρνητικά γεγονότα, δηλαδή ατομικοί τύποι για τους οποίους δηλώνεται

ρητά ότι είναι ψευδείς. Στην PROLOG δεν υπάρχει η δυνατότητα αναπαράστασης τέτοιας γνώσης, παρά μόνο θετικής γνώσης. Η ύπαρξη κάποιου γεγονότος δηλώνει την αλήθεια του, ενώ θεωρείται πως ό,τι δεν αναφέρεται ρητά στη μνήμη της PROLOG, τότε δεν ισχύει. Αυτό ονομάζεται "*υπόθεση κλειστού κόσμου*" (*closed-world assumption*) και δίνει τη δυνατότητα ύπαρξης ενός είδους περιορισμένης άρνησης, της "*άρνησης ως αποτυχίας*" (*negation-as-failure*).
Σύμφωνα με τα παραπάνω, οποιαδήποτε σχέση δεν μπορεί να αποδειχθεί από το σύστημα θεωρείται ως ψευδής. Η υλοποίηση αυτή της άρνησης διαφέρει σαφώς από την κλασική έννοια της άρνησης στη λογική, αλλά χρησιμοποιήθηκε καθώς απαλλάσσει τον προγραμματιστή από την υποχρέωση του ορισμού όλης την αρνητικής πληροφορίας για κάποια εφαρμογή.
Το κατηγόρημα **not** δέχεται ως όρισμα μια οποιαδήποτε κλήση της PROLOG. Αν η κλήση μπορεί να αποδειχθεί τότε το κατηγόρημα αποτυγχάνει. Αν όχι τότε πετυχαίνει.

?- not(member(a,[a,b,c])).
no
?- not(member(d,[a,b,c])).
yes

Πολλές εκδόσεις της PROLOG παρέχουν το κατηγόρημα **\+** ως ισοδύναμο του **not** για να αποφευχθεί η σύγχυση της άρνησης της PROLOG με την άρνηση της λογικής.

1.3.1.3 Η PROLOG στην Τεχνητή Νοημοσύνη

Αναπαράσταση Προβλημάτων

Αναζήτηση πρώτα σε πλάτος (BFS)

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-gobfs(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος. Στην υλοποίηση που ακολουθεί γίνεται έλεγχος για βρόχους στις καταστάσεις που επισκέπτεται ο αλγόριθμος.

Σχεδιασμός Ενεργειών

Έστω το πρόβλημα μεταφοράς φορτίων (transportation logistics). Συγκεκριμένα, έστω δύο πόλεις, η Αθήνα και η Θεσσαλονίκη. Κάθε πόλη έχει δύο τοποθεσίες, το αεροδρόμιο και το κέντρο της. Κάθε πόλη διαθέτει ένα φορτηγό, το οποίο μπορεί να μετακινείται μεταξύ των δύο τοποθεσιών της πόλης, αλλά όχι από τη μία πόλη στην άλλη. Επιπλέον, υπάρχει ένα αεροπλάνο που μπορεί να μετακινείται μεταξύ των δύο αεροδρομίων. Τέλος, υπάρχει ένα φορτίο, το οποίο αρχικά βρίσκεται στο κέντρο της Θεσσαλονίκης. Το φορτίο μπορεί να φορτωθεί/ξεφορτωθεί στα φορτηγά των δύο πόλεων καθώς και στο αεροπλάνο.
Αρχικά το φορτίο δεν είναι φορτωμένο πουθενά και βρίσκεται στο κέντρο της Θεσσαλονίκης. Έστω επίσης ότι αρχικά το φορτηγό της Θεσσαλονίκης βρίσκεται στο κέντρο της Θεσσαλονίκης, το αεροπλάνο βρίσκεται στο αεροδρόμιο της Θεσσαλονίκης και το φορτηγό της Αθήνας βρίσκεται στο αεροδρόμιο της Αθήνας.
Το ζητούμενο είναι να μεταφερθεί το φορτίο στο κέντρο της Αθήνας.
Το σύστημα σχεδιασμού ενεργειών που ακολουθεί δέχεται περιγραφές προβλημάτων κατά STRIPS και χρησιμοποιεί μια απλή αναζήτηση κατά πλάτος για την επί-

λυσή τους. Για τη λειτουργία του απαιτεί τον ορισμό των στατικών γεγονότων του προβλήματος σαν απλά γεγονότα PROLOG, ενώ τα δυναμικά γεγονότα της αρχικής κατάστασης πρέπει να περιλαμβάνονται σε ένα γεγονός της μορφής:
initial([*fact1*, *fact2*, …]).
όπου οι όροι με πλάγια γράμματα πρέπει να αντικατασταθούν από τα δυναμικά γεγονότα που αληθεύουν στην αρχική κατάσταση. Παρόμοια, οι στόχοι του προβλήματος πρέπει να περιλαμβάνονται σε ένα γεγονός της μορφής:
goal([*fact1*, *fact2*, …]).
Τέλος τα σχήματα των ενεργειών πρέπει να οριστούν σαν κανόνες της μορφής:
operator(*name*(V1, V2, …) % το όνομα και οι παράμετροι της ενέργειας
[*prec1*, *prec2*, …], % η λίστα προϋποθέσεων
[*del1*, *del2*, …], % η λίστα διαγραφής
[*add1*, *add2*, …]):- % η λίστα προσθήκης
***fact1*, *fact2*, …. . % Στατικές προϋποθέσεις εφαρμογής**.

1.3.1.4 Ένα Απλό Πρόγραμμα

Ο καλύτερος τρόπος για να μάθει κανείς μια γλώσσα προγραμματισμού είναι να πειραματιστεί σε έναν υπολογιστή με τη βοήθεια του manual. Το έντυπο αυτό σε καμμία περίπτωση δεν αντικαθιστά το manual της γλώσσας, αλλά έχει γραφτεί με μορφή "παραδειγμάτων συνεχούς ροής" προσπαθώντας να εξομοιώσει τον διάλογο με τον υπολογιστή, έτσι ώστε αν μετά την ανάγνωση αυτού του κειμένου κάποιος βρεθεί αντιμέτωπος με έναν compiler Prolog να αισθάνεται σε αρκετά πλεονεκτική θέση. Στα παραδείγματα που ακολουθούν σημειώνουμε με έντονους χαρακτήρες τόσο τα προγράμματα, όσο και οποιαδήποτε inputs δίνουμε στη γλώσσα.
Οι παρακάτω τρείς γραμμές1 αποτελούν ένα πρόγραμμα Prolog:
man(peter).
man(jimmy).
woman(helen).
Με αυτές τις τρείς εντολές δηλώνουμε ότι ο Peter είναι άνδρας (man), όπως επίσης και ο Jimmy, ενώ η Helen είναι γυναίκα (woman). Παρατηρήστε ότι **κάθε εντολή στην Prolog τελειώνει με τελεία**. Αυτός είναι απαράβατος κανόνας. Κάθε μιά από τις τρείς παραπάνω δηλώσεις-εντολές λέγεται ***γεγονός*** ή ***fact*** ή πιό γενικά ***συνάρτηση***2 και αποτελείται από ένα ***κατηγόρημα*** (αλλιώς ***predicate*** ή ***functor*** – εδώ κατηγορήματα είναι τα man και woman) και ένα ***όρισμα*** ή ***argument*** (peter, jimmy,helen). Μια συνάρτηση μπορεί να έχει πολλά ορίσματα. Το πλήθος τους λέγεται ***arity***. Για τη συνάρτηση man το arity είναι 1 και συμβολίζουμε3: man/1
Αφού γράψουμε το παραπάνω πρόγραμμα σε έναν editor και πούμε στην Prolog να το εκτελέσει (θα εξηγήσουμε στην ενότητα 2 πώς γίνονται όλα αυτά), θα βρεθούμε μπροστά στο ***prompt*** της Prolog: ?- Τώρα η Prolog περιμένει τις ερωτήσεις μας.
Μπορούμε για παράδειγμα να επιβεβαιώσουμε ότι έχει καταλάβει τις δηλώσεις που κάναμε:
?- **man(peter).**
Yes
?- **woman(helen).**

Yes
Σε κάθε μια από τις ερωτήσεις μας η Prolog απαντάει με ένα "Yes" αφού συμφωνούν με τα facts που γνωρίζει (το πρόγραμμά μας). Μπορούμε ακόμα να ρωτήσουμε και για γεγονότα που δεν έχουν 1 Στα παραδείγματα που ακολουθούν σημειώνουμε με έντονους χαρακτήρες τόσο τα προγράμματα, όσο και οποιαδήποτε inputs δίνουμε στη γλώσσα.
2 Κάτω από την έννοια **συνάρτηση** περιλαμβάνονται και οι κανόνες, στους οποίους θα αναφερθούμε στην ενότητα 7.
3 Επειδή η Prolog επιτρέπει τον ορισμό διαφορετικών συναρτήσεων με ίδιο functor αλλά διαφορετικά arities, το πλήρες όνομα μιας
συνάρτησης περιέχει και το αντίστοιχο arity. Έτσι, οι συναρτήσεις func/1 και func/2 είναι διαφορετικές.
σχέση με τις αρχικές μας δηλώσεις:
?- **man(helen).**
No
?- **woman(jimmy).**
No
?- **woman(jenny).**
No
Η απάντηση της Prolog είναι "No". Προσέξτε ότι στην τρίτη ερώτηση (για την Jenny) η απάντηση είναι αρνητική ενώ πιό λογικοφανής θα ήταν η απάντηση: "ΔΕΝ ΞΕΡΩ", αφού στο πρόγραμμά μας δεν έχουμε πεί τίποτα για την Jenny. Βλέπουμε λοιπόν ότι η Prolog έχει **δύο *επίπεδα λογικής***: αν κάτι το γνωρίζει, απαντάει “Yes” (η πρόταση είναι TRUE). Αν κάτι δεν το γνωρίζει απαντάει "No" (η πρόταση είναι FALSE). Έτσι λοιπόν και στις δυο πρώτες ερωτήσεις η Prolog δεν απαντά "no"
βάσει κάποιου συλλογισμού της μορφής: "γνωρίζω ότι το woman(helen) είναι TRUE, άρα το man(helen) θα είναι FALSE, επομένως απαντώ No", αλλά λέει: "θέλω να ελέγξω αν το man(helen) είναι TRUE. Υπάρχει το man(helen) μέσα στις δηλώσεις του προγράμματος; Όχι. Επομένως το man(helen) είναι FALSE και απαντώ No". Αν δηλαδή το πρόγραμμά μας είχε και μια τέταρτη
δήλωση που θα έλεγε ότι το man(helen) είναι γεγονός, τότε η Prolog θα απαντούσε "Yes" και στο man(helen) και στο woman(helen). Εκ πρώτης όψεως αυτή η συμπεριφορά δεν δείχει και πολύ έξυπνη. Όμως, αναλογιστείτε ότι για την Prolog οι λέξεις man και woman που χρησιμοποιήσαμε ως predicate-names, δεν έχουν κανένα σημασιολογικό νόημα. Θα μπορούσαμε να είχαμε γράψει rpd001
και qlx422 στη θέση τους. Δε μπορεί κανείς να αποκλείσει το ενδεχόμενο για το rpd001(helen) και το qlx422(helen) να είναι και τα δυο TRUE.
Στις ερωτήσεις μας μπορούμε ακόμα να χρησιμοποιήσουμε και ***λογικούς τελεστές***. Ο τελεστής άρνησης (NOT) στην Prolog συμβολίζεται1 με “\+”.
?- **\+ man(peter).**
No
?- **\+ woman(peter).**
Yes
Εδώ η Prolog εξετάζει τη λογική τιμή της παράστασης που ακολουθεί το \+, το man(peter) ή το woman(peter), και στη συνέχεια προσδίδει την αντίθετη τιμή στην ολική παράσταση. Αυτό φαίνεται χαρακτηριστικά στο παρακάτω παράδειγμα:
?- **\+ woman(jenny).**
Yes
?- **\+ man(jenny).**

1 Στο Edinburgh standard ο τελεστής άρνησης συμβολίζεται με “not”. Πχ: “not woman(jenny)”. Στην SWI-Prolog το not δουλεύει, αλλά ως όνομα συνάρτησης. Δηλαδή πρέπει να γράψουμε “not(woman(jenny))”. Στο ISO standard ο τελεστής μετονομάστηκε σε \+ ώστε να μην μπερδεύει τους νεο-εισαγόμενους στη γλώσσα επειδή λειτουργεί λίγο διαφορετικά από ότι το σημασιολογικό NOT με το οποίο έχουμε συνηθίσει. Σε κάθε περίπτωση, η Prolog μας επιτρέπει να αλλάξουμε τα ονόματα των τελεστών ή να ορίσουμε νέους. Στην ενότητα 18 θα δούμε πώς γίνεται αυτό.
Yes
Οι απαντήσεις της Prolog είναι απόλυτα λογικές αν σκεφτούμε ότι η έννοια "jenny" είναι απολύτως άγνωστη στο πρόγραμμά μας. Αν αντί για "jenny" είχαμε τη λέξη "house", οι απαντήσεις θα ήταν ίδιες, και στην περίπτωση αυτή είναι εμφανές ότι η Prolog έχει δίκιο! Με άλλα λόγια, ο τελεστής \+ στην πραγματικότητα έχει τη σημασία του “μη αποδείξιμου”. Είναι μη αποδείξιμο ότι το woman(jenny) ισχύει; Ναι. Είναι μη αποδείξιμο ότι το man(jenny) ισχύει; Επίσης ναι.
Οι λογικοί τελεστές AND και OR συμβολίζονται στην Prolog με "**,**" και "**;**" αντίστοιχα:
?- **man(peter), man(jimmy).**
Yes
δηλαδή ισχύει **και** το man(peter) **και** το man(jimmy).
?- **man(peter),** \+ **man(jimmy).**
No
"No", επειδή το not man(jimmy) είναι FALSE.
?- **man(peter); \+ man(jimmy).**
Yes
Αλλά εδώ αρκεί που το man(peter) είναι TRUE, γιατί οι δυο συναρτήσεις είναι συνδεδεμένες με λογικό OR.
Εδώ πρέπει να πούμε ότι το "," εκτός από AND έχει και την έννοια του διαχωριστή ορισμάτων, όταν βρίσκεται μέσα στην παρένθεση μιας συνάρτησης. Πχ. **f(x,y)**. Σε αυτήν την περίπτωση το "," απλώς διαχωρίζει το x από το y. Δεν έχει την έννοια του x AND y. Συναρτήσεις με περισσότερα από ένα ορίσματα θα εξετάσουμε λεπτομερώς στην ενότητα 4. Ενα δεύτερο σημείο που θα πρέπει να τονίσουμε είναι ότι στον κώδικα του προγράμματός μας δε μπορούμε να έχουμε γεγονότα ορισμένα με τελεστές. Δηλαδή κάτι τέτοιο:
\+ man(helen).
man(peter); man(jimmy).
είναι απαράδεκτο για πρόγραμμα Prolog, αφού οι ορισμοί δεν είναι **πλήρεις**. Πχ. για την helen λέμε τι δεν είναι, αλλά δε λέμε τι είναι!1
Στα παραδείγματα που είδαμε ως τώρα, οι απαντήσεις της Prolog ήταν λακωνικές. Ενα "Yes" ή ένα 1 Δε μπορούμε να ορίσουμε κάτι με την άρνηση μιας έννοιας. Βέβαια υπάρχουν τρόποι να κάνουμε την Prolog να καταλάβει και τέτοιες ασαφείς έννοιες, αλλά απαιτούν πιο σύνθετο προγραμματισμό.

"no" ανάλογα με τη λογική τιμή της ερώτησης που κάναμε. Μπορούμε όμως να κάνουμε και ερωτήσεις που απαιτούν περισσότερες πληροφορίες. Για παράδειγμα, μπορούμε να ρωτήσουμε ποιούς άντρες αναγνωρίζει το πρόγραμμά μας:
?- **man(X).**
X = peter **;**
X = jimmy **;**
No

Με αυτή την ερώτηση συναντάμε για πρώτη φορά την έννοια της ***μεταβλητής***. Θα αναρωτηθήκατε ίσως γιατί τόσην ώρα γράφαμε τα κύρια ονόματα (peter, jimmy, helen) με μικρά γράμματα. Ο λόγος είναι ότι **κάθε λέξη με κεφαλαίο πρώτο γράμμα, η Prolog τη θεωρεί μεταβλητή**. Ετσι, το **peter** είναι μια σταθερά, ενώ το **Peter** θα ήταν μεταβλητή. Στην ερώτηση που κάναμε, το X στο man(X)
είναι επίσης μεταβλητή και η ερώτηση έχει την εξής έννοια: "*βρές τις κατάλληλες τιμές για το X, έτσι ώστε το man(X) να είναι TRUE*". Χαρακτηριστικό είναι ότι η Prolog βρίσκει **όλες** τις τιμές της μεταβλητής (ή των μεταβλητών, αν υπάρχουν πολλές) που ικανοποιούν την ερώτησή μας. Σημειώστε ότι η Prolog τυπώνει τις τιμές μια-μια. Γράφει μόνο την πρώτη, και στη συνέχεια περιμένει να
πατήσουμε ένα πλήκτρο. Αν πατήσουμε το πλήκτρο ';' θα μας τυπώσει την επόμενη τιμή που ικανοποιεί την ερώτηση που κάναμε (ή θα γράψει 'No', αν δεν υπάρχει άλλη). Αν πατήσουμε το 'Space' δεν θα ψάξει για άλλη τιμή. Έτσι (πατώντας το ;), το X παίρνει πρώτα την τιμή 'peter', μετά την τιμή 'jimmy' και στο τέλος η Prolog απαντά 'No', εννοώντας ότι δεν υπάρχουν άλλες τιμές για τη μεταβλητή X που ικανοποιούν την ερώτηση. Παρατηρήστε ότι η Prolog βρήκε πρώτα τη λύση X=peter και μετά την X=jimmy. Αυτό οφείλεται στη σειρά με την οποία είναι γραμμένα τα γεγονότα στο πρόγραμμά μας. Η σειρά λοιπόν έχει σημασία. Και η σειρά στο πρόγραμμα, αλλά και η σειρά που κάνουμε τις ερωτήσεις1:

?- **man(X); woman(X).**
X = peter ;
X = jimmy ;
X = helen ;
No
?- **woman(X); man(X).**
X = helen ;
X = peter ;
X = jimmy ;
No

Στην πρώτη ερώτηση ζητήσαμε από την Prolog να βρεί πρώτα τους άντρες, ενώ στη δεύτερη ζητήσαμε πρώτα τις γυναίκες. Βλέπουμε λοιπόν ότι οι λογικοί τελεστές στην Prolog έχουν μια τάση να μη συμπεριφέρονται απόλυτα αντιμεταθετικά, όπως έχουμε συνηθίσει στη λογική. A AND B στην Prolog σημαίνει να εξεταστεί πρώτα το A και μετά το B, ενώ στη λογική δεν υπήρχε τέτοια υπόνοια. Και μπορεί να πεί κανείς: εντάξει, τί μας πειράζει η σειρά αφού το σύνολο των λύσεων είναι αυτό που έχει τελικά σημασία; Δυστυχώς η σειρά μερικές φορές επηρρεάζει και το σύνολο των λύσεων! Προσέξτε το ακόλουθο αξιομνημόνευτο παράδειγμα:

1 Αλλά δεν έχει σημασία η σχετική θέση των συναρτήσεων, πχ. αν στον ορισμό των γεγονότων γράφαμε πρώτα το **woman(helen)** και μετά τα **man(peter)** και **man(jimmy)**, οι απαντήσεις θα ήταν ίδιες.

?- **man(X), \+ woman(X).**
X = peter ;
X = jimmy ;
No
?- **\+ woman(X), man(X).**
No

Την πρώτη φορά που συναντά κανείς αυτή την περίπτωση, απορεί (και με το δίκιο του) γιατί έχει την εντύπωση ότι η Prolog είναι μια γλώσσα που (θα έπρεπε να) συμπεριφέρεται λογικά. Όμως η Prolog είναι στην πραγματικότητα μια διαδικαστική γλώσσα που **προσποιείται** ότι συμπεριφέρεται λογικά,
χωρίς να τα καταφέρνει πάντα (όπως φάνηκε στο παράδειγμα). Μένει τώρα να κατανοήσουμε γιατί η Prolog αντέδρασε τόσο αρνητικά στη δεύτερη ερώτησή μας, ενώ υπήρχαν τιμές για το X που θα την ικανοποιούσαν. Στην προσπάθειά μας να καταλάβουμε το συλλογισμό της Prolog, της απευθύνουμε τη μισή μόνο από την επίμαχη ερώτηση:
?- **\+ woman(X).**
No
Δεν υπάρχει τιμή για το X η οποία κάνει το woman(X) FALSE, ώστε να κάνει το \+ woman(X) TRUE; Βεβαίως και υπάρχει, το X=peter. Αλλά η Prolog ποτέ δε βρίσκει αυτή την τιμή, γιατί "σκέφτεται" ως εξής:
• Θέλω να ικανοποιήσω το **\+ woman(X)**.
• Ελέγχω πρώτα το **woman(X)**.
• Ψάχνω λοιπόν όλα τα γεγονότα του προγράμματος που έχουν για όνομα συνάρτησης (κατηγόρημα) τη λέξη 'woman'.
• Βρίσκω μόνο ένα, το **woman(helen)**.
• Επομένως, το **woman(X)** γίνεται TRUE με X=helen (μοναδική τιμή).
• Αρα το **\+ woman(X)** γίνεται FALSE για X=helen και αφού το X στο **woman(X)** δε μπορεί να πάρει άλλη τιμή, ούτε το **\+ woman(X)** μπορεί να πάρει άλλη τιμή, επομένως είναι πάντα FALSE και γράφω "No".
Βλέπουμε λοιπόν ότι υπάρχουν εκφράσεις όπως η **woman(X)** που μπορούν να επιστρέψουν κάποιες τιμές για τα ορίσματά τους (***γεννήτριες τιμών*** ή ***εκφράσεις-γεννήτριες***) και εκφράσεις όπως η **\+ woman(X)** που δεν επιστρέφουν τιμές για τα ορίσματά τους. Αυτές (οι δεύτερες) μπορούν να χρησιμοποιηθούν μόνο για να ελέγξουν αν οι τιμές των ορισμάτων τους είναι αποδεκτές ή όχι (***ελεγκτές τιμών*** ή ***εκφράσεις-ελεγκτές***).
Έτσι, στην ερώτηση:
?- **man(X), \+ woman(X).**
X = peter ;
X = jimmy ;
No
η **man(X)** είναι έκφραση-γεννήτρια ενώ η **\+ woman(X)** είναι έκφραση-ελεγκτής. Η **man(X)**

επιστρέφει τις τιμές 'peter' και 'jimmy' για το X και η **\+ woman(X)** ελέγχει αν οι τιμές αυτές την ικανοποιούν. Αντίθετα, στην ερώτηση:
?- **\+ woman(X), man(X).**
No
η **\+ woman(X)** ως έκφραση-ελεγκτής που είναι, δε μπορεί να δώσει τιμές στο X. Θα μπορούσε μόνο να ελέγξει αν μια τιμή που ήδη έχει πάρει η μεταβλητή X είναι ικανοποιητική ή όχι. Όμως η X δεν έχει ακόμα καμμιά τιμή. Επομένως η **\+ woman(X)** γίνεται FALSE, και η Prolog ούτε καν ελέγχει τη **man(X)** αφού συνδέεται με τη **\+ woman(X)** με λογικό AND (και TRUE να γίνει η **man(X)** δεν αλλάζει τίποτα, αφού ολόκληρη η ερώτηση έχει πια αποτύχει).
Χαρακτηριστικό είναι ότι παρόμοια ερώτηση με OR:

?- **\+ woman(X); man(X).**
X = peter ;
X = jimmy ;
No
προχωράει πέρα από το ***fail***1 της **\+ woman(X)** και βρίσκει τιμές που ικανοποιούν τη **man(X)**.
Και ερχόμαστε με ένα τελευταίο παράδειγμα να απορρίψουμε την αρχή εκείνη της λογικής που λέει ότι NOT(NOT(A))=A, ή αλλιώς: **δυο αρνήσεις ισοδυναμούν με μια κατάφαση**.
?- **\+ \+ woman(helen).**
Yes
Απόλυτα λογικό, η απάντηση είναι ίδια όπως και στο:
?- **woman(helen).**
Yes
Ας γενικεύσουμε την ερώτηση:
?- **woman(X).**
X = helen ;
No
...και τώρα αν βάλουμε και τη διπλή άρνηση, η απάντηση θα πρέπει να είναι η ίδια:
?- \+ \+ **woman(X).**
X = _G278 ;
No
...ή όχι; Και πρώτα-πρώτα να πούμε ότι εκείνο το περίεργο "_G278" (καθώς και οποιαδήποτε 1 fail=αποτυχία. Οι έκφράσεις: "η συνάρτηση έκανε fail", "χτύπησε fail" ή "έγινε FALSE" χρησιμοποιούνται πιο συχνά μεταξύ των προγραμματιστών της Prolog από την πιο άμεση: "η συνάρτηση είναι FALSE", επειδή για τις περισσότερες συναρτήσεις δεν είναι εμφανές από την αρχή αν θα καταλήξουν σε TRUE ή FALSE. Το "έγινε fail" υπονοεί μια διαδικασία ελέγχου: αρχικά δεν ξέραμε τί είναι, στη συνέχεια έκανε fail, άρα είναι FALSE. Ενώ το "είναι FALSE" σημαίνει ότι είναι ολοφάνερα FALSE εξ' αρχής.

έκφραση που αποτελείται από ένα ***underscore***1 ακολουθούμενο από έναν τετραψήφιο κωδικό) είναι μια "***εσωτερική μεταβλητή***"2 της Prolog, ένας καταχωρητής του οποίου την ύπαρξη θα έπρεπε να αγνοούμε, κάτω από κανονικές συνθήκες. Εδώ τί ακριβώς έγινε και εμφανίστηκε ως τιμή στη μεταβλητή X; Ας δούμε πάλι πώς "σκέφτηκε" η Prolog: Κατ' αρχήν, δεν επιχείρησε την ενέργεια που θα έκανε αμέσως ένας άνθρωπος: δυο αρνήσεις στη σειρά διαγράφονται και μένει μόνο το **woman(X)**. Και σωστά δεν το έπραξε επειδή το \+ δεν είναι
ακριβώς άρνηση αλλά σημαίνει “μη αποδείξιμο”. Έτσι προσπάθησε να εξαντλήσει τα περιθώρια απόδειξης της ερώτησης που κάναμε.
Ξεκίνησε από το **woman(X)** και βρήκε κάποια τιμή για το X (X=helen). Την τιμή αυτή την αποθήκευσε προσωρινά στην εσωτερική μεταβλητή "_G278". Κατόπιν προσπάθησε να δεί αν η έκφραση **\+ woman(X)** είναι TRUE για X=helen. Απέτυχε, επομένως η _G278 αδειάζει για να δεχτεί την επόμενη τιμή του X που κάνει TRUE την \+ woman(X). Τέτοια τιμή δε βρέθηκε, και η **\+ woman(X)** γίνεται FALSE. Στη συνέχεια εξετάζεται η έκφραση **\+ \+ woman(X)** η οποία είναι
TRUE (αφού η **\+ woman(X)** βρέθηκε FALSE) για την τιμή του X που ήδη υπάρχει στην _G278.

Αλλά η _G278 έχει αδειάσει και η Prolog αφού δε βρίσκει τιμή για το X να επιστρέψει, αρκείται να μας δώσει το όνομα της εσωτερικής μεταβλητής που θά 'πρεπε να έχει την τιμή του X. Σημείωση: Αν ο παραπάνω συλλογισμός σας φάνηκε περίπλοκος ή δυσνόητος μην ανησυχείτε γιατί οι προγραμματιστές της Prolog ΣΕ ΚΑΜΜΙΑ ΠΕΡΙΠΤΩΣΗ δεν αναλύουν με αυτόν τον τρόπο τη λογική των προγραμμάτων τους. Ο συλλογισμός δόθηκε μόνο για λόγους πληρότητας και για να δείξει ότι η Prolog μερικές φορές δε συμπεριφέρεται όσο "λογικά" θα περιμέναμε, αλλά πάντοτε υπάρχει κάποιος λόγος για αυτό.
Εδώ και επτά περίπου σελίδες ασχολούμαστε με ένα πρόγραμμα τριών γραμμών το οποίο δεν έχει **ούτε μια εντολή της Prolog**. Πραγματικά, κοιτάξτε ξανά τον κώδικα:
man(peter).
man(jimmy).
woman(helen).
Αυτό είναι όλο κι όλο. Δεν έχει εντολές που τυπώνουν ή εισάγουν δεδομένα, δεν έχει μεταβλητές που παίρνουν τιμές, ακόμα και τα ονόματα στα κατηγορήματα και στα ορίσματα ήταν δικές μας επιλογές. Παρ' όλα αυτά είναι ένα πρόγραμμα Prolog. Και πώς φαίνεται ότι ένα πρόγραμμα είναι πρόγραμμα Prolog, αν όχι από τις εντολές του; Μα φυσικά από τη σύνταξη. Μπορεί να μη
χρησιμοποιήσαμε καμμιά από τις πολυάριθμες built-in συναρτήσεις της Prolog, όμως δεν αποφύγαμε να χρησιμοποιήσουμε τις παρενθέσεις και τις τελείες. Ακόμα, φροντίσαμε τα ονόματα να είναι γραμμένα με μικρά γράμματα και όχι με κεφαλαία. Όλα αυτά τα στοιχεία αποτελούν το συντακτικό της Prolog το οποίο είναι τελικά αυτό που χαρακτηρίζει τη γλώσσα. Η Prolog λοιπόν είναι μια γλώσσα που χαρακτηρίζεται από το συντακτικό της, σε αντίθεση με άλλες γλώσσες προγραμματισμού που χαρακτηρίζονται από το λεξιλόγιό τους. Οχι πως έχει καμμία ιδιαίτερη σημασία αυτό από μόνο του, αλλά μας προετοιμάζει να γνωρίσουμε μια νέα φιλοσοφία στον τρόπο προγραμματισμού, μια φιλοσοφία που θα στηρίζεται περισσότερο στη μορφή παρά στις λέξεις, περισσότερο στο "πνεύμα" παρά στο "γράμμα" του κώδικα.

1.3.1.5 Prolog O bjects

Οι σταθερές, οι μεταβλητές και οι αριθμοί που χρησιμοποιούμε στην Prolog λέγονται με ένα όνομα ***prolog objects*** και διακρίνονται σε:
α) CONSTANTS (Σταθερές)
Διακρίνονται με τη σειρά τους σε ***atoms*** και ***integers***. Τα atoms είναι ονόματα που αρχίζουν με μικρό πρώτο γράμμα ή περικλείονται σε απλά εισαγωγικά. Δεν είναι strings1 γιατί δε μπορούν να αποσυντεθούν (κάτω από κανονικές συνθήκες) σε sub-atoms.
Οι ακόλουθες λέξεις **είναι** atoms:
john jOHN 'JOHN' b019 tell_mE 'c d'
Οι ακόλουθες λέξεις **δεν είναι** atoms:
18ab John _alpha tell-me
Για τους integers δε χρειάζονται ειδικές εξηγήσεις:
0 1 2 -41 6221
Η "καθαρή" Prolog δεν έχει πραγματικούς αριθμούς (reals). Δεν τους χρειάζεται. Οι μόνοι αριθμοί που έχουν έννοια στη λογική είναι οι ακέραιοι (integers). Επειδή όμως οι προγραμματιστές είναι εξαιρετικά ευαίσθητοι σε τέτοιου είδους “καινοτομίες”, όλες

σχεδόν οι εκδόσεις της Prolog που κυκλοφορούν, διαθέτουν κάποιες βιβλιοθήκες για χειρισμό πραγματικών αριθμών (εξαγωγή ριζών,
τριγωνομετρικές συναρτήσεις, λογάριθμοι κτλ.)

β) VARIABLES (Μεταβλητές)

Μεταβλητή είναι οποιαδήποτε λέξη αρχίζει με κεφαλαίο λατινικό γράμμα ή με underscore και δεν περικλείεται σε εισαγωγικά. Παραδείγματα μεταβλητών:

Answer Tell_me WHAT _get _ _45E8

Τη σημασία της ανώνυμης μεταβλητής την έχουμε συζητήσει ήδη. Μεταβλητές που **αρχίζουν** με underscore **δεν** είναι ανώνυμες και **δεν** τυγχάνουν "ειδικού χειρισμού" από την Prolog. Προσοχή μόνο στις μεταβλητές που μετά το underscore ακολουθεί τετραψήφιος κωδικός. Έτσι συμβολίζει η Prolog τις εσωτερικές της μεταβλητές (εσωτερικοί καταχωρητές) και αν τύχει να χρησιμοποιούμε το ίδιο όνομα για μεταβλητή του προγράμματός μας, πολλά "διασκεδαστικά" γεγονότα μπορούν να
συμβούν κατά την εκτέλεσή του.

COMPOUND OBJECTS (Σύνθετα Αντικείμενα)

Μια συνάρτηση με ένα ή περισσότερα ορίσματα αποτελεί compound object, πχ:

time(23,15,20)

Τα ορίσματα ενός compound object μπορούν να είναι διαφορετικών τύπων:

date(wednesday,16,8,2006)

ή ακόμα και άλλα compound objects:

now(date(wednesday,16,8,2006),time(23,15,20))

Η Prolog χειρίζεται τα compound objects ως αυτόνομες οντότητες κι έτσι μπορεί μια μεταβλητή να πάρει τιμή ένα ολόκληρο compound object:

X = now(date(wednesday,16,8,2006),time(23,15,20)

1.3.1.6 Μερικές Εντολές

Η Prolog έχει μια πλούσια βιβλιοθήκη με συναρτήσεις.Θα εξηγήσουμε τις πιό συνηθισμένες (και χρήσιμες).

πρώτα-πρώτα η εντολή write/1:

?- **write(hello).**
hello
Yes
?- **write('hello there').**
hello there
Yes
?- **write('και Ελληνικά').**
και Ελληνικά
Yes
?- **X=hello, write(X).**
hello
X = hello ;
Yes

Απλώς γράφει την τιμή που είναι καταχωρημένη στο μοναδικό της argument, χωρίς να αλλάζει γραμμή για την επόμενη εκτύπωση (το "Yes" της Prolog είναι αυτό που αλλάζει τη γραμμή στα παραπάνω παραδείγματα). Αυτό φαίνεται καθαρότερα αν γράψουμε διαδοχικές εντολές write/1:
?- **write(one), write(two), write(three).**
onetwothree
Yes

Βέβαια, μπορούμε να προσθέσουμε κενά με τη write/1:
?- **write(one), write(' '), write(two), write(' three').**
one two three
Yes

ή να χρησιμοποιήσουμε τη συνάρτηση tab/1: tab(X) σημαίνει "*γράψε Χ κενά*":
?- **write(one), tab(4), write(two), tab(4), write(three).**
one two three
Yes

Η Γλώσσα Προγραμματισμού Prolog v.2.0
Αλλαγή γραμμής εκτελεί η συνάρτηση nl/0:
?- **write(one), nl, write(two), nl, write(three).**
one
two
three
Yes
Και ακολουθούν δυο συναρτήσεις που η χρησιμότητά τους δε φαίνεται από την πρώτη στιγμή, η true/0 που επιτυγχάνει πάντα:
?- **true.**
Yes
...και η fail/0 που αποτυγχάνει πάντα:
?- **fail.**
No

1.3.1.7 Κανόνες Prolog

Η δύναμη της Prolog έγκειται στο συνδυασμό των γεγονότων με λογικούς κανόνες για τη δημιουργία νέων γεγονότων.
Γενικά, ένας κανόνας στην Prolog έχει τη μορφή:
Head :- Body.
Ενα αριστερό μέλος (Head) συνδέεται με ένα δεξί μέλος (Body) με ένα ειδικό σύμβολο (:) που ακριβώς επειδή συνδέει ένα Head με ένα Body λέγεται **neck**. Ο κανόνας τελειώνει με τελεία. Η σημασία του κανόνα είναι: "το Head ισχύει, αν ισχύει το Body". Παρατηρούμε λοιπόν ότι το neck ισοδυναμεί με λογικό-IF. Το Body είναι η υπόθεση και το Head είναι το συμπέρασμα.
Το Body μπορεί να είναι μια ακολουθία συναρτήσεων συνδεδεμένων με λογικούς τελεστές, ενώ το Head επιτρέπεται να είναι μόνο μία συνάρτηση. Διακρίνουμε εδώ μια

αντιστοιχία των κανόνων Prolog με τα Horn clauses που γνωρίζουμε από τη Λογική. Τα γεγονότα της Prolog μπορούν να θεωρηθούν ως ειδική περίπτωση κανόνων χωρίς Body (το Head ισχύει, χωρίς καμμία υπόθεση) ή
ακόμα και ως:
fact :- true.

Ετσι ολοκληρώνουμε τη γενίκευση λέγοντας ότι κάθε εντολή ενός προγράμματος Prolog είναι της μορφής
"Head :- Body." όπου ένα από τα Head, Body μπορεί να λείπει

Προγράμματα στην Prolog

Η Prolog χρησιμοποιείται ευρέως σε εφαρμογές Τεχνητής Νοημοσύνης, οι οποίες απαιτούν εξαγωγή συμπερασμάτων από κάποια δεδομένα.
Ένα πρόγραμμα Prolog αποτελείται από ένα σύνολο
λογικών προτάσεων (clauses):
– ***Γεγονότα*** (facts).
(π.χ., ο Σωκράτης είναι άνθρωπος.)
– ***Κανόνες*** (rules).
(π.χ., κάθε άνθρωπος είναι θνητός.)

1.3.1.8 Παραδείγματα στην Prolog με εικόνες

Εφαρμογή σε χρονικό συλλογισμό

```
load_bullet(3).
shoot(fred, 5).

dead(X, T) :- dead(X, T-1).
dead(X, T) :- gun_loaded(T-1), shoot(X, T-1).
gun_loaded(T) :- load_bullet(T-1).
gun_loaded(T) :- gun_loaded(T-1), \+ shoot(X, T-1).


? dead(fred, 8).
```

Εφαρμογή σε παιχνίδια

```
evaluate_and_choose([Move|Rest], Board, Player, D, MaxMin, Record, Best):-
        make_move(Move, Board, NextBoard),
        minimax(D, NextBoard, Player, MaxMin, _MoveX, Value),
        update(Move, Value, Record, NewRecord),
        evaluate_and_choose(Rest, Board, Player, D, MaxMin, NewRecord, Best).
```

```
minimax(0, Board, Player, MaxMin, _Move, Value):-
        value(Board, Player, V), Value is V*MaxMin.
minimax(D, Board, Player, MaxMin, Move, Value):-
        D>0, D1 is D-1, MinMax is - MaxMin, opponent(Player, Opponent),
        find_all_moves(Board, Player, Moves), write(D), write(Move), nl,
        evaluate_and_choose(Moves, Board, Opponent, D1, MinMax,
          (nil,-1000), (Move,Value)).
```

Εφαρμογή σε βάσεις δεδομένων

```
course(   course_code(CS(100)),
          course_info(     title("Computing Machinery and Intelligence"),
                           credits(4), semester_offered(winter) ),
          professor_info( name("A."), surname("Turing"), title("Professor") ),
          teaching_info(
             teaching_days_time_rooms([
                  lesson(day("Monday"), from(1030), to(1159), room(123)),
                  lesson(day("Thursday"), from(1030), to(1159), room(123)),
             ]),   lab_days_time_rooms([]) ),
          other_info( obligatory(yes), prerequisites([MATH(001)]) ) ).
```

```
available_course(CompletedCourses, Semester, Course) :-  % define so that
          % all the prerequisites of Course are in CompletedCourses
          % and Course is offered in Semester
```

1.3.2 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 δακτυλογραφούμενη, ελεύθερης μορφής, 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 τρέχουσα έκδοση αυτού του προτύπου είναι αυτή του 2003, η ISO/IEC 14882:2003. Μια καινούρια έκδοση είναι υπό ανάπτυξη, γνωστή ανεπίσημα με την ονομασία C++0x.

1.3.2.1 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.

1.3.2.2 Χαρακτηριστικά

Η C++ κληρονόμησε το μεγαλύτερο μέρος της σύνταξής της C και τον προεπεξεργαστή της C.Το παρακάτω είναι ένα πρόγραμμα hello world που χρησιμοποιεί την λειτουργία stream της C++ για να γράψει ένα μήνυμα στην κύρια έξοδο.

```
# include <iostream>

int main()
{
  std::cout << "Hello, world!\n";
}
```

1.3.2.3 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).

1.3.3 Java(αντικειμενοστραφής προγραμματισμός)

1.3.3.1 Ιστορία

Στις αρχές του 1991, η *Sun* αναζητούσε το κατάλληλο εργαλείο για να αποτελέσει την πλατφόρμα ανάπτυξης λογισμικού σε μικρο-συσκευές (έξυπνες οικιακές συσκευές έως πολύπλοκα συστήματα παραγωγής γραφικών). Τα εργαλεία της εποχής ήταν γλώσσες όπως η C++ και η C. Μετά από διάφορους πειραματισμούς προέκυψε το συμπέρασμα ότι οι υπάρχουσες γλώσσες δεν μπορούσαν να καλύψουν τις ανάγκες τους. Ο "πατέρας" της Java, James Gosling, που εργαζόταν εκείνη την εποχή για την Sun, έκανε ήδη πειραματισμούς πάνω στη C++ και είχε παρουσιάσει κατά καιρούς κάποιες πειραματικές γλώσσες (C++ ++) ως πρότυπα για το νέο εργαλείο που αναζητούσαν στην *Sun*. Τελικά

μετά από λίγο καιρό κατέληξαν με μια πρόταση για το επιτελείο της εταιρίας, η οποία ήταν η γλωσσα *Oak*. Το όνομά της το πήρε από το ομώνυμο δένδρο (βελανιδιά) το οποίο ο Gosling είχε έξω από το γραφείο του και έβλεπε κάθε μέρα.

1.3.3.2 Τα χαρακτηριστικά της Java

Ένα από τα βασικά πλεονεκτήματα της Java έναντι των περισσότερων άλλων γλωσσών είναι η ανεξαρτησία του λειτουργικού συστήματος και πλατφόρμας. Τα προγράμματα που είναι γραμμένα σε *Java* τρέχουνε ακριβώς το ίδιο σε Windows, Linux, Unix και Macintosh (σύντομα θα τρέχουν και σε Playstation καθώς και σε άλλες κονσόλες παιχνιδιών) χωρίς να χρειαστεί να ξαναγίνει μεταγλώττιση (compiling) ή να αλλάξει ο πηγαίος κώδικας για κάθε διαφορετικό λειτουργικό σύστημα. Για να επιτευχθεί όμως αυτό χρειαζόταν κάποιος τρόπος έτσι ώστε τα προγράμματα γραμμένα σε Java να μπορούν να είναι «κατανοητά» από κάθε υπολογιστή ανεξάρτητα του είδους επεξεργαστή (Intel x86, IBM, Sun SPARC, Motorola) αλλά και λειτουργικού συστήματος (Windows, Unix, Linux, BSD, MacOS). Ο λόγος είναι ότι κάθε κεντρική μονάδα επεξεργασίας κατανοεί διαφορετικό κώδικα μηχανής. Ο συμβολικός (*assembly*) κώδικας που μεταφράζεται και εκτελείται σε Windows είναι διαφορετικός από αυτόν που μεταφράζεται και εκτελείται σε έναν υπολογιστή Macintosh. Η λύση δόθηκε με την ανάπτυξη της *Εικονικής Μηχανής* (*Virtual Machine* ή VM ή ΕΜ στα ελληνικά).

1.3.3.3 Η εικονική μηχανή της Java

Αφού γραφεί κάποιο πρόγραμμα σε Java, στη συνέχεια μεταγλωτίζεται μέσω του μεταγλωττιστή javac, ο οποίος παράγει έναν αριθμό από αρχεία .class (κώδικας byte ή bytecode). Ο κώδικας byte είναι η μορφή που παίρνει ο πηγαίος κώδικας της Java όταν μεταγλωττιστεί. Όταν πρόκειται να εκτελεστεί η εφαρμογή σε ένα μηχάνημα, το Java Virtual Machine που πρέπει να είναι εγκατεστημένο σε αυτό θα αναλάβει να διαβάσει τα αρχεία .class. Στη συνέχεια τα μεταφράζει σε γλώσσα μηχανής που να υποστηρίζεται από το λειτουργικό σύστημα και τον επεξεργαστή, έτσι ώστε να εκτελεστεί (να σημειωθεί εδώ ότι αυτό συμβαίνει με την παραδοσιακή Εικονική Μηχανή (Virtual Machine). Πιο σύγχρονες εφαρμογές της εικονικής Μηχανής μπορούν και μεταγλωττίζουν εκ των προτέρων τμήματα bytecode απευθείας σε κώδικα μηχανής (εγγενή κώδικα ή native code) με αποτέλεσμα να βελτιώνεται η ταχύτητα). Χωρίς αυτό δε θα ήταν δυνατή η εκτέλεση λογισμικού γραμμένου σε Java. Πρέπει να σημειωθεί ότι η JVM είναι λογισμικό που εξαρτάται από την πλατφόρμα, δηλαδή για κάθε είδος λειτουργικού συστήματος και αρχιτεκτονικής επεξεργαστή υπάρχει διαφορετική έκδοση του. Έτσι υπάρχουν διαφορετικές JVM για Windows, Linux, Unix, Macintosh, κινητά τηλέφωνα, παιχνιδομηχανές κλπ.

Οτιδήποτε θέλει να κάνει ο προγραμματιστής (ή ο χρήστης) γίνεται μέσω της εικονικής μηχανής. Αυτό βοηθάει στο να υπάρχει μεγαλύτερη ασφάλεια στο σύστημα γιατί η εικονική μηχανή είναι υπεύθυνη για την επικοινωνία χρήστη - υπολογιστή. Ο

προγραμματιστής δεν μπορεί να γράψει κώδικα ο οποίος θα έχει καταστροφικά αποτελέσματα για τον υπολογιστή γιατί η εικονική μηχανή θα τον ανιχνεύσει και δε θα επιτρέψει να εκτελεστεί. Από την άλλη μεριά ούτε ο χρήστης μπορεί να κατεβάσει «κακό» κώδικα από το δίκτυο και να τον εκτελέσει. Αυτό είναι ιδιαίτερα χρήσιμο για μεγάλα κατανεμημένα συστήματα όπου πολλοί χρήστες χρησιμοποιούν το ίδιο πρόγραμμα συγχρόνως.

1.3.3.4 Ο συλλέκτης απορριμάτων(Garbage Collector)

Ακόμα μία ιδέα που βρίσκεται πίσω από τη *Java* είναι η ύπαρξη του συλλέκτη απορριμμάτων (*Garbage Collector*). Συλλογή απορριμμάτων είναι μία κοινή ονομασία που χρησιμοποιείται στον τομέα της πληροφορικής για να δηλώσει την ελευθέρωση τμημάτων μνήμης από δεδομένα που δε χρειάζονται και δε χρησιμοποιούνται άλλο. Αυτή η απελευθέρωση μνήμης στη Java είναι αυτόματη και γίνεται μέσω του συλλέκτη απορριμμάτων. Υπεύθυνη για αυτό είναι και πάλι η εικονική μηχανή η οποία μόλις «καταλάβει» ότι ο σωρός (heap) της μνήμης (στη Java η συντριπτική πλειοψηφία των αντικειμένων αποθηκεύονται στο σωρό σε αντίθεση με τη C++ όπου αποθηκεύονται κυρίως στη στοίβα) κοντεύει να γεμίσει ενεργοποιεί το συλλέκτη απορριμμάτων. Έτσι ο προγραμματιστής δε χρειάζεται να ανησυχεί για το πότε και αν θα ελευθερώσει ένα συγκεκριμένο τμήμα της μνήμης, ούτε και για σφάλματα δεικτών. Αυτό είναι ιδιαίτερα σημαντικό γιατί είναι κοινά τα σφάλματα προγραμμάτων που οφείλονται σε λανθασμένο χειρισμό της μνήμης.

1.3.3.5 Επιδόσεις

Παρόλο που η εικονική μηχανή προσφέρει όλα αυτά (και όχι μόνο) τα πλεονεκτήματα, η *Java* αρχικά ήταν πιο αργή σε σχέση με άλλες προγραμματιστικές γλώσσες υψηλού επιπέδου (high-level) όπως η C και η C++. Εμπειρικές μετρήσεις στο παρελθόν είχαν δείξει ότι η C++ μπορούσε να είναι αρκετές φορές γρηγορότερη από την Java. Ωστόσο γίνονται προσπάθειες από τη Sun για τη βελτιστοποίηση της εικονικής μηχανής, ενώ υπάρχουν και άλλες υλοποιήσεις της εικονικής μηχανής από διάφορες εταιρίες (όπως της IBM), οι οποίες μπορεί σε κάποια σημεία να προσφέρουν καλύτερα και σε κάποια άλλα χειρότερα αποτελέσματα. Επιπλέον με την καθιέρωση των μεταγλωττιστών JIT (Just In Time), οι οποίοι μετατρέπουν τον κώδικα byte απευθείας σε γλώσσα μηχανής, η διαφορά ταχύτητας από τη C++ έχει μικρύνει κατά πολύ.

1.3.3.6 Εργαλεία ανάπτυξης

Όλα τα εργαλεία που χρειάζεται κάποιος για να γράψει Java προγράμματα έρχονται δωρεάν, από το περιβάλλον ανάπτυξης μέχρι εργαλεία *build* όπως το Apache Ant και βιβλιοθήκες, ενώ υπάρχουν πολλές διαφορετικές υλοποιήσεις της *Εικονικής Μηχανής* και του *μεταγλωττιστή* (πχ the GNU Compiler for Java) της *Java*.

2 Περίληψη

Με αφορμή τον εκπαιδευτικό χαρακτήρα του παιχνιδιού, αποφάσισα να το υλοποιήσω, με τη μόνη διαφορά ότι ο χρήστης που θα το χρησιμοποιεί θα έχει τη δυνατότητα να παίζει σκάκι με τον υπολογιστή και όχι με έναν άλλο παίκτη. Το πρόγραμμα αυτό είναι υλοποιημένο σε γλώσσα προγραμματισμού C και χρησιμοποιεί τεχνητή νοημοσύνη για να μπορεί να βρίσκει την καλύτερη κίνηση από όλες τις δυνατές κινήσεις κάθε φορά. Νικητής βγαίνει ο υπολογιστής σε περίπτωση που ο χρήστης χάσει το βασιλιά του.

2.1 Κίνητρο για τη διεξαγωγή της εργασίας

Με τους γρήγορους ρυθμούς που προστάζει η εποχή μας, ο κόσμος έχει κλειστεί στον εαυτό του και πολλές φορές ενώ έχει διάθεση για παιχνίδι, δεν μπορεί να βρει κάποιον να έχει το χρόνο και τη διάθεση να έχει το ίδιο ενδιαφέρον με αυτόν. Κίνητρο, λοιπόν, για το συγκεκριμένο θέμα πτυχιακής μου έδωσε η ανάγκη των ανθρώπων για παιχνίδι, αλλά χωρίς να χρειαστεί να ξοδέψουν χρόνο και χρήματα για να χαλαρώσουν. Οπότε, μπορεί ο καθένας θα είναι σε θέση (αρκεί να διαθέτει υπολογιστή) να μπορεί να παίξει σκάκι με τον υπολογιστή όσες φορές θέλει.
Έτσι, το πρόγραμμα αυτό, θα μπορεί όχι μόνο να διασκεδάσει, αλλά και να εκπαιδεύσει τους χρήστες του.

2.2 Σκοπός και στόχοι της εργασίας

Ο στόχος της συγκεκριμένης πτυχιακής εργασίας ήταν να μπορέσω να δημιουργήσω ένα πρόγραμμα, το οποίο θα μπορεί με αλληλεπιδραστικό τρόπο να παίξει σκάκι με οποιονδήποτε χρήστη. Ο σκοπός για τον οποίο δημιουργήθηκε είναι για να μπορεί ο οποιοσδήποτε χρήστης, με οποιαδήποτε γνώση στο σκάκι να παίξει. Γίνονται συνέχεια έλεγχοι για τα πιόνια που επιλέγει να κινήσει ο χρήστης, έτσι ώστε, ακόμα και να μη γνωρίζει να παίζει, να μαθαίνει μέσα από το πρόγραμμα αυτό.

2.3 Δομή της εργασίας

Η πτυχιακή εργασία αποτελείται από πέντε κεφάλαια και ορισμένα υποκεφάλαια. Το αμέσως επόμενο κεφάλαιο που θα αναλυθεί είναι η μεθοδολογία που χρησιμοποίησα για την ολοκλήρωση του προγράμματος. Αμέσως μετά ακολουθεί το κεφάλαιο που αφορά στον τρόπο δράσης μου για την υλοποίηση, παράδοση και παρουσίαση της εργασίας. Το τέταρτο κεφάλαιο αφορά στο κύριο μέρος της πτυχιακής και χωρίζεται σε επιμέρους κεφάλαια. Πιο συγκεκριμένα, θα αναλυθούν οι απαιτήσεις του συστήματος, ο

σχεδιασμός υλοποίησης του προγράμματος, αλλά και ο τρόπος με τον οποίο υλοποιήθηκε. Τέλος, στο πέμπτο κεφάλαιο αναφέρονται τα αποτελέσματα της εργασίας, κάποια συμπεράσματα, αλλά και μελλοντική εργασία που μπορεί να γίνει για να επεκταθεί το συγκεκριμένο πρόγραμμα.

3 Μεθοδολογία Υλοποίησης

Το πρόγραμμα που θα υλοποιήσω αφορά στην υλοποίηση του παιχνιδιού σκάκι με τεχνητή νοημοσύνη. Το αρχικό πρόβλημα που πρέπει να αντιμετωπιστεί είναι η μέθοδος που θα χρησιμοποιήσω για να μπορεί ο υπολογιστής να υπολογίζει την καλύτερη και πιο συμφέρουσα κίνηση με σκοπό να μπορέσει να κερδίσει τον αντίπαλο παίκτη.

Η χρήση της τεχνητής νοημοσύνης στο συγκεκριμένο πρόγραμμα θα γίνει για να μπορέσει το πρόγραμμα σε ένα βαθμό να μιμηθεί τις γνωστικές ικανότητες του χρήστη που έχει απέναντί του και να βγάζει αποτελέσματα και να επιλέγει κινήσεις οι οποίες θα βασίζονται στην ανθρώπινη λογική.

Οι κυριότεροι αλγόριθμοι που υπάρχουν και διάβασα μέχρι να αποφασίσω ποιον θα χρησιμοποιήσω είναι οι ακόλουθοι :

3.1 Αλγόριθμοι αναζήτητης

Οι αλγόριθμοι αναζήτησης δεν αντανακλούν πάντα τον τρόπο της ανθρώπινης σκέψης, αποτελούν όμως μία σαφή μαθηματική μεθοδολογία. Ο λόγος για την ύπαρξη τόσων αλγορίθμων είναι ότι κάθε ένας έχει διαφορετική *πολυπλοκότητα (complexity)* που τον καθιστά περισσότερο ή λιγότερο *αποδοτικό (efficient)* σε απαίτηση μνήμης ή χρόνο εκτέλεσης από άλλους αλγόριθμους κατά τη διάρκεια της αναζήτησης λύσεων σε συγκεκριμένη κατηγορία προβλημάτων.

Αλγόριθμοι Αναζήτησης

Τυφλοί

Όνομα Αλγορίθμου	Συντομογραφία	Ελληνική Ορολογία
Depth-First Search	DFS	Αναζήτηση Πρώτα σε Βάθος
Breadth-First Search	BFS	Αναζήτηση Πρώτα σε Πλάτος
Iterative Deepening	ID	Επαναληπτική Εκβάθυνση
Bi-directional Search	BiS	Αναζήτηση Διπλής Κατεύθυνσης
Branch and Bound	B&B	Επέκταση και Οριοθέτηση
Beam Search	BS	Ακτινωτή Αναζήτηση

Ευριστικοί

Hill Climbing	HC	Αναρρίχηση Λόφων
Best-First Search	BestFS	Αναζήτηση Πρώτα στο Καλύτερο
A* (A-star)	A*	A* (Άλφα Άστρο)

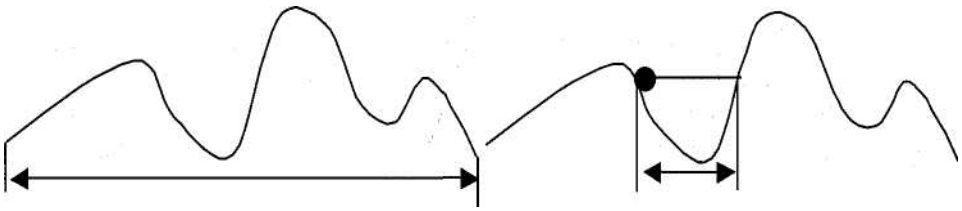
Παιχνιδιών 2 ατόμων

Minimax	Minimax	Αναζήτηση Μεγίστου-Ελαχίστου
Alpha-Beta	AB	Άλφα-Βήτα

3.2 Χώρος Αναζήτησης

Ένα πρόβλημα ορίστηκε προηγουμένως ως η τετράδα (I, G, T, S), όπου I είναι η αρχική κατάσταση, G είναι το σύνολο των τελικών καταστάσεων, T είναι το σύνολο των τελεστών μετάβασης και S είναι ο χώρος καταστάσεων. Σκοπός ενός αλγορίθμου αναζήτησης είναι να προσπαθήσει να βρει τη λύση μέσα στο χώρο καταστάσεων. Φυσιολογικά ένας τέτοιος αλγόριθμος πρέπει να εξετάσει μόνον το υποσύνολο του χώρου καταστάσεων στο οποίο ανήκει και η αρχική κατάσταση. Δοθέντος ενός προβλήματος (I, G, T, S), *χώρος αναζήτησης (search space-SP)* είναι το σύνολο όλων των καταστάσεων που είναι προσβάσιμες από την αρχική κατάσταση. Τυπικά, μία κατάσταση S ονομάζεται *προσβάσιμη (accessible)* αν υπάρχει μια ακολουθία τελεστών μετάβασης $t_1, t_2, ..., t_k \in T$ τέτοια ώστε S= $t_k(...(t_2(t_1(I)))...)$.

Η διαφορά μεταξύ του χώρου καταστάσεων και του χώρου αναζήτησης είναι λεπτή. Ο χώρος αναζήτησης είναι υποσύνολο του χώρου καταστάσεων, δηλαδή SP$\subseteq S$. Η διαφορά έγκειται στο γεγονός ότι εξ' ορισμού ο χώρος αναζήτησης εξαρτάται από την αρχική κατάσταση, ενώ ο χώρος καταστάσεων όχι. Μόνον όταν όλες οι καταστάσεις του χώρου καταστάσεων είναι προσβάσιμες από την αρχική κατάσταση, ο χώρος αναζήτησης και ο χώρος καταστάσεων ταυτίζονται. Για παράδειγμα, στο Σχήμα, η μπίλια μπορεί να βρεθεί σε οποιοδήποτε σημείο της επιφάνειας (χώρος καταστάσεων). Αν όμως αυτή αφεθεί ελεύθερη σε ένα σημείο της επιφάνειας, οι καταστάσεις οι οποίες είναι προσβάσιμες είναι αυτές που ενεργειακά είναι δυνατές (χώρος αναζήτησης).



Χώρος Καταστάσεων

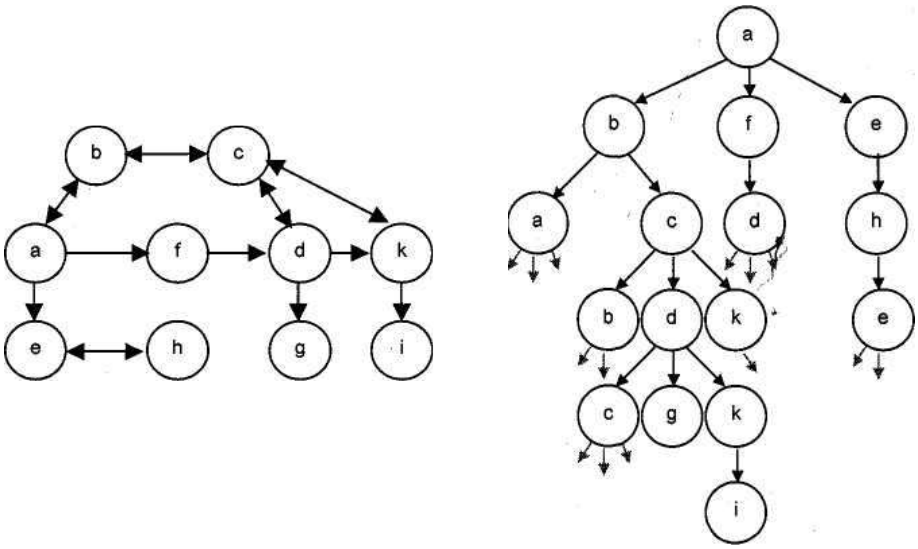
Χώρος Αναζήτησης

Όπως ο χώρος καταστάσεων, έτσι και ο χώρος αναζήτησης μπορεί να αναπαρασταθεί με γράφο. Είναι πάντα εφικτό να μετατραπεί ο γράφος σε *δένδρο αναζήτησης (search tree),* το οποίο όμως μπορεί να έχει μονοπάτια απείρου μήκους. Τα δένδρα αυτά είναι OR-δένδρα (OR-trees) γιατί σε κάθε κόμβο υπάρχουν εναλλακτικοί τελεστές που μπορεί να εφαρμοστούν. Ο Πίνακας περιέχει την αντιστοιχία μίας δομής δένδρου με το χώρο αναζήτησης ενός προβλήματος.

Τμήμα Δένδρου	**Αναπαράσταση**
Κόμβος (Node)	Κατάσταση

Ρίζα (Root)	Αρχική Κατάσταση
Φύλλο (Tip, Leaf)	Τελική Κατάσταση ή Αδιέξοδο *(Dead Node*), δηλαδή κατάσταση στην οποία δεν μπορεί να εφαρμοστεί κανένας τελεστής μετάβασης
Κλαδί (Branch)	Τελεστής Μετάβασης που μετατρέπει μια κατάσταση-Γονέα (Parent State) σε μία άλλη κατάσταση-Παιδί (Child State)
Λύση (Solution)	Μονοπάτι (Path) που ενώνει την αρχική με μία τελική κατάσταση
Επέκταση (Expansion)	Η διαδικασία παραγωγής όλων των καταστάσεων-παιδιών ενός κόμβου
Παράγοντας Διακλάδωσης (Branching Factor)	0 αριθμός των καταστάσεων-παιδιών που προκύπτουν από μία επέκταση. Επειδή δεν είναι σταθερός αριθμός, αναφέρεται και ως Μέσος Παράγοντας Διακλάδωσης (Average Branching Factor)

Στο παρακάτω Σχήμα απεικονίζεται ένας χώρος αναζήτησης και το αντίστοιχο OR-δένδρο. Οι κόμβοι *α* μέχρι k είναι καταστάσεις. Η ρίζα α είναι η αρχική κατάσταση. Τα φύλλα g και i είναι αντίστοιχα η τελική κατάσταση και ένα αδιέξοδο. Οι συνδέσεις μεταξύ κόμβων είναι τα κλαδιά του δένδρου. Μία λύση στο πρόβλημα αναζήτησης αποτελούν τα κλαδιά που συνδέουν με τη σειρά τους κόμβους a-b, b-c, c-d, d-g. *Ο παράγοντας διακλάδωσης (branching factor)* εκφράζει τον αριθμό των καταστάσεων που προκύπτουν από μία άλλη κατάσταση. Για παράδειγμα, ο παράγοντας διακλάδωσης είναι 3 για τον κόμβο α, 2 για *τον κόμβο f,* 1 για τον *e,* κ.ο.κ. Όπως προαναφέρθηκε, το αντίστοιχο δένδρο μπορεί να έχει και ακολουθία κλαδιών απείρου μήκους, όπως φαίνεται από στο παρακάτω Σχήμα.



Το πρόβλημα που παρουσιάζεται στους χώρους αναζήτησης πραγματικών προβλημάτων είναι ο γρήγορος ρυθμός με τον οποίον αναπτύσσεται το δένδρο. Έστω, για παράδειγμα, ένα τυπικό πρόβλημα μεσαίου μεγέθους, όπου το δένδρο έχει μέσο παράγοντα διακλάδωσης 10 και η λύση βρίσκεται σε βάθος 20, τότε ο χώρος αναζήτησης μπορεί να φτάσει την τάξη του 10^{20}. Το φαινόμενο αυτό της εκθετικής αύξησης του αριθμού των κόμβων του δένδρου ονομάζεται *συνδυαστική έκρηξη (combinatorial explosion).* Λόγω της συνδυαστικής έκρηξης αυξάνονται εκθετικά και οι απαιτήσεις ενός αλγορίθμου σε μνήμη και χρόνο, με αποτέλεσμα να είναι πρακτικά αδύνατον να βρεθεί λύση σε πραγματικό χρόνο.

3.3 Χαρακτηριστικά Αλγορίθμων

Ένας αλγόριθμος είναι μία αυστηρά καθορισμένη ακολουθία βημάτων-εντολών, η οποία επιδιώκει να λύσει ένα πρόβλημα [1]. Μετά την εφαρμογή κάποιου αλγορίθμου στο χώρο αναζήτησης ενός προβλήματος P=(I, G, T, S) προκύπτει το *λυμένο πρόβλημα (solved problem),* το οποίο ορίζεται ως μία τετράδα P_s=(V, A, F, G_s) όπου:

- V είναι το σύνολο των καταστάσεων που εξέτασε ο αλγόριθμος αναζήτησης,
- A είναι ο αλγόριθμος που χρησιμοποιήθηκε,
- F είναι το σύνολο των λύσεων που βρέθηκαν, και
- G_s είναι το σύνολο των τελικών καταστάσεων που εξετάστηκαν.

Η αντιπαράθεση της τετράδας P ενός προβλήματος, του χώρου αναζήτησης και της παραπάνω τετράδας P_s ενός λυμένου προβλήματος επιτρέπει κάποιες συγκρίσεις, οι οποίες με τη σειρά τους οδηγούν σε ορισμούς νέων εννοιών. Κατ' αρχήν, το σύνολο V είναι υποσύνολο του χώρου αναζήτησης, $V \subseteq SP$. Ο πληθάριθμος του V (ο αριθμός των καταστάσεων που περιέχει) είναι ένα από τα χαρακτηριστικά της αποδοτικότητας του αλγορίθμου. Όταν το σύνολο των καταστάσεων που εξετάζει ο αλγόριθμος για να βρει τις απαιτούμενες λύσεις είναι ίσο με το χώρο αναζήτησης, δηλαδή V=SP, τότε ο αλγόριθμος ονομάζεται *εξαντλητικός (exhaustive).*

Ένας αλγόριθμος δε λύνει πάντα κάποιο πρόβλημα, έστω και αν υπάρχει κάποια λύση. Τότε τα σύνολα G_S και F είναι κενά. Ένας αλγόριθμος αναζήτησης ονομάζεται *πλήρης (complete)* αν εγγυάται ότι θα βρει μία λύση για οποιαδήποτε τελική κατάσταση, αν τέτοια λύση υπάρχει. Σε αντίθετη περίπτωση, ο αλγόριθμος ονομάζεται *ατελής (incomplete).*

Η πληρότητα ενός αλγορίθμου μπορεί να αποδειχθεί μόνο με μαθηματικές μεθόδους. Ωστόσο για ορισμένες περιπτώσεις είναι σίγουρο πως αν ο αλγόριθμος δε βρει λύση, τότε οπωσδήποτε δεν υπάρχει λύση στο πρόβλημα, όπως για παράδειγμα στην

περίπτωση που ένας αλγόριθμος είναι εξαντλητικός. Η μη εύρεση λύσης από έναν πλήρη αλγόριθμο είναι εξίσου σημαντική με την εύρεση λύσης στο ίδιο πρόβλημα.

Μερικά προβλήματα έχουν διατεταγμένο το σύνολο των τελικών καταστάσεων, σύμφωνα με τη σημαντικότητα - αξία της κάθε τελικής κατάστασης. Μία λύση ονομάζεται *βέλτιστη (optimal)* αν οδηγεί στην καλύτερη, σύμφωνα με τη διάταξη, τελική κατάσταση. Όταν δεν υπάρχει διάταξη, μία λύση ονομάζεται *βέλτιστη* αν είναι η συντομότερη *(shortest),* δηλαδή αν περιέχει το μικρότερο αριθμό τελεστών μετάβασης που οδηγούν σε κάποια τελική κατάσταση. Διαφορετικοί αλγόριθμοι βρίσκουν λύσεις διαφορετικής ποιότητας. Ένας αλγόριθμος αναζήτησης καλείται *αποδεκτός (admissible)* αν εγγυάται ότι θα βρει τη *βέλτιστη λύση,* αν μια τέτοια λύση υπάρχει.

3.4 Διαδικασία Επιλογής ενός Αλγορίθμου Αναζήτησης

Δοθέντος ενός προβλήματος, είναι σημαντικό να επιλεγεί ο καταλληλότερος αλγόριθμος για την επίλυση του. Η επιλογή αυτή γίνεται βάσει κάποιων κριτηρίων τα οποία όμως δε μπορούν να τυποποιηθούν. Η επιλογή εξαρτάται κυρίως από τη φύση του προβλήματος και σε μεγάλο βαθμό από τους συμβιβασμούς που πρέπει να γίνουν. Για παράδειγμα, μπορεί κάποιος να είναι διατεθειμένος να θυσιάσει την αποδοτικότητα σε χώρο ή χρόνο προς χάριν της καλύτερης λύσης ή να θυσιάσει την πληρότητα, επιδιώκοντας τη γρήγορη εύρεση οποιασδήποτε λύσης. Εν συντομία, η επιλογή ενός αλγορίθμου βασίζεται στα εξής κριτήρια :

- τον αριθμό των καταστάσεων που αυτός επισκέπτεται
- τη δυνατότητα εύρεσης λύσεων εφόσον αυτές υπάρχουν
- τον αριθμό των λύσεων
- την ποιότητα των λύσεων
- την αποδοτικότητα του σε χρόνο
- την αποδοτικότητα του σε χώρο (μνήμη)
- την ευκολία υλοποίησης του

Στα παραπάνω κριτήρια εντάσσεται και η έννοια του *κλαδέματος* ή *αποκοπής καταστάσεων (pruning)* του χώρου αναζήτησης. Αποκοπή είναι η διαδικασία κατά την οποία ο αλγόριθμος απορρίπτει, κάτω από ορισμένες συνθήκες, κάποιες καταστάσεις και μαζί με αυτές όλο το υποδένδρο που εκτυλίσσεται κάτω από τις καταστάσεις αυτές. Η αποκοπή μπορεί να βασίζεται είτε σε αντικειμενικά κριτήρια όταν είναι σίγουρο ότι δεν υπάρχει λύση από εκεί και κάτω ή σε αυθαίρετα, ευριστικά όπως αναφέρεται στη συνέχεια, κριτήρια. Για παράδειγμα, η συνέχιση της αναζήτησης από μία κατάσταση μπορεί να οδηγήσει σε λύση, αλλά το κόστος υπολογισμού της να είναι τόσο μεγάλο, ώστε να αποφασιστεί να κλαδευτεί ο χώρος αναζήτησης που συνδέεται με αυτήν την κατάσταση.

Οι αλγόριθμοι αναζήτησης χωρίζονται σε δύο μεγάλες κατηγορίες, τους *τυφλούς* και τους *ευριστικούς*. Οι τυφλοί διατάσσουν το μέτωπο της αναζήτησης βάσει της χρονικής δημιουργίας των νέων καταστάσεων. Οι ευριστικοί αλγόριθμοι όμως, διατάσσουν το μέτωπο αναζήτησης σύμφωνα με κάποια κριτήρια που αξιολογούν τις νέες καταστάσεις ως "καλύτερες" ή "χειρότερες" από κάποιες άλλες. Για παράδειγμα, ένα κριτήριο για το αν μία κατάσταση είναι "καλύτερη" από κάποια άλλη είναι η εκτιμούμενη απόσταση της από την τελική κατάσταση.

3.5 Τυφλοί αλγόριθμοι

3.5.1 Αναζήτηση Πρώτα σε Βάθος (Depth First Search - DFS)

Όπως φανερώνει η ονομασία του αλγορίθμου *πρώτα σε βάθος (Depth-First Search-DFS),* η αναζήτηση επιλέγει προς επέκταση την κατάσταση που βρίσκεται πιο βαθιά στο δένδρο. Στην περίπτωση που υπάρχουν περισσότερες από μία καταστάσεις στο ίδιο βάθος ο *DFS* επιλέγει τυχαία μία από αυτές και για ευκολία θεωρείται ότι επιλέγεται η αριστερότερη. Ο αλγόριθμος *DFS* περιγράφεται ως εξής:

Βάλε την αρχική κατάσταση στο μέτωπο της αναζήτησης.

1. Αν το μέτωπο της αναζήτησης είναι κενό τότε σταμάτησε.
2. Βγάλε την πρώτη κατάσταση από το μέτωπο της αναζήτησης.
3. Αν είναι η κατάσταση μέλος του κλειστού συνόλου τότε πήγαινε στο δεύτερο βήμα.
4. Αν η κατάσταση είναι μία από τις τελικές, τότε ανέφερε τη λύση.
5. Αν θέλεις και άλλες λύσεις πήγαινε στο βήμα 2. Αλλιώς σταμάτησε.
6. Εφάρμοσε τους τελεστές μετάβασης για να βρεις τις καταστάσεις-παιδιά.
7. Βάλε τις καταστάσεις-παιδιά στην αρχή του μετώπου της αναζήτησης.
8. Βάλε την κατάσταση-γονέα στο κλειστό σύνολο.
9. Πήγαινε στο βήμα 2.

Το μέτωπο της αναζήτησης είναι μια δομή στοίβας (Stack LIFO - Last In First Out), δηλαδή οι νέες καταστάσεις τοποθετούνται πάντα στην αρχή της στοίβας και η αναζήτηση συνεχίζεται με μία από αυτές. Το βασικό *μειονέκτημα* του αλγορίθμου αναζήτησης πρώτα σε βάθος (Depth - First Search - DFS) είναι ότι δεν εγγυάται ότι η πρώτη λύση που θα βρεθεί είναι η βέλτιστη (μονοπάτι με το μικρότερο μήκος ή με μικρότερο κόστος). Επίσης, αν δεν υπάρχει έλεγχος βρόχων ή αν ο χώρος αναζήτησης είναι μη πεπερασμένος, ο αλγόριθμος DFS μπορεί να μπλεχτεί σε κλαδιά μεγάλου μήκους ή ατέρμονα κλαδιά του δένδρου (κλαδιά με άπειρο μήκος). Ο αλγόριθμος αυτός μπορεί να μη βρει ποτέ μια τελική κατάσταση αν και μπορεί να περάσει από

πολύ κοντά της. Συνεπώς, ο DFS εν γένει θεωρείται ατελής. Στις περιπτώσεις όμως που ο χώρος αναζήτησης είναι πεπερασμένος και χρησιμοποιείται κλειστό σύνολο, ο αλγόριθμος αναζήτησης πρώτα σε βάθος θα βρει λύση, εάν μια τέτοια υπάρχει.

3.5.2 Αναζήτηση Πρώτα σε Πλάτος (Breadth First Search - BFS)

Ο αλγόριθμος αναζήτησης πρώτα σε πλάτος (Breadth First Search - BFS) εξετάζει πρώτα τις καταστάσεις που βρίσκονται στο ίδιο βάθος και μόνον όταν τις εξετάσει όλες συνεχίζει στην επέκταση καταστάσεων στο αμέσως επόμενο επίπεδο. Η αναζή¬τηση μοιάζει να προχωρά κατά κύματα, όπου το πρώτο κύμα είναι οι καταστάσεις του πρώτου επιπέδου του δένδρου, το δεύτερο κύμα οι καταστάσεις του δεύτερου επιπέδου, κ.ο.κ.
Περιγραφικά ο αλγόριθμος είναι:

1. Βάλε την αρχική κατάσταση στο μέτωπο της αναζήτησης.
2. Αν το μέτωπο της αναζήτησης είναι κενό τότε σταμάτησε.
3. Βγάλε την πρώτη κατάσταση από το μέτωπο της αναζήτησης.
4. Αν είναι η κατάσταση μέλος του κλειστού συνόλου τότε πήγαινε στο βήμα 2.
5. Αν η κατάσταση είναι μία τελική τότε ανέφερε τη λύση.
6. Αν θέλεις και άλλες λύσεις πήγαινε στο βήμα 2. Αλλιώς σταμάτησε.
7. Εφάρμοσε τους τελεστές μεταφοράς για να βρεις τις καταστάσεις-παιδιά.
8. Βάλε τις καταστάσεις-παιδιά στο τέλος του μετώπου της αναζήτησης.
9. Βάλε την κατάσταση-γονέα στο κλειστό σύνολο.
10. Πήγαινε στο βήμα 2.

Η διαφορά του αλγορίθμου BFS (Breadth First Search) από τον αλγόριθμο DFS (Depth First Search) εντοπίζεται στην περιγραφή του καθενός. Και συγκεκριμένα σε μία μόνο λέξη, "τέλος" αντί "αρχή". Εδώ το μέτωπο της αναζήτησης είναι μια δομή ουράς (Queue FIFO, δηλαδή First In First Out) και όχι στοίβας, ποτέ δεν επεκτείνεται μία κατάσταση αν δεν επεκταθούν πρώτα όλες οι κατα¬στάσεις που βρίσκονται σε μικρότερο βάθος, γιατί απλά οι τελευταίες μπήκαν στο μέτωπο της αναζήτησης νωρίτερα.
Βασικό πλεονέκτημα του αλγορίθμου αναζήτησης πρώτα σε πλάτος (BFS - Breadth First Search) είναι ότι βρίσκει πάντα την καλύτερη λύση (μικρότερη σε μήκος). Είναι πλήρης, δηλαδή θα βρει λύση σε κάποιο πρόβλημα, αν τέτοια υπάρχει. Σε μία ακραία περίπτωση, όταν δηλαδή ένα δένδρο αναζήτησης έχει άπειρο πλάτος, (υπάρχουν άπειροι τελεστές που εφαρμόζονται σε μία κατάσταση), τότε ο BFS έχει πρόβλημα στην ανάπτυξη του δένδρου αναζήτησης. Τέτοιου είδους προβλήματα όμως είναι σπάνια και για αυτό ο BFS (Breadth First Search) θεωρείται δίκαια πλήρης.

3.5.3 Αλγόριθμος Επαναληπτικής Εκβάθυνσης (Iterative Deepening - ID)

Ο αλγόριθμος επαναληπτικής εκβάθυνσης (Iterative Deepening - ID) συνδυάζει με τον καλύτερο τρόπο τους DFS (Depth First Search) και BFS (Breadth First Search). Ο ID είναι

κατά βάση DFS, αλλά η αναζή¬τηση γίνεται σε στάδια. Κάθε στάδιο είναι η εφαρμογή του DFS σε ορισμένο μόνο βάθος, ανεξάρτητα από το συνολικό βάθος του δένδρου αναζήτησης. Όταν η αναζή¬τηση στο προκαθορισμένο βάθος ολοκληρωθεί, ο DFS επαναλαμβάνεται με την ίδια αρχική κατάσταση, αλλά σε μεγαλύτερο βάθος. Αναλυτικότερα η περιγραφή του ID είναι:

1. Όρισε το αρχικό βάθος αναζήτησης (συνήθως 1).
2. Εφάρμοσε τον αλγόριθμο DFS μέχρι αυτό το βάθος αναζήτησης.
3. Αν έχεις βρει λύση σταμάτησε.
4. Αύξησε το βάθος αναζήτησης (συνήθως κατά 1).
5. Πήγαινε στο βήμα 2.

Σοβαρό μειονέκτημα του Αλγόριθμου Επαναληπτικής Εκβάθυνσης (ID) είναι ότι όταν αρχίζει ο αλγόριθμος αναζήτησης πρώτα σε βάθος (DFS) με διαφορετικό βάθος δε θυμάται τίποτα από την προηγούμενη αναζήτηση. Αυτό όμως αντισταθμίζεται από τα πλεονεκτήματα του ID, δηλαδή ότι δεν κινδυνεύει να χαθεί σε κάποιο κλαδί με άπειρο μήκος, αφού το βάθος αναζήτησης είναι προκαθορισμένο. Άρα ο ID είναι πλήρης. Επίσης, αν το βάθος αυξάνεται κατά 1 σε κάθε κύκλο και ο ID βρει λύση, τότε αυτή η λύση θα είναι η καλύτερη. Και αυτό γιατί αν υπήρχε άλλη καλύτερη λύση αυτή θα βρισκόταν σε προηγούμενο κύκλο αναζήτησης.

3.5.4 Αναζήτηση Διπλής Κατεύθυνσης (Bidirectional Search - BiS)

Η ιδέα της αναζήτησης διπλής κατεύθυνσης (Biodirectional Search - BiS) πηγάζει από τη δυνατότητα του παραλληλισμού (parallelism) στα υπολογιστικά συστήματα. Αν υπάρχουν για παράδειγμα δύο επεξεργαστές, ο ένας μπορεί να αρχίσει να αναζητά τη λύση από την αρχική προς μία τελική κατάσταση και ο άλλος από μία τελική προς την αρχική, μοιράζοντας τη δουλειά και (θεωρητικά) μειώνοντας το χρόνο αναζήτη¬σης [1]. Ο BiS έχει δύο προϋποθέσεις κάτω από τις οποίες μπορεί να εφαρμοστεί:

- οι τελεστές μετάβασης σε ένα πρόβλημα είναι αντιστρέψιμοι (reversible), δηλαδή αν για κάθε τελεστή που εφαρμόζεται σε μια κατάσταση S1 και προκύπτει η κα¬τάσταση S2, υπάρχει ο αντίστροφος τελεστής που εφαρμόζεται στην S2 και προ¬κύπτει η S1, και
- είναι πλήρως γνωστή η τελική κατάσταση (δηλαδή δεν είναι γνωστά μόνον μερι-κά χαρακτηριστικά της).

Στα μειονεκτήματα του BiS (Biodirectional Search) συγκαταλέγεται το γεγονός ότι υπάρχει επιπλέον κόστος που οφείλεται στην επικοινωνία μεταξύ των δύο αναζητήσεων. Η αναζήτηση προχωρά και από τις δύο πλευρές ταυτόχρονα με την ελπίδα ότι κάποια από τις κα¬ταστάσεις που θα επεκταθεί θα είναι κοινή και στις δύο πλευρές. Αυτό προϋποθέτει ότι ο αλγόριθμος πρέπει να θυμάται ποιες καταστάσεις έχει επισκεφθεί και από τις δύο πλευρές έτσι ώστε να αναγνωρίσει την ύπαρξη κοινής κατάστασης.

Αν βρεθεί κοινή κατάσταση και από τις δύο αναζητήσεις, η λύση είναι η ένωση των μονοπατιών από την κοινή κατάσταση έως την αρχική και έως την τελική κατά-

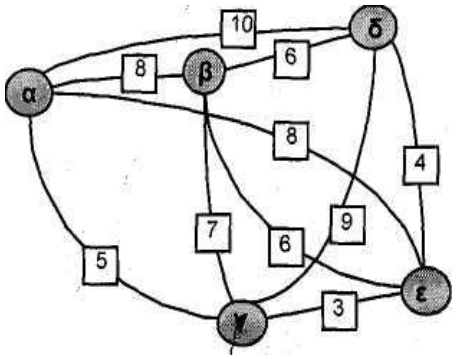
σταση. Δεν υπάρχει περιορισμός όσον αφορά τον αλγόριθμο με τον οποίο γίνονται οι αναζητήσεις, μάλιστα μπορεί να είναι και διαφορετικοί για κάθε νήμα της αναζήτησης (thread). Πρέπει να σημειωθεί, ωστόσο, ότι ο BiS δε μειώνει την πολυπλοκότητα, αλλά κληρονομεί την πολυπλοκότητα των αλγορίθμων τους οποίους χρησιμοποιεί.

3.5.5 Επέκταση και Οριοθέτηση (Branch and Bound – B&B)

Στις προηγούμενες ενότητες, καλύτερες λύσεις θεωρούνται όπως διαπιστώνεται, αυτές που περιείχαν λιγότερους σε αριθμό τελεστές. Αυτό υπονοεί ότι όλοι οι τελεστές μετάβασης έχουν το ίδιο κόστος, κάτι το οποίο δε συμβαίνει σε όλα τα προβλήματα. Για παρά¬δειγμα, το κόστος μετακίνησης από μία πόλη σε μία άλλη εξαρτάται από την, από-σταση μεταξύ των δύο πόλεων. Αν ένας αλγόριθμος αναζητάει τη συντομότερη διαδρομή από μία αρχική πόλη σε μία άλλη πόλη-προορισμό, θα πρέπει να ληφθεί υπόψη και το συνολικό κόστος μετακίνησης. Αυτό εκφράζεται από τη συνολική απόσταση μεταξύ της αρχικής πόλης και της πόλης προορισμού, δηλαδή από το ά-θροισμα του επιμέρους κόστους κάθε τελεστή. Φυσικά, ένας απλός αλλά αφελής τρόπος θα ήταν να εφαρμοστεί ένας αλγόριθμος που βρίσκει όλες τις λύσεις, να τις αξιολογεί και να καταλήγει σε αυτή με το μικρότερο συνολικά κόστος. Κάτι τέτοιο όμως θα απαιτούσε την εξαντλητική αναζήτηση σε όλο το χώρο αναζήτησης. Μία καλύτερη προσέγγιση στο πρόβλημα θα ήταν κατά τη διάρκεια της αναζήτησης να μην εξετάζονται καταστάσεις οι οποίες είναι σίγουρο ότι δε θα οδηγήσουν στην κα¬λύτερη λύση. Έτσι, κλαδεύεται μεγάλο (πιθανά) τμήμα του χώρου αναζήτησης.

Ο αλγόριθμος επέκτασης και οριοθέτησης (Branch and Bound – B&B) εφαρμόζε-ται σε προβλήματα όπου αναζητείται η βέλτιστη λύση, δηλαδή εκείνη με το ελάχιστο κόστος όπως ο χρονοπρογραμματισμός εργασιών. Σε αυτήν την κατηγορία των προβλημάτων οι τελεστές μετάβασης δεν είναι όλοι του ίδιου κόστους. Επιπλέον το κόστος είναι μία μονότονη συνάρτηση, δηλαδή αυξάνει συνεχώς καθώς εξελίσσεται η αναζήτηση. Αυτό οφείλεται στο ότι όλοι οι τελεστές έχουν πάντα θετικό κόστος.

Η λειτουργία του B&B επιδεικνύεται μέσω ενός προβλήματος. Το πρόβλημα παριστά¬νεται με ένα γράφο που αποτελείται από N κόμβους οι οποίοι αντιπροσωπεύουν αντίστοιχα N πόλεις. Κάποιος πρέπει να ξεκινήσει από μία πόλη, να επισκεφτεί μία φορά την κάθε πόλη και να επιστρέψει στην αρχική πόλη με το λιγότερο συνολικό κόστος, διανύοντας δηλαδή την ελάχιστη δυνατή απόσταση. Για απλότητα όλοι οι κόμβοι ενώνονται μεταξύ τους (πλήρης γράφος) και η αρχή είναι δεδομένη, για πα¬ράδειγμα η πόλη α (Σχήμα 4.3). Ο χώρος αναζήτησης του προβλήματος απεικονίζε¬ται με γράφο στο Σχήμα. Αν ο γράφος αυτός αναπτυσσόταν σε δένδρο, το δένδρο θα είχε 3 επιλογές από την αρχική κατάσταση, 3 επιλογές από κάθε κατάσταση που προκύπτει από την αρχική κ.ο.κ., δηλαδή 3! διαφορετικές λύσεις με το αντίστοι¬χο κόστος, από τις οποίες μία θα είναι η βέλτιστη.



Το πρόβλημα ανήκει στην κατηγορία προβλημάτων με λύση μη-πολυωνυμικού χρόνου (NP-Complete). Η εξαντλητική εξέταση (N-1)! μονοπατιών για N πόλεις είναι ένας πολύ μεγάλος αριθμός ακόμα και για μικρό αριθμό πόλεων. Για παράδειγμα, αν N=20, τότε 19!= 1,216351003088χ1017, που σημαίνει ότι ακόμη και ο πιο γρήγορος υπολογιστής που βρίσκει 1 εκατομμύριο λύσεις το δευτερόλεπτο θα χρειαζόταν 385 χρόνια για να λύσει το πρόβλημα.

Τα προβλήματα ελαχιστοποίησης κόστους έχουν πολλές εφαρμογές. Για παρά-δειγμα, η αυτόματη συναρμολόγηση ψηφιακών πλακετών, όπου ένας μηχανικός βρα-χίονας μεταφέρει τα ολοκληρωμένα κυκλώματα και τα εναποθέτει στη σωστή τους θέση, είναι πρόβλημα ελαχιστοποίησης κόστους. Μία μη βέλτιστη διαδρομή του βραχίονα μπορεί να έχει τεράστιο χρονικό και κατά συνέπεια οικονομικό κόστος σε μία εταιρία συναρμολό¬γησης.

Η λειτουργία του B&B βασίζεται στο κλάδεμα καταστάσεων (pruning) και κατά συνέπεια στην ελάττωση του χώρου αναζήτησης. Όταν το πρόβλημα είναι τέτοιο ώστε να εγγυάται ότι η αναζήτηση σε ένα υποδένδρο, ως προς τη σημαντικότητα της λύσης, δε θα είναι χρήσιμη, τότε το υποδένδρο αυτό κλαδεύεται, δηλαδή η αναζήτη¬ση δεν προχωρά σε αυτό. Για παράδειγμα, έστω ότι σε ένα πρόβλημα έχει βρεθεί μία λύση, της οποίας το κόστος είναι 26, ενώ καθώς η αναζήτηση συνεχίζεται, σε μία τρέχουσα κατάσταση η διαδρομή κοστολογείται μέχρι στιγμής ως 27. Τότε δεν υπάρ¬χει λόγος να γίνει η επέκταση αυτής της κατάστασης, αφού είναι σίγουρο πως η επέ¬κταση αυτή δε θα οδηγήσει στην ανεύρεση μιας καλύτερης λύσης. Συνεπώς, ο αλγό¬ριθμος μπορεί να κλαδέψει την κατάσταση αυτή και μαζί της φυσικά όλο το υποδέν¬δρο που ξεκινά από αυτήν. Η περιγραφή του B&B είναι:

1. Βάλε την αρχική κατάσταση στο μέτωπο της αναζήτησης.
2. Αρχική τιμή της καλύτερης λύσης είναι το +οο (όριο).
3. Αν το μέτωπο της αναζήτησης είναι κενό, τότε σταμάτησε. Η καλύτερη μέχρι τώρα λύση είναι και η βέλτιστη.
4. Βγάλε την πρώτη σε σειρά κατάσταση από το μέτωπο της αναζήτησης.
5. Αν η κατάσταση ανήκει στο κλειστό σύνολο, τότε πήγαινε στο 3.
6. Αν η κατάσταση είναι τελική, τότε ανανέωσε τη λύση ως την καλύτερη μέχρι τώρα και ανανέωσε την τιμή του ορίου με την τιμή που αντιστοιχεί στην τελική κατάσταση. Πήγαινε στο 3.

7. Εφάρμοσε τους τελεστές μεταφοράς για να παράγεις τις καταστάσεις-παιδιά και την τιμή που αντιστοιχεί σε αυτές.
8. Βάλε τις καταστάσεις-παιδιά, των οποίων η τιμή δεν υπερβαίνει το όριο, μπροστά στο μέτωπο της αναζήτησης.
9. Βάλε την κατάσταση-γονέα στο κλειστό σύνολο.
10. Πήγαινε στο 3.

Υπάρχουν διάφορες παραλλαγές του B&B, ανάλογα με το ποια κατάσταση επεκτείνεται πρώτη. Μία άλλη βελτίωση που επιδέχεται ο B&B είναι να επεκταθεί αυτή η κατάσταση που κατά εκτίμηση οδηγεί στη λύση με το μικρότερο κόστος.
Τέλος, ο B&B μπορεί να συνδυαστεί με δυναμικό προγραμματισμό (dynamic programming) όπου το κλάδεμα δε γίνεται μόνο σε σύγκριση με το τρέχον όριο, δηλαδή τη βέλτιστη λύση μέχρι εκείνη τη στιγμή, αλλά γίνεται και για κάθε κατά¬σταση που είναι περιττή. Για παράδειγμα, έστω δύο μονοπάτια το αβε13 και το αε8. Εφόσον υπάρχει διαδρομή από το α στο ε με κόστος 8 δεν υπάρχει νόημα να επεκτα¬θεί το μονοπάτι αβε με κόστος 13, που είναι μεγαλύτερο, γιατί ποτέ το αβε δε θα καταλήξει σε βέλτιστη λύση.
Η πρώτη εφαρμογή της μεθόδου BB στην επίλυση του προβλήματος του job shop προτάθηκε από τον Balas, και στη συνέχεια ακολούθησαν πολλοί άλλοι μεταξύ των οποίων και οι Carlier, Carlier και Pinson, Appelgate και Cook, Brucker, Perregaard και Clausen, Boyd και Burlingame και Martin. Αν και η μελέτη των μεθόδων BB δείχνει ότι επιτεύχθηκαν σημαντικές βελτιώσεις στη επίλυση του προβλήματος με τη χρήση τους, αυτό θα πρέπει κυρίως να αποδοθεί στη βελτίωση της τεχνολογίας των ηλεκτρονικών υπολογιστών, παρά στις τεχνικές που χρησιμοποιήθηκαν
Από αυτούς τους αλγορίθμους, οι minimax, alpha-beta αφορούν σε παιχνίδια δυο ατόμων. Για να αποφασίσω ποιον από τους δυο θα χρησιμοποιήσω, χρειάστηκε να καταγράψω τα χαρακτηριστικά τους.

3.6 Αλγόριθμοι Ευριστικής Αναζήτησης

Ευριστικός μηχανισμός (heuristic) είναι μία στρατηγική, βασισμένη στη γνώση για το συγκεκριμένο πρόβλημα, η οποία χρησιμοποιείται σα βοήθημα στη γρήγορη επίλυσή του.

Ο ευριστικός μηχανισμός υλοποιείται με ευριστική συνάρτηση (heuristic function), που έχει πεδίο ορισμού το σύνολο των καταστάσεων ενός προβλήματος και πεδίο τιμών το σύνολο τιμών που αντιστοιχεί σε αυτές. Ευριστική τιμή (heuristic value) είναι η τιμή της ευριστικής συνάρτησης και εκφράζει το πόσο κοντά βρίσκεται μία κατάσταση σε μία τελική. Η ευριστική τιμή δεν είναι η πραγματική τιμή της απόστασης από μία τερματική κατάσταση, αλλά μία εκτίμηση (estimate) που πολλές φορές μπορεί να είναι και λανθασμένη.

3.6.1 Αναζήτηση με Αναρρίχηση Λόφων

Ο αλγόριθμος HC
1.Η αρχική κατάσταση είναι η τρέχουσα κατάσταση.
2.Αν η κατάσταση είναι μία τελική τότε ανέφερε τη λύση και σταμάτησε.
3.Εφάρμοσε τους τελεστές μετάβασης για να βρεις τις καταστάσεις-παιδιά.
4.Βρες την καλύτερη κατάσταση σύμφωνα με την ευριστική συνάρτηση.
5.Η καλύτερη κατάσταση γίνεται η τρέχουσα κατάσταση.
6.Πήγαινε στο βήμα 2.

3.6.1.1 Ο αλγόριθμος HC (Ψευδοκώδικας)

```
algorithm hc(InitialState, FinalStates)
begin
Closed←Ø;
CurrentState←InitialState;
while CurrentState ∉ FinalStates do
Children ←Expand(CurrentState);
if Children= Ø then exit;
EvaluatedChildren←Heuristic(Children);
CurrentState←best(EvaluatedChildren);
endwhile;
end.
```

Ο HC χρησιμοποιείται σε προβλήματα όπου πρέπει να βρεθεί μία λύση πολύ γρήγορα, έστω και αν αυτή δεν είναι η καλύτερη, παίρνοντας όμως και το ρίσκο να μη βρεθεί καμία λύση, έστω και αν τέτοια υπάρχει.
! Πλεονεκτήματα:
Πολύ αποδοτικός και σε χρόνο και σε μνήμη,
! Μειονεκτήματα:
Είναι ατελής.
Βασικά προβλήματα του HC:
$ Πρόποδες (foothill).
$ Οροπέδιο (plateau).
$ Κορυφογραμμή (ridges).

Βελτιώσεις:
Εξαναγκασμένη αναρρίχηση λόφου (Enforced Hill-Climbing - EHC)
Προσομοιωμένη εξέλιξη (Simulated Annealing - SA)
Αναζήτηση με απαγορευμένες καταστάσεις (Tabu Search - TS).

3.6.2 Αναζήτηση Πρώτα στο Καλύτερο

Ο αλγόριθμος αναζήτηση πρώτα στο καλύτερο (Best-First - BestFS) κρατά όλες τις καταστάσεις στο μέτωπο αναζήτησης.

1.Βάλε την αρχική κατάσταση στο μέτωπο αναζήτησης.
2.Αν το μέτωπο αναζήτησης είναι κενό τότε σταμάτησε.
3.Πάρε την πρώτη σε σειρά κατάσταση από το μέτωπο αναζήτησης.
4.Αν η κατάσταση είναι μέλος του κλειστού συνόλου τότε πήγαινε στο 2.
5.Αν η κατάσταση είναι μία τελική τότε ανέφερε τη λύση και σταμάτα.
6.Εφάρμοσε τους τελεστές μεταφοράς για να παράγεις τις καταστάσεις-παιδιά.
7.Εφάρμοσε την ευριστική συνάρτηση σε κάθε παιδί.
8.Βάλε τις καταστάσεις-παιδιά στο μέτωπο αναζήτησης.
9.Αναδιάταξε το μέτωπο αναζήτησης, έτσι ώστε η κατάσταση με την καλύτερη ευριστική τιμή να
είναι πρώτη.
10. Βάλε τη κατάσταση-γονέα στο κλειστό σύνολο.
11. Πήγαινε στο βήμα 2.

3.6.2.1 Ο αλγόριθμος BestFS (Ψευδοκώδικας)

```
algorithm bestfs(InitialState, FinalStates)
begin
Closed←Ø;
EvaluatedInitialState←Heuristic(<InitialState>)
Frontier←<EvaluatedInitialState>;
CurrentState←best(Frontier);
while CurrentState ∉ FinalStates do
Frontier←delete(CurrentState,Frontier);
if CurrentState ∉ ClosedSet then
begin
Children ←Expand(CurrentState);
EvaluatedChildren←Heuristic(Children);
Frontier←Frontier ^ EvaluatedChildren;
Closed←ClosedU{CurrentState};
end;
if Frontier= Ø then exit;
CurrentState←best(Frontier);
endwhile;
end.
```

Πλεονεκτήματα:
Προσπαθεί να δώσει μια γρήγορη λύση σε κάποιο πρόβλημα. Το αν τα καταφέρει ή όχι εξαρτάται πολύ από τον ευριστικό μηχανισμό.
Είναι πλήρης.
! Μειονεκτήματα:
Το μέτωπο αναζήτησης μεγαλώνει με υψηλό ρυθμό και μαζί του ο χώρος που χρειάζεται για την αποθήκευσή του, καθώς και ο χρόνος για την επεξεργασία των στοιχείων του.
Δεν εγγυάται ότι η λύση που θα βρεθεί είναι η βέλτιστη.

3.6.3 Αναζήτηση A*

Συνδυάζουν την άπληστη αναζήτηση με την αναζήτηση με βάση το κόστος. Χρησιμοποιούν δηλαδή στην ταξινόμηση την συνάρτηση f(s) = d(s) + h(s)
• Η f(s) μας δίνει ουσιαστικά μία εκτίμηση της καλύτερης λύσης που περνάει από τον κόμβο s.
• Προϋπόθεση ότι η h(s) δεν υπερεκτιμά ποτέ το κόστος για να φτάσουμε στο κοντινότερο στόχο (αποδεκτή συνάρτηση).
• Όπως μπορεί να αποδειχτεί και αυστηρά αυτή η ιδιότητα της f(s) μας εγγυάται ότι ο αλγόριθμος θα βρει πάντα λύση και μάλιστα θα βρει πρώτα την καλύτερη λύση. Ο αλγόριθμος A* δηλαδή, είναι πλήρης και βέλτιστος.

Άλλη σημαντική Ιδιότητα: πληροφόρηση
• Πληροφόρηση: αν h1(n) & h2(n) είναι αποδεκτές συναρτήσεις (δηλ. Δεν υπερεκτιμούν την απόσταση σε στόχο) και & h1(n) >= h2(n), τότε οι κόμβοι που ο A* εξετάζει χρησιμοποιώντας την h1 είναι υποσύνολο αυτών που εξετάζει χρησιμοποιώντας την h2
• δηλαδή η h1(n) θα οδηγήσει σε πιο γρήγορη αναζήτηση
• ακραία περίπτωση: h2(n) = 0 = καθόλου πληροφόρηση - τότε ότι παραπάνω ξέρει η h1 θα βελτίωνε την αναζήτηση
• θα λέμε ότι η h1 είναι πιο πληροφορημένη από την h2

Σταθερότητα: Αν η h “σπάνια” υπερεκτιμά την απόσταση το πολυ s, τότε ο A* σπάνια θα βρεί λύσεις που είναι χειρότερες από την καλύτερη
• αρα, είναι χρήσιμο να έχουμε καλή εκτίμηση της h
• Πληρότητα: Ο A* θα τελειώσει και θα βρεί τη βέλτιστη λύση ακόμα και σε άπειρους χώρους αναζήτησης, αν υπάρχει βέλτιστη λύση και όλα τα κόστη είναι θετικά
• Μονότονη συνάρτηση: Αν για όλα τα n, m όπου m είναι απόγονος του n και h(n) - h(m) <= cost(n, m), τότε όποτε επισκεφτόμαστε ένα κόμβο, θα είμαστε εκεί με τον συντομότερο δρόμο
• δεν χρειάζεται να “θυμόμαστε” τα μονοπάτια
• ακραία περίπτωση: h(n) = 0 [κατά πλάτος]

Αλγόριθμος:
1. Δημιούργησε μία ουρά (queue) που αρχικά έχει τη ρίζα (root).
2. Έως ότου η ουρά είναι άδεια ή ο στόχος έχει βρεθεί, έλεγξε αν το πρώτο στοιχείο είναι ο στόχος.
2α. Αν είναι μην κάνεις τίποτα.
2β. Αν δεν είναι, τότε:
- βγάλε το πρώτο στοιχείο
- βάλε τα παιδιά του πρώτου στοιχείου πίσω στην ουρά
- ταξινόμησε όλη την ουρά με βάση την f(s) (ο καλύτερος κόμβος μπροστά)
3. Αν έχει βρεθεί ο στόχος, τότε έχουμε επιτυχία, αλλιώς, αποτυχία.

Σύγκριση αλγορίθμων

Όνομα αλγορίθμου	Πολυπλοκότητα χρόνου	Πολυπλοκότητα χώρου	Πληρότητα	Βέλτιστη Λύση
Κατά πλάτος	$O(b^m)$	$O(b^m)$	**ΝΑΙ**	ΟΧΙ
Κατά βάθος	$O(b^m)$	O(bm)	ΟΧΙ	ΟΧΙ
Αναρρίχησης	$O(b^m)$	O(bm)	ΟΧΙ	ΟΧΙ
Σταδιακής εκβάθυνσης	$O(b^m)$	O(bm)	**ΝΑΙ**	ΟΧΙ
Με βάση το κόστος	$O(b^m)$	$O(b^m)$	αναλόγως την d(s)	αναλόγως την d(s)
Περιορισμένης δέσμης	$O(k^m)$	$O(k^m)$ ή O(km)	ΟΧΙ	ΟΧΙ
Άπληστη αναζήτηση	$O(b^m)$	$O(b^m)$	αναλόγως την h(s)	**ΟΧΙ**
A*	$O(b^m)$	$O(b^m)$	**ΝΑΙ**	**ΝΑΙ**

3.6.4 Παιχνίδια

• Το παίξιμο παιχνιδιών από ηλεκτρονικούς υπολογιστές αποτέλεσε αντικείμενο έρευνας από της αρχές της Τεχνητής Νοημοσύνης.
• Το παίξιμο παιχνιδιών, όπως το σκάκι, θεωρείται ότι απαιτεί ανεπτυγμένη νοημοσύνη και άρα η νίκη κάποιου υπολογιστή θα “αποδείκνυε” ότι και οι υπολογιστές μπορούν να διαθέτουν νοημοσύνη - τεχνητή νοημοσύνη.
• Ιστορικά, μεγάλοι ερευνητές, όπως ο Claude Shannon και Alan Turing, έχουν ασχοληθεί με την συγγραφή προγραμμάτων που παίζουν σκάκι. Βέβαια, σύντομα διαπίστωσαν ότι αποτελεί αφελή απλούστευση ναθεωρήσει κανείς ότι η μηχανή διαθέτει νοημοσύνη μόνοκαι μόνο επειδή μπορεί να νικήσει στο σκάκι.

Τα παιχνίδια με τα οποία θα ασχοληθούμε είναι αυτά που μπορούν να οριστούν αυστηρά, παίζονται από δύο παίκτες και είναι σχετικά απλό να αναπαρασταθεί η κατάστασή τους.
• Γενικά, τα παιχνίδια θα μπορούσαμε να τα κατατάξουμε στην κατηγορία των προβλημάτων όπου η αρχική γνώση δεν είναι αρκετή για την λύση του προβλήματος. Αυτό συμβαίνει διότι δεν ξέρουμε πως σκέφτεται ο αντίπαλος
και άρα δεν μπορούμε να γνωρίζουμε ποιο θα είναι το τελικό αποτέλεσμα κάποιας δική μας κίνησης. Ωστόσο, γνωρίζουμε όλα τα πιθανά ενδεχόμενα και έτσι μπορούμε στην γενική περίπτωση να τα εξετάσουμε και να κρίνουμε ποίο από αυτά μας συμφέρει περισσότερο.

Γενική Θεώρηση: Έστω ότι έχουμε τους παίκτες Χ και Υ.
Ο κάθε παίκτης, σε κάθε κίνηση που κάνει, προσπαθεί να μεγιστοποιήσει την πιθανότητά του να κερδίσει.
• Ο Χ σε κάθε κίνηση που κάνει προσπαθεί να μεγιστοποιήσει την πιθανότητα του να κερδίσει ενώ ο Υ σε κάθε κίνηση του προσπαθεί να ελαχιστοποιήσει την πιθανότητα αυτή (την πιθανότητα να κερδίσει ο Χ).
• Βολεύει να ονομάζουμε τους παίκτες με τα ονόματα MAX και MIN. MAX είναι ο παίκτης που θέλουμε να νικήσει και ο αλγόριθμος προσπαθεί να διαλέξει τις καλύτερες κινήσεις για αυτόν ενώ MIN είναι ο αντίπαλος.

3.6.4.1 Παίξιμο παιχνιδιών

• Ανεπαρκής η αρχική πληροφορία για την επίλυση τουπροβλήματος.
• Απαιτείται ειδική διαδικασία αναζήτησης.
• Για μη τετριμμένα παιχνίδια ο χώρος αναζήτησης είναι εξαιρετικά μεγάλος - πρακτικά είναι αδύνατο η εξαντλητική αναζήτηση.
• Σε πολλά παιχνίδια το ζητούμενο δεν είναι μόνο να νικήσει ο παίκτης αλλά να πετύχει και το μεγαλύτερο δυνατό σκορ (βαθμολογία)
• Στην περίπτωση που σημασία έχει μόνο η νίκη ή η ισοπαλία (όπως το σκάκι) μπορούμε να δώσουμε το σκορ 1 για αυτόν που κερδίζει, 0 για αυτόν που κάνει ισοπαλία και -1 για αυτόν που χάνει.

Παράδειγμα - τα 7 δεκάρικα

• Έστω ότι παίζουμε το παιχνίδι με τα 7 δεκαράκια (Grundy’s game).
• Σε αυτό το παιχνίδι έχουμε 7 δεκαράκια τοποθετημένα αρχικά σε μία στοίβα. Οι δύο παίκτες παίζουν ο ένας μετά τον άλλο.
• Ο κάθε παίκτης, στην σειρά του πρέπει να διασπάσει μία στοίβα σε δύο άλλες στοίβες με άνισο αριθμό από δεκάρικα.
• Ο παίκτης που δεν μπορεί να το κάνει χάνει.

Δεν αρκεί να βρούμε ένα μονοπάτι στο γράφο από την αρχική κατάσταση σε κάποια νικηφόρα κατάσταση. Η ικανότητα του MIN να διαλέγει ποια θα είναι η κίνησή του κάνει την αναζήτηση πιο περίπλοκη.

• Ο πιο κατάλληλος τρόπος για να αναπαραστήσουμε την αναζήτηση είναι να χρησιμοποιήσουμε δέντρα AND-OR. Οι απόγονοι των OR κόμβων αντιπροσωπεύουν τις επιλογές κινήσεων που έχει ο παίκτης MAX ενώ οι απόγονοι των AND κόμβων αντιπροσωπεύουν τις επιλογές που έχει ο MIN παίκτης.
• Για να αξίζει κάποια κίνηση του MAX παίκτη (OR κόμβος) πρέπει κάθε επιτρεπτή επόμενη κίνηση του MIN (κάθε παιδί του OR κόμβου) να συμφέρει τον MAX.

3.7 Αλγόριθμος minimax

Για μια ισχύουσα κατάσταση, ο αλγόριθμος αυτός θα αποφασίσει ποια θα είναι η καλύτερη επόμενη κίνηση. Ο ένας παίκτης θα ονομάζεται max και ο άλλος min. Ο αλγόριθμος αυτός θα επιλέγει την καλύτερη κίνηση με δεδομένο ότι ο αντίπαλος παίκτης θα επιλέγει πάντα την καλύτερη κίνηση. Το μέτρο υπεροχής του ενός ή άλου αντιπάλου δίνεται από μια συνάρτηση η οποία ονομάζεται συνάρτηση αξιολόγησης. Η εξαντλητική αναζήτηση των δέντρων αναζήτησης είναι ανέφικτη. Το ζητούμενο είναι να χτιστεί το δέντρο μέχρι κάποιο βάθος και να βρεθεί η καλύτερη κίνηση από την παρούσα κατάσταση.

Η διαδικασία MIN-MAX

• Κατασκευάζουμε το πλήρες δέντρο του παιχνιδιού (το δέντρο AND-OR).
• Βαθμολογούμε τους κόμβους φύλα (τελικές καταστάσεις) σύμφωνα με την βαθμολογία που θα έπαιρνε ο παίκτης MAX αν το παιχνίδι κατέληγε σε εκείνη την κατάσταση.
• Εφαρμόζουμε τα παρακάτω δύο βήματα επαναληπτικά μέχρι όλοι οι κόμβοι του δέντρου να έχουν βαθμολογηθεί:
• Για κάθε κόμβο MAX (AND κόμβος) τα παιδιά του οποίου έχουν όλα βαθμολογηθεί υπολογίζουμε το μέγιστο αυτών των βαθμών και του αναθέτουμε αυτή την τιμή.
• Για κάθε κόμβο MIN (OR κόμβος) τα παιδιά του οποίου έχουν όλα βαθμολογηθεί υπολογίζουμε το ελάχιστο αυτών των βαθμών και του αναθέτουμε αυτή την τιμή.

3.8 Αλγόριθμος alpha-beta

Ο αλγόριθμος Άλφα-Βήτα (Alpha-Beta - AB) αποφεύγει την αναζήτηση καταστάσεων που ικανοποιούν ορισμένες συνθήκες και την αξιολόγηση καταστάσεων. Είναι όμοιος με τον minimax, αλλά με κλάδεμα υποδένδρων. Το κλάδεμα που κάνει ο AB, δεν είναι ευριστικό γιατί βασίζεται σε πραγματικά νούμερα. Αρχικά, όλοι οι κόμβοι που είναι μέγιστοι έχουν τιμή –άπειρο και αυτοί που είναι ελάχιστοι έχουν τιμή +άπειρο. Το κλάδεμα γίνεται αν σε ένα επίπεδο αντιστοιχούν ελάχιστοι κόμβοι και έχουμε βρει μια τιμή a για κάποιο κόμβο, τότε θα “κλαδευτούν” όλα τα υπόδεντρα που έχουν τη ρίζα τους στο ίδιο ή σε χαμηλότερο επίπεδο και έχουν τιμή <=a. Το

ίδιο γίνεται και αν σε ένα επίπεδο αντιστοιχούν μέγιστοι κόμβοι, τότε θα κλαδευτούν υπόδεντρα με τιμή >=a.

Ο αλγόριθμος Alfa-Beta εκμεταλλεύεται ορισμένες ιδιότητες του αλγορίθμου MIN-MAX και επιτρέπει τον υπολογισμό της επόμενης κίνησης χωρίς να υπολογίσει την τιμή όλων των κόμβων του δέντρου.
• Δεν επηρεάζει την απόφαση του αλγορίθμου MIN-MAX αφού κάνει περικοπές μόνο σε κλαδιά τα οποία σίγουρα δεν θα επηρεάσουν την τελική απόφαση.
• Κάνει περικοπή μόνο ασήμαντων κλαδιών : MIN (X1, X2, ..., Xk-1, Xk) = MIN (X1, X2, ..., Xk-1) αν υπάρχει i με 0<i<k και Xk > Xi.
MAX (X1, X2, ..., Xk-1, Xk) = MAX (X1, X2, ..., Xk-1) αν υπάρχει I με 0<i<k και Xk < Xi.

Έστω, για παράδειγμα ότι πρέπει να υπολογιστεί η τιμή ενός MIN κόμβου. Έχουμε υπολογίσει την τιμή του πρώτου παιδιού του και για το επόμενο παιδί του ξέρουμε (με κάποιον τρόπο) ότι η τιμή του θα είναι μεγαλύτερη από την τιμή του πρώτου παιδιού. Μπορούμε με σιγουριά να πούμε ότι η ακριβής τιμή του δεύτερου παιδιού δεν θα αλλάξει σε τίποτα την τελική τιμή του κόμβου. Δεν αλλοιώνει το αποτέλεσμα του MIN-MAX.
• Μπορεί να χρησιμοποιηθεί και κατα την διάρκεια ανάπτυξης του δέντρου.
• Έχει σημασία η σειρά εξέτασης των παιδιών ενός κόμβου.
• Σταματάμε την εξέταση ενός κόμβου όταν ισχύει Alfa>Beta.

1. Θέσε α = ένας πολύ μεγάλος αρνητικός αριθμός.
2. Θέσε β = ένας πολύ μεγάλος θετικός αριθμός.
3. Θέσε ρ = η θέση από την οποία πρέπει να παίξει ο παίκτης MAX.
4. Η minmax τιμή του κόμβου ρ = ΤΙΜΗ_MAX(ρ, α, β)
ΤΙΜΗ_MAX(ρ, α, β)
1. Αν Είναι_Φύλλο(ρ) τότε επέστρεψε Αξία_Κόμβου(ρ).
2. Διαφορετικά εκτέλεσε τα παρακάτω βήματα για κάθε παιδί π του ρ.
2.1. Θέσε α = MAX(α, ΤΙΜΗ_MIN(π, α, β))
2.2. Αν α ≥ β επέστρεψε β.
3. Επέστρεψε α.
ΤΙΜΗ_MIN(ρ, α, β)
1. Αν Είναι_Φύλλο(ρ) τότε επέστρεψε Αξία_Κόμβου(ρ).
2. Διαφορετικά εκτέλεσε τα παρακάτω βήματα για κάθε παιδί π του ρ.
2.1. Θέσε β = MIN(β, ΤΙΜΗ_MAX(π, α, β))
2.2. Αν α ≥ β επέστρεψε α.
3. Επέστρεψε β.

Αποφάσισα να χρησιμοποιήσω τον αλγόριθμο minimax και να τον υλοποιήσω.

4 Σχέδιο δράσης για την εκπόνηση της εργασίας

Κατά την ανάθεση της πτυχιακής εργασίας, σχεδίασα τον τρόπο δράσης μου και έθεσα χρονοδιαγράμματα για να ολοκληρωθεί, παραδοθεί και να παρουσιαστεί εμπρόθεσμα.

4.1 Σημαντικοί στόχοι για την ολοκλήρωση της πτυχιακής

Οι στόχοι που έθεσα για από την έναρξη μέχρι την ολοκλήρωση της εργασίας συνοψίζονται στον παρακάτω πίνακα.

Εργασία	Ημέρες
Ανάλυση του προβλήματος	10
εύρεση αλγορίθμων αναζήτησης	15
Επιλογή αλγορίθμου	1
Ολοκλήρωση κώδικα	50
Έλεγχος λειτουργίας προγράμματος	5
Συγγραφή αναφοράς εργασίας	32
Υποβολής αίτησης αξιολόγησης εργασίας	1
Προετοιμασία παρουσίασης αναφοράς	6
Παρουσίαση αναφοράς	1

5 Κύριο μέρος της πτυχιακής

Σε αυτό το κεφάλαιο θα αναλύσω την εργασία που υλοποίησα. Αρχικά, θα αναλύσω το πρόβλημα που έπρεπε να λύσω και θα προχωρήσω στην επεξήγηση του τρόπου υλοποίησης του προγράμματος

5.1 Ανάλυση του προβλήματος

Το πρόγραμμα που υλοποίησα είναι το παιχνίδι σκάκι με τεχνητή νοημοσύνη. Για να μπορεί ο χρήστης να παίζει με τον υπολογιστή έπρεπε να βρεθεί ηο κατάλληλος αλγόριθμος αναζήτησης. Όπως αναφέρθηκε και στο κεφάλαιο δυο, ο αλγόριθμος που επέλεξα ήταν ο minimax. Αφού επέλεξα αλγόριθμο αναζήτησης, έπρεπε να βρω και κάποιο αλγόριθμο αξιολόγησης. Τον αλγόριθμο αξιολόγησης τον χρειάζομαι για να μπορώ να υπολογίζω τη συνολική αξία του κάθε πιονιού σε όλες τις δυνατές κινήσεις που μπορεί να κάνει από τη θέση που βρίσκεται. Αφού υπολογιστεί η συνολική αξία οποιασδήποτε δυνατής κίνησης, θα επιλεχθεί εκείνη με το μικρότερο αριθμό. Έτσι, το πιόνι που έχει τη συγκεκριμένη αξία, θα μετακινηθεί στην καλύτερη θέση που μπορεί να επιλέξει από όλες τις δυνατές θέσεις. Αυτά είναι τα δυο κυριότερα θέματα που κλήθηκα να λύσω για να μπορέσω να προχωρήσω στην υλοποίηση του προγράμματος.

5.2 Απαιτήσεις συστήματος

Το συγκεκριμένο πρόγραμμα υλοποιήθηκε σε λειτουργικό περιβάλλον SUSE που έχω εγκατεστημένο στον υπολογιστή μου. Αφού ολοκληρώθηκε, μεταφέρθηκε και σε λειτουργικό σύστημα Windows XP, Windows Vista και Windows 7. Λειτουργεί το ίδιο καλά και σωστά, με την προϋπόθεση ο υπολογιστής με λειτουργικό σύστημα Windows να διαθέτει C compiler για Windows. Το πρόγραμμα έχει τις ελάχιστες απαιτήσεις και μπορεί να εκτελεστεί σε υπολογιστή με οποιαδήποτε μνήμη και επεξεργαστή από τα προαναφερθέντα λειτουργικά συστήματα.

5.3 Σχεδιασμός υλοποίησης

Συνάρτηση αξιολόγησης για το σκάκι

Υπεροχή κομματιών: αρχικά δίνεται στο κάθε πιόνι μια αξία, π.χ. Βασιλιάς=8, Άλογο=3, Πιόνι=1 κτλ. Έτσι θα δοθούν τιμές για όλα τα πιόνια στο παιχνίδι.

Υπεροχή θέσης: Κάθε κομμάτι που βρίσκεται στα κεντρικά τετράγωνα παίρνει επιπλέον 4 βαθμούς.
Απειλές: Κάθε πιόνι, ανάλογα με την ιδιότητά του, θα έχει και αντίστοιχο βαθμό απειλής, π.χ. απειλή βασίλισσας=10, απειλή βασιλιά=15, απειλή στρατηγού=9 κτλ. Αν σε επόμενη κίνηση απειλείται ο ο πύργος, θα προστεθεί στη συνολική αξία του πύργου της συγκεκριμένης θέσης στη συνολική του αξία, ενώ αν ένα πιόνι, εκεί που θα μετακινηθεί θα απειλεί το βασιλιά, θα αφαιρεθεί από τη συνολική του αξία -15 που αντιστοιχή στο βασιλιά.
Τελική τιμή: Η τιμή που προκύπτει στο τέλος, για να επιλεγεί η καλύτερη κίνηση, αφορά σε απειλές, υπεροχή θέσης και υπεροχή κομματιών. Θα επιλεγεί η κίνηση που η συνολική τελική τιμή της είναι η μικρότερη δυνατή και αυτή η κίνηση θα υλοποιηθεί.

Περιγραφή τρόπου λειτουργίας προγράμματος

Αρχικά, ζητείται από το χρήστη να επιλέξει πιόνι που θα θελήσει να μετακινήσει. Αφού δώσει θέση προέλευσης, γίνεται έλεγχος αν στη συγκεκριμένη θέση υπάρχει πιόνι του παίκτη και, σε περίπτωση που υπάρχει, ελέγχεται αν μπορεί να μετακινηθεί το συγκεκριμένο πιόνι, π.χ. πριν ξεκινήσει το παιχνίδι, δεν μπορεί να επιλεγεί ο στρατηγός ή η βασίλισσα, γιατί έχουν άλλα πιόνια γύρω τους. Αφού επιλεγεί, λοιπόν, η θέση προέλευσης, ζητείται από το χρήστη να δώσει θέση που επιθυμεί να μετακινήσει το πιόνι που διάλεξε. Αφού δώσει θέση προορισμού, γίνεται έλεγχος αν μπορεί να πάει εκεί το συγκεκριμένο πιόνι (αν βρίσκεται μέσα στις επιτρεπτές κινήσεις του) στη θέση που επιλέχθηκε. Αν δεν είναι επιτρεπτή η κίνηση, ζητείται νέα θέση, αλλιώς πραγματοποιείται η μετακίνηση. Μετά είναι η σειρά του υπολογιστή και, αφού παίξει, έρχεται ξανά η σειρά του παίκτη. Η διαδικασία αυτή ακολουθείται μέχρι να χάσει ο παίκτης ή ο υπολογιστής το βασιλιά του.

Τρόπος λειτουργίας τεχνητής νοημοσύνης

Μόλις ξεκινήσει το πρόγραμμα και δεν έχει παίξει ακόμα ο παικτης, ο υπολογιστής αποθηκεύει όλες τις δυνατές κινήσεις που μπορεί να κάνει για όλες τις δυνατές κινήσεις που μπορεί να επιλέξει να πραγματοποιήσει ο χρήστης. Για την κάθε μια από τις κινήσεις και για το κάθε πιόνι του που βρίσκεται πάνω στη σκακιέρα, υπολογίζονται όλες οι αξίες και επιλέγεται η καλύτερη δυνατή (δηλαδή αυτή με το μικρότερο αριθμό). Έτσι, με αυτό τον τρόπο μπορεί να επιλέξει την καλύτερη δυνατή κίνηση, ανάλογα με τις κινήσεις που μπορεί να κάνει την επόμενη φορά που θα έρθει η σειρά του παίκτη.

5.4 Υλοποίηση

Ακολουθεί ο κώδικας του προγράμματός μου και αναλυτική περιγραφή του.

5.4.1 Επεξήγηση βιβλιοθηκών, καθολικών μεταβλητών, δομών

Ορίζω τις βιβλιοθήκες που περιέχουν τις πιο σημαντικές συναρτήσεις συστήματος που θα χρησιμοποιήσω κατά την υλοποίηση του προγράμματός μου.

```
#include <stdio.h>
#include <stdlib.h>
```

Για να απλοποιήσω τη δήλωση ορισμένων μεταβλητών, έκανα χρήση της εντολής #define, η οποία είναι δουλεία του πρόεπεξεργαστή της C και πριν γίνει compile το πρόγραμμα, ο προεπεξεργαστής θα πάει και να αντικαταστήσει όλα τα define με τους αντίστοιχους αριθμούς που βρίσκονται δίπλα τους.

Το N το χρησιμοποιώ για να φτιάξω τη σκακιέρα, η οποία θα είναι μεγέθους 8x8.

```
#define N 8
```

Πιο συγκεκριμένα, τα παρακάτω define αναφέρονται στα άσπρα πιόνια που θα χρησιμοποιήσω και πιο αναλυτικά, w_rook αναφέρεται στον άσπρο πύργο, το w_knight αναφέρεται στο άσπρο άλογο, το w_queen στην άσπρη βασίλισσα, το w_king στον άσπρο βασιλιά και το w_pawn στο άσπρο πιόνι, το w_general αναφέρεται στον άσπρο στατηγό.

```
#define w_rook 100
#define w_knight 101
#define w_queen 102
#define w_king 103
#define w_pawn 104
#define w_general 105
```

Τα παρακάτω define αναφέρονται στα μαύρα πιόνια που θα χρησιμοποιήσω και πιο αναλυτικά, b_rook αναφέρεται στο μαύρο πύργο, το b_knight αναφέρεται στο μαύρο άλογο, το b_queen στη μαύρη βασίλισσα, το b_king στο μαύρο βασιλιά, το b_pawn στο μαύρο πιόνι και το b_general αναφέρεται στο μαύρο στρατηγό.

```
#define b_rook 106
#define b_knight 107
#define b_queen 108
#define b_king 109
#define b_pawn 110
#define b_general 111
```

Τα παρακάτω define αναφέρονται στους δυο παίκτες του παιχνιδιού και πιο συγκεκριμένα, το white αφορά στον άνθρωπο που θα παίζει και το black στον υπολογιστή.

```
#define white 112
#define black 113
```

Τα παρακάτω define αναφέρονται στην αξία του κάθε πιονιού στο παιχνίδι, όπου rookVal είναι η αξία του πύργου, generalVal η αξία του στρατηγού, knightVal η αξία του αλόγου, queenVal, η αξία της βασίλισσας και kingVal η αξία του βασιλιά. Οι αριθμοί αυτοί έχουν προκύψει ανάλογα με τη σπουδαιότητα του κάθε πιονιού και τη σημαντικότητά τους κατά τη διάρκεια ενός παιχνιδιού.

```
#define rookVal 4
#define generalVal 5
#define knightVal 3
#define queenVal 7
#define kingVal 8
#define pawnVal 1
#define centerVal 4
```

Τα παρακάτω define αναφέρονται στους βαθμούς του κάθε πιονιού και ανάλογα αν απειλεί ή απειλείται, ο αριθμός που βρίσκεται δίπλα του θα προστίθεται ή θα αφαιρείται από τη συνολική αξία, όπως θα αναφέρουμε παρακάτω. Πιο συγκεκριμένα, το threatQueenVal αναφέρεται στην απειλή της βασίλισσας, το threatGeneralVal αναφέρεται στην απειλή του στρατηγού, threatRookVal αναφέρεται στην απειλή του πύργου, threatKnightVal αναφέρεται στην απειλή του αλόγου, threatPawnVal αναφέρεται στην απειλή του πιονιού και το threatKingVal αναφέρεται στην απειλή του βασιλιά. Επειδή ο βασιλιάς είναι το κυριότερο πιόνι στο παιχνίδι, έχω δώσει πολύ μεγάλο αριθμό στη συγκεκριμένη μεταβλητή, σε σχέση με όλες τις υπόλοιπες. Οι αριθμοί αυτοί έχουν προκύψει ανάλογα με τη σπουδαιότητα του κάθε πιονιού και συγκριτικά με αυτή βγήκαν και οι απειλές τους.

```
#define threatQueenVal 10
#define threatGeneralVal 9
#define threatRookVal 8
#define threatKnightVal 7
#define threatPawnVal 5
#define threatKingVal 15
```

Η μεταβλητή player είναι μια global-καθολική μεταβλητή, η οποία ορίζεται στην αρχή του προγράμματος έξω από κάθε συνάρτηση. Αυτό γίνεται για να μπορεί να τη βλέπει κάθε συνάρτηση του προγράμματος και η main, γιατί αφορά στην εναλλαγή των παιχτών, δηλαδή αν θα παίζει ο παίκτης ή ο υπολογιστής.

```
int player;
```

Με τη χρήση των typedef, εκχωρώ διαφορετικά, πιο μικρά ονόματα για τις δομές που θα χρησιμοποιήσω για την επίλυση του προγράμματος. Πιο συγκεκριμένα, αντί να χρησιμοποιώ το όνομα struct status κάθε φορά που θέλω να πάρω ένα

στιγμιότυπο της δομής status, θα χρησιμοποιώ το status και θα έχω ακριβώς το ίδιο αποτέλεσμα. Με τον ίδιο τρόπο ορίζονται και τα υπόλοιπα typedef.

```
typedef struct status status;
typedef struct moveList moveList;
typedef struct pc pc;
typedef struct humanList humanList;
```

Η δομή moveList αφορά στην αποθήκευση όλων των δυνατών κινήσεων που μπορούν να γίνουν από οποιαδήποτε δοσμένη θέση. Τα πεδία της είναι τρεις ακέραιοι αριθμοί a, b, v και ένας δείκτης στον επόμενο κόμβο τύπου moveList. Η μεταβλητή a είναι η δυνατή θέση που μπορεί να πάει ένα πιόνι στη σκακιέρα στον άξονα x (τον κάθετο άξονα) και η μεταβλητή b είναι η δυνατή θέση που μπορεί να μετακινηθεί ένα πιόνι στη σκακιέρα στον άξονα y (τον οριζόντιο άξονα), δηλαδή ένα πιόνι μπορεί να πάει στη θέση (a, b). Η μεταβλητή v αφορά στην αξία που θα έχει το πιόνι στη θέση που θα μετακινηθεί. Ο αριθμός αυτός προκύπτει από πολλές συνιστώσες, όπως αν θα απειλείται στη νέα θέση που θα πάει ή αν θα μετακινηθεί σε κεντρικό τετράγωνο στη σκακιέρα. Τέλος, ο δείκτης *next αφορά στη σύνδεση των κόμβων με όλες τις δυνατές κινήσεις που μπορεί να κάνει ένα πιόνι προς όλα τα γειτονικά τετράγωνα της σκακιέρας.

```
struct moveList
{
  int a;
  int b;
  int v;  //value for next moves
  moveList *next;
};
```

Η δομή status αφορά στην αποθήκευση της τρέχουσας κατάστασης της σκακιέρας για κάθε ένα από τα πιόνια που βρίσκονται σε αυτήν. Αποτελείται από δυο ακέραιους αιρθμούς και ένα δείκτη. Πιο συγκεκριμένα, η μεταβλητή kind αφορά στο είδος του πιονιού που βρίσκεται στο συγκεκρινμένο τετράγωνο της σκακιέρας και η μεταβλητή value είναι η αξία του πιονιού αυτού. Ο δείκτης *ptr είναι του τύπου δομής moveList και συνδέει το συγκεκριμένο πιόνι που βρίσκεται στη σκακιέρα με όλες τις δυνατές κινήσεις που μπορεί να κάνει αυτό προς όλα τα γειτονικά τετράγωνα της σκακιέρας.

```
struct status
{
  int kind;
  int value;
  struct moveList *ptr;
};
```

Η δομή struct pc αφορά στην κάθε δυνατή κίνηση που μπορεί να κάνει κάθε πιόνι του υπολογιστή. Ο κάθε κόμβος της λίστας που θα δημιουργηθεί θα αφορά στη

θέση προέλευσης του πιονιού, στη θέση προορισμού του πιονιού, στη νέα αξία που θα έχει στη νέα θέση που θα πάει και στο είδος του συγκεκριμένου πιονιού. Τα πεδία που έχει η δομή αυτή είναι έξι ακέραιοι αριθμοί και ένας δείκτης σε επόμενο κόμβο. Πιο αναλυτικά, η μεταβλητή pc_from1 είναι η θέση που βρίσκεται το συγκεκριμένο πιόνι στον κάθετο άξονα και η μεταβλητή pc_from2 είναι η θέση που βρίσκεται το συγκεκριμένο πιόνι στον οριζόντιο άξονα, δηλαδή η αρχική θέση του πιονιού είναι η (pc_from1, pc_from2). Η μεταβλητή pc_to1 είναι η θέση που θα μετακινηθεί το συγκεκριμένο πιόνι στον κάθετο άξονα και η μεταβλητή pc_to2 είναι η θέση που θα μετακινηθεί το συγκεκριμένο πιόνι στον οριζόντιο άξονα, δηλαδή μια από τις πιθανές δυνατές θέσεις που μπορεί να μετακινηθεί το πιόνι αυτό είναι η (pc_to1, pc_to2). Η μεταβλητή new_val είναι η αξία που θα έχει το πιόνι στη νέα θέση που θα μεταβεί και η μεταβλητή pc_kind είναι το είδος του πιονιού που αναλύουμε τη συγκεκριμένη στιγμή για να βρούμε τις πιθανές κινήσεις του. Τέλος, ο δείκτης *next είναι ένας δείκτης στον επόμενο κόμβο στη λίστα με όλες τις πιθανές θέσεις πάνω στη σκακιέρα που μπορεί να μετακινηθεί το συγκεκριμένο πιόνι.

```
struct pc
{
  int pc_from1;
  int pc_from2;
  int pc_to1;
  int pc_to2;
  int new_val;
  int pc_kind;
  pc *next;
};
```

Η δομή struct humanList αφορά στην κάθε δυνατή κίνηση που μπορεί να κάνει κάθε πιόνι του παίκτη. Ο κάθε κόμβος της λίστας που θα δημιουργηθεί θα αφορά στη θέση προέλευσης του πιονιού, στη θέση προορισμού του πιονιού, στη νέα αξία που θα έχει στη νέα θέση που θα πάει και στο είδος του συγκεκριμένου πιονιού. Τα πεδία που έχει η δομή αυτή είναι πέντε ακέραιοι αριθμοί και ένας δείκτης σε επόμενο κόμβο. Πιο αναλυτικά, η μεταβλητή pl_from1 είναι η θέση που βρίσκεται το συγκεκριμένο πιόνι στον κάθετο άξονα και η μεταβλητή pl_from2 είναι η θέση που βρίσκεται το συγκεκριμένο πιόνι στον οριζόντιο άξονα, δηλαδή η αρχική θέση του πιονιού είναι η (pl_from1, pl_from2). Η μεταβλητή pl_to1 είναι η θέση που θα μετακινηθεί το συγκεκριμένο πιόνι στον κάθετο άξονα και η μεταβλητή pl_to2 είναι η θέση που θα μετακινηθεί το συγκεκριμένο πιόνι στον οριζόντιο άξονα, δηλαδή μια από τις πιθανές δυνατές θέσεις που μπορεί να μετακινηθεί το πιόνι αυτό είναι η (pl_to1, pl_to2). Η μεταβλητή pl_kind είναι το είδος του πιονιού που αναλύουμε τη συγκεκριμένη στιγμή για να βρούμε τις πιθανές κινήσεις του. Τέλος, ο δείκτης *next είναι ένας δείκτης στον επόμενο κόμβο στη λίστα με όλες τις πιθανές θέσεις πάνω στη σκακιέρα που μπορεί να μετακινηθεί το συγκεκριμένο πιόνι.

```
struct humanList
{
```

```
  int pl_from1;
  int pl_from2;
  int pl_to1;
  int pl_to2;
  int pl_kind;
  humanList *next;
};
```

Ορίζουμε και κάποιες καθολικές μεταβλητές, οι οποίες θα είναι ορατές σε όλες τις συναρτήσεις του προγράμματος καθ'όλη τη διάρκεια εκτέλεσής του. Πιο συγκεκριμένα, η μεταβλητή *head είναι ένα δείκτης τύπου δομής pc και αφορά στην εκκίνηση της λίστας με τις κινήσεις του υπολογιστή, δηλαδή να είναι προσβάσιμη ανά πάσα στιγμή χρειαστούμε. Αρχικοποιείται με NULL, που σημαίνει ότι δεν έχει δημιουργηθεί ακόμη τέτοια λίστα. Η μεταβλητή *pl_head είναι ένας δείκτης στη λίστα με όλες τις πιθανές κινήσεις του παίκτη. Η μεταβλητή board[N][N] είναι τύπου δομής status και είναι η σκακιέρα του παιχνιδιού. Είναι διαστάσεων 8x8, αφού το N το ορίσαμε παραπάνω. Η μεταβλητή kingThreatened είναι ένας ακέραιος αριθμός και αφορά στην απειλή του βασιλιά στην τρέχουσα χρονική στιγμή. Αν είναι 0, τότε ο βασιλιάς δεν απειλείται, ενώ αν γίνει 1 τότε απειλείται.

```
pc *head = NULL;
humanList *pl_head = NULL;
status board[N][N];
int kingThreatened=0;
```

5.4.2 Επεξήγηση συναρτήσεων

moveList *insert(moveList *head, int x, int y);

Η συνάρτηση *insert αφορά στην εισαγωγή όλων των δυνατών κινήσεων για δοσμένη θέση προέλευσης x, y που δίνονται σαν όρισμα στη συνάρτηση. Το όρισμα *head είναι ένας δείκτης που δείχνει στην αρχή της λίστας με τις δυνατές κινήσεις moveList. Η συνάρτηση αυτή, αφού κάνει την εισαγωγή του κόμβου που θα δημιουργήσει στη λίστα, θα επιστρέψει το head της λίστας για να μπορούμε να τη χρησιμοποιήσουμε όποτε τη θέλουμε.

```
moveList *insert(moveList *head, int x, int y)
{
  moveList *tmp = head;
  moveList *new = (moveList *)malloc(sizeof(moveList));
  new->a = x;
  new->b = y;
  new->v = 0;
  if(head == NULL)
```

```
    return new;
  else
  {
    while(tmp->next != NULL)
          tmp = tmp->next;
    tmp->next = new;
    new->next = NULL;
    return head;
  }
}
```

void initialize();

Η συνάρτηση initialize, αρχικοποιεί τη σκακιέρα, βάζοντας πάνω σε κάθε τετράγωνο το είδος του πιονιού ή το κενό, το δείκτη προς όλες τις δυνατές κινήσεις του πιονιού αυτού και την αξία του. Δεν επιστρέφει κάτι.

```
void initialize()
{
  int i, j;
  moveList *p = NULL;

  for(i=2; i<6; i++)  //edw tha mpoun ta kena
  {
    for(j=0; j<N; j++)
    {
          board[i][j].kind = 0;
          board[i][j].ptr = NULL;
    }
  }


  /**********Apodosi arxikwn timwn sta aspra pionia***************/

  board[0][0].kind = w_rook;       /*purgos*/
  board[0][0].value = rookVal;
  board[0][0].ptr = NULL;

  board[0][7].kind = w_rook;  /*purgos*/
  board[0][7].value = rookVal;
  board[0][7].ptr = NULL;

            /*alogo*/
  board[0][1].kind = w_knight;
```

```
board[0][1].value = knightVal;
board[0][1].ptr = NULL;

board[0][6].kind = w_knight;
board[0][6].value = knightVal;
board[0][6].ptr = NULL;

          /*aksiwmatikos*/
board[0][2].kind = w_general;
board[0][2].value = generalVal;
board[0][2].ptr = NULL;

board[0][5].kind = w_general;
board[0][5].value = generalVal;
board[0][5].ptr = NULL;

  /*basilissa*/
board[0][3].kind = w_queen;
board[0][3].value = queenVal;
board[0][3].ptr = NULL;

  /*basilias*/
board[0][4].kind = w_king;
board[0][4].value = kingVal;
board[0][4].ptr = NULL;

for(i=0; i<N; i++)                    /*aspra pionia*/
{
  board[1][i].kind = w_pawn;
  board[1][i].value = pawnVal;
  board[1][i].ptr = NULL;
}
/*************************************************************/

/**********Apodosi arxikwn timwn sta maura pionia***************/

  /*purgos*/
board[7][0].kind = b_rook;
board[7][0].value = -rookVal;
board[7][0].ptr = NULL;

board[7][7].kind = b_rook;
board[7][7].value = -rookVal;
board[7][7].ptr = NULL;

  /*vasilissa*/
board[7][3].kind = b_queen;
```

```
  board[7][3].value = -queenVal;
  board[7][3].ptr = NULL;

    /*vasilias*/
  board[7][4].kind = b_king;
  board[7][4].value = -kingVal;
  board[7][4].ptr = NULL;

  for(i=0; i<N; i++)
  {
    board[6][i].kind = b_pawn;
    board[6][i].value = -pawnVal;
    board[6][i].ptr = NULL;
  }

    /*alogo*/
  board[7][1].kind = b_knight;
  board[7][1].value = -knightVal;
  board[7][1].ptr = NULL;

  board[7][6].kind = b_knight;
  board[7][6].value = -knightVal;
  board[7][6].ptr = NULL;

    /*aksiwmatikos*/
  board[7][2].kind = b_general;
  board[7][2].value = -generalVal;
  board[7][2].ptr = NULL;

  board[7][5].kind = b_general;
  board[7][5].value = -generalVal;
  board[7][5].ptr = NULL;

  return ;
}
```

void print();

Η συνάρτηση print χρησιμοποιείται για να τυπώνει την τρέχουσα κατάσταση της σκακιέρας κάθε φορά που αυτή θα καλείται. Πιο συγκεκριμένα, όπου βρίσκει τύπο πύργο, θα τυπώνει T, όπου βρίσκει άλογο, θα τυπώνει H, όπου βρίσκει αξιωματικό, θα τυπώνει G, όπου βρίσκει βασίλισσα θα τυπώνει Q, όπου βρίσκει βασιλιά θα τυπώνει K και όπου βρίσκει απλά πιόνια θα τυπώνει S. Τα πιόνια του υπολογιστή θα έχουν τα ίδια σύμβολα, απλά θα τυπώνονται με κόκκινο χρώμα. Η συνάρτηση δεν επιστρέφει τίποτα.

```
void print()
{
  int i, j;
  printf("   0   1   2   3   4   5   6   7 \n");
  for(i=0; i<N; i++) {
    printf("%d ", i);
    for(j=0; j<N; j++)
    {
          if( board[i][j].kind == w_rook)
            printf("| T |");
          else if(board[i][j].kind == b_rook)
            printf("|\033[1;31m T \033[0m|");
          else if(board[i][j].kind == w_knight)
            printf("| H |");
          else if (board[i][j].kind == b_knight)
            printf("|\033[1;31m H \033[0m|");
          else if(board[i][j].kind == w_general)
            printf("| G |");
          else if (board[i][j].kind == b_general)
            printf("|\033[1;31m G \033[0m|");
          else if(board[i][j].kind == w_pawn)
            printf("| S |");
          else if (board[i][j].kind == b_pawn)
            printf("|\033[1;31m S \033[0m|");
          else if(board[i][j].kind == w_king)
            printf("| K |");
          else if (board[i][j].kind == b_king)
            printf("|\033[1;31m K \033[0m|");
          else if(board[i][j].kind == w_queen)
            printf("| Q |");
          else if (board[i][j].kind == b_queen)
            printf("|\033[1;31m Q \033[0m|");
          else
            printf("| - |");
    }
    printf("\n");
  }
  return ;
}
```

void checkMoves(pc *pList);

Η συνάρτηση checkMoves, ελέγχει για ένα συγκεκριμένο πιόνι του υπολογιστή αν απειλείται ή αν απειλεί και αντίστοιχα θα προσθέτει ή θα αφαιρεί την αντίστοιχη τιμή ανάλογα με το είδος του πιονιού που απειλεί ή απειλείται. Η συνάρτηση δεν επιστρέφει τίποτα, απλά αποθηκεύει στο πεδίο pList->new_val την τιμή που θα έχει

το πιόνι αφού υπολογίσει τις απειλές. Αυτό γίνεται για να μπορεί αργότερα να αξιολογήσει τις δυνατές κινήσεις και να βρει την πιο κατάλληλη.

```
void checkMoves(pc *pList)
{
  //pc *pList = head;
  humanList *hList = pl_head;
    while(hList)
    {
        if((pList->pc_to1 == hList->pl_to1) && (pList->pc_to2 == hList->pl_to2))
        {
          if (pList->pc_kind == b_king)
            pList->new_val += threatKingVal;
          else if (pList->pc_kind == b_queen)
            pList->new_val += threatQueenVal;
          else if (pList->pc_kind == b_rook)
            pList->new_val += threatRookVal;
          else if (pList->pc_kind == b_knight)
            pList->new_val += threatKnightVal;
          else if (pList->pc_kind == b_general)
            pList->new_val += threatGeneralVal;
          else if (pList->pc_kind == b_pawn)
            pList->new_val += threatPawnVal;
        }
        if ((pList->pc_from1 == hList->pl_to1) && (pList->pc_from2 == hList->pl_to2)
&&
            pList->pc_kind == b_king)
        {
          kingThreatened = 1;  //se periptwsi pou apeileitai o vasilias gia mia apo tis
        }
        hList = hList->next;    //epomenes kiniseis tou human, ekei pou vrisketai twra
    }
  return;
}
```

void Evaluate (pc *tmp);

Η συνάρτηση Evaluate είναι ουσιαστικά η συνάρτηση αξιολόγησης όλων των πιθανών κινήσεων του υπολογιστή. Κάθε φορά που καλείται, υπολογίζει για το πιόνι που βρίσκεται στη θέση προέλευσης (tmp->pc_from1, tmp->pc_from2) την τιμή που θα έχει το πιόνι αν μετακινηθεί στη θέση (tmp->pc_to1, tmp->pc_to2). Στο τέλος καλεί και τη συνάρτηση checkMoves που αναλύσαμε παραπάνω για να υπολογίσει τις απειλές. Δεν επιστρέφει κάτι η συνάρτηση.

```
void Evaluate (pc *tmp)
{
```

```
  int value;
  int i = tmp->pc_to1;
  int j = tmp->pc_to2;
  int x = tmp->pc_from1;
  int y = tmp->pc_from2;

  //axiologisi tis listas me oles tis pithanes kiniseis
    if (board[x][y].kind == b_rook)
          tmp->new_val = -rookVal;
    else if (board[x][y].kind == b_knight)
          tmp->new_val = -knightVal;
    else if (board[x][y].kind == b_queen)
          tmp->new_val = -queenVal;
    else if (board[x][y].kind == b_king)
          tmp->new_val = -kingVal;
    else if (board[x][y].kind == b_pawn)
          tmp->new_val = -pawnVal;
    else if (board[x][y].kind == b_general)
          tmp->new_val = -generalVal;
    if( (i>=2 && i<=5) && (j>=2 && j<=5) )   /*kentrika tetragwna*/
    {
          if(board[x][y].kind>=100 && board[x][y].kind<=105)
            tmp->new_val += centerVal;
          if(board[x][y].kind>=106 && board[x][y].kind<=111)
            tmp->new_val -= centerVal;
    }
    if (player == black && (board[i][j].kind == w_queen))
          tmp->new_val -= threatQueenVal;
    if (player == black && (board[i][j].kind == w_rook))
          tmp->new_val -= threatRookVal;
    if (player == black && (board[i][j].kind == w_knight))
          tmp->new_val -= threatKnightVal;
    if (player == black && (board[i][j].kind == w_general))
          tmp->new_val -= threatGeneralVal;
    if (player == black && (board[i][j].kind == w_pawn))
          tmp->new_val -= threatPawnVal;
    if (player == black && board[i][j].kind == w_king)
            tmp->new_val -= threatKingVal;
    checkMoves (tmp);
  return;
}
```

int moveKnight (int x1,int y1,int flag1,int flag2,moveList *tmp);

Η συνάρτηση moveKnight κοιτάζει το προς κίνηση πιόνι το οποίο είναι άλογο και φτιάχνει μια λίστα με όλες τις δυνατές κινήσεις. Πιο συγκεκριμένα, οι κινήσεις που

μπορεί να κάνει το άλογο είναι κάτω αριστερά (από τη θέση (1,1) στη θέση (3,0)), κάτω δεξιά (από τη θέση (1,1) στη θέση (3,2)), πάνω αριστερά (από τη θέση (4,2) στη θέση (2,1)), πάνω δεξιά (από τη θέση (4,2) στη θέση (2,3)), αριστερά πάνω (από τη θέση (4,2) στη θέση (3,0)), αριστερά κάτω (από τη θέση (4,2) στη θέση (5,0)), δεξιά πάνω (από τη θέση (4,2) στη θέση (3,4)) και δεξιά κάτω (από τη θέση (4,2) στη θέση (5,4)). Ανάλογα με την κίνηση που θα ζητηθεί κάθε φορά, ελέγχει αν μπορεί να γίνει η κίνηση. Αν μπορεί να γίνει, θα την αποθηκεύσει στη λίστα με τις δυνατές κινήσεις, καθώς επίσης και στη συγκεκριμένηθέση στη σκακιέρα και θα επιστρέψει 0, αλλιώς θα επιστρέψει -1.

```
int moveKnight (int x1,int y1,int flag1,int flag2,moveList *tmp)
{  //flag1, flag2 einai antipala pionia
  if (board[x1+2][y1-1].kind == 0 || (board[x1+2][y1-1].kind >= flag1 &&
     board[x1+2][y1-1].kind <= flag2))
  {
    if ((x1 >= 0 && x1 <= N-3) && (y1 != 0))
          tmp = insert(tmp,x1+2,y1-1);  //katw aristera (1,1)->(3,0)
  }
  if (board[x1+2][y1+1].kind == 0 || (board[x1+2][y1+1].kind >= flag1 &&
     board[x1+2][y1+1].kind <= flag2))
  {
    if ((x1 >= 0 && x1 <= N-3) && (y1 != N-1))
          tmp = insert(tmp,x1+2,y1+1);  //katw deksia (1,1)->(3,2)
  }
  if (board[x1-2][y1-1].kind == 0 || (board[x1-2][y1-1].kind >= flag1 &&
     board[x1-2][y1-1].kind <= flag2))
  {
    if ((x1 >= N-6 && x1 <= N-1) && (y1 != 0))
          tmp = insert(tmp,x1-2,y1-1);  //panw aristera (4,2)->(2,1)
  }
  if (board[x1-2][y1+1].kind == 0 || (board[x1-2][y1+1].kind >= flag1 &&
     board[x1-2][y1+1].kind <= flag2))
  {
    if ((x1 >= N-6 && x1 <= N-1) && (y1 != N-1))
          tmp = insert(tmp,x1-2,y1+1);  //panw deksia (4,2)->(2,3)
  }
  if (board[x1-1][y1-2].kind == 0 || (board[x1-1][y1-2].kind >= flag1 &&
     board[x1-1][y1-2].kind <= flag2))
  {
    if ((y1 >= N-6 && y1 <= N-1) && x1 != 0)
          tmp = insert(tmp,x1-1,y1-2);  //aristera panw (4,2)->(3,0)
  }
  if (board[x1+1][y1-2].kind == 0 || (board[x1+1][y1-2].kind >= flag1 &&
     board[x1+1][y1-2].kind <= flag2))
  {
    if ((y1 >= N-6 && y1 <= N-1) && x1 != N-1)
          tmp = insert(tmp,x1+1,y1-2);  //aristera katw (4,2)->(5,0)
```

```
  }
  if (board[x1-1][y1+2].kind == 0 || (board[x1-1][y1+2].kind >= flag1 &&
     board[x1-1][y1+2].kind <= flag2))
  {
    if ((y1 >= 0 && y1 <= N-3) && (x1 != 0))
          tmp = insert(tmp,x1-1,y1+2);  //deksia panw (4,2)->(3,4)
  }
  if (board[x1+1][y1+2].kind == 0 || (board[x1+1][y1+2].kind >= flag1 &&
     board[x1+1][y1+2].kind <= flag2))
  {
    if ((y1 >= 0 && y1 <= N-3) && (x1 != N-1))
          tmp = insert(tmp,x1+1,y1+2);  //deksia katw (4,2)->(3,0)
  }
  board[x1][y1].ptr = tmp;
  if(tmp == NULL)
    return -1;
  return 0;
}
```

int moveKing (int x1,int y1,int flag1,int flag2,moveList *tmp);

Η συνάρτηση moveKing κοιτάζει το προς κίνηση πιόνι το οποίο είναι ο βασιλιάς και φτιάχνει μια λίστα με όλες τις δυνατές κινήσεις. Οι κινήσεις που μπορεί να κάνει ο βασιλιάς είναι πάνω, κάτω, αριστερά, δεξιά, πάνω δεξιά, πάνω αριστερά, κάτω δεξιά και κάτω αριστερά. Ανάλογα με την κίνηση που θα ζητηθεί κάθε φορά, ελέγχει αν μπορεί να γίνει η κίνηση. Αν μπορεί να γίνει, θα την αποθηκεύσει στη λίστα με τις δυνατές κινήσεις, καθώς επίσης και στη συγκεκριμένηθέση στη σκακιέρα και θα επιστρέψει 0, αλλιώς θα επιστρέψει -1.

```
int moveKing (int x1,int y1,int flag1,int flag2,moveList *tmp)
{
  if (board[x1+1][y1].kind == 0 || (board[x1+1][y1].kind >= flag1 &&
     board[x1+1][y1].kind <= flag2))
  {
    if (x1 != N-1)
          tmp = insert (tmp,x1+1,y1);  //katw
  }
  if (board[x1+1][y1-1].kind == 0 || (board[x1+1][y1-1].kind >= flag1 &&
     board[x1+1][y1-1].kind <= flag2))
  {
    if (x1 != N-1 && y1 != 0)
          tmp = insert (tmp,x1+1,y1-1);  //katw aristera
  }
  if (board[x1][y1-1].kind == 0 || (board[x1][y1-1].kind >= flag1 &&
     board[x1][y1-1].kind <= flag2))
  {
```

```
    if (y1 != 0)
          tmp = insert (tmp,x1,y1-1);  //aristera
  }
  if (board[x1-1][y1-1].kind == 0 || (board[x1-1][y1-1].kind >= flag1 &&
     board[x1-1][y1-1].kind <= flag2))
  {
    if (x1 != 0 && y1 != 0)
          tmp = insert (tmp,x1-1,y1-1);  //panw aristera
  }
  if (board[x1-1][y1].kind == 0 || (board[x1-1][y1].kind >= flag1 &&
     board[x1-1][y1].kind <= flag2))
  {
    if (x1 != 0)
          tmp = insert (tmp,x1-1,y1);  //panw
  }
  if (board[x1-1][y1+1].kind == 0 || (board[x1-1][y1+1].kind >= flag1 &&
     board[x1-1][y1+1].kind <= flag2))
  {
    if (x1 != 0 && y1 != N-1)
          tmp = insert (tmp,x1-1,y1+1);  //panw deksia
  }
  if (board[x1][y1+1].kind == 0 || (board[x1][y1+1].kind >= flag1 &&
     board[x1][y1+1].kind <= flag2))
  {
    if (y1 != N-1)
          tmp = insert (tmp,x1,y1+1);  //deksia
  }
  if (board[x1+1][y1+1].kind == 0 || (board[x1+1][y1+1].kind >= flag1 &&
     board[x1+1][y1+1].kind <= flag2))
  {
    if (x1 != N-1 && y1 != N-1)
          tmp = insert (tmp,x1+1,y1+1);  //katw deksia
  }
  board[x1][y1].ptr = tmp;
  if(tmp == NULL)
    return -1;
  return 0;
}
```

moveList *moveRook (int x1,int y1,int flag1,int flag2,moveList *tmp);

Η συνάρτηση moveRook κοιτάζει το προς κίνηση πιόνι το οποίο είναι ο πύργος και φτιάχνει μια λίστα με όλες τις δυνατές κινήσεις. Οι κινήσεις που μπορεί να κάνει ο πύργος είναι πάνω, κάτω, αριστερά, δεξιά. Ανάλογα με την κίνηση που θα ζητηθεί κάθε φορά, ελέγχει αν μπορεί να γίνει η κίνηση. Αν μπορεί να γίνει, θα την αποθηκεύσει στη λίστα με τις δυνατές κινήσεις, καθώς επίσης και στη

συγκεκριμένηθέση στη σκακιέρα και θα επιστρέψει τον κόμβο με την κίνηση, αλλιώς θα επιστρέψει NULL. Αυτό γίνεται, γιατί τις κινήσεις αυτές τις κάνει και η βασίλισσα, οπότε θα κληθεί και εκεί, γι'αυτό και χρειαζόμαστε τον κόμβο αυτό.

```
moveList *moveRook (int x1,int y1,int flag1,int flag2,moveList *tmp) /*flag1, flag2 oi
*/
{                                                                    /*times antipalwn*/
  int i;
  for (i = y1+1; i < N; i++)  //deksia kinisi
  {
    if( (player == white) && (board[x1][i].kind >= 100 && board[x1][i].kind <= 105) )
          break;
    if( (player == black) && (board[x1][i].kind >= 106 && board[x1][i].kind <= 111) )
          break;
    if ( (board[x1][i].kind == 0) || (board[x1][i].kind >= flag1 &&
           board[x1][i].kind <= flag2))
    {
          if(board[x1][i].kind >= flag1 && board[x1][i].kind <= flag2)
          {
            tmp = insert (tmp,x1,i);
            break;
          }
          else
            tmp = insert (tmp,x1,i);
    }

  }
  for (i = x1+1; i < N; i++)  //katw kinisi
  {
    if( (player == white) && (board[i][y1].kind >= 100 && board[i][y1].kind <= 105) )
          break;
    if( (player == black) && (board[i][y1].kind >= 106 && board[i][y1].kind <= 111) )
          break;
    if ((board[i][y1].kind == 0) || (board[i][y1].kind >= flag1 &&
           board[i][y1].kind <= flag2))
    {
          if(board[i][y1].kind >= flag1 && board[i][y1].kind <= flag2)
          {
            tmp = insert (tmp,i,y1);
            break;
          }
          else
            tmp = insert (tmp,i,y1);
    }
  }
  for (i = y1-1; i >= 0; i--)  //aristeri kinisi
  {
```

```
    if( (player == white) && (board[x1][i].kind >= 100 && board[x1][i].kind <= 105) )
          break;
    if( (player == black) && (board[x1][i].kind >= 106 && board[x1][i].kind <= 111) )
          break;
    if ((board[x1][i].kind == 0) || (board[x1][i].kind >= flag1 &&
           board[x1][i].kind <= flag2))
    {
          if(board[x1][i].kind >= flag1 && board[x1][i].kind <= flag2)
          {
            tmp = insert (tmp,x1,i);
            break;
          }
          else
            tmp = insert (tmp,x1,i);
    }
  }
  for (i = x1-1; i >= 0; i--)  //panw kinisi
  {
    if( (player == white) && (board[i][y1].kind >= 100 && board[i][y1].kind <= 105) )
          break;
    if( (player == black) && (board[i][y1].kind >= 106 && board[i][y1].kind <= 111) )
          break;

    if ((board[i][y1].kind == 0) || (board[i][y1].kind >= flag1 &&
           board[i][y1].kind <= flag2))
    {
          if(board[i][y1].kind >= flag1 && board[i][y1].kind <= flag2)
          {
            tmp = insert (tmp,i,y1);
            break;
          }
          else
            tmp = insert (tmp,i,y1);
    }
  }
  board[x1][y1].ptr = tmp;
  return board[x1][y1].ptr;
}
```

moveList *moveGeneral (int x1,int y1,int flag1,int flag2,moveList *tmp);

Η συνάρτηση moveGeneral κοιτάζει το προς κίνηση πιόνι το οποίο είναι ο αξιωματικός και φτιάχνει μια λίστα με όλες τις δυνατές κινήσεις. Οι κινήσεις που μπορεί να κάνει ο αξιωματικός είναι πάνω δεξία, κάτω δεξιά, πάνω αριστερά, κάτω αριστερά. Ανάλογα με την κίνηση που θα ζητηθεί κάθε φορά, ελέγχει αν μπορεί να γίνει η κίνηση. Αν μπορεί να γίνει, θα την αποθηκεύσει στη λίστα με τις δυνατές

κινήσεις, καθώς επίσης και στη συγκεκριμένηθέση στη σκακιέρα και θα επιστρέψει τον κόμβο με την κίνηση, αλλιώς θα επιστρέψει NULL. Αυτό γίνεται, γιατί τις κινήσεις αυτές τις κάνει και η βασίλισσα, οπότε θα κληθεί και εκεί, γι'αυτό και χρειαζόμαστε τον κόμβο αυτό.

```
moveList *moveGeneral (int x1,int y1,int flag1,int flag2,moveList *tmp)
{
  int i,j;

  i = x1 + 1;
  j = y1 - 1;
  while(i<N && j>=0)  //katw aristera
  {
    if( (player == white) && (board[i][j].kind >= 100 && board[i][j].kind <= 105) )
          break;
    if( (player == black) && (board[i][j].kind >= 106 && board[i][j].kind <= 111) )
          break;

    if (board[i][j].kind == 0 || (board[i][j].kind >= flag1 &&
                              board[i][j].kind <= flag2))
    {
          if(board[i][j].kind >= flag1 && board[i][j].kind <= flag2)
          {
            tmp = insert (tmp,i,j);
            break;
          }
          else
            tmp = insert (tmp,i,j);
    }
    i++;
    j--;
  }

  i = x1 - 1;
  j = y1 - 1;
  while(i>=0 && j>=0) //panw aristera
  {
    if( (player == white) && (board[i][j].kind >= 100 && board[i][j].kind <= 105) )
          break;
    if( (player == black) && (board[i][j].kind >= 106 && board[i][j].kind <= 111) )
          break;

    if (board[i][j].kind == 0 || (board[i][j].kind >= flag1 &&
                              board[i][j].kind <= flag2))
    {
          if(board[i][j].kind >= flag1 && board[i][j].kind <= flag2)
          {
```

```
            tmp = insert (tmp,i,j);
            break;
          }
          else
            tmp = insert (tmp,i,j);
    }
    i--;
    j--;
}

i = x1 - 1;
j = y1 + 1;
while(i>=0 && j<N)  //panw deksia
{
    if( (player == white) && (board[i][j].kind >= 100 && board[i][j].kind <= 105) )
          break;
    if( (player == black) && (board[i][j].kind >= 106 && board[i][j].kind <= 111) )
          break;

    if (board[i][j].kind == 0 || (board[i][j].kind >= flag1 &&
                                  board[i][j].kind <= flag2))
    {
          if(board[i][j].kind >= flag1 && board[i][j].kind <= flag2)
          {
            tmp = insert (tmp,i,j);
            break;
          }
          else
            tmp = insert (tmp,i,j);
    }
    i--;
    j++;
}
i = x1 + 1;
j = y1 + 1;
while(i<N && j<N)  //katw deksia
{
    if( (player == white) && (board[i][j].kind >= 100 && board[i][j].kind <= 105) )
          break;
    if( (player == black) && (board[i][j].kind >= 106 && board[i][j].kind <= 111) )
          break;

    if (board[i][j].kind == 0 || (board[i][j].kind >= flag1 &&
                                  board[i][j].kind <= flag2))
    {
          if(board[i][j].kind >= flag1 && board[i][j].kind <= flag2)
          {
```

```
            tmp = insert (tmp,i,j);
            break;
          }
          else
            tmp = insert (tmp,i,j);
      }
      i++;
      j++;
    }
    board[x1][y1].ptr = tmp;
    return board[x1][y1].ptr;
}
```

int moveQueen (int x1,int y1,int flag1,int flag2,moveList *tmp);

Η συνάρτηση moveQueen κοιτάζει το προς κίνηση πιόνι το οποίο είναι η βασίλισσα και φτιάχνει μια λίστα με όλες τις δυνατές κινήσεις. Οι κινήσεις που μπορεί να κάνει ο βασίλισσα είναι πάνω δεξία, κάτω δεξιά, πάνω αριστερά, κάτω αριστερά, πάνω, κάτω, αριστερά, δεξιά. Τις μισές από τις κινήσεις αυτές μπορεί να τις κάνει ο πύργος και τις υπόλοιπες ο αξιωματικός. Οπότε, καλεί τις δυο αυτές συναρτήσεις που αναλύσαμε παραπάνω και ελέγχει την επιστρεφόμενη τιμή. Αν είναι NULL, σημαίνει ότι η συγκεκριμένη κίνηση δεν μπορεί να γίνει και επιστρέφει -1, αλλιώς την εισάγει στη σκακιέρα στη συγκεκριμένη θέση και επιστρέφει 0.

```
int moveQueen (int x1,int y1,int flag1,int flag2,moveList *tmp)
{
  tmp = moveRook (x1,y1,flag1,flag2,tmp);
  tmp = moveGeneral (x1, y1, flag1, flag2, tmp);

  board[x1][y1].ptr = tmp;
  if(tmp == NULL)
    return -1;
  return 0;
}
```

Η συνάρτηση moveCtrl με τη βοήθεια των συναρτήσεων που αναλύσαμε ακριβώς παραπάνω για τις κινήσεις του βασιλιά, της βασίλισσας, του πύργου, του αλόγου και του αξιωματικού, ελέγχει (ανάλογα με την επιστρεφόμενη τιμή τους) αν μπορεί να γίνει η συγκεκριμένη κίνηση και αν είναι επιτρεπτή, τότε πραγματοποιεί τη μετακίνηση του πιονιού που έχει ζητηθεί. Κάνει, επίσης, έλεγχο για τα απλά πιόνια και πραγματοποιεί αντίστοιχη μετακίνηση, αν αυτή επιτρέπεται. Αν έγινε η μετακίνηση, θα επιστρέψει 0, αλλιώς -1.

```
int moveCtrl(int x1, int y1)
{
  moveList *tmp = NULL;
```

```
  int mk;
  if(board[x1][y1].kind == 0)
    return -1;
  else if((board[x1][y1].kind >= 106 && board[x1][y1].kind <= 111) && (player ==
white))
    return -1;
  else if((board[x1][y1].kind >= 100 && board[x1][y1].kind <= 105) && (player ==
black))
    return -1;
  else
  {
    if(board[x1][y1].kind == w_pawn)  /*white pawns*/
    {
          if (player == white)
          {
            if(board[x1+1][y1].kind != 0 && (board[x1+1][y1-1].kind >= 100 &&
              board[x1+1][y1-1].kind <= 105) && (board[x1+1][y1+1].kind >= 100
              && board[x1+1][y1-1].kind <= 105))
              return -1;
          }
          if (player == black)
          {
            if(board[x1-1][y1].kind != 0 && board[x1-1][y1-1].kind != 0 &&
              board[x1-1][y1+1].kind != 0)
              return -1;
          }
          if (x1 == 1 && board[x1+2][y1].kind == 0)
            tmp = insert(tmp, x1+2, y1);
          if(board[x1+1][y1].kind == 0)  /*kinhsh mprosta*/
            tmp = insert(tmp, x1+1, y1);
          if( (board[x1+1][y1-1].kind>=106 && board[x1+1][y1-1].kind<=111) && y1 !=
0)
            tmp = insert(tmp, x1+1, y1-1); /*diagwnia aristerh kinhsh*/
          if( (board[x1+1][y1+1].kind>=106 && board[x1+1][y1+1].kind<=111) && y1
!= N-1)
            tmp = insert(tmp, x1+1, y1+1); /*diagwnia deksia kinhsh*/
          board[x1][y1].ptr = tmp;
    }
    if(board[x1][y1].kind == b_pawn)  //black pawns
    {
          if (x1 == 6 && board[x1-2][y1].kind == 0)
            tmp = insert(tmp, x1-2, y1);
          if(board[x1-1][y1].kind == 0)  //kinhsh mprosta
            tmp = insert(tmp, x1-1, y1);
          if( (board[x1-1][y1-1].kind>=100 && board[x1-1][y1-1].kind<=105) && y1 !=
0)
            tmp = insert(tmp, x1-1, y1-1); //diagwnia aristerh kinhsh
```

```
        if( (board[x1-1][y1+1].kind>=100 && board[x1-1][y1+1].kind<=105) && y1 !=
N-1)
          tmp = insert(tmp, x1-1, y1+1); //diagwnia deksia kinhsh
        board[x1][y1].ptr = tmp;
    }
    if (board[x1][y1].kind == w_knight)  //white knight
        return moveKnight (x1,y1,106,111,tmp);
    if (board[x1][y1].kind == b_knight)  //black knight
        return moveKnight (x1,y1,100,105,tmp);
    if (board[x1][y1].kind == w_king)  //white king
    {
        if (moveKing (x1,y1,106,111,tmp) == -1)  //an den mporei na kinithei
          return -1;
    }
    if (board[x1][y1].kind == b_king)  //black king
        return moveKing (x1,y1,100,105,tmp);
    if (board[x1][y1].kind == w_rook)  //white rook
    {
        tmp = moveRook (x1,y1,106,111,tmp);
        if(tmp == NULL)
          return -1;
    }
    if (board[x1][y1].kind == b_rook)  //black rook
    {
        tmp = moveRook (x1,y1,100,105,tmp);
        if(tmp == NULL)
          return -1;
    }
    if (board[x1][y1].kind == w_queen)  //white queen
        return moveQueen (x1,y1,106,111,tmp);
    if (board[x1][y1].kind == b_queen)  //black queen
        return moveQueen (x1,y1,100,105,tmp);

    if (board[x1][y1].kind == w_general)  //white general
    {
        tmp = moveGeneral (x1,y1,106,111,tmp);
        if (tmp == NULL)
          return -1;
    }
    if (board[x1][y1].kind == b_general)  //black general
    {
        tmp = moveGeneral (x1,y1,100,105,tmp);
        if (tmp == NULL)
          return -1;
    }
    return 0;
  }
```

```
}
```

int check(int x1, int y1, int x2, int y2);

Η συνάρτηση check ελέγχει αν η κίνηση που επέλεξε ο χρήστης είναι επιτρεπτή, δηλαδή αν ανήκει στη λίστα με όλες τις επιτρεπόμενες κινήσεις. Αν επιτρέπεται, θα επιστρέψει 0, αλλιώς -1.

```
int check(int x1, int y1, int x2, int y2)
{
  moveList *found = NULL, *tmp;
  tmp = board[x1][y1].ptr;
  if(tmp == NULL)   /*den uparxei diathesimi kinisi*/
    return -1;
  else {
    while(tmp)
    {
          if( (tmp->a == x2) && (tmp->b == y2) )
            return 0;
          tmp = tmp->next;
    }
    return -1;
  }
}
```

void makeMove(int x1, int y1);

Η συνάρτηση makeMove πραγματοποιεί την κίνηση. Ανάλογα με την επιστρεφόμενη τιμή της συνάρτησης moveCtrl, ζητάει από το χρήστη να δώσει έγκυρη θέση προέλευσης και ανάλογα με την επιστρεφόμενη τιμή της συνάρτησης check, ζητάει από το χρήστη να δώσει έγκυρη θέση προορισμού. Αν οι τιμές είναι σωστές, τότε αλλάζει την κατάσταση της σκακιέρας με τις νέες τιμές. Η συνάρτηση δεν επιστρέφει τίποτα.

```
void makeMove(int x1, int y1)
{
  int flag, x2, y2;
  while ( (flag = moveCtrl(x1, y1)) < 0 )
  {
    printf("Edwses mi egkuri thesi. Prospathise ksana: ");
    scanf("%d %d", &x1, &y1);
  }
  printf("Dwse teliki thesi: ");
  scanf("%d %d", &x2, &y2);
```

```
  while( (flag = check(x1, y1, x2, y2)) < 0)
  {
    printf("Edwses mi egkuro proorismo. Prospathise ksana: ");
    scanf("%d %d", &x2, &y2);
  }
  board[x2][y2].kind = board[x1][y1].kind;
  board[x2][y2].ptr = NULL;
  board[x1][y1].kind = 0;
  board[x1][y1].ptr = NULL;
  return ;
}
```

void movePc (int x,int y);

Η συνάρτηση movePc αναλύει όλες τις δυνατές κινήσεις που μπορεί να κάνει ο υπολογιστής δοσμένης θέσης προέλευσης (x, y). Η συνάρτηση δεν επιστρέφει τίποτα.

```
void movePc (int x,int y)
{
  moveList *tmp = board[x][y].ptr;
  pc *cur;
  pc *new;
  while (tmp != NULL)
  {
    new = (pc *)malloc (sizeof(pc));
    new->pc_from1 = x;
    new->pc_from2 = y;
    new->new_val = tmp->v;
    new->pc_to1 = tmp->a;
    new->pc_to2 = tmp->b;
    new->pc_kind = board[x][y].kind;
    if (head == NULL)
          head = new;
    else
    {
          cur = head;
          while (cur->next != NULL)
            cur = cur->next;
          cur->next = new;
          new->next = NULL;
    }
    tmp = tmp->next;
  }
}
```

void ins_pl(humanList *new);

void pl_moves(int x, int y);

Οι συναρτήσεις ins_pl και pl_moves υπολογίζουν όλες τις δυνατές κινήσεις που νπορεί να κάνει ο παίκτης και τις αποθηκεύει στην αντίστοιχη λίστα του, ανάλογα με το πιόνι που θέλει να μετακινήσει κάθε φορά. Παίρνει όλες τις δυνατές θέσεις που υπάρχουν στο συγκεκριμένο τετράγωνο στη σκακιέρα και τις περνάει στη λίστα με όλες τις δυνατές κινήσεις που μπορεί να κάνει το συκεκριμένο πιόνι. Οι συναρτήσεις δεν επιστρέφουν κάτι.

```
void ins_pl(humanList *new)
{
  humanList *cur;
  if (pl_head == NULL)
    pl_head = new;
  else
  {
    cur = pl_head;
    while (cur->next != NULL)
          cur = cur->next;
    cur->next = new;
  }
  return ;
}

void pl_moves(int x, int y)
{
  moveList *tmp = board[x][y].ptr;
  humanList *new;
  while (tmp != NULL)
  {
    new = (humanList *)malloc (sizeof(humanList));
    if (board[x][y].kind == w_pawn)
    {
          if (y != 0 && x != N-1)
          {
            new = (humanList *)malloc (sizeof(humanList));
            new->pl_from1 = x;
            new->pl_from2 = y;
            new->pl_to1 = x+1;
            new->pl_to2 = y-1;
            new->pl_kind = w_pawn;
            ins_pl(new);
          }
          if (y != N-1 && x != N-1)
          {
            new = (humanList *)malloc (sizeof(humanList));
            new->pl_from1 = x;
```

```
            new->pl_from2 = y;
            new->pl_to1 = x+1;
            new->pl_to2 = y+1;
            new->pl_kind = w_pawn;
            ins_pl(new);
          }
    }
    else
    {
          x = tmp->a;
          y = tmp->b;
          moveCtrl (x,y);
          new->pl_from1 = x;
          new->pl_from2 = y;
       new->pl_to1 = tmp->a;
          new->pl_to2 = tmp->b;
          ins_pl(new);
    }
    tmp = tmp->next;
  }
}
```

void myFree();

Η συνάρτηση myFree απελευθερώνει το χώρο που καταλαμβάνουν οι λίστες humanList και pcList κάθε φορά που πραγματοποιείται μια κίνηση, για να μπορεί να υπολογίσει τις νέες κινήσεις από την αρχή και να μην καταλαμβάνουν πολύ χώρο στη μνήμη του υπολογιστή. Η συνάρτηση δεν επιστρέφει κάτι.

```
void myFree()
{
  humanList *tmp = pl_head;
  pc *lol = head;
  while(tmp != NULL)  /*kanw free th lista*/
  {
    tmp = tmp->next;
    //free(pl_head)
    pl_head = NULL;
    pl_head = tmp;
  }

  while(lol != NULL)     /*kanw free th lista*/
  {
    lol = lol->next;
    //free(head);
    head =NULL;
```

```
    head = lol;
  }


  return ;
}
```

int main();

Η συνάρτηση main χρησιμοποιείται για να "τρέξει" το πρόγραμμα. Καλεί όλες τις απαραίτητες συναρτήσεις και συγκεκριμένα, ζητάει από το χρήστη να δώσει τη θέση του πιονιού που θέλει να μετκινήσει, αλλά και τη θέση προορισμού του. Αφού δημιουργήσει τις λίστες με όλες τις πιθανές κινήσεις, καλεί τις συναρτήσεις και στη συνέχεια επιλέγεται η καλύτερη κίνηση για τον υπολογιστή ανάλογα με την κίνηση που ενδέχεται να κάνει ο χρήστης στην επόμενη κίνησή του. Επίσης, ελέγχει και βρίσκει το νικητή του παιχνιδιού.

```
int main()
{
  int i, x1, y1, x2, y2, flag1, flag2,colour,color,k,l,value,valueking;
  moveList *tmp;
  pc *tmp2;
  humanList *tmp3;
  initialize();
  print();
  player = white;    /*arxika paizoun ta aspra*/
  color = 0,colour = 0;
  while(1)
  {
    if(x1==-1)
          break;
    if(player == white)
          printf("\nPlayer : white\n");
    else
         printf("\nPlayer : black\n");
    if (player == white)
    {
          printf("Dwse arxikh 8esh: ");
          scanf("%d %d", &x1, &y1);

      makeMove(x1, y1);
    }
          //system("clear");

    for (k = 0; k < N; k++)
    {
```

```
        for (l = 0 ; l < N; l++)
        {
          if(board[k][l].kind == 0)
            continue;
          tmp = board[k][l].ptr;
          if (board[k][l].kind == b_king)
            color++;
          if (board[k][l].kind >= 106 && board[k][l].kind <= 111)
          {
            moveCtrl (k,l);
            movePc (k,l);
          }
          if (board[k][l].kind >= 100 && board[k][l].kind <= 105)
          {
            moveCtrl (k,l);
            pl_moves(k, l);  //ftiaxnei ti lista me oles tis dunates kiniseis tou human
          }
        }
    }
    if (color == 0)
    {
        system ("clear");
        print ();
        printf ("Exase o upologistis\n");
        return 0;
    }
    else
        color = 0;

    if(player == black)
    {
        tmp2 = head;
        while (tmp2 != NULL)
        {
          Evaluate (tmp2);
          tmp2 = tmp2->next;
        }
        printf ("\n");
        tmp2 = head;
        value = tmp2->new_val;
        while (tmp2 != NULL)  //minimax
        {
          printf("->val=%d - kind=%d - %d,%d : %d,%d\n", tmp2->new_val, tmp2-
>pc_kind, tmp2->pc_from1, tmp2->pc_from2, tmp2->pc_to1, tmp2->pc_to2);
          if (kingThreatened == 0)
          {
            if (-tmp2->new_val > -value)
```

```
                value = tmp2->new_val;
            }
            else
            {
              if (-tmp2->new_val > -valueking)
                valueking = tmp2->new_val;
            }
            tmp2 = tmp2->next;
          }
          printf("----------------------->%d\n", value);
          tmp2 = head;
          while (tmp2 != NULL)
          {
            if (tmp2->new_val == value && kingThreatened == 0)
              break;
            if (tmp2->new_val == valueking && kingThreatened == 1)
              break;
            tmp2 = tmp2->next;
          }
          board[tmp2->pc_to1][tmp2->pc_to2].kind  =  board[tmp2->pc_from1][tmp2-
>pc_from2].kind;
          board[tmp2->pc_from1][tmp2->pc_from2].ptr = NULL;
          board[tmp2->pc_from1][tmp2->pc_from2].kind = 0;
          board[tmp2->pc_to1][tmp2->pc_to2].ptr = NULL;
          print ();
      }
      for (k = 0; k < N; k++)  //elegxw an exase o paiktis
      {
          for (l = 0; l < N; l++)
          {
            if (board[k][l].kind == w_king)
              colour++;
          }
      }
      if (colour == 0)
      {
          system ("clear");
          print ();
          printf ("Exases\n");
          return 0;
      }
      else
                colour = 0;
    myFree();
      if(player == white)
          player = black;
      else if (player == black)
```

```
            player = white;
    }
    return 0;
}
```

6 Αποτελέσματα

Το συγκεκριμένο πρόγραμμα, αφού υλοποιήθηκε και έγιναν δοκιμές για το σωστό τρόπο λειτουργίας του, είναι ένα πλήρες πρόγραμμα παιχνιδιού σκακιού. Εκτός από τα πλαίσια της πτυχιακής εργασίας, το συγκεκριμένο πρόγραμμα μπορεί να χρησιμοποιηθεί από άτομα οποιασδήποτε ηλικίας και οποιουδήποτε επιπέδου. Μπορεί να χρησιμοποιηθεί σε ατομικό ή ομαδικό επίπεδο με τη μορφή τουρνουά. Μπορεί να χρησιμοποιηθεί για διασκέδαση ή ακόμα και για εκμάθηση από μικρής ηλικίας άτομα για να μάθουν να σκέφτονται σωστά και ορθά. Δεδομένου ότι ο υπολογιστής χρησιμοποιεί ένα συγκεκριμένο αλγόριθμο για να παίξει το παιχνίδι σκάκι και αλληλεπιδρά ανάλογα με τις κινήσεις του χρήστη που το χρησιμοποιεί, θα μπορούσαν να κρατηθούν στατιστικά στοιχεία αν το πρόγραμμα χρησιμοποιούταν σε σχολεία. Θα μπορούσαν να διαπιστώσουν το επίπεδο σκέψης του κάθε μαθητή, αλλά και να τους βοηθήσουν να αναπτύξουν ένα καλύτερο τρόπο σκέψης. θα μπορούσε ακόμα να προωθήσει τη σπουδαιότητα της πληροφορικής και τον τρόπο με τον οποίο μπορεί κανείς χωρίς να είναι υποχρεωμένος να βρει αντίπαλο να μπορεί να παίξει και να περάσει ευχάριστα την ώρα του.

6.1 Συμπεράσματα

Το πρόγραμμα αυτό, μετά την ολοκλήρωσή του, είναι ένα πλήρες πρόγραμμα με έναν πολύ γνωστό αλγόριθμο αξιολόγησης και μπορεί να υποστηριχθεί άνετα από οποιονδήποτε χρήστη και με την τεχνητή νοημοσύνη να αλληλεπιδρά με τις εκάστοτε κινήσεις του χρήστη. Ο υπολογιστής είναι πάντα “ένα βήμα μπροστά”, διότι υπολογίζει την καλύτερη δυνατή κίνηση που μπορεί να κάνει σύμφωνα με τις καλύτερες κινήσεις που μπορεί να επιλέξει να πραγματοποιήσει ο χρήστης.

6.2 Μελλοντική εργασία και επεκτάσεις

Το συγκεκριμένο πρόγραμμα θα μπορούσε να επεκταθεί μελλοντικά για να μπορεί να αναφέρεται σε περισσότερους τομείς. Μια σημαντική επέκταση που θα μπορούσε να γίνει είναι να παρουσιάζονταν κάποιες από τις πιο σωστές κινήσεις που θα μπορούσε να κάνει ο χρήστης, έτσι ώστε να μπορεί, όχι απλά να παίζει, αλλά και να μαθαίνει πώς να παίζει σωστά σκάκι. Μια άλλη επέκταση που θα μπορούσε να γίνει είναι να προστιθόταν χρόνος που θα είχε στη διάθεσή του ο χρήστης να κάνει την κάθε κίνηση. Εκτός από τις προαναφερθείσες επεκτάσεις, το πρόγραμμα αυτό μπορεί να δεχθεί πάρα πολλές παραλλαγές, ανάλογα με τις ανάγκες και στο κοινό το οποίο αναφέρεται.

7 Βιβλιογραφία

1)Ding-Zhu Du,Minimax and Application(Monconvex Optimization and its applications)
2)Diego Rasskin-Gutman,Chess Metaphors(Artificial Intelligence and the human mind)
3)Mario Carpo,The Alphabet and the Algorithm
4)Holger Hoos,Stochastic Local Search:Foundations &Applications(The Morgan Kaufmann Series in Artificial Intelligence)Sep 30,2004
5)Wolfgang Banzhaf,Genetic Programming:An Introduction(The Morgan Kaufmann Series in Artificial)
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(Computer Science)
8)E.Tyugu,Algorithms and Architectures of Artificial Intelligence(Frontiers in Artificial Intelligence and Applications)
9)Jaume Bacardit,Will Browne,Jan Drugowitsch,Mausilla Butz,Learning Classifier Systems(Lecture notes in Artificial Intelligence)
10)Graham Williams,Data Mining:Theory,Methodology,Techniques and Applications
11)Barry O'Sullivan,Recent Advances in Constraints:14th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming,CSCLP 2009/ Lecture notes inArtificial Intelligence (Mar 31,2011)
12)Gordon McCall,Advances in Artificial Intelligence:11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence,AI'96,Toronto/ Lecture notes inArtificial Intelligence (May 8,1996)