ΚΕΦΑΛΑΙΟ 11: Τεχνικές Λογικού Προγραμματισμού για Επίλυση Προβλημάτων

Λέξεις Κλειδιά:
Λογική και Αλγοριθμική, Προβλήματα Παραγωγής και Δοκιμής, Προβλήματα Γράφων. Αναπαράσταση και αναζήτηση σε γράφους.

Περίληψη
Το κεφάλαιο αποτελεί επιστέγασμα όλων των προηγουμένων, καθώς παραθέτει παραδείγματα τεχνικών αναπαράστασης και επίλυσης σύνθετων προβλημάτων σε Prolog. Αντιπαρατίθεται η λογική προσέγγιση σε ένα πρόβλημα με την αντίστοιχη αλγοριθμική. Κλασσικές τεχνικές όπως εκείνη της "Παραγωγής και Δοκιμής" (Generate and Test) παρουσιάζονται μέσω κλασικών παραδειγμάτων, όπως η εύρεση λύσης σε ακέραιες εξισώσεις και άλλα παρόμοια μη-αριθμητικά προβλήματα ικανοποίησης περιορισμών. Παρουσιάζεται επίσης η πλέον δημοφιλής αναπαράσταση προβλημάτων της Τεχνητής Νοημοσύνης, η αναπαράσταση μέσω γράφων, καθώς και η υλοποίηση τυφλών και ευριστικών αλγορίθμων αναζήτησης, οι οποίοι οδηγούν στην επίλυση σύνθετων προβλημάτων.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

- κατανοήσει τις βασικές διαφορές μεταξύ λογικής και αλγοριθμικής προσέγγισης,
- αντιληφθεί την βασική αλγοριθμική για την εύρεση διαδρομής σε ένα γράφο,
- ορίζει προβλήματα χώρου καταστάσεων στην Τεχνητή Νοημοσύνη,
- καταλάβει τις βασικές αρχές αναζήτησης και επίλυσης προβλημάτων στην Τεχνητή Νοημοσύνη,
- υλοποιήσει επιλύσεις με Παραγωγή και Δοκιμή και Αλγόριθμους Αναζήτησης Πρώτα σε Βάθος, Πρώτα σε Πλάτος και Πρώτα στο Καλύτερο.

Παράδειγμα κίνητρο

Η αυτόματη επίλυση προβλημάτων αποτελεί βασικό στόχο της Τεχνητής Νοημοσύνης (ΤΝ). Τα προβλήματα που καλείται να λύσει η ΤΝ ώστε το αποτέλεσμα (η λύση) να είναι καλύτερο ή τουλάχιστον όσο καλό είναι το αποτέλεσμα που παράγει ο άνθρωπος, είναι εξαιρετικής πολυπλοκότητας. Το μεγάλο πλεονέκτημα είναι ότι μία υπολογιστική μηχανή έχει πολύ μεγάλη υπολογιστική ισχύ, εκτελώντας 10^{15} εντολές ανά δευτερόλεπτο.

Ένα τέτοιο πρόβλημα είναι το λεγόμενο πρόβλημα του πλανόδιου πωλητή (Travelling Salesperson Problem ή TSP). Το πρόβλημα παριστάνεται με ένα γράφο που αποτελείται από Ν κόμβους οι οποίοι αντιπροσωπεύουν αντίστοιχα Ν τοποθεσίες. Κάποιος πρέπει να ξεκινήσει από έναν κόμβο, να επισκεφτεί μία φορά την κάθε τοποθεσία και να επιστέψει στην αρχική με το λιγότερο συνολικό κόστος, διανύοντας δηλαδή την ελάχιστη δυνατή απόσταση. Θεωρούμε ότι όλοι οι κόμβοι ενώνονται μεταξύ τους (πλήρης γράφος). Το πρόβλημα έχει πολλές εφαρμογές σε σειριακό σχεδιασμό ενεργειών με κόστος για κάθε ενέργεια. Οι πιθανές διαδρομές είναι (N-1)! και μία από αυτές είναι η βέλτιστη λύση. Το πρόβλημα ανήκει στην κατηγορία προβλημάτων με λύση μη-πολυωνυμικού χρόνου (NP-complete). Η εξαντλητική εξέταση (N-1)! μονοπατιών για Ν πόλεις είναι ένας πολύ μεγάλος αριθμός ακόμα και για μικρό αριθμό πόλεων. Για παράδειγμα, αν N=20, τότε υπάρχουν $19!=1.216\times10^{17}$ λύσεις, που σημαίνει ότι ακόμη και ένας υπολογιστής που εξετάζει 10^6 λύσεις το δευτερόλεπτο θα χρειαζόταν 385 χρόνια για να βρει τη βέλτιστη λύση.

Μονοπάτι Μάθησης

Οι ενότητες αυτού του κεφαλαίου χωρίζονται σε δύο ομάδες, την 11.1, την 11.2, την 11.3 που αποτελούν κλασσικές εφαρμογές και αναδεικνύουν βασικές τεχνικές του λογικού προγραμματισμού, και τις ενότητες 11.4 και 11.5 που δείχνουν βασικούς αλγορίθμους επίλυσης προβλημάτων. Οι τρεις πρώτες μπορούν να διαβαστούν ανεξάρτητα. Πριν την κατανόηση των 11.4 και 11.5 απαιτείται η κατανόηση της εύρεσης διαδρομής σε γράφους που περιγράφεται αναλυτικά στην ενότητα 11.3.

11.1 Λογική εναντίον Αλγοριθμικής

Ένα από τα σημαντικότερα θέματα στον προγραμματισμό είναι η αποτελεσματικότητα (efficiency) της εκτέλεσης ενός προγράμματος. Η αποτελεσματικότητα αυτή μπορεί να μετρηθεί με πολλούς παράγοντες, αλλά το πιο σημαντικό από αυτά είναι ο χρόνος εκτέλεσης και οι απαιτήσεις μνήμης που καταναλώνεται κατά τη διάρκεια της εκτέλεσης. Έχουν γίνει σημαντικές προσπάθειες για τη δημιουργία αποτελεσματικών μεταγλωττιστών και βελτιστοποιήσεων για όλες τις γλώσσες προγραμματισμού που θα μειώνουν το χρόνο εκτέλεσης και το χώρο διαχείρισης. Η Prolog δεν αποτελεί εξαίρεση στην ανάγκη ταχύτερης εκτέλεσης. Αυτή η ανάγκη ενισχύεται από το γεγονός ότι υπάρχει μια αντιστρόφως ανάλογη σχέση ανάμεσα στη λογική έκφραση ενός προβλήματος και σε κάποιον διαδικαστικό αλγόριθμο που το λύνει. Όσο πιο δηλωτικός είναι ο κώδικας τόσο ο χρόνος εκτέλεσης αυξάνεται, και όσο πιο διαδικαστικός είναι τόσο ο χρόνος εκτέλεσης μειώνεται. Ένα πρώτο παράδειγμα είδαμε στο Κεφάλαιο 7, με την υλοποίηση του κατηγορήματος εύρεσης μεγίστου μιας λίστας (max/2). Θα εξεταστεί εδώ αναλυτικότερα με μια μελέτη περίπτωσης, την ταξινόμηση αριθμών μέσα σε μία λίστα.

Ταξινόμηση Λίστας: η Λογική Προσέγγιση

Ταξινόμηση μία λίστας αριθμών είναι το πρόβλημα εύρεσης μίας μετάθεσης (permutation) των αριθμών μέσα στην λίστα, έτσι ώστε αυτοί να εμφανίζονται σε αύξουσα (ή φθίνουσα) σειρά. Στην Prolog αυτό μπορεί να εκφραστεί ως ο ορισμός:

```
nsort(List, PermList): -
  permutation(List, PermList),
  is_sorted(PermList).
```

όπου το κατηγόρημα permutation/2 παράγει όλες τις πιθανές μεταθέσεις μιας λίστας, και is_sorted/2 ένα κατηγόρημα που εξετάζει κατά πόσον μία λίστα από αριθμούς είναι ταξινομημένη. Το permutation/2 είναι συνήθως ενσωματωμένο κατηγόρημα στην Prolog ενώ το is_sorted/1 μπορεί εύκολα να οριστεί αναδρομικά (μία λίστα είναι ταξινομημένη αν η ουρά της είναι ταξινομημένη και το πρώτο στοιχείο της είναι μεγαλύτερο από το δεύτερο στοιχείο).

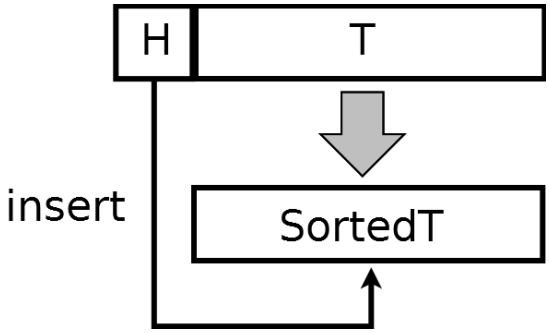
Ο παραπάνω ορισμός αποτελεί την ουσία του δηλωτικού προγραμματισμού, με την έννοια ότι δείχνει "τί" είναι η ταξινόμηση και όχι "πώς" γίνεται. Πρακτικά, ο μηχανισμός εκτέλεσης της Prolog δημιουργεί μεταθέσεις τις οποίες το κατηγόρημα is_sorted εξετάζει αν είναι ταξινομημένες. Αν κάποια δεν είναι, μία νέα μετάθεση δημιουργείται μέσω της οπισθοδρόμησης για να εξεταστεί και αυτή με τη σειρά της, με τη διαδικασία να τελειώνει όταν βρεθεί η κατάλληλη λίστα. Η διαδικασία αυτή ονομάζεται “παραγωγή και δοκιμής” (generate and test) ή “προσπάθεια και αποτυχία” (trial and error) στην οποία θα αναφερθούμε εκτενώς στη συνέχεια.

Ταξινόμηση Λίστας: η Αναδρομική Προσέγγιση

Εφαρμόζοντας την αρχή της αναδρομής, “για να ταξινομήσω μία λίστα με αριθμούς, αρκεί να υποθέσω ότι η ουρά της είναι ταξινομημένη, οπότε αυτό που απομένει είναι να εισάγω την κεφαλή της λίστας στη σωστή της θέση” (Σχήμα 11.1). Σε ορισμό Prolog, αυτό εκφράζεται:

```
isort([], []).
isort([H|T], SortedList):-
  isort(T, SortedT),
  insert_sorted(H, SortedT, SortedList).
```

όπου το κατηγόρημα insert_sorted/3, εισάγει ένα αριθμό σε μία ταξινομημένη λίστα στη σωστή του θέση, δηλαδή ανάμεσα σε ένα μεγαλύτερο και ένα μικρότερο αριθμό. Μπορείτε να δείτε ένα video για την υλοποίηση του isort/2 εδώ.

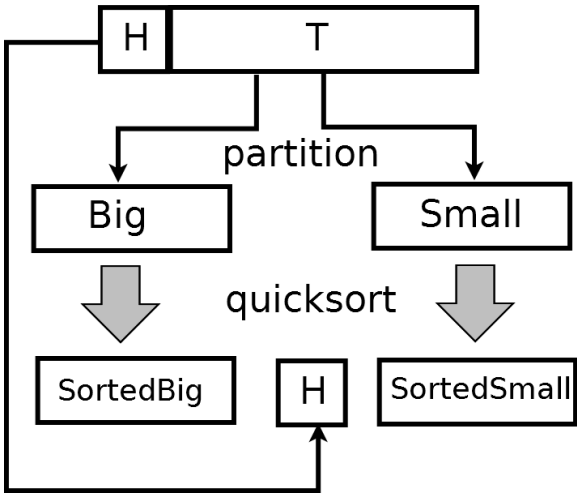


Σχήμα 11.1: *Αναδρομική ταξινόμηση με εισαγωγή*

Ο ορισμός αυτός είναι ένας τυπικός αναδρομικός ορισμός, όπως πιθανότερα θα σκεφτόταν ένας εξοικειωμένος προγραμματιστής της Prolog. Όπως και ο προηγούμενος είναι δηλωτικός αλλά με κάποια επιπλέον στοιχεία διαδικασίας εύρεσης της λύσης, όπως κάθε αναδρομικός ορισμός.

Ταξινόμηση Λίστας: η Αλγοριθμική Προσέγγιση

Αλγοριθμικά η ταξινόμηση μπορεί να εκφραστεί με πολλούς τρόπους, ένας από αυτούς είναι η γρήγορη ταξινόμηση (quicksort). Σύμφωνα με αυτόν τον αλγόριθμο, η αρχική λίστα χωρίζεται σε δύο λίστες, μια που περιέχει τους αριθμούς μεγαλύτερους από την κεφαλή της λίστας και μια που περιέχει τους μικρότερους. Στη συνέχεια, αυτές οι δύο λίστες ταξινομούνται με τον ίδιο τρόπο (αναδρομή) και μετά συνενώνονται σε μία μεγαλύτερη έχοντας στη μέση την αρχική κεφαλή. Προφανώς η συνολική λίστα είναι και ταξινομημένη. Γραφικά, ο αλγόριθμος απεικονίζεται στο Σχήμα 11.2.



Σχήμα 11.2: *Γρήγορη ταξινόμηση*

Ο αντίστοιχος Prolog κώδικας είναι:

```
qsort([],[]).
qsort([H|T], SortedList):-
  partition(H, T, Big, Small),
  qsort(Big, SortedBig),
  qsort(Small, SortedSmall),
  append(SortedBig, [H | SortedSmall], SortedList).
```

όπου το κατηγόρημα partition/4 χωρίζει μία λίστα L με βάση έναν αριθμό H, σε δύο άλλες λίστες, την Big που περιέχει όλους τους αριθμούς μεγαλύτερους από το H και την Small που περιέχει όλους τους αριθμούς μικρότερους από το H.

Είναι εμφανές ότι ο ορισμός αυτός είναι ο πιο διαδικαστικός (αλγοριθμικός) από τους προηγούμενους, και συμβιβάζει την εύκολη, γρήγορη και ευανάγνωστη κωδικοποίηση με δηλωτικά χαρακτηριστικά με την αποδοτικότητα σε χρόνο εκτέλεσης.

Επιλύοντας τα διλήμματα;

Θα ήταν ενδιαφέρον κάποιος να τρέξει κάποια "πειράματα" για να βεβαιώσει του λόγου το αληθές. Η αποδοτικότητα της εκτέλεσης ενός προγράμματος Prolog μετριέται με:

- cputime: ο χρόνος εκτέλεσης για την απάντηση
- inferences: ο συνολικός αριθμός λογικών συνεπαγωγών, δηλαδή των κλήσεων σε κατηγορήματα
- localused: ο όγκος της τοπικής στοιβάδας (local stack) που χρησιμοποιήθηκε, για τις κλήσεις των κατηγορημάτων
- gloabalused: ο όγκος της καθολικής στοιβάδας (global stack) που χρησιμοποιήθηκε, για τους όρους, τις λίστες, τις μεταβλητές και τις τιμές τους.

Το ενσωματωμένο κατηγόρημα statistics/2 με όρισμα ένα από τα παραπάνω επιστρέφει την αντίστοιχη τιμή. Έτσι, ο παρακάτω ορισμός μπορεί να χρησιμοποιηθεί για να μετρήσει την απόδοση όλων των προγραμμάτων ταξινόμησης που παρουσιάστηκαν:

```
test(What,Predicate,List,Result):-
  Call=..[Predicate,L,Result],
  statistics(What,V1),
  Call,
  statistics(What,V2),
  V is V2-V1,
  nl,write(What),write('='),write(V),nl.
```

Αξίζει να τονιστεί ότι ακόμη και μετά από έναν ικανοποιητικό αριθμό "πειραμάτων" πάνω σε διαφορετικές λίστες, δεν είναι ξεκάθαρος ο νικητής σε όλες τις κατηγορίες των μετρήσεων. Μπορεί να βρεθεί ότι κάποια κωδικοποίηση υπερτερεί των άλλων ως προς το χρόνο, αλλά υστερεί ως προς τον αποθηκευτικό χώρο που απαιτεί.

Στις περισσότερες υλοποιήσεις της Prolog υπάρχουν ενσωματωμένα κατηγορήματα για ταξινόμηση λίστας, όπως τα sort/2 και keysort/2, τα οποία λειτουργούν και για άλλους τύπους όρων (χαρακτήρες, συμβολοσειρές κλπ).

11.2 Παραγωγή και Δοκιμή

Η στρατηγική της **Παραγωγής και Δοκιμής (Genarate and Test)** ή **Προσπάθειας και Αποτυχίας (Trial and Error)** είναι η πιο αφελής στρατηγική επίλυσης προβλημάτων. Η διαδικασία είναι η δημιουργία μιας πιθανής λύσης και ο έλεγχος για το αν αυτή η λύση τηρεί κάποιες προδιαγραφές. Αν όχι, μια άλλη λύση δημιουργείται και επανελέγχεται, με τη διαδικασία να επαναλαμβάνεται μέχρις ότου βρεθεί η σωστή λύση το πρόβλημα. Αυτό μοιάζει με το να προσπαθεί να ανοίξει κανείς ένα χρηματοκιβώτιο, παράγοντας έναν-έναν τους πιθανούς συνδυασμούς και δοκιμάζοντας αν κάθε ένας από αυτούς ανοίγει την πόρτα. Όσο και αν αυτή η στρατηγική φαίνεται αδύνατη για τον άνθρωπο, αυτή μπορεί να χρησιμοποιηθεί από προγράμματα λόγω της μεγάλης ταχύτητας εκτέλεσης. Φυσικά, η στρατηγική αυτή έχει καλά αποτελέσματα σε προβλήματα μικρού σχετικά μεγέθους.

Στην Prolog η στρατηγική της παραγωγής και δοκιμής υλοποιείται μέσω της οπισθοδρόμησης. Έτσι, κατηγορήματα παράγουν λύσεις που δοκιμάζονται από άλλα κατηγορήματα. Αν αυτά αποτύχουν τότε δημιουργούνται μέσω οπισθοδρόμησης άλλες λύσεις που δοκιμάζονται και αυτές με τη σειρά τους.

Εύρεση λύσης σε ακέραιες εξισώσεις

Ένα κλασικό παράδειγμα επίλυσης προβλήματος με παραγωγή και δοκιμή είναι αυτό της εύρεσης ακέραιων τιμών που ικανοποιούν αριθμητικές παραστάσεις. Για παράδειγμα, έστω το σύστημα των παραστάσεων: χ + ψ < 15 και χ - ψ = 5, όπου το χ ανήκει στο διάστημα [1,10] και το ψ στο διάστημα [1,5].

Το παρακάτω πρόγραμμα επιλύει το σύστημα παράγοντας τιμές για τα χ και ψ και δοκιμάζοντάς τες για το αν πληρούν τους περιορισμούς που τέθηκαν:

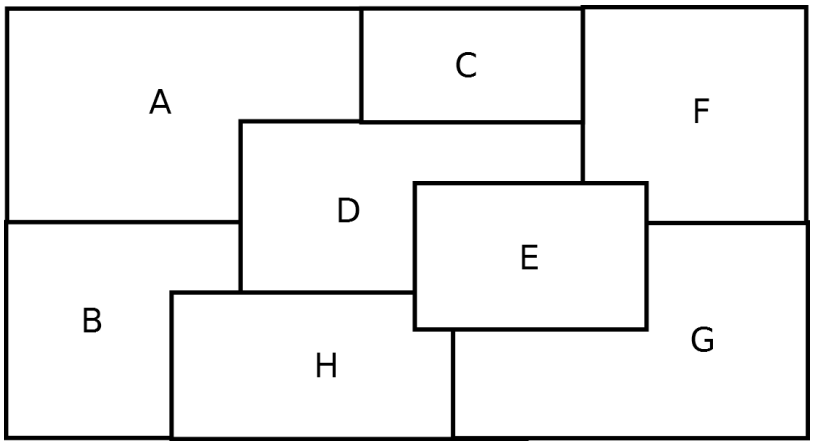
```
solve(X, Y) :-
  member(X, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  member(Y, [1, 2, 3 ,4 ,5]),
  T is X+Y,
  T < 15,
  5 is X-Y.
```

Οι μεταβλητές X και Y δεσμεύονται σταδιακά με τιμές από τις λίστες [1..10] και [1..5] αντίστοιχα. Αμέσως μετά ελέγχονται οι περιορισμοί 5 is X-Y και X+Y<15. Αν και οι δύο αληθεύουν οι τιμές των X και Y αποτελούν λύση. Αν δεν αληθεύει ένας από αυτούς τότε η οπισθοδρόμηση παράγει τις επόμενες τιμές στη λίστα μέσω του member/2.

Άλλα προβλήματα ικανοποίησης περιορισμών

Υπάρχουν και άλλα προβλήματα, μη αριθμητικά, που εντάσσονται στη γενικότερη κατηγορία των **προβλημάτων ικανοποίησης περιορισμών (constraint satisfation problems**). Εκτενέστερη αναφορά στην επίλυση τέτοιων προβλημάτων μέσω Prolog θα γίνει σε επόμενο κεφάλαιο (Κεφάλαιο 12). Εδώ, απλά αναφέρονται στα πλαίσια της στρατηγικής με παραγωγή και δοκιμή. Τέτοια προβλήματα είναι το κρυπταριθμητικό παζλ (για παράδειγμα η εύρεση αριθμητικής τιμής για κάθε γράμμα της εξίσωσης SEND+MORE=MONEY), το πρόβλημα των N Βασιλισσών (τοποθέτηση N βασιλισσών σε μια σκακιέρα N x N, ώστε να μην απειλεί η μία την άλλη), το πρόβλημα χρωματισμού ενός χάρτη και πολλά άλλα.

Στο τελευταίο, το πρόβλημα του χρωματισμού ενός χάρτη, πρέπει να χρωματίσουμε ένα λευκό χάρτη με χρώματα έτσι ώστε η κάθε περιοχή να μην έχει το ίδιο χρώμα με οποιαδήποτε γειτονική της (Σχήμα 11.3).



Σχήμα 11.3: *Παράδειγμα ενός χάρτη προς χρωματισμό*

Ξεκινούμε με τη βασική αναπαράσταση του χάρτη σαν λίστα με ζευγάρια που έχουν περιοχή και το αντίστοιχο χρώμα της (αρχικά μη δεσμευμένη μεταβλητή):

```
template_map([(a,_),(b,_),(c,_),(d,_),(e,_),(f,_),(g,_),(h,_)]).
```

τα χρώματα που είναι διαθέσιμα:

```
colours([red, green, blue, yellow, orange]).
```

και τις γειτονικές περιοχές:

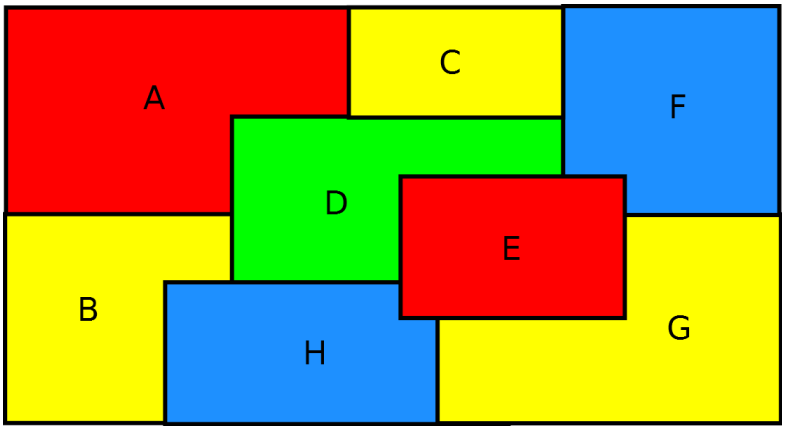
```
next(a,b).
next(a,c).
```

```
next(a,d).
next(b,h).
next(b,d).
...
neighbourgh(X,Y):-next(X,Y),!.
neighbourgh(X,Y):-next(Y,X),!.
```

Με βάση τα παραπάνω, ο κύριος ορισμός είναι:

```
map(Map):-
  colours(C),
  template_map(Map),
  solve_map(Map,C).
```

που παραπέμπει τη λύση του προβλήματος στο κατηγόρημα solve_map/3 που ουσιαστικά θα δώσει τιμές στις μεταβλητές χρώματος.



Σχήμα 11.4: *Παράδειγμα λύσης ενός χάρτη που χρωματίστηκε*

Ο ορισμός είναι φυσικά αναδρομικός: "*Για να χρωματίσουμε μία λίστα από χώρες, υποθέτουμε ότι έχουμε χρωματίσει τις υπόλοιπες χώρες (ουρά) και διαλέγουμε ένα χρώμα για την πρώτη χώρα (κεφαλή) που δεν έρχεται σε συγκρούεται με τις γειτονικές*". Αυτό σε μορφή κώδικα εκφράζεται:

```
solve_map([],_).
solve_map([(Country,Colour)|Rest],AllColours):-
  solve_map(Rest,AllColours),
  member(Colour,AllColours),
  not(conflict((Country,Colour),Rest)).
```

το οποίο με τη σειρά του παραπέμπει τη λύση στο κατηγόρημα conflict/2 που ελέγχει αν ο χρωματισμός μιας χώρας είναι ίδιος με το χρωματισμό μιας γειτονικής. Ο ορισμός είναι και πάλι αναδρομικός τηρώντας την αρχή της αναδρομής:

```
conflict((Country1,Colour),[(Country2,Colour)|_]):-
  neighbourgh(Country1,Country2),!.
conflict((Country,Colour),[_|Rest]):-
  conflict((Country,Colour),Rest).
```

Η κατάλληλη ερώτηση δίνει και τις σωστή απάντηση (Σχήμα 11.4) που μπορεί να είναι περισσότερες από μια:

```
?-map(M).
M=[(a,red),(b,yellow),(c,yellow),(d,green),(e,red),(f,blue),
(g,yellow),(h,blue)]
```

11.3 Αναζήτηση σε γράφο

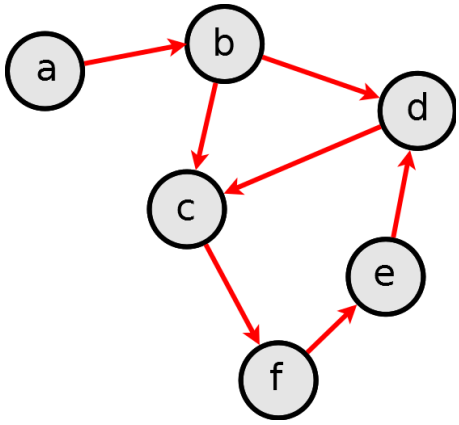
Πολλά από τα προβλήματα στον τομέα της Τεχνητής Νοημοσύνης, και όχι μόνον, ανάγονται σε προβλήματα αναζήτησης διαδρομής σε ένα γράφο. Περισσότερες λεπτομέρειες θα αναφερθούν σε παρακάτω ενότητα.

Ένας γράφος ορίζεται ως ένα σύνολο από κορυφές (κόμβους) και ακμές (συνδέσεις) μεταξύ των κόμβων. Οι γράφοι μπορεί να είναι κατευθυνόμενοι (υπάρχει ένας τρόπος μετάβασης από κόμβο σε κόμβο) ή μη-κατευθυνόμενοι (υπάρχει αμφίδρομη σύνδεση μεταξύ των κόμβων).

Ένα από τα πλέον κοινά προβλήματα στους γράφους είναι να βρούμε ένα μονοπάτι (αλληλουχία κόμβων ή ακμών) από έναν αρχικό κόμβο σε έναν άλλον τελικό. Πολλές φορές επιβάλλεται η τήρηση ορισμένων περιορισμών σχετικά με την τελική διαδρομή, για παράδειγμα απαιτείται η συντομότερη ή αυτή με το ελάχιστο κόστος διαδρομή. Όλα αυτά θα δοθούν βήμα-βήμα στις ενότητες που ακολουθούν.

Αναπαράσταση Γράφου

Το Σχήμα 11.5 αναπαριστά ένα κατευθυνόμενο γράφο.

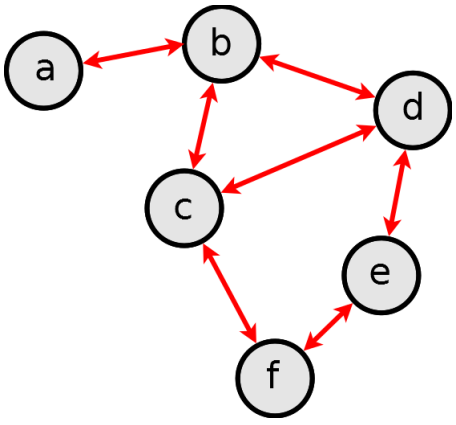


Σχήμα 11.5: *Ένας κατευθυνόμενος γράφος*

Ένας γράφος αναπαριστά σχέσεις μεταξύ κόμβων, όπως στο παράδειγμα του κοινωνικού δικτύου και του οικογενειακού δένδρου στα αρχικά κεφάλαια. Όπως σε εκείνα, έτσι και στο γενικό αφαιρετικό γράφο που παρουσιάζεται εδώ, οι συνδέσεις μπορεί να είναι ένα σύνολο από γεγονότα της μορφής:

```
link(a, b).
link(b, c).
link(b, d).
...
```

Το Σχήμα 11.6 αναπαριστά ένα μη-κατευθυνόμενο γράφο (οι ακμές δεν έχουν κατεύθυνση):



Σχήμα 11.6: *Ένας μη-κατευθυνόμενος γράφος*

Ένας απλός τρόπος για να αναπαραστήσουμε ένα μη-κατευθυνόμενο γράφο, είναι να διπλασιάσουμε τον αριθμό των γεγονότων για να δηλώσουμε την αντίστροφη σύνδεση, για παράδειγμα link(b,a). Αντ' αυτού, υπάρχει ένας καλύτερος τρόπος, με δύο κανόνες της μορφής:

```
next(X, Y) :- link(X, Y).
```

```
next(X, Y) :- link(Y, X).
```

όπου το κατηγόρημα next/2 θα είναι αυτό που αναπαριστά την αμφίδρομη σχέση μεταξύ των κόμβων.

Κάποιος θα μπορούσε να αναρωτηθεί γιατί ένας και μόνο επιπλέον κανόνας:

```
link(X, Y) :- link(Y, X).
```

δεν αρκεί για την αμφίδρομη σχέση; Η απορία έχει βάση εφόσον μια ερώτηση όπως η:

```
?- link(b,a).
```

αφού έψαχνε στα γεγονότα δε θα έβρισκε το link(b,a) αλλά μέσω του κανόνα θα επαλήθευε το link(a,b). Αυτό είναι σωστό. Όμως σε περίπτωση που ο δεσμός δεν υπάρχει, όπως για παράδειγμα στην ερώτηση:

```
?-link(a,c).
```

η εκτέλεση του προγράμματος δε θα τερμάτιζε ποτέ, λόγω του ατέρμονου βρόχου που δημιουργείται στην εναλλασσόμενη αναζήτηση του link(a,c) και link(c,a).

Μια άλλη αναπαράσταση των δεσμών μεταξύ κόμβων είναι με γεγονότα της μορφής:

```
links(a,[b]).
links(b,[a,c,d]).
links(c,[b,d,f]).
...
```

Ένας χωριστός κανόνας μπορεί να οριστεί για τον έλεγχο της σχέσης μεταξύ δύο κόμβων:

```
next(X, Y) :-
  links(X, ListofNodes),
  member(Y, ListofNodes).
```

Αυτή η αναπαράσταση απαιτεί λιγότερες φράσεις γεγονότα για όλο το γράφο αλλά χρειάζεται περισσότερο χρόνο για να βρει ή να επιβεβαιώσει δεσμούς ανάμεσα σε κόμβους λόγω της επιπλέον αναζήτησης του member/2 μέσα στη λίστα.

Ύπαρξη διαδρομής

Η αναζήτηση μιας διαδρομής που συνδέει δύο μακρινούς κόμβους είναι μάλλον κοινή για όλες τις εφαρμογές που περιλαμβάνουν γράφους ή δέντρα. Η αναζήτηση μπορεί να επιτευχθεί με πολλούς διαφορετικούς τρόπους. Υπάρχει πληθώρα τέτοιων αλγορίθμων αναζήτησης οι οποίοι δεδομένου ενός γράφου μπορούν να βρουν διαδρομές οι οποίες πληρούν ορισμένα κριτήρια. Αρχικά θα αναφερθούμε στον απλούστερο αλγόριθμο από όλους, την Αναζήτηση Πρώτα Κατά Βάθος (Depth-First Search), ο οποίος αντιστοιχεί στην στρατηγική εκτέλεσης της Prolog και είναι ως εκ τούτου πιο εύκολο να εφαρμοστεί σε αυτήν.

Έστω η ερώτηση:

```
?- exists_path(a, f).
yes
```

η οποία επιβεβαιώνει την ύπαρξη ενός διαδρομής ανάμεσα στο a και στο f. Το πρόγραμμα που υλοποιεί την εύρεση της διαδρομής είναι παρόμοιο με αυτό της εύρεσης διασύνδεσης μεταξύ φίλων στο κοινωνικό δίκτυο του κεφαλαίου της αναδρομής:

```
exists_path(Node, FinalNode):-
  next(Node, FinalNode).
exists_path(Node, FinalNode):-
  next(Node, SomeNextNode),
  exists_path(SomeNextNode, FinalNode).
```

Εύρεση διαδρομής

Το παραπάνω πρόγραμμα δεν είναι ιδιαίτερα ενδιαφέρον γιατί δεν επιστρέφει τη διαδρομή που συνδέει τους δύο κόμβους. Έστω η ερώτηση:

```
?-  path(a, f, Route).
Route = [a, b, c, f] ;
Route = [a, b, d, e, f] ;
```

```
Route = [a, b, d, c, f] ;
...
```

η οποία επιστρέφει πολλά διαφορετικά μονοπάτια σαν μία λίστα από κόμβους αρχίζοντας από τον αρχικό a μέχρι τον τελικό κόμβο f. Το πρόγραμμα παίρνει την εξής μορφή:

```
path(Node, FinalNode, [Node, FinalNode]):-
  next(Node, FinalNode).
path(Node, FinalNode, [Node | RestRoute]):-
  next(Node, SomeNextNode),
  path(SomeNextNode, FinalNode, RestRoute).
```

που ερμηνεύεται ως εξής: "*η διαδρομή μεταξύ δύο γειτονικών κόμβων Node και FinalNode περιέχει αυτούς τους κόμβους, αλλιώς (αν δεν είναι γειτονικοί), υποθέτοντας ότι υπάρχει μονοπάτι μεταξύ ενός γειτονικού κόμβου SomeNextNode μέχρι τον τελικό FinalNode που δίνεται σε μία λίστα, τότε το τελικό μονοπάτι είναι αυτή η λίστα με την προσθήκη του αρχικού κόμβου Node*".

Εύρεση διαδρομής χωρίς βρόχους

Παρόλο που το παραπάνω πρόγραμμα φαίνεται να δουλεύει σωστά, υπάρχει περίπτωση να εκτελείται ατέρμονα, λόγω των κυκλικών διαδρομών που εμφανίζονται στο γράφο. Για να μη γίνεται αυτό πρέπει να υπάρχει πρόβλεψη, ώστε η διαδικασία της αναζήτησης να μην περνά από τους ίδιους κόμβους.

Αυτό απαιτεί μία λίστα που θα κρατά τους κόμβους που έχει επισκεφθεί η αναζήτηση και έναν επιπλέον έλεγχο για το αν ένας προς επίσκεψη κόμβος είναι ήδη μέσα στη λίστα ή όχι. Το νέο πρόγραμμα γίνεται:

```
path_loopcheck(InitialNode, FinalNode, Route):-
  path_loopcheck (InitialNode, FinalNode, [InitialNode], Route).


path_loopcheck(Node, FinalNode, _, [Node, FinalNode]) :-
  next(Node, FinalNode).

path_loopcheck(Node, FinalNode, VisitedNodes, [Node | RestRoute]) :-
  next(Node, SomeNextNode),
  not(member(SomeNextNode, VisitedNodes)),
  path_loopcheck(SomeNextNode, FinalNode,
        [SomeNextNode|VisitedNodes], RestRoute).
```

όπου VisitedNodes είναι η λίστα που αναπαριστά τους κόμβους που έχει ήδη επισκεφθεί η διαδικασία. Σε κάθε αναδομική κλήση ένας νέος κόμβος προστίθεται σε αυτή τη λίστα.

Παράμετρος Συσσώρευσης

Η λίστα που αποθηκεύει τους κόμβους ονομάζεται και **παράμετρος συσσώρευσης (accumulative parameter)** γιατί συσσωρεύει πληροφορία σε κάθε αναδρομική κλήση. Μια παράμετρος συσσώρευσης αρχικοποιείται με μία τιμή, όπως στο παραπάνω παράδειγμα τον αρχικό κόμβο InitialNode.

Η παράμετρος συσσώρευσης είναι μια συνηθισμένη πρακτική στην Prolog που βοηθά στο να μπει η αναδρομική κλήση στο τέλος του ορισμού. Για παράδειγμα, ο ορισμός του αριθμού στοιχείων σε μία λίστα μπορεί να γραφεί και ως:

```
list_length(List, R) :-
  list_length(List, 0, R).
```

όπου το μηδέν είναι η αρχική τιμή της παραμέτρου συσσώρευσης . Το κατηγόρημα list_length/3 ορίζεται ως εξής:

```
list_length([], L, L).
list_length([H | T], SoFar, L) :-
  NewLength is SoFar + 1,!,
  list_length(T, NewLength, L).
```

Η συνθήκη τερματισμού αναδεικνύει ότι το τελικό αποτέλεσμα, δηλαδή το μήκος της λίστας, που είναι όσο η συσσωρευμένη αξία γίνεται μέσω της εκτέλεσης του προγράμματος. Η διαφορά μεταξύ αυτής της

κωδικοποίησης του list_length και αυτής που παρουσιάστηκε σε προηγούμενο κεφάλαιο, είναι ότι η αναδρομή εδώ είναι πιο αποτελεσματική ως προς τη μνήμη που απαιτείται για την εκτέλεση. Η εξοικονόμηση μνήμης ονομάζεται **βελτιστοποίηση της τελευταίας κλήσης (last call optimisation)**. Εφόσον η αναδρομική κλήση είναι η τελευταία κλήση στον ορισμό και όλες οι μεταβλητές μέχρι εκείνο το σημείο είναι δεσμευμένες, η μνήμη που έχει χρησιμοποιηθεί μπορεί να επαναχρησιμοποιηθεί. Αυτό έχει σαν αποτέλεσμα η μνήμη να είναι σταθερή ανεξάρτητα από τον αριθμό των αναδρομικών κλήσεων.

Από την άλλη πλευρά όμως, η παραπάνω εκδοχή απαιτεί μια επιπλέον παράμετρο. Παρατηρήστε ότι η επιπλέον παράμετρος παραμένει αδέσμευτη μεταβλητή και δεσμεύεται με συγκεκριμένη τιμή μόνο στη συνθήκη τερματισμού. Θα μπορούσαμε να την παρομοιάσουμε ως μια "τεμπέλικη μεταβλητή" (lazy variable), επειδή βρίσκεται σε αναμονή για τη δέσμευση της κατά τη διάρκεια όλων των αναδρομικών κλήσεων.

Με βάση τα παραπάνω, κάποιος θα μπορούσε να ξαναγράψει τον ορισμό του path_loopcheck/4 με μια παράμετρο συσσώρευσης που επιστρέφει το τελικό μονοπάτι σαν τη λίστα με όλους τους κόμβους που έχουν επισκεφθεί:

```
path_loopcheck _alt(InitialNode, FinalNode, Route) :-
  path_loopcheck _alt(InitialNode, FinalNode, [InitialNode], Route).

path_loopcheck _alt(Node, FinalNode, Route, [FinalNode|Route]):-
  next(Node, FinalNode).

path_loopcheck _alt(Node, FinalNode, VisitedNodes, Route) :-
  next(Node, SomeNextNode),
  not member(SomeNextNode, VisitedNodes),
  path_loopcheck _alt(SomeNextNode, FinalNode,
             [SomeNextNode|VisitedNodes], Route).
```

Το μόνο μειονέκτημα αυτής της εναλλακτικής λύσης είναι ότι η διαδρομή εμφανίζεται με αντίστροφη σειρά, από τον τελικό προς τον αρχικό κόμβο:

```
?- path_loopcheck _alt(a, f, Route).
Route = [f,c,b,a] ;
...
```

κάτι που θα μπορούσε εύκολα να διευθετηθεί μέσω της αλλαγής του ορισμού:

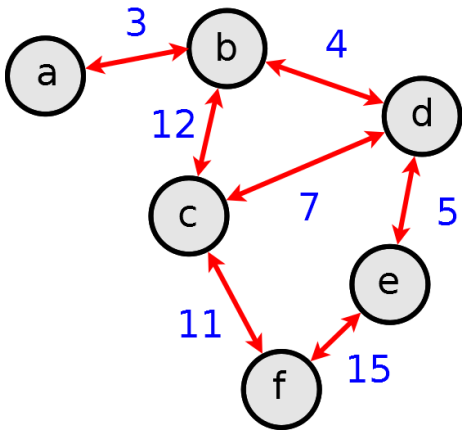
```
path_loopcheck _alt(Node, FinalNode, Route, FinalRoute):-
  next(Node, FinalNode),
  reverse([FinalNode|Route], FinalRoute).
```

Στους παραπάνω ορισμούς πρέπει να σημειώσουμε την ύπαρξη παρόμοιων κατηγορημάτων αλλά με διαφορετικό αριθμό ορισμάτων, για παράδειγμα το path_loopcheck/3 και το path_loopcheck/4, και το list_length/2 και list_length/3. Παρόλο που τα ονόματα των κατηγορημάτων είναι τα ίδια, για την Prolog πρόκειται για δύο τελείως διαφορετικά κατηγορήματα.

Εύρεση διαδρομής και κόστους

Κάποιοι γράφοι εμπεριέχουν κόστος μεταξύ κόμβων (Σχήμα 11.7). Η αναπαράσταση του γράφουν αλλάζει, έτσι ώστε να δηλώνεται το βάρος (κόστος) μεταξύ των κόμβων ως εξής:

```
link(a,b,3).
link(b,d,4).
link(b,c,12).
link(d,c,7).
...
```



Σχήμα 11.7: *Ένας γράφος με βάρη*

Το συνολικό κόστος μιας διαδρομής Cost υπολογίζεται ταυτόχρονα με την εύρεσή της:

```
path_weight(InitialNode, FinalNode, Route, Cost):-
  path_weight(InitialNode, FinalNode, [InitialNode], Route, Cost).

path_weight(Node, FinalNode, Route, [FinalNode|Route], Cost) :-
  next(Node, FinalNode, Cost).

path_weight(Node, FinalNode, VisitedNodes, Route, TotalCost):-
  next(Node, NextNode, Cost),
  not member(SomeNextNode, VisitedNodes),
  path_weight(NextNode, FinalNode, [NextNode|VisitedNodes], Route,
RestCost),
  TotalCost is RestCost + Cost.
```

Και πάλι, μια παράμετρος συσσώρευσης μπορεί να χρησιμοποιηθεί για το κόστος.

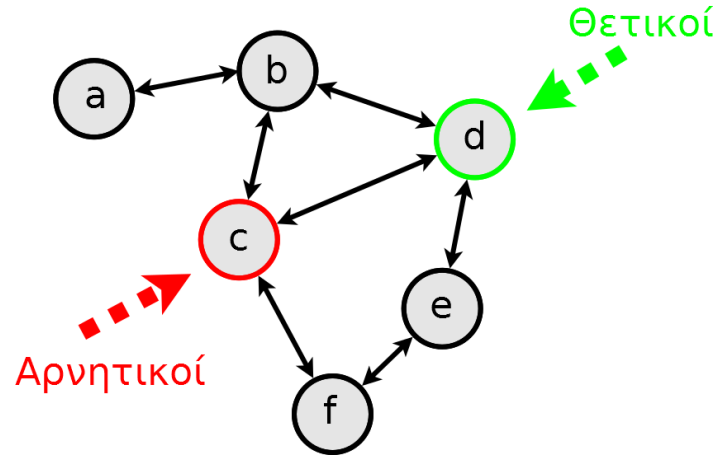
Εύρεση διαδρομής με κριτήρια

Υπάρχουν περιπτώσεις όπου η παραγόμενη διαδρομή πρέπει να ικανοποιεί κάποια κριτήρια. Για παράδειγμα, η διαδρομή πρέπει να περάσει από συγκεκριμένους κόμβους (positive constraint) ή/και δεν πρέπει να περάσει από κάποιους άλλους κόμβους (negative constraint), όπως φαίνεται στο Σχήμα 11.8. Έκφραση των κριτηρίων γίνεται με κατηγορήματα. Για παράδειγμα, αν η διαδρομή πρέπει να περιέχει το d αλλά δεν πρέπει να περιέχει το c, οι ορισμοί είναι:

```
negative_constraint(Path):-
  member(c, Path).
```

και

```
positive_constraint(Path):-
  member(d, Path).
```



Σχήμα 11.8: *Ένας γράφος με θετικούς και αρνητικούς περιορισμούς*

Ο ορισμός του κατηγορήματος εύρεσης διαδρομής γίνεται:

```
path_constraint(InitialNode, FinalNode, Route) :-
  path_constraint(InitialNode, FinalNode, [InitialNode], Route).

path_constraint(Node, FinalNode, Route, [FinalNode|Route]):-
  next(Node, FinalNode),
  positive_constraint([FinalNode|Route]).

path_constraint(Node, FinalNode, VisitedNodes, Route):-
  next(Node, SomeNextNode),
  not member(SomeNextNode, VisitedNodes),
  not negative_constraint([SomeNextNode|VisitedNodes]),
  path_constraint(SomeNextNode, FinalNode,
        [SomeNextNode|VisitedNodes], Route).
```

Έτσι, η απάντηση περιέχει διαδρομές που ικανοποιούν αυτούς τους περιορισμούς:

```
?- path_constraint(a, f, Route).
Route = [a,b,d,e,f]
```

11.4 Επίλυση Προβλημάτων ΤΝ με Τυφλή Αναζήτηση

Η επίλυση ενός προβλήματος αποτελεί θεμελιώδη διαδικασία στην Τεχνητή Νοημοσύνη. Σε αυτό το βιβλίο, δεν είναι σκοπός μας να αναλύσουμε με λεπτομέρεια πως αυτό συμβαίνει. Περισσότερες λεπτομέρειες αναφέρονται εκτενώς σε βιβλία Τεχνητής Νοημοσύνης, όπως το βιβλίο "Τεχνητή Νοημοσύνη" των Ι. Βλαχάβα, Π. Κεφαλά, Ν. Βασιλειάδη, Φ. Κόκκορα, Η. Σακελλαρίου. Εδώ θα παρουσιάσουμε τρεις βασικούς αλγορίθμους επίλυσης προβλημάτων βασισμένα σε καταστάσεις (state-based problems) για να δείξουμε την ομοιότητα που παρουσιάζει η διαδικασία επίλυσης με αυτή της εύρεσης μίας διαδρομής σε γράφο.

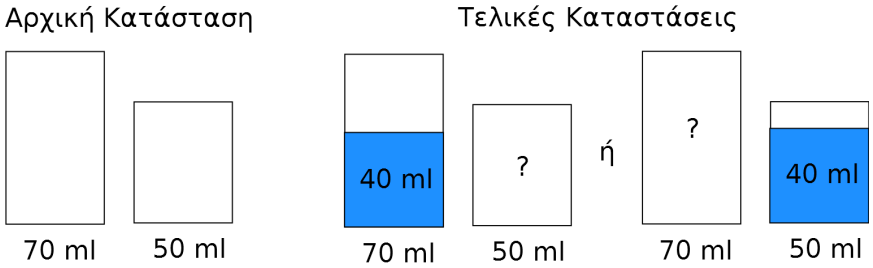
Η έννοια του Προβλήματος

Κατάσταση (state) ενός κόσμου είναι ένα στιγμιότυπο (instance) μίας συγκεκριμένης χρονικής στιγμής κατά την εξέλιξη του κόσμου. Τα προβλήματα βασισμένα σε καταστάσεις ορίζονται ως:

- μια αρχική κατάσταση (initial state)
- μια ή περισσότερες τελικές καταστάσεις (goal states)
- ενέργειες που μπορούν να αλλάξουν τις καταστάσεις του προβλήματος (operators).

Λύση σε ένα πρόβλημα αποτελεί η σειρά των ενεργειών που θα γίνουν για να γεφυρώσουν την αρχική με μία από τις τελικές καταστάσεις. Για λόγους απλούστευσης εδώ θα θεωρήσουμε λύση τη σειρά των καταστάσεων από την αρχική μέχρι την τελική.

Για παράδειγμα, στο πρόβλημα των ποτηριών, δύο ποτήρια A και B χωρίς υποδιαιρέσεις χωράνε ακριβώς το A 70 ml και το B 50 ml αντίστοιχα. Τα ποτήρια μπορούν να γεμίζουν μέχρι το χείλος από μία βρύση και να αδειάζουν είτε το ένα μέσα στο άλλο ή στο νεροχύτη, με τελικό στόχο να υπάρχουν στο τέλος ακριβώς 40 ml σε κάποιο από τα δύο. Λύση στο πρόβλημα αποτελεί η σειρά των ενεργειών που θα γίνουν ώστε το ένα από τα δύο ποτήρια να έχει τελικά 40 ml (Σχήμα 11.9).



Σχήμα 11.9: *Η αρχική και οι τελικές καταστάσεις του προβλήματος των ποτηριών*

Η χωρητικότητα των δύο ποτηριών A και B είναι:

```
glassA(70).
glassB(50).
```

Θα περιγράψουμε μια κατάσταση με τον όρο state(Description,Path), όπου:

- Description είναι ένας όρος που περιγράφει το στιγμιότυπο, και
- Path είναι η διαδρομή μέσα σε λίστα που έχει ακολουθηθεί μέχρι αυτή την κατάσταση

Η παράμετρος Path είναι χρήσιμη γιατί αποθηκεύει τη μέχρι τώρα διαδρομή και στην περίπτωση της τελικής κατάστασης τη λύση του προβλήματος. Δεν είναι άμεσα διαισθητικό αλλά η ανάγκη της θα φανεί παρακάτω. Συνεπώς, η αρχική κατάσταση είναι η:

```
initial_state(state((0,0),[])).
```

και οι τελικές καταστάσεις:

```
final_state(state((_,40),_Path)).
final_state(state((40,_),_Path)).
```

Οι ενέργειες που μπορούν να γίνουν σε οποιαδήποτε κατάσταση (εφόσον ικανοποιούνται ορισμένες συνθήκες) είναι συνολικά οκτώ:

- άδειασε όλο το περιεχόμενο του B στο A
- άδειασε όλο το περιεχόμενο του A στο B
- άδειασε μέρος του B στο A για να γεμίσει
- άδειασε μέρος του A στο B για να γεμίσει
- γέμισε το A από τη βρύση
- γέμισε το B από τη βρύση
- άδειασε το A
- άδειασε το B

Για παράδειγμα η ενέργεια "άδειασε όλο το περιεχόμενο του B στο A" περιγράφεται ως εξής:

```
operator(state((V1,V2),Path),state((Vsum,0),[(V1,V2)|Path])) :-
  V2 > 0,
  Vsum is V1 + V2,
  glassA(G1),
  Vsum =< G1.
```

Στα δύο ορίσματα αναφέρονται:

- η τρέχουσα κατάσταση state((V1,V2),Path), και
- η επόμενη κατάσταση state((Vsum,0),[(V1,V2)|Path])

Άλλες ενδεικτικές ενέργειες είναι η "άδειασε μέρος του B στο A για να γεμίσει":

```
operator(state((V1,V2),Path),state((G1,Vdiff),[(V1,V2)|Path])) :-
  V2 > 0,
  V1 >= 0,
  glassA(G1),
  Vdiff is V2 - ( G1 - V1 ),
  Vdiff > 0.
```

η ενέργεια "γέμισε το A από τη βρύση":

```
operator(state((V1,V2),Path),state((G1,V2),[(V1,V2)|Path])) :-
  glassA(G1), V1\=G1.
```

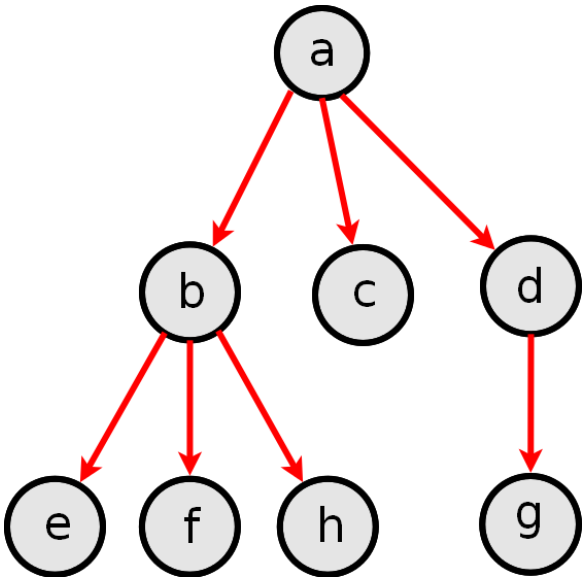
και τέλος η ενέργεια "άδειασε το A":

```
operator(state((V1,V2),Path),state((0,V2),[(V1,V2)|Path])).
```

Όμοια υλοποιούνται και οι υπόλοιπες αντίστοιχες τέσσερις ενέργειες.

Η εύρεση λύσης ενός προβλήματος ως εύρεση διαδρομής σε γράφο

Μετά τον ορισμό ενός προβλήματος ακολουθεί η προσπάθεια επίλυσής του. Ξεκινώντας από την αρχική κατάσταση μπορούμε να δημιουργήσουμε τις επόμενες καταστάσεις που ακολουθούν. Για παράδειγμα από την (0,0) μπορούν να δημιουργηθούν δύο επόμενες καταστάσεις η (70,0) και η (0,50) γεμίζοντας το A ή B ποτήρι αντίστοιχα. Μετά, από κάθε μια από αυτές τις νέες καταστάσεις με άλλες ενέργειες δημιουργούνται άλλες νέες καταστάσεις και αυτή η διαδικασία συνεχίζεται έως ότου συναντήσει κάποιος μια από τις τελικές καταστάσεις. Η διαδικασία δημιουργίας νέων καταστάσεων ονομάζεται επέκταση (expansion) μιας τρέχουσας κατάστασης. Με τη διαρκή επέκταση καταστάσεων, αυτό που δημιουργείται είναι ένα δένδρο (μη κυκλικός γράφος) που ονομάζεται και **δένδρο αναζήτησης (search tree)**. Κι αυτό γιατί σε αυτό το δένδρο αναζητείται η λύση ενός προβλήματος, ως η διαδρομή που ενώνει την αρχική και μία τελική κατάσταση. Στο Σχήμα 11.10 φαίνεται ένα από δένδρο αναζήτησης με αρχική κατάσταση τη και a τελική την g. Η λύση σε αυτό το δένδρο είναι η διαδρομή a-d-g.



Σχήμα 11.10: *Ένα δένδρο αναζήτησης με όλες τις καταστάσεις ενός προβλήματος με αρχική κατάσταση την a και τελική κατάσταση τη g.*

Το Prolog κατηγόρημα για την επέκταση μιας κατάστασης και δημιουργία των επομένων καταστάσεων πρέπει να έχει την εξής αφαιρετική μορφή:

expand(*Τρέχουσα Κατάσταση,Λίστα Επόμενων Καταστάσεων*):-

findall(*Νέα Κατάσταση*,

τέτοια ώστε

operator(*Τρέχουσα Κατάσταση, Επόμενη Κατάσταση*),

και η επόμενη κατάσταση να μην ανήκει στη μέχρι τώρα διαδρομή,

Λίστα Επόμενων Καταστάσεων).

Το κατηγόρημα expand/2 εφαρμόζει όλους τους πιθανούς τελεστές/ενέργειες που μπορεί να γίνουν στην τρέχουσα κατάσταση και βάζει όλες τις νέες καταστάσεις σε μία λίστα. Υπάρχει ένας επιπλέον έλεγχος ο οποίος αποφεύγει να βάλει στη λίστα καταστάσεις που υπάρχουν ήδη στη διαδρομή που έχει δημιουργηθεί μέχρι τώρα.

Ο πλήρης ορισμός του κατηγορήματος expand/2 που εφαρμόζεται ανεξάρτητα από το προς επίλυση πρόβλημα είναι:

```
expand(state(Description,Path),NextStates):-
  findall(state(NewDescription,[Description|Path]),
   (
    operator(
     state(Description,Path),
      state(NewDescription,[Description|Path])
            ),
     not(member(NewDescription,[Description|Path]))
    ),
    NextStates).
```

Το κατηγόρημα expand/2 παίζει ακριβώς το ρόλο του κατηγορήματος next/2 στους γράφους.

Άρα λοιπόν, η επίλυση ενός προβλήματος ανάγεται σε εύρεση διαδρομής σε ένα γράφο. Το θέμα είναι με ποιον τρόπο θα γίνει η αναζήτηση, δηλαδή με ποιον αλγόριθμο θα δημιουργήσουμε τις νέες καταστάσεις. Αυτή η επιλογή είναι κρίσιμη γιατί μπορεί να συντελέσει στην εύρεση ή όχι της λύσης, στην εύρεση μιας οποιαδήποτε λύσης αντί της "καλύτερης", στο χρόνο επίλυσης κλπ. Όλα αυτά όμως είναι αντικείμενο της Τεχνητής Νοημοσύνης και δε θα μας απασχολήσουν σε αυτό το βιβλίο.

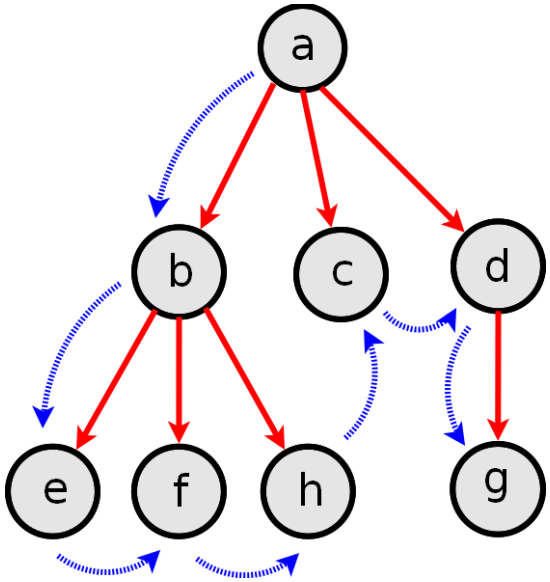
Εύρεση λύσης με αναζήτηση Πρώτα σε Βάθος

Η αναζήτησης της λύσης/διαδρομής απαιτεί μια συγκεκριμένη μεθοδολογία/αλγόριθμο που να καθορίζει ποιο είναι το επόμενο βήμα. Στην επίλυση προβλημάτων ένας τέτοιος αλγόριθμος πρέπει να καθορίσει προς τα που θα κινηθεί η αναζήτηση. Για παράδειγμα, αν επεκταθεί η κατάσταση a θα δώσει τις b,c και d. Σε ποια από τις νέες τρεις θα συνεχίσει η αναζήτηση; Αν υποθέσουμε ότι ανάμεσα σε καταστάσεις του ίδιου επιπέδου/βάθους διαλέγουμε εκείνη που βρίσκεται πιο αριστερά στο δένδρο, δηλαδή την b, αυτή θα δημιουργήσει άλλες τρεις f, e και h οι οποίες θα προστεθούν στις προηγούμενες εναλλακτικές. Έτσι τώρα όλες οι υποψήφιες καταστάσεις είναι οι c, d, f, e και h. Ανάμεσα σε καταστάσεις διαφορετικού επιπέδου/βάθους ποια κατάσταση θα επιλεγεί για επέκταση;

Η επιλογή καθορίζει και τον αλγόριθμο. Για παράδειγμα:

- αν επιλεγεί προς επέκταση η βαθύτερη στο δένδρο και ανάμεσα στις ισοβαθείς η αριστερότερη, τότε διεξάγεται Αναζήτηση Πρώτα σε Βάθος (Depth-First Search)
- αν επιλεγεί προς επέκταση η ρηχότερη στο δένδρο και ανάμεσα στις πιο ρηχές η αριστερότερη, τότε διεξάγεται Αναζήτηση Πρώτα σε Πλάτος (Breadth-First Search)

Όλες οι υποψήφιες προς επέκταση καταστάσεις αποτελούν μία λίστα που ονομάζεται Ανοιχτή Λίστα (Open List) ή Σύνορο της Αναζήτησης (Frontier). Ο αλγόριθμος Αναζήτηση Πρώτα σε Βάθος επιλέγει την πρώτη κατάσταση σε αυτή τη λίστα. Η Ανοιχτή Λίστα διαμορφώνεται έτσι ώστε οι νέες καταστάσεις να προστίθενται στο τέλος της λίστας.



Σχήμα 11.11: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα σε Βάθος τις καταστάσεις στο δένδρο αναζήτησης*

Το βίντεο που δείχνει τη σειρά με την οποία ο αλγόριθμος επισκέπτεται ένα δένδρο αναζήτησης (Σχήμα 11.11) είναι Animation of Depth-First Search ή εναλλακτικά αυτό.

Ο αλγόριθμος Αναζήτηση Πρώτα σε Βάθος υλοποιείται σε Prolog ως εξής:

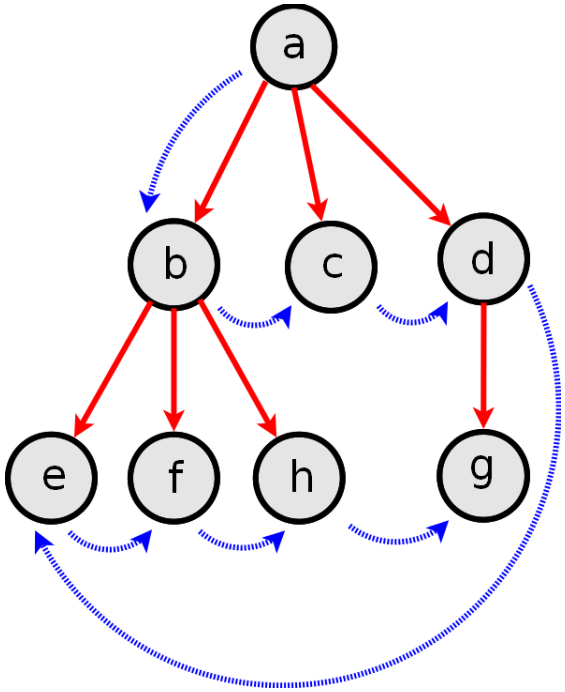
```
dfs:-
  initial_state(S),
  dfs([S],[],Solution),
  write(Solution),nl.

dfs([State|_],State):-
  final_state(State).

dfs([State|RestOpen],Solution):-
  expand(State,NextStates),
  append(NextStates,RestOpen,NewOpenList),
  !,
  dfs(NewOpenList,Solution).
```

Εύρεση λύσης με αναζήτηση Πρώτα σε Πλάτος

Ο αλγόριθμος Αναζήτηση Πρώτα σε Πλάτος επιλέγει πάλι την πρώτη κατάσταση σε αυτή τη λίστα. Αλλά σε αντίθεση με τα παραπάνω, η Ανοιχτή Λίστα διαμορφώνεται έτσι ώστε οι νέες καταστάσεις να προστίθενται στην αρχή της λίστας.



Σχήμα 11.12: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα σε Πλάτος τις καταστάσεις στο δένδρο αναζήτησης*

Το βίντεο που δείχνει τη σειρά με την οποία ο αλγόριθμος επισκέπτεται ένα δένδρο αναζήτησης (Σχήμα 11.12) είναι Animation of Breath-First Search.

Ο αλγόριθμος Αναζήτηση Πρώτα σε Πλάτος υλοποιείται σε Prolog ως εξής:

```
bfs:-
  initial_state(S),
  bfs([S],[],Solution),
  write(Solution),nl

bfs([State|_],State):-
   final_state(State).

bfs([State|RestOpen],Solution):-
  expand(State,NextStates),
  append(RestOpen, NextStates,NewOpenList),
  !,
  bfs(NewOpenList,Solution).
```

Είναι εύκολο να παρατηρήσει κανείς ότι η διαφορά των δύο αλγορίθμων έγκειται στη σειρά των ορισμάτων του append/3.

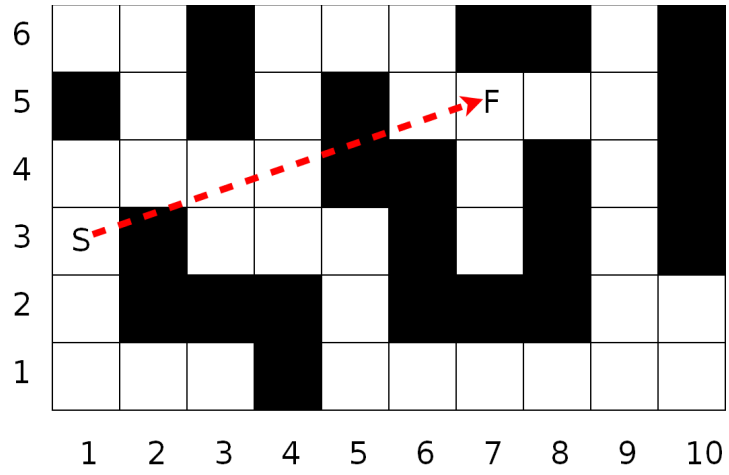
11.5 Προχωρημένα Θέματα: Επίλυση Προβλημάτων ΤΝ με Ευριστική Αναζήτηση

Οι δύο αλγόριθμοι που παρουσιάστηκαν παραπάνω ανήκουν στην κατηγορία των Αλγορίθμων **Τυφλής Αναζήτησης (Blind Search)**, γιατί δεν έχουν κάποια ένδειξη ποια διαδρομή πρέπει να ακολουθήσουνε για να βρεθούν πιο κοντά στην τελική κατάσταση. Υπάρχουν όμως προβλήματα στα οποία οι καταστάσεις μπορούν να ταξινομηθούν σε "μάλλον καλύτερες" και "μάλλον χειρότερες" με βάση την "απόστασή" τους από την τελική κατάσταση. Για παράδειγμα στο πρόβλημα εύρεσης διαδρομής σε ένα λαβύρινθο στο Σχήμα 11.13 η κατάσταση του να βρεθεί η αναζήτηση στο (2,4) μοιάζει καλύτερη από τη (1,1). Άρα, αν στην ανοιχτή λίστα υπήρχαν αυτές οι δύο καταστάσεις, θα ήταν πιο υποσχόμενο να διαλέξουμε τη φαινομενικά καλύτερη, δηλαδή τη (2,4), εφόσον η απόσταση της από την τελική (αγνοώντας τα εμπόδια) είναι μικρότερη από αυτήν της (1,1).

Οι αλγόριθμοι που βασίζονται σε μία τέτοια ταξινόμηση καταστάσεων ονομάζονται Αλγόριθμοι **Ευριστικής Αναζήτησης (Heuristic Search)**. Η μέθοδος (συνήθως μια αριθμητική παράσταση που δίνει μία τιμή) βάση της οποίας γίνεται η αξιολόγηση κάθε κατάστασης ονομάζεται **Ευριστική Συνάρτηση (Heuristic Function)**. Ένας τέτοιος είναι ο αλγόριθμος **Αναζήτησης Πρώτα στο Καλύτερο (Best-First Search)**.

Η έννοια του Προβλήματος με Ευριστικό Μηχανισμό

Έστω το πρόβλημα αναζήτησης διαδρομής σε ένα λαβύρινθο με εμπόδια, όπως φαίνεται στο Σχήμα 11.13 .



Σχήμα 11.13: *Το πρόβλημα εύρεσης διαδρομής σε λαβύρινθο με αρχική κατάσταση την S και τελική κατάσταση την F.*

Τα εμπόδια μπορούν να αναπαρασταθούν ως γεγονότα της μορφής:

```
blocked_square((1,5)).
blocked_square((2,2)).
blocked_square((2,3)).
blocked_square((3,2)).
...
```

Οι αρχική και η τελική κατάσταση:

```
initial_state(state((1,3),[])).
final_state(state((7,5),_PATH)).
```

Οι ενέργειες/τελεστές είναι τέσσερις (μπορώ να κινηθώ ένα τετράγωνο προς τα δεξιά, αριστερά, πάνω και κάτω):

```
operator(state(Parent,Path),state(Child,[Parent|Path])):-
  can_go(Parent,Child),
  not(blocked_square(Child)),
  valid(Child).

can_go((X,Y),(NX,Y)):- NX is X + 1.
can_go((X,Y),(NX,Y)):- NX is X - 1.
can_go((X,Y),(X,NY)):- NY is Y + 1.
can_go((X,Y),(X,NY)):- NY is Y - 1.

valid((X,Y)):-
  X>0,Y>0,dimension(LX,LY),X =<LX,Y=<LY.
```

όπου dimension/2 δίνει τις διαστάσεις του λαβύρινθου:

```
dimension(10,6).
```

Η ευριστική συνάρτηση που χρησιμοποιείται σε αυτό το πρόβλημα είναι η λεγόμενη απόσταση Manhattan που υπολογίζει την απόσταση από την τελική ως άθροισμα απόλυτων τιμών των πλευρών:

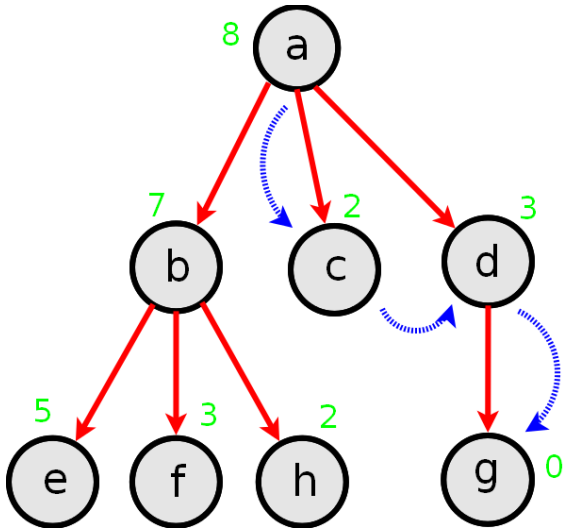
```
heuristic_Func(state((X,Y),_),HV):-
  final_state(state((XF,YF),_)),
```

```
HV is abs(X-XF) + abs(Y-YF).
```

Εναλλακτικά χρησιμοποιείται ως ευριστική συνάρτηση και η Ευκλείδειος απόσταση που υπολογίζει την απόσταση από την τελική ως τη ρίζα του αθροίσματος των πλευρών.

Εύρεση λύσης με αναζήτηση Πρώτα στο Καλύτερο

Ο αλγόριθμος **Πρώτα στο Καλύτερο (Best-First Search)** χρησιμοποιεί την ευριστική συνάρτηση για να αξιολογήσει την κάθε κατάσταση και επισκέπτεται αυτήν με την μικρότερη τιμή. Έτσι στο Σχήμα 11.14, η σειρά με την οποία θα επεκτείνει τις καταστάσεις του δένδρου αναζήτησης είναι η a, c, d και g.



Σχήμα 11.14: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα στο Καλύτερο τις καταστάσεις στο δένδρο αναζήτησης*

Για να είναι δυνατή η ταξινόμηση των καταστάσεων, η ανοιχτή λίστα έχει δυάδες της μορφής (Ευριστική Τιμή, Κατάσταση). Ο αλγόριθμος υλοποιείται στην Prolog ως εξής:

```
bestfs:-
  initial_state(S),
  evaluateStates([S],EVSTATES),
  bestfs(EVSTATES,Solution),
  write(Solution),nl.

bestfs([(_,State)|_],State):-
  final_state(State).

bestfs([(_,State)|OPEN],Solution):-
  expand(State,NextStates),
  evaluateStates(NextStates,EV_States),
  sort(EV_States,NextSorted),
  merge(NextSorted,OPEN,NewOpen),
  !,
  bestfs(NewOpen,Solution).
```

όπου το κατηγόρημα evaluateStates/2 καλεί το heuristic_Func/2 για αναθέσει ευριστικές τιμές σε κάθε νέα κατάσταση πριν αυτές προστεθούν στην ανοιχτή λίστα:

```
evaluateStates(States,Evaluated):-
  findall((HV,State),
          (member(State,States), heuristic_Func(State,HV)),
          Evaluated).
```

Τα ενσωματωμένα κατηγορήματα sort/2 και merge/2 ταξινομούν μια λίστα και ενώνουν δύο ταξινομημένες λίστες αντίστοιχα.

Βιβλιογραφία

Οι αλγόριθμοι αναζήτησης για επίλυση προβλημάτων είναι από τα βασικά θέματα που αναφέρονται σε οποιοδήποτε βιβλίο σχετικό με Τεχνητή Νοημοσύνη (Βλαχάβας I. et al., 2011), (Russel & Norvig, 2009), (Luger, 2008). Η πρώτη προσέγγιση των αλγορίθμων με την γλώσσα Prolog έγινε με εκτενή τρόπο στο κλασσικό βιβλίο του Bratko και τις μετέπειτα εκδόσεις του. Οι αλγόριθμοι στο κεφάλαιο αυτό έχουν κάποιες διαφορές από του (Bratko, 2011), κυρίως βέβαια διότι η ανάπτυξή τους παρουσιάζεται σταδιακά, όπως παραδείγματος χάριν συμβαίνει με το παράδειγμα της αναζήτησης διαδρομής σε γράφο.

Βλαχάβας, I. και Κεφαλάς, Π. και Βασιλειάδης, N. και Κόκκορας, Φ. και Σακελλαρίου Η. (2011). *Τεχνητή Νοημοσύνη*. Γ' Έκδοση. ISBN: 978-960-8396-64-7. Εκδόσεις Πανεπιστημίου Μακεδονίας.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition. Prentice Hall.

Luger G.F. (2008). *Artificial Intelligence: Structures and Strategies for Complex Problem-Solving*. 6th edition. Addison Wesley Longman.

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Άλυτες Ασκήσεις

11.1 Για να συμπληρωθεί ο ορισμός του κατηγορήματος nsort/2, να γραφεί ο ορισμός του κατηγορήματος is_sorted/2 που εξετάζει αν μία λίστα από αριθμούς είναι διατεταγμένη κατά φθίνουσα σειρά. Για παράδειγμα:

```
?- is_sorted([10,8,5,3,2]).
yes
?-is_sorted([10,5,2,8,3]).
no
```

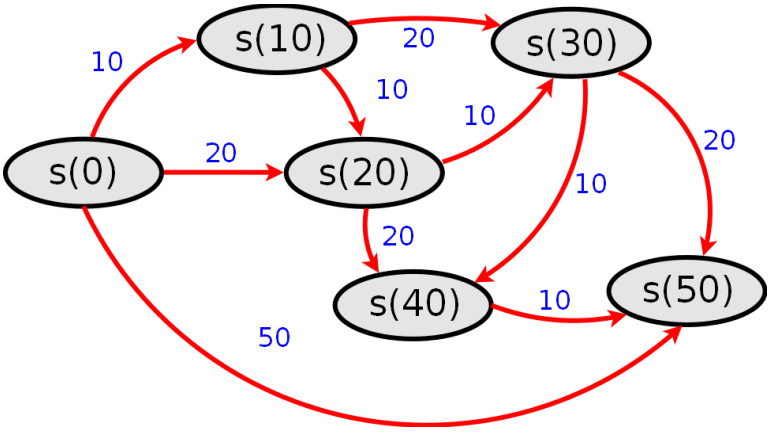
11.2 Για να συμπληρωθεί ο ορισμός του κατηγορήματος isort/2, να γραφεί ο ορισμός του κατηγορήματος insert_sorted/3, το οποίο εισάγει ένα αριθμό σε μία ταξινομημένη λίστα κατά φθίνουσα σειρά στη σωστή του θέση, δηλαδή ανάμεσα σε ένα μεγαλύτερο και ένα μικρότερο αριθμό. Για παράδειγμα:

```
?-insert_sorted(3,[10,8,5,2],L).
L=[10,8,5,3,2]
?-insert_sorted(10,[8,5,3,2],L).
L=[10,8,5,3,2]
?-insert_sorted(2,[10,8,5,3],L).
L=[10,8,5,3,2]
```

11.3 Για να συμπληρωθεί ο ορισμός του κατηγορήματος quicksort/2, να γραφεί ο ορισμός του κατηγορήματος partition/4, το οποίο χωρίζει μία λίστα L με βάση έναν αριθμό X, σε δύο άλλες λίστες, την Big που περιέχει όλους τους αριθμούς μεγαλύτερους από το X και την Small που περιέχει όλους τους αριθμούς μικρότερους από το X. Για παράδειγμα:

```
?-partition(4,[10,7,2,1,8,3,5],Big,Small).
Big=[10,7,8,5]
Small=[2,1,3]
```

11.4 Ένα μηχάνημα αυτόματης πώλησης εισιτηρίων σε αστικά λεωφορεία μοντελοποιείται από την ακόλουθη μηχανή πεπερασμένων καταστάσεων. Η τιμή του εισιτηρίου είναι 50 λεπτά, και το μηχάνημα μπορεί να δεχθεί νομίσματα των 10, 20 και 50 λεπτών. Ο αυτόματος πωλητής επιστρέφει εισιτήριο μόνο αν το ακριβές αντίτιμο εισάγεται στη μηχανή. Η μηχανή πεπερασμένων καταστάσεων που περιγράφει την λειτουργία του αυτόματου πωλητή παρουσιάζεται στο Σχήμα 11.15:



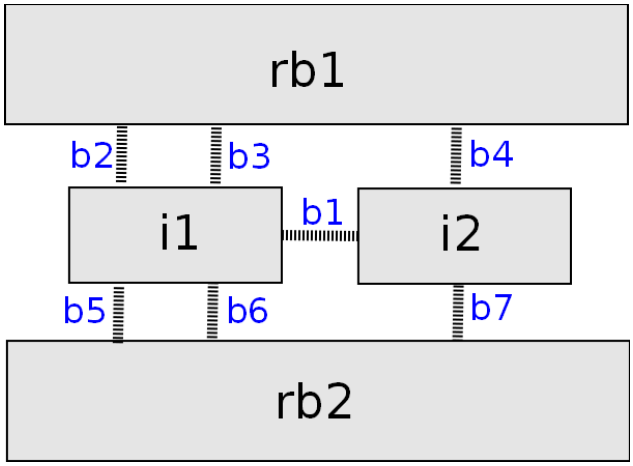
Σχήμα 11.15: *Η μηχανή πεπερασμένων καταστάσεων που περιγράφει την λειτουργία ενός αυτόματου πωλητή*

(α) Να υλοποιήσετε ένα κατηγόρημα transitions/3 το οποίο περιγράφει τις μεταβάσεις του παραπάνω αυτομάτου. Το κατηγόρημα θα έχει σαν ορίσματα δύο καταστάσεις και το νόμισμα που απαιτείται για την μετάβαση από την πρώτη κατάσταση στη δεύτερη.

(β) Να υλοποιήσετε το κατηγόρημα coins_to_insert/3, το οποίο δεδομένης μιας αρχικής κατάστασης και μιας επιθυμητής στην οποία θα "πάει" το αυτόματο,να επιστρέφει σε λίστα τα νομίσματα που απαιτούνται.

(γ) Μπορείτε να υπολογίσετε πόσοι διαφορετικοί τρόποι υπάρχουν (συνδυασμοί νομισμάτων) για να αγοράσει κάποιος εισιτήριο? Θα πρέπει να σημειωθεί ότι δύο ακολουθίες όπου τα νομίσματα εμφανίζονται διαφορετικά, θεωρούνται διαφορετικές μεταξύ τους.

11.5 Η πόλη του Konigsberg καταλαμβάνει τις δύο όχθες (rb1, rb2) και τα δύο νησιά (i1, i2) του ποταμού Pregel και τα προηγούμενα συνδέονται με γέφυρες (b1, b2, ...b7), όπως φαίνεται σχηματικά στο Σχήμα 11.16.



Σχήμα 11.16: *Η πόλη του Konigsberg*

(α) Να ορίσετε ένα κατηγόρημα connection/3 το οποίο πετυχαίνει όταν τα πρώτα δύο ορίσματα του είναι τοποθεσίες, δηλαδή όχθες ή νησιά και το τρίο μια γέφυρα που τα ενώνει. Για παράδειγμα:

```
?- connection(rb1,i2,Bridge).
Bridge = b4
yes
?- connection(rb1,i1,Bridge).
Bridge = b2 ;
Bridge = b3 ;
No
?- connection(rb1,rb2,Bridge).
```

```
No
```

(β) Το κύριο τουριστικό αξιοθέατο της πόλης είναι οι γέφυρες, έτσι οι περιηγητές προσπαθούν να σχεδιάσουν διαδρομές που περιλαμβάνουν διέλευση από αυτές. Να ορίσετε ένα κατηγόρημα walk/3 το οποίο πετυχαίνει όταν τα δύο πρώτα του ορίσματα είναι τοποθεσίες και το τρίτο μια λίστα από γέφυρες τις οποίες θα πρέπει να διέλθει ο περιηγητής για να φτάσει από την πρώτη τοποθεσία στην δεύτερη. Θα πρέπει σημειωθεί ότι ο περιηγητής θα πρέπει να διέλθει κάθε γέφυρα μόνο μια φορά, ασχέτως πόσες φορές θα επισκεφτεί μια τοποθεσία (όχθη ή νησί). Για παράδειγμα:

```
?- walk(rb1,i1,L).
L = [b2] ;
L = [b3] ;
L = [b2, b1, b4, b3]
..(υπάρχουν και πολλές άλλες λύσεις)...
Yes
?- walk(rb1,rb2,L).
L = [b2, b5] ;
L = [b2, b6] ;
L = [b2, b1, b7] ;
L = [b2, b1, b4, b3, b6] ;
..(υπάρχουν και πολλές άλλες λύσεις)...
```

(γ) Ο Euler ήταν ο πρώτος που απέδειξε ότι δεν υπάρχει διαδρομή που να διέρχεται από οποιαδήποτε τοποθεσία και καταλήγει σε οποιαδήποτε τοποθεσία που να διέρχεται από όλες τις γέφυρες μόνο μια φορά. Να ορίσετε ένα κατηγόρημα euler/0 που να αποδεικνύει ότι δεν υπάρχει τέτοια διαδρομή.

```
?- euler.
yes
```

11.6 Ο λεγόμενος και γρίφος του Einstein λέει ότι: "Πέντε άνδρες έχουν διαφορετικές εθνικότητες και ζουν σε πέντε διαφορετικά σπίτια. Εξασκούν πέντε διαφορετικά επαγγέλματα και καθένας τους έχει ένα αγαπημένο κατοικίδιο και ένα αγαπημένο ποτό. Κάθε σπίτι είναι βαμμένο με διαφορετικό χρώμα.

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The Japanese is a painter.
- The Italian drinks tea.
- The Norwegian lives in the first house on the left.
- The owner of the fox drinks water.
- The owner of the green house drinks coffee.
- The green house is on the right of the white one.
- The sculptor breeds snails.
- The diplomat lives in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian's house is next to the blue one.
- The violinist drinks fruit juice.
- The fox is in a house next to that of the doctor.
- The horse is in a house next to that of the violinist."

Γράψτε ένα Prolog πρόγραμμα που να απαντά στην ερώτηση "*Ποιος έχει τη ζέβρα?*". Η λύση εκφράζεται σαν λίστα που αρχικά έχει πολλές αδέσμευτες μεταβλητές:

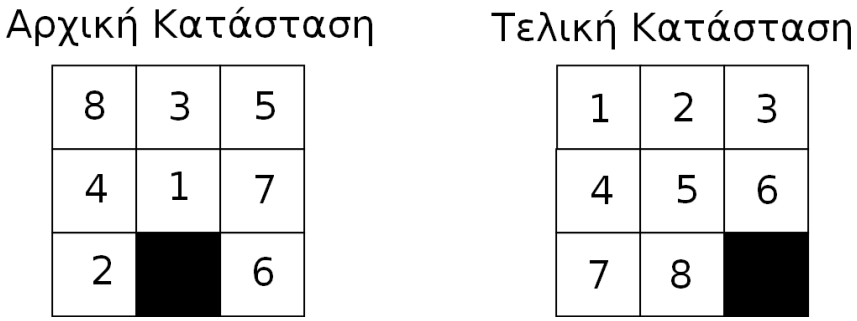
```
houseplan([
  house(_,norwegian,_,_,_),
  house(_,_,_,_,_),
  house(_,_,_,milk,_),
  house(_,_,_,_,_),
  house(_,_,_,_,_)
        ]).
```

Το τελικό πρόγραμμα θα δείχνει κάπως έτσι:

```
who_owns_zebra(Owner):-
  houseplan(Houses),
... (many prolog lines) ...
  next_to(house(_,_,_,_,seven_stars),  house(_,Owner,zebra,_,_),
Houses),
... (many Prolog lines)....
```

Προφανώς πρέπει να ορίσετε το κατηγόρημα next_to/3 που αληθεύει όταν τα δύο πρώτα ορίσματα είναι διπλανά σε μία λίστα που δίνεται ως τρίτο όρισμα.

11.7 Το N-παζλ είναι ένα γνωστό παζλ με συρόμενα τετράγωνα που αποτελείται από αριθμημένα τετράγωνα και έναν κενό χώρο (Σχήμα 11.17). Τα τετράγωνα βρίσκονται αρχικά σε μια (ψεύδο) τυχαία σειρά. Ο στόχος του παζλ είναι να αναδιατάξετε τα τετράγωνα με κινήσεις που ολισθαίνουν τετράγωνα στο κενό χώρο.



Σχήμα 11.17: *Αρχική και τελική κατάσταση του 8-παζλ*

Συνήθως οι ευρετικές συναρτήσεις που χρησιμοποιούνται για αυτό το πρόβλημα είναι η μέτρηση των πλακιδίων ή το άθροισμα των αποστάσεων manhattan. Μια δυνατή αναπαράσταση του παζλ θα μπορούσε να είναι μια λίστα με τη μορφή [2,3,4,5,1,e, 6,7,8], όπου το πρώτο στοιχείο του πίνακα αντιστοιχεί στη θέση παζλ (1,1), η δεύτερη σε η θέση (1,2) κ.α.

α) Να αναπαραστήσετε το πρόβλημα του παζλ σε Prolog.

β) Να υλοποιήσετε το κατηγόρημα heuristic_Func(S, HV) για το παζλ.

γ) Να κάνετε μερικά πειράματα για να συγκρίνετε την απόδοση των τριών αλγορίθμων που παρουσιάστηκαν σε αυτό το κεφάλαιο.

11.8 Να υλοποιήσετε ένα πρόγραμμα Prolog που να λύνει την κρυπταριθμητική εξίσωση:

DONALD + GERALD = ROBERT,

αναθέτοντας έναν διαφορετικό αριθμό σε κάθε γράμμα.