1 ΠΡΟΒΛΗΜΑΤΑ ΤΝ ΚΑΙ LISP

1.1 Αναζήτηση και Στρατηγικές Αναζήτησης

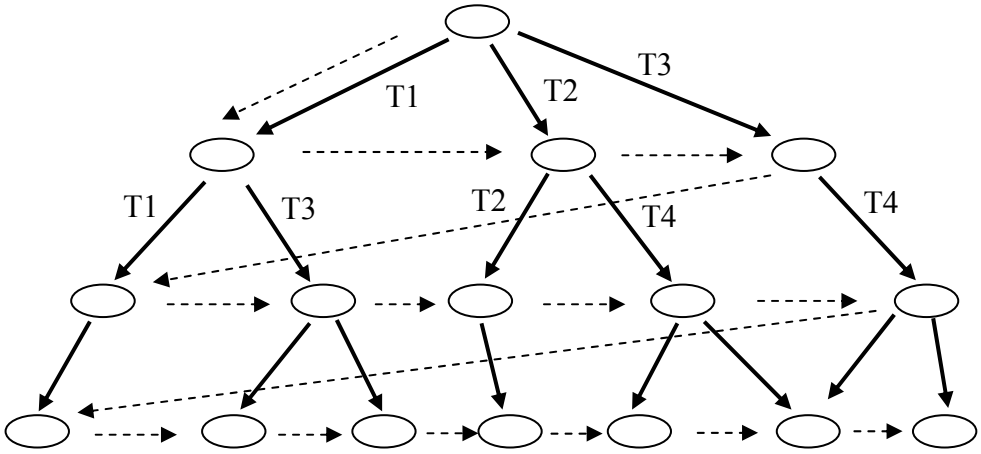
Ένας τρόπος επίλυσης προβλημάτων με μεθόδους Τεχνητής Νοημοσύνης (ΤΝ) είναι η 'αναζήτηση λύσης' (search). Σύμφωνα μ' αυτήν, ένα πρόβλημα παριστάνεται ως μια (αρχική) 'κατάσταση' και η επίλυσή του ως μια σειρά μεταβάσεων από την κατάσταση αυτή σε μια (τελική) κατάσταση, που αποτελεί τη λύση του προβλήματος, δια μέσου άλλων (ενδιάμεσων) καταστάσεων.

Η παραγωγή των ενδιάμεσων καταστάσεων γίνεται με τη χρήση των 'τελεστών μετάβασης'. Ένας τελεστής μετάβασης προσδιορίζει την επόμενη κατάσταση με βάση την προηγούμενη. Ένα πρόβλημα μπορεί να έχει ένα ή περισσότερους τελεστές μετάβασης, που ουσιαστικά αναπαριστούν τις ενέργειες που μπορούν να γίνουν για να φτάσουμε στη λύση του προβλήματος. Για να μπορεί να εφαρμοστεί ένας τελεστής πρέπει να ικανοποιούνται ορισμένες προϋποθέσεις, που αποτελούν μέρος του ορισμού του τελεστή.

Δεδομένου ότι οι τελεστές μετάβασης είναι συνήθως περισσότεροι από ένας, τίθεται το ζήτημα του ποιόν θα εφαρμόσουμε κάθε φορά. Κατ' αρχή αποκλείονται αυτοί των οποίων οι προϋποθέσεις δεν ικανοποιούνται. Για τους υπόλοιπους καθορίζουμε μια σειρά με βάση κάποια 'στρατηγική αναζήτησης'. Υπάρχουν δύο κατηγορίες στρατηγικών αναζήτησης, οι 'τυφλές' και οι 'ευριστικές'.

Οι τυφλές στρατηγικές δεν λαμβάνουν υπ' όψιν το συγκεκριμένο πρόβλημα. Δύο τέτοιες στρατηγικές είναι οι 'αναζήτηση κατά πλάτος' και 'αναζήτηση κατά βάθος' (με οπισθοδρόμηση). Η αναζήτηση κατά πλάτος (breadth-first search) εφαρμόζει κατ' αρχήν όλους τους εφαρμόσιμους τελεστές (με μια συγκεκριμένη σειρά) στην αρχική κατάσταση και παράγει τόσες νέες καταστάσεις όσοι και οι εφαρμόσιμοι τελεστές. Οι καταστάσεις αυτές, επειδή η διαδικασία της αναζήτησης μπορεί να

παρασταθεί σαν ένα δέντρο, που λέγεται ‘δέντρο αναζήτησης’, λέγονται ‘παιδιά’ της αρχικής κατάστασης (που ονομάζεται ‘ρίζα’ του δέντρου). Στη συνέχεια, η αναζήτηση κατά πλάτος εφαρμόζει όλους τους εφαρμόσιμους τελεστές (με την ίδια σειρά) στο πρώτο από τα παιδιά που έχουν παραχθεί (και παράγει τα αντίστοιχα παιδιά του), μετά στο δέυτερο κ.ο.κ. Όταν τελειώσουν τα παιδιά της αρχικής κατάστασης, συνεχίζει με το πρώτο παιδί του πρώτου παιδιού της αρχικής κατάστασης κ.ο.κ. Όταν βρει τη λύση (τελική κατάσταση ή κατάσταση στόχου) σταματά.



Σχήμα 11.1 Αναζήτηση κατά πλάτος

Στο Σχήμα 11.1 απεικονίζεται ο τρόπος αναζήτησης κατά πλάτος, δηλ. η σειρά δημιουργίας των καταστάσεων (ως κόμβων ενός δέντρου). Τα T1, T2, T3 και T4 παριστάνουν τους τελεστές μετάβασης.

Ο αλγόριθμος της κατά πλάτος αναζήτησης σε ψευδοκώδικα παρουσιάζεται παρακάτω:

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα) και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω X, είναι ο στόχος
   2.1 Αν είναι, τότε σταμάτα (επιτυχία)
   2.2 Αν δεν είναι, τότε
      2.2.1 Παράγαγε τα παιδιά του X και βάλε το X στην closed
      2.2.2 Διάγραψε όσα παιδιά του X υπάρχουν στην closed
      2.2.3 Εισάγαγε τα υπόλοιπα παιδιά του X στο τέλος της open
3. Σταμάτα (αποτυχία)

Βασικά, χρησιμοποιούνται δύο λίστες, η open και η closed, όπου αποθηκεύονται στην μεν open οι ανοικτές καταστάσεις, δηλ. αυτές που δεν έχουν ακόμη αναπτυχθεί (δηλ. δεν έχουν παραχθεί τα παιδιά τους), στη δε closed, οι κλειστές, δηλ. αυτές που

έχουν αναπτυχθεί. Το βήμα 2.2.2 υπάρχει για να διαγραφούν όσες καταστάσεις έχουν ήδη αναπτυχθεί, διότι δεν προσφέρουν κάτι καινούργιο.

Μια υλοποίηση του παραπάνω αλγορίθμου σε LISP είναι η παρακάτω.

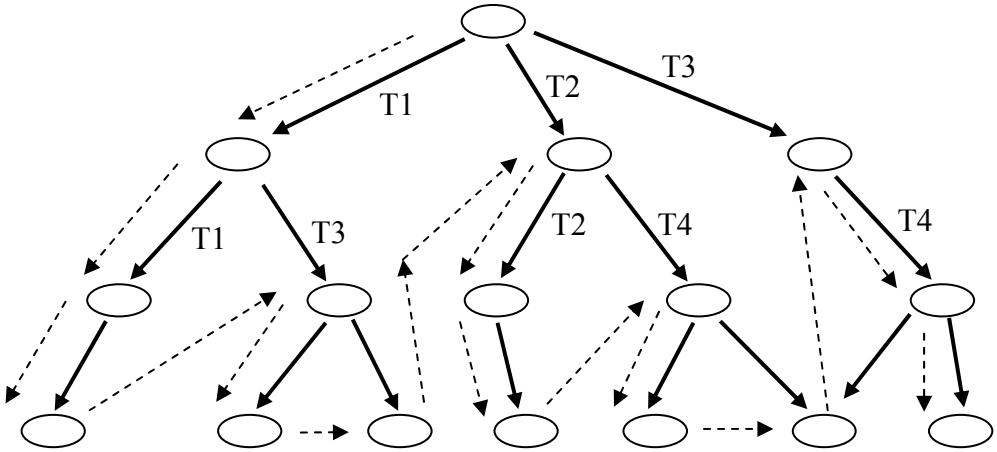
```
(defun breadth-first-search (init-state final-states)
      (let ((open (list init-state))
            (closed nil))
        (breadth-solve open closed final-states))

(defun breadth-solve (open closed final-states)
      (if (null open) nil
         (let ((cur-state (pop open))
            (if (member cur-state final-states)
               (show-solution-path cur-state closed)
               (let* ((childs (make-childs cur-state))
                        (closed (push cur-state closed))
                        (childs (remove-closed-childs childs closed))
                        (open (append open childs)))
                  (breadth-solve open closed final-states))))))
```

Αυτός είναι βέβαια ένας σκελετός. Υπάρχουν συναρτήσεις που πρέπει να οριστούν στη συνέχεια, όπως οι show-solution-path, make-childs και remove-closed-childs.

Η αναζήτηση κατά βάθος (depth-first search) ξεκινά με τον ίδιο τρόπο, αλλά στη συνέχεια εφαρμόζει όλους τους εφαρμόσιμους τελεστές στο πρώτο παιδί της αρχικής κατάστασης, μετά στο πρώτο παιδί του πρώτου παιδιού της αρχικής κατάστασης κ.ο.κ. μέχρις ότου είτε βρεθεί η λύση (τελική κατάσταση), οπότε σταματά, είτε βρει αδιέξοδο, δηλ. μια κατάσταση όπου κανένας τελεστής δεν μπορεί να εφαρμοστεί, οπότε κάνει ‘οπισθοδρόμηση’ (backtracking). Η οπισθοδρόμηση αναγκάζει τη διαδικασία να γυρίσει πίσω σε μια κατάσταση όπου εφαρμόζει τον επόμενο τελεστή από εκείνον που εφήρμοσε, παράγει τα παιδιά της, συνεχίζει με το πρώτο από αυτά κ.ο.κ.

Στο Σχήμα 11.2 απεικονίζεται ο τρόπος αναζήτησης κατά βάθος, δηλ. η σειρά δημιουργίας των καταστάσεων (ως κόμβων ενός δέντρου). Τα T1, T2, T3 και T4 παριστάνουν τους τελεστές μετάβασης.



Σχήμα 11.2 Αναζήτηση κατά βάθος με οπισθοδρόμηση

Ο αλγόριθμος της κατά βάθος αναζήτησης σε ψευδοκώδικα παρουσιάζεται παρακάτω. Παρατηρείστε ότι η μόνη διαφορά έγκειται στο βήμα 2.2.3 όπου τα παραγόμενα παιδιά αποθηκεύονται στην αρχή αντί στο τέλος της λίστας.

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα) και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω X, είναι ο στόχος
   2.1 Αν είναι, τότε σταμάτα (επιτυχία)
   2.2 Αν δεν είναι, τότε
      2.2.1 Παράγαγε τα παιδιά του X και βάλε το X στην closed
      2.2.2 Διάγραψε όσα παιδιά του X υπάρχουν στην closed
      2.2.3 Εισάγαγε τα υπόλοιπα παιδιά του X στην αρχή της open
3. Σταμάτα (αποτυχία)

Η αντίστοιχη υλοποίηση σε LISP διαφέρει μόνο σε μια γραμμή:

(open (append childs open))

αντί (open (append open childs)), που καθορίζει το πού τοποθετούνται τα παραγόμενα παιδιά στη λίστα open.

Είτε με τη μία είτε με την άλλη στρατηγική ουσιαστικά αναζητούμε τη λύση δημιουργώντας όλες τις δυνατές καταστάσεις που μπορούν να δημιουργηθούν. Το ζητούμενο όμως στην Τεχνητή Νοημοσύνη είναι πώς θα μπορέσουμε να το αποφύγουμε αυτό κάνοντας την αναζήτηση ‘έξυπνη’. Αυτό επιτυγχάνεται με τη χρήση ‘ευριστικών’ (heuristics), δηλ. έξυπνων τρικ ή κανόνων που συντομεύουν την αναζήτηση. Τα ευριστικά αυτά σχετίζονται με το συγκεκριμένο πρόβλημα κάθε

φορά. Έτσι, δεν αναπτύσσουμε όλα τα παιδιά μιας κατάστασης, αλλά μόνο τα ‘καλύτερα’, δηλ. αυτά που με βάση το ευριστικό μας «υπόσχονται» πιο σύντομη αναζήτηση. Μ’ αυτόν τον τρόπο δημιουργούνται ευριστικές στρατηγικές, που είναι παραλλαγές των παραπάνω τυφλών στρατηγικών. Π.χ. η στρατηγική που ονομάζεται ‘αναζήτηση δέσμης’ (ή ‘ακτινωτή αναζήτηση’) (beam search) λειτουργεί όπως η αναζήτηση κατά πλάτος, μόνο που κάθε φορά επιλέγει τα καλύτερα m από τα παιδιά κάθε κατάστασης για να αναπτύξει.

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα) και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω Χ, είναι ο στόχος
   2.1 Αν είναι, τότε σταμάτα (επιτυχία)
   2.2 Αν δεν είναι, τότε
      2.2.1 Παράγαγε τα παιδιά του Χ και βάλε το Χ στην closed
      2.2.2 Διάγραψε όσα παιδιά του Χ υπάρχουν στην closed
      2.2.3 Διάταξε τα παιδιά του Χ με βάση το ευριστικό
      2.2.4 Εισάγαγε τα παιδιά στην αρχή της open
3. Σταμάτα (αποτυχία)

```
(defun hill-search (init-state final-states)
      (let ((open (list init-state))
            (closed nil))
        (beam-solve open closed final-states))

(defun hill-solve (open closed final-states)
      (if (null open) nil
         (let ((cur-state (pop open))
            (if (member cur-state final-states)
               (show-solution-path cur-state closed)
               (let* ((childs (make-childs cur-state))
                        (closed (push cur-state closed))
                        (childs (remove-closed-childs childs closed))
                        (childs (sort-childs childs))
                        (open (append childs open)))
                  (breadth-solve open closed final-states))))))
```

Επίσης, η ‘αναρρίχηση λόφων’ (hill climbing) είναι μια παραλλαγήν της αναζήτησης κατά βάθος, όπου δεν επιλέγεται κάθε φορά το πρώτο παιδί για ανάπτυξη, αλλά το καλύτερο, με βάση το ευριστικό (υπάρχουν διάφορες παραλλαγές του αλγορίθμου). Αυτές οι στρατηγικές βρίσκουν εν γένει τη λύση σε λιγότερα

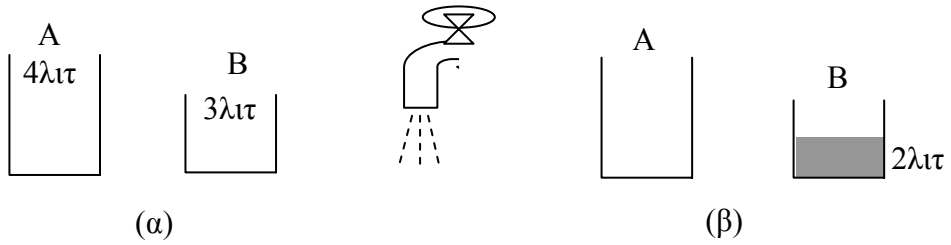
βήματα, αλλά χρειάζεται προσοχή στον ορισμό του ευρετικού και στη επιλογή του m (πόσο μικρό ή μεγάλο πρέπει να είναι).

Στα παραπάνω δύο πλαίσια φαίνεται ο αλγόριθμος αναρρίχησης λόφων σε ψευδοκώδικα και σε LISP. Παρατηρείστε ότι η διαφορά από τον αλγόριθμο αναζήτησης κατά βάθος έγκειται στο ότι έχει προστεθεί ένα επί πλέον βήμα για την διάταξη των παιδιών μιας κατάστασης. Επομένως, εδώ πρέπει να υλοποιηθεί επί πλέον και η συνάρτηση sort-childs, που εξαρτάται από το ευριστικό (βλ. παρακάτω).

1.2 Το Πρόβλημα των Δύο Δοχείων

Ας δούμε για παράδειγμα το γνωστό πρόβλημα των δύο δοχείων: «Υπάρχουν δύο δοχεία χωρητικότητας 4 και 3 λίτρων αντίστοιχα και μια βρύση. Κατ' αρχήν, τα δοχεία είναι άδεια. Θέλουμε να απομονώσουμε στο δεύτερο δοχείο ποσότητα 2 λίτρων. Οι δυνατές ενέργειες είναι: γέμισμα των δοχείων από τη βρύση, άδειασμα των δοχείων στο έδαφος, άδειασμα του ενός δοχείου στο άλλο, μερικώς ή ολικώς.». Στο Σχήμα 11.3 φαίνεται η αρχική (α) και μια τελική κατάσταση (β) του προβλήματος.

Για να μπορέσουμε να λύσουμε τομπρόβλημα με αναζήτηση θα πρέπει πρώτα να βρούμε ένα τρόπο αναπαράστασης μιας κατάστασης, ώστε να μπορέσουμε να αναπαραστήσουμε την αρχική και την τελική (ή τις τελικές) καταστάσεις. Μετά πρέπει να προσδιορίσουμε τους τελεστές μετάβασης, το ευριστικό και μετά να εφαρμόσουμε μια στρατηγική αναζήτησης.



Σχήμα 11.3 Το πρόβλημα των δύο δοχείων

Για το πρόβλημα των δύο δοχείων, ορίζουμε σαν αναπαράσταση μιας τυχαίας κατάστασης μια λίστα με δύο στοιχεία:

(x y)

όπου x η ποσότητα νερού στο δοχείο των 4 λίτρων (με δυνατές τιμές 0, 1, 2, 3, 4) και y στο δοχείο των 3 λίτρων (με δυνατές τιμές 0, 1, 2, 3). Οπότε η αρχική και οι τελικές καταστάσεις παριστάνονται:

Αρχική κατάσταση: (0 0)

Τελικές καταστάσεις : (2 2), (0 2), (1 2), (3 2), (4 2)

Παρατηρείστε ότι το κοινό στοιχείο των τελικών καταστάσεων είναι ότι το δεύτερο στοιχείο κάθε λίστας, που παριστάνει την ποσότητα νερού στο δοχείο Β, είναι '2'. Σκοπός μα είναι να φτάσουμε σε μια από αυτές, ξεκινώντας από την αρχική.

Οι τελεστές μετάβασης έχουν ως εξής:

ΤΕΛΕΣΤΗΣ:ΠΕΡΙΓΡΑΦΗ	**ΠΡΟΫΠΟΘΕΣΕΙΣ**	**ΑΠΟΤΕΛΕΣΜΑ**
Τ1: Γέμισε το Α	x<4	(4 y)
Τ2: Γέμισε το Β	y<3	(x 3)
Τ3:Άδειασε το Α	x>0	(0 y)
Τ4: Άδειασε το Β	y>0	(x 0)
Τ5: Άδειασε το Α στο Β	x>0, y<3	Αν x≥3-y τότε ((x-(3-y)) 3), αλλιώς (0 (y+x))
Τ6: Άδειασε το Β στο Α	x<4, y>0	Αν y≥4-x τότε (4 (y-(4-x))), αλλιώς ((y+x) 0)

Στη συνέχεια δίνουμε τρόπους υλοποίησης των τελεστών μετάβασης. Π.χ. ο τελεστής Τ1 υλοποιείται ως εξής:

```
(defun t1 (state)
  (let((x (car state))
       (y (second state)))
    (if (<x 4)
        (let ((newstate (list 4 y)))
          (prog1 newstate
                 (format t "~%~3a ΓΕΜΙΣΕ ΤΟ ΔΟΧΕΙΟ Α ~3a" state newstate))))))
```

Παράδειγμα εφαρμογής: (t1 '(1 3)) → (1 3) ΓΕΜΙΣΕ ΤΟ ΔΟΧΕΙΟ Α (4 3)

Δηλ.εκτυπώνεται η αρχική κατάσταση, η περιγραφή του τελεστή και η κατάσταση που προκύπτει μετά την εφαρμογή του.

Ομοίως υλοποιείται και ο Τ2. Ο Τ5 μπορεί να υλοποιηθεί ως εξής:

```
(defun t5 (state)
   (let ((x (car state))
         (y (second state)))
     (if (and (> x 0) (< y 3))
         (let ((z (- 3 y)))
           (if (or (> x z) (= x z))
              (let ((newstate (list (- x (- 3 y)) 3)))
                 (print-state-t5 state newstate))
              (let ((newstate (list 0 (+ y x))))
                 (print-state-t5 state newstate)))))))
```

```
(defun print-state-t5 (state newstate)
  (prog1 newstate
         (format  t "~3a ΑΔΕΙΑΣΕ ΤΟ ΔΟΧΕΙΟ Α ΣΤΟ Β ~3a" state newstate)))
```

Παράδειγμα εφαρμογής: (t5 '(3 1)) → (4 0) ΑΔΕΙΑΣΕ ΤΟ ΔΟΧΕΙΟ Α ΣΤΟ Β (1 3)

Μια ευριστική συνάρτηση είναι αυτή που υλοποιεί το ευριστικό, παριστάνεται δε ως h(n), όπου n είναι είναι μια κατάσταση. Στην περίπτωση των δύο δοχείων μπορούμε να ορίσουμε σαν ευριστικό σε μια κατάσταση (x, y) το πόσο κοντά στα 2 λίτρα βρίσκεται η ποσότητα νερού στα δύο δοχεία. Επίσης, συνήθως επιδιώκουμε να είναι h(n)=0 στις καταστάσεις-στόχους. Γι' αυτό, εκφράζουμε την ευριστική συνάρτηση ως εξής:

$$h(n) = \begin{cases} (|2-x|+|2-y|)/2 & \text{αν } x, y \neq 2 \\ 0 & \text{αν } x=2 \text{ ή } y=2 \end{cases}$$

Αυτό μπορεί να υλοποιηθεί σε LISP ως εξής:

```
(defun h-value (state)
    (let ((x (car state))
          (y (second state)))
      (if (or (= x 2) (= y 2)) 0
          (float (/ (+ (abs (- 2 x)) (abs (- 2 y))) 2)))))
```

Παραδείγματα εφαρμογής: (h-value '(3 1)) → 1, (h-value '(3 2)) → 0

Στη συνέχεια, απαιτείται η υλοποίηση της συνάρτησης sort-childs, όπως προαναφέραμε. Αυτή μπορεί να γίνει ως ακολούθως. Η συνάρτηση sort-childs είναι η συνάρτηση πρώτου επιπέδου:

```
(defun sort-childes (states)
   (sort-states (compute-h states)))
```

η οποία παίρνει ένα όρισμα (states), που είναι μια λίστα καταστάσεων, καλεί την compute-h, που υπολογίζει τις ευριστικές τιμές των καταστάσεων, και την sort-states, που πραγματοποιεί τη διάταξη, και τελικά επιστρέφει τη λίστα διατεταγμένη.

Παράδειγμα εφαρμογής: (sort-nodes‘((4 3) (3 3) (4 1) (1 3))) →
((3 3) (1 3) (4 3) (4 1))

Η compute-h παίρνει σαν είσοδο μια λίστα καταστάσεων και επιστρέφει μια λίστα ζευγών, όπου κάθε ζεύγος αποτελείται από μια καταστάση και την αντίστοιχη ευρετική τιμή.

```
(defun compute-h (states)
    (let ((newstates nil))
        (dolist (state states (reverse newstates))
            (push (cons state (h-value state)) newstates))))
```

Παράδειγμα εφαρμογής: (compute-h ‘((4 3) (3 3) (4 1) (1 3))) →
(((4 3) . 1.5) ((3 3) . 1.0) ((4 1) . 1.5) ((1 3) . 1.0))

Η sort-states παίρνει σαν είσοδο την έξοδο της compute-h και επιστρέφει μια λίστα με διατεταγμένες τις καταστάσεις με βάση τις ευριστικές τιμές τους.

```
(defun sort-states (h-states)
  (do* ((n-states h-states (remove-state min-state n-states))
        (min-state (find-minstate h-states) (find-minstate n-states))
        (s-states nil))
       ((null n-states) (reverse s-states))
    (push (car min-state) s-states)))
```

Παράδειγμα εφαρμογής:
(sort-states ‘(((4 3) . 1.5) ((3 3) . 1.0) ((4 1) . 1.5) ((1 3) . 1.0))) →
((3 3) (1 3) (4 3) (4 1))

```
(defun find-minstate (states)
  (let ((minstate (car states)))
    (dolist (state (cdr states) minstate)
      (if (< (cdr state) (cdr minstate))
        (setf minstate state)))))

(defun remove-state (minstate states)
  (let ((newstates nil))
    (dolist (state states (reverse newstates))
      (if (not (equalp state minstate))
        (push state newstates)))))
```

Η sort-states, που είναι η συνάρτηση που πραγματοποιεί τη διάταξη, για να το κάνει αυτό χρησιμοποιεί (καλεί) δύο άλλες συναρτήσεις, τις find-minstate και remove-state, από τις οποίες η μεν πρώτη βρίσκει το ελάχιστο στοιχείο μιας λίστας καταστάσεων, ενώ η δεύτερη αφαιρεί μια κατάσταση από τη λίστα καταστάσεων (βλ. παραπάνω πλαίσιο). Ουσιαστικά, η sort-states, με τη βοήθεια των δύο αυτών συναρτήσεων, υλοποιεί τον αλγόριθμο διάταξης με επιλογή (ένας άλλος τέτοιος αλγόριθμος θα μπορούσε να είναι ο αλγόριθμος bubble sort).

Βιβλιογραφία

1. G. L. Steele JR, “Common Lisp, The Language”, Digital Press, 1984 (διαθέσιμο on-line στη διεύθυνση http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html)
2. P. H. Winston and B. K. P. Horn, LISP, 3rd Edition, Addison Wesley, 1989.
3. G. F. Luger and W. A. Stubblefield, Artificial Intelligence and the Design of Expert Systems (Κεφ. 7), Benjamin/Cummings, 1989.
4. D. S. Touretzky, “COMMON LISP: A Gentle Introduction to Symbolic Computation”, Benjamin/Cummings, 1990 (διαθέσιμο on-line στη διεύθυνση http://www.cs.cmu.edu/~dst/LispBook/)