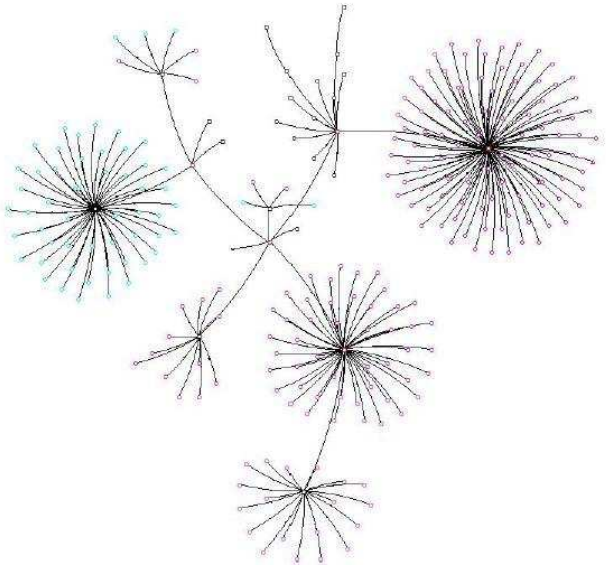
ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΟΜΕΑΣ ΘΕΩΡΗΤΙΚΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

Αλγόριθμοι και Πολυπλοκότητα

Β. Ζησιμόπουλος



Αθήνα, 2008

Ευχαριστίες

Ευχαριστώ θερμά τους συνεργάτες μου στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών Μαρία Λιάζη, Γεράσιμο Πολλάτο, Ορέστη Τελέλη, Δημήτρη Ψούνη και τους μεταπτυχιακούς φοιτητές Γεράσιμο Δημητρίου, Χρήστο Σόφη, Αθανάσιο Περπερή και Πέτρο Χριστόπουλο, οι οποίοι βοήθησαν στην επιμέλεια και στην μείωση των λαθών αυτής της έκδοσης. Αναμφίβολα θα υπάρχουν και άλλα λάθη τα οποία ευελπιστούμε να ελαχιστοποιήσουμε στην επόμενη έκδοση.

Β. Ζησιμόπουλος

Περιεχόμενα

1 Εισαγωγή **1**
- 1.1 Ανάλυση Αλγορίθμων . . . 2
- 1.2 Ασυμπτωτική πολυπλοκότητα . . . 5
  - 1.2.1 Κλάσεις Πολυπλοκότητας . . . 8
  - 1.2.2 Ιδιότητες ασυμπτωτικών συμβολισμών . . . 12
  - 1.2.3 Ασκήσεις . . . 14

2 Αναδρομή **21**
- 2.1 Divide and Conquer . . . 21
  - 2.1.1 Merge-Sort . . . 21
  - 2.1.2 Binary Search . . . 25
  - 2.1.3 Οι πύργοι του Hanoi . . . 26
  - 2.1.4 Η ακολουθία Fibonacci . . . 28
  - 2.1.5 Ο αλγόριθμος Min-Max . . . 30
  - 2.1.6 QuickSort . . . 32
- 2.2 Ασυμπτωτική προσέγγιση μερικών αθροισμάτων . . . 35
- 2.3 Δένδρα Απόφασης . . . 36
- 2.4 The Master Theorem . . . 37
  - 2.4.1 Το Master θεώρημα . . . 39
  - 2.4.2 Παραδείγματα . . . 40
- 2.5 Αναδρομικές γραμμικές εξισώσεις . . . 40
  - 2.5.1 Γραμμικές αναδρομές τάξης k . . . 41
  - 2.5.2 Αναδρομές διαμερίσεων . . . 41
  - 2.5.3 Αναδρομές πλήρεις, γραμμικές ή πολυωνυμικές . . . 42
  - 2.5.4 Παραδείγματα Γραμμικών Αναδρομικών Εξισώσεων . . . 42
  - 2.5.5 Γραμμικές Αναδρομικές Εξισώσεις με σταθερούς συντελεστές 43
  - 2.5.6 Εξισώσεις μετατρεπόμενες σε γραμμικές αναδρομές . . . 44

3 Σωροί **47**
- 3.1 Δένδρα . . . 47
- 3.2 Η δομή σωρού . . . 48
  - 3.2.1 Υλοποίηση με πίνακα ενός σωρού . . . 49
  - 3.2.2 Εισαγωγή ενός νέου στοιχείου στο σωρό . . . 49
  - 3.2.3 Διαγραφή ενός στοιχείου από το σωρό . . . 50

3.2.4 Κατασκευή σωρού 52
3.3 Ταξινόμηση με τη βοήθεια σωρού 55

4 Κατακερματισμός (Hashing) 57
4.1 Εισαγωγή 57
4.2 Πίνακες Άμεσης Διευθυνσιοδότησης (Direct Address Tables) . . . 57
4.3 Πίνακες Κατακερματισμού (Hash Tables) 59
4.3.1 Επίλυση Συγκρούσεων με Λίστες (Collision Resolution by Chaining) 60
4.3.2 Ανάλυση Κατακερματισμού με Λίστες 61
4.4 Συναρτήσεις Κατακερματισμού (Hash Functions) 63
4.4.1 Η Μέθοδος της Διαίρεσης (Division Method) 63
4.4.2 Η Πολλαπλασιαστική Μέθοδος (Multiplication Method) . 64
4.5 Ανοιχτή Διευθυνσιοδότηση (Open Addressing) 65
4.5.1 Γραμμική Διερεύνηση (Linear Probing) 67
4.5.2 Τετραγωνική Διερεύνηση (Quadratic Probing) 67
4.5.3 Διπλός Κατακερματισμός (Double Hashing) 67
4.5.4 Ανάλυση του Κατακερματισμού Ανοιχτής Διευθυνσιοδότησης 68

5 Γράφοι 71
5.1 Γενικά 71
5.2 Ορισμοί 74
5.2.1 Κατευθυνόμενοι γράφοι 75
5.2.2 Μη κατευθυνόμενοι γράφοι 75
5.2.3 Κύκλοι και μονοπάτια 75
5.3 Ειδικές Κατηγορίες Γράφων 78
5.3.1 Πλήρεις Γράφοι 78
5.3.2 Διμερείς Γράφοι 79
5.3.3 Κανονικοί Γράφοι 80
5.3.4 Επίπεδοι γράφοι 80
5.3.5 Υπογράφοι (subgraph) 80
5.3.6 Συνεκτικοί γράφοι 81
5.3.7 Χορδικοί Γράφοι (Chordal Graphs) 81
5.3.8 Δέντρα (trees) 82
5.4 Πράξεις επι των γράφων 83
5.4.1 Ισομορφισμός γράφων 83
5.4.2 Συμπλήρωμα ενός Γράφου 83
5.4.3 Διαγραφή (deletion) 84
5.5 Αποθήκευση Γράφων 85
5.5.1 Πίνακας Γειτνίασης 85
5.5.2 Πίνακας Πρόσπτωσης 87
5.5.3 Λίστες Γειτνίασης 88

6 Αλγόριθμοι Γράφων **91**
6.1 Εξερεύνηση Γράφων 91
6.1.1 Γενική μορφή ενός αλγορίθμου εξερεύνησης 91
6.1.2 Εξερεύνηση Πρώτα Κατά Πλάτος 92
6.1.3 Εξερεύνηση Πρώτα Κατά Βάθος 93
6.1.4 Εφαρμογές των αλγορίθμων εξερεύνησης 95
6.2 Συνεκτικότητα 98
6.3 Τοπολογική Ταξινόμηση 100
6.4 Πράξεις σε ασύνδετα σύνολα (disjoint sets) 102
6.5 Ελάχιστα Μονοπάτια 105
6.5.1 Ο αλγόριθμος του Dijkstra 108
6.5.2 Ο αλγόριθμος του Bellman 112
6.5.3 Ο αλγόριθμος του Bellman για γράφους χωρίς κύκλο . . . 118
6.6 Δέντρα Επικάλυψης 119
6.6.1 Πρόβλημα εύρεσης ενός δέντρου επικάλυψης ελάχιστου κόστους 121
6.6.2 Αλγόριθμος Prim 124
6.6.3 Αλγόριθμος Prim (nearest neighbour) 127
6.6.4 Αλγόριθμος Kruskal 129
6.7 ΔΕΕΚ και το πρόβλημα του Πλανόδιου Πωλητή (TSP) 133

7 Άπληστοι Αλγόριθμοι **137**
7.1 Γενική μορφή ενός Άπληστου Αλγόριθμου 137
7.2 Ανάθεση ενός πόρου 138
7.3 Αποθήκευση αρχείων σε δίσκους 141
7.4 Το Διακριτό Πρόβλημα Σακιδίου (Discrete Knapsack) 143
7.5 Το Συνεχές Πρόβλημα Σακιδίου (Continuous Knapsack) 144

8 Δυναμικός Προγραμματισμός **145**
8.1 Εισαγωγικά 145
8.2 Βασικά στοιχεία του δυναμικού προγραμματισμού 146
8.2.1 Βέλτιστα διασπώμενη δομή (optimal substructure) 146
8.2.2 Επικαλυπτόμενα υποπροβλήματα (Overlapping subproblems) 148
8.3 Εφαρμογές 148
8.3.1 Χρονοδρομολόγηση γραμμής παραγωγής 149
8.3.2 Αλυσιδωτός πολλαπλασιασμός πινάκων 152
8.3.3 Το πρόβλημα του σακιδίου 156

9 Εξαντλητική Αναζήτηση **161**
9.1 Το πρόβλημα των βασιλισσών 161

10 NP-πληρότητα **167**
10.1 Εισαγωγή 167
10.2 Οι κλάσεις πολυπλοκότητας P και NP 168
10.3 Προβλήματα Απόφασης και $NP-$πληρότητα 170
10.3.1 Το πρόβλημα της ικανοποιησιμότητας 171

10.4 Προβλήματα Συνδυαστικής Βελτιστοποίησης 172
10.5 Προβλήματα $NP-$δύσκολα 172
10.6 Αναγωγές 173
10.7 Αγνώστου Κατάστασης Προβλήματα 174

Κεφάλαιο 1

Εισαγωγή

Η έννοια του αλγορίθμου είναι θεμελιώδης στον χώρο της επιστήμης των υπολογιστών. Χωρίς αλγόριθμους δεν υπάρχουν προγράμματα και χωρίς προγράμματα δεν υπάρχει τίποτα για να τρέξει στον υπολογιστή.

Η δημιουργία αποδοτικών αλγορίθμων αποτέλεσε πρόκληση για τους επιστήμονες των περασμένων δεκαετιών. Όχι μόνο για την επίλυση με την χρήση υπολογιστή, προβλημάτων που ανέκυπταν σε όλους τους τομείς της επιστήμης, αλλά και για την συνεχή βελτίωσή τους, ώστε να τα λύνουν γρηγορότερα και καλύτερα.

Σήμερα ακόμη περισσότερο, η ανάγκη σχεδίασης αποδοτικών αλγορίθμων, είναι μεγαλύτερη γιατί και τα προβλήματα που αντιμετωπίζει η ανθρωπότητα είναι μεγαλύτερα, δυσκολότερα και πιο περίπλοκα:

- Η αποκωδικοποίηση του ανθρωπίνου DNA, που αποσκοπεί στην αναγνώριση των 100.000 περίπου γονιδίων του ανθρώπου. Η αναζήτηση των γονιδίων γίνεται σε περίπου 3 εκατομμύρια βασικά χημικά ζευγάρια που δημιουργούν το ανθρώπινο γονιδίωμα, τα οποία είναι πεπλεγμένα και αταξινόμητα.

- Στο διαδίκτυο, του οποίου το ολοένα και αυξανόμενο μέγεθος δημιουργεί προβλήματα στον εντοπισμό της χρήσιμης πληροφορίας. Για παράδειγμα, οι μηχανές αναζήτησης πρέπει μέσα σε κλάσματα δευτερολέπτου να απαντούν στα ερωτήματά μας, ψάχνοντας σε μερικά δισεκατομμύρια ιστοσελίδων. Ακόμη περισσότερο, η δρομολόγηση πακέτων πληροφορίας πρέπει να βρίσκει σε μηδενικό χρόνο τον πιο γρήγορο δρόμο, ώστε να είμαστε ικανοποιημένοι με την ταχύτητα πρόσβασης των δεδομένων.

- Στο ηλεκτρονικό εμπόριο και γενικότερα στην επικοινωνία μέσω διαδικτύου, η εμπιστοσύνη στην υπηρεσία είναι προϋπόθεση, λόγω κινδύνου διαρροής προσωπικών δεδομένων (πιστωτικές κάρτες, διευθύνσεις κ.λ.π.). Το πρόβλημα απαιτεί τη δημιουργία γρήγορων και απαραβίαστων αλγορίθμων κρυπτογράφησης της διακινούμενης πληροφορίας.

- Στον εμπορικό κόσμο, η επιλογή στρατηγικών αποτελεί ένα περίπλοκο πρόβλημα. Για παράδειγμα, τα προβλήματα δρομολόγησης σε μια αεροπορική εταιρεί-

α, του σχεδιασμού οδικών αξόνων, της επιλογής μηχανών σε ένα εργοστάσιο, της επιλογής διαφημιστικής εκστρατείας, επιλέγονται από διαφορετικές δυνατές στρατηγικές με σκοπό την κοινή ωφέλεια ή το κέρδος. Αποδοτικοί αλγόριθμοι χρειάζονται ώστε να βρίσκεται η καλύτερη δυνατή στρατηγική από τις διαθέσιμες στρατηγικές, σε ικανοποιητικό χρόνο, παρά τα τεράστια μεγέθη δεδομένων των προβλημάτων.

Τι είναι όμως ένας αλγόριθμος; Πως ξεχωρίζουμε έναν καλό από έναν κακό αλγόριθμο; Πότε μπορούμε να κρίνουμε ότι ένας αλγόριθμος είναι ικανοποιητικός; Πως κατασκευάζονται αποδοτικοί αλγόριθμοι; Πως μπορούμε να εφαρμόζουμε έξυπνες αλγοριθμικές ιδέες στα προβλήματα που αντιμετωπίζουμε; Αυτά τα ερωτήματα είναι που πρέπει να απαντήσουμε, ώστε να κάνουμε τους αλγόριθμους ένα πολύ ισχυρό εφόδιο στο οπλοστάσιό μας.

1.1 Ανάλυση Αλγορίθμων

Ένας αλγόριθμος είναι μια υπολογιστική διαδικασία που παίρνει μία ή περισσότερες τιμές σαν είσοδο και παράγει μία ή περισσότερες τιμές σαν έξοδο. Έτσι ένας αλγόριθμος είναι μία ακολουθία υπολογιστικών βημάτων που μετασχηματίζει την είσοδο σε έξοδο.

Η εκτέλεση ενός αλγορίθμου πρέπει να είναι τυφλή εφαρμογή των κανόνων ή των υπολογιστικών βημάτων από τα οποία αυτός αποτελείται. Κανένας αλγόριθμος δηλαδή δεν 'σκέφτεται' αλλά ακολουθεί βήμα προς βήμα τους κανόνες του. Ένας αλγόριθμος καλά ορισμένος πρέπει να έχει τα εξής χαρακτηριστικά:

1. Να είναι πεπερασμένος, δηλαδή να τερματίζει πάντα μετά από έναν πεπερασμένο αριθμό βημάτων.

2. Να είναι επακριβώς ορισμένος. Κάθε βήμα του αλγορίθμου πρέπει να είναι επακριβώς ορισμένο, δηλαδή οι ενέργειες που πρέπει να γίνουν δεν πρέπει να είναι αμφίσημες. Για τον λόγο αυτό για την περιγραφή του αλγορίθμου ακολουθείται κάποια γλώσσα προγραμματισμού (που προσφέρει τον επακριβή ορισμό των εννοιών), ή κάποιου είδους ψευδογλώσσα.

3. Τα δεδομένα εισόδου, εξόδου και τα βοηθητικά δεδομένα που δημιουργούνται κατά τη διάρκεια εκτέλεσης του αλγορίθμου, πρέπει να κωδικοποιούνται από κάποιο πεπερασμένο αριθμό συμβόλων (π.χ., το σύνολο $\{0, 1\}$).

4. Να είναι ορθός. Δηλαδή, η έξοδος να είναι λύση στο υπό μελέτη πρόβλημα. Η ορθότητα του αλγορίθμου θα πρέπει να είναι δυνατόν να αποδειχθεί με μαθηματικά εργαλεία.

Μία διαδικασία που έχει όλα τα χαρακτηριστικά που αναπτύξαμε αλλά δεν είναι σίγουρο αν είναι πεπερασμένη, ονομάζεται υπολογιστική μέθοδος. Για παράδειγμα, ένα πρόγραμμα που υπολογίζει επακριβώς τα ψηφία του π δεν είναι αλγόριθμος γιατί το πρόγραμμα αυτό δεν τερματίζει ποτέ, δεν πρόκειται κάποια στιγμή (έστω σε άπειρο χρόνο) να ολοκληρώσει τον υπολογισμό και να παράγει μία έξοδο σαν

αποτέλεσμα. Αντίθετα, αλγόριθμο μπορούμε να χαρακτηρίσουμε ένα πρόγραμμα που υπολογίζει τα πρώτα 2^{1000} ψηφία του π.

Επιπρόσθετα δεν σημαίνει ότι κάθε αλγόριθμος μπορεί να προγραμματιστεί, δηλαδή μια καλά ορισμένη αλγοριθμική διαδικασία δεν είναι απαραίτητο ότι είναι προγραμματίσιμη. Αυτό μπορεί να συμβαίνει λόγω περιορισμού της χρησιμοποιούμενης γλώσσας, αδυναμίας των υπολογιστών να εκφράσουν μαθηματικές έννοιες κ.λ.π. Ωστόσο οι εφαρμογές και τα παραδείγματα που θα χρησιμοποιήσουμε στην συνέχεια, δεν θα υπεισέρχονται σε αυτές τις ειδικές κατηγορίες. Οπότε χάριν απλότητας θα εννοήσουμε ταυτόσημες τις έννοιες αλγόριθμος και πρόγραμμα.

Ένα πρόγραμμα θα είναι χρήσιμο, αν εκτελείται σε 'λογικό' χρόνο (δηλαδή δεν χρειάζεται αιώνες για να ολοκληρωθεί) και δεσμεύει 'λογικό' χώρο μνήμης (όχι παραπάνω από όσο μπορούμε να έχουμε σε ένα καθημερινό σύστημα). Για να μπορούμε να διαχωρίσουμε τα χρήσιμα από τα άχρηστα προγράμματα, θα πρέπει να βρεθεί ένας τρόπος(μέτρο) για τον χαρακτηρισμό του προγράμματος. Αυτός είναι η πολυπλοκότητα χρόνου (δηλαδή πόσο γρήγορος είναι ο αλγόριθμος), και η πολυπλοκότητα χώρου (δηλαδή πόση μνήμη χρειάζεται το πρόγραμμα για να τρέξει).

Η πολυπλοκότητα ενός αλγορίθμου εκφράζεται σαν συνάρτηση της διάστασης του υπό μελέτη προβλήματος. Η διάσταση του προβλήματος είναι το πλήθος των ατομικών δεδομένων για επεξεργασία, όπως για παράδειγμα το πλήθος των στοιχείων μιας ακολουθίας, ή ενός πίνακα, τα bits ενός αριθμού, κ.ο.κ. Ας δούμε τώρα την έννοια της πολυπλοκότητας με ένα παράδειγμα:

Το Πρόβλημα της Αναζήτησης

Δίνεται μια ακολουθία n στοιχείων $S = (a_1, a_2, \ldots, a_n)$ και ένα στοιχείο x. Είναι το x μέλος του S και αν ναι σε ποια θέση;

Ένας απλός αλγόριθμος είναι ο εξής: Ψάξε ένα ένα τα στοιχεία της ακολουθίας. Αν βρεις αυτό που ψάχνεις σταμάτα. Ας τον διατυπώσουμε σε αλγοριθμική γλώσσα (ψευδογλώσσα) όπως φαίνεται στο σχήμα 1.1.

ΣΕΙΡΙΑΚΗ ΑΝΑΖΗΤΗΣΗ (S, x)

1. $i = 1$
2. while $i \neq n + 1$ and $a_i \neq x$ do
3. $\quad i = i + 1$
4. end while
5. if $i > n$
6. $\quad$ return -1
7. else
8. $\quad$ return i
9. end if

Σχήμα 1.1: Αλγόριθμος αναζήτησης ενός στοιχείου x σε μονοδιάστατο πίνακα S.

Πόσες πράξεις θα κάνει ο αλγόριθμος αυτός για κάποια δεδομένα εισόδου; Αν το x είναι το πρώτο στοιχείο της ακολουθίας, τότε θα κάνει 3 συγκρίσεις (από τον πρώτο έλεγχο της WHILE και την εντολή IF) και μία καταχώρηση (την $i = 1$). Αν το x είναι το δεύτερο στοιχείο της ακολουθίας, τότε θα κάνει 5 συγκρίσεις και 3 καταχωρήσεις και γενικά αν είναι το i-οστό στοιχείο θα κάνει $2i + 1$ συγκρίσεις και $i + 1$ καταχωρήσεις. Αντίθετα, αν το x δεν ανήκει στην ακολουθία, τότε ο αλγόριθμος θα εκτελέσει $2n$ συγκρίσεις (στην τελευταία επανάληψη υποθέτουμε ότι θα κάνει και τις δύο συγκρίσεις) και $n + 1$ καταχωρήσεις.

Η πρώτη περίπτωση είναι η καλύτερη δυνατή (δηλαδή ο αλγόριθμος θα εκτελέσει τον ελάχιστο αριθμό βημάτων). Η δεύτερη περίπτωση είναι η χειρότερη δυνατή (θα εκτελέσει το μέγιστο δυνατό αριθμό βημάτων).

Αν θεωρήσουμε ως D_n το σύνολο των διαφορετικών εισόδων στο πρόβλημα και $d \in D_n$ μια είσοδό του, τότε οι ορίζονται τα παρακάτω μέτρα, κάθε ένα από τα οποία μας δίνει μία διαφορετική αίσθηση της αποδοτικότητάς του αλγορίθμου:

- Πολυπλοκότητα στην βέλτιστη περίπτωση:
  $C_{\beta\pi}(n) = \min\{\text{κόστος}(d),\, d \in D_n\}$
- Πολυπλοκότητα στην χείριστη περίπτωση:
  $C_{\chi\pi}(n) = \max\{\text{κόστος}(d),\, d \in D_n\}$
- Πολυπλοκότητα κατά μέσο όρο:
  $C_{\mu o}(n) = \sum_{d \in D_n} p(d)\ \text{κόστος}(d)$

Η κατά μέσο όρο πολυπλοκότητα, είναι πόσες πράξεις κάνει (ή πόση μνήμη δεσμεύει) ο αλγόριθμος κατά μέσο όρο. Έτσι μετράμε τις πράξεις (κόστος(d)) που κάνει ο αλγόριθμος για κάθε δυνατή είσοδο $d \in D_n$ που δίνεται με πιθανότητα $p(d)$.

Για παράδειγμα, έστω ότι σαν ακολουθία μας δίνεται η ταξινομημένη ακολουθία $1, \ldots, n$ και σαν στοιχείο προς εξερεύνηση μας δίνεται ισοπίθανα ένα από τα $\{1, 2, \ldots, 2n\}$. Τότε η πολυπλοκότητα κατά μέσο όρο (για συγκρίσεις μόνο) είναι:

$$\begin{aligned} C_{\mu o}(n) &= \frac{1}{2n}3 + \frac{1}{2n}5 + \ldots + \frac{1}{2n}(2n+1) + \frac{n}{2n}(2n+3) \\ &= \frac{1}{2n}\sum_{i=1}^{n}(2i+1) + \frac{2n+3}{2} = \frac{3n+5}{2} \end{aligned}$$

Βλέπουμε λοιπόν ότι όσον αφορά τις συγκρίσεις (που για τον αλγόριθμο αυτό είναι στοιχειώδης πράξη, δηλαδή αυτή που γίνεται περισσότερο και είναι υπολογιστικά βαρύτερη από κάθε άλλη) ο αλγόριθμος έχει 3 στην βέλτιστη, $2n + 3$ στην χείριστη και περίπου $1.5n$ κατά μέσο όρο. Θα μπορούσαμε να πούμε ότι αντίστοιχα κάθε μέγεθος από αυτά μας δίνει μια αισιόδοξη, μια απαισιόδοξη και μία ρεαλιστική άποψη του χρόνου εκτέλεσης του προβλήματος. Ωστόσο όλα μαζί είναι χρήσιμα για να έχουμε μία ολοκληρωμένη άποψη για τον αλγόριθμο.

Ωστόσο, δεν είναι απαραίτητο να έχουμε τέτοιες αποκλίσεις ανάμεσα στις τρεις πολυπλοκότητες. Ας θεωρήσουμε το ακόλουθο πρόβλημα:

Εσωτερικό γινόμενο δύο διανυσμάτων

Έστω $a = (a_i), b = (b_i)$ διανύσματα του $\mathbb{R}^n$. Ζητείται το εσωτερικό γινόμενο αυτών.

ΕΣΩΤΕΡΙΚΟ ΓΙΝΟΜΕΝΟ (a, b)
1. $sp = 0$
2. for $i = 1$ to n do
3. $\quad sp = sp + a_i b_i$
4. end for
5. return sp

Σχήμα 1.2: Υπολογισμός του εσωτερικού γινομένου δύο ίσης διάστασης διανυσμάτων a και b.

Σε αυτόν τον αλγόριθμο ανεξάρτητα από τους αριθμούς των ακολουθιών θα γίνουν n προσθέσεις, n πολλαπλασιασμοί και n καταχωρήσεις. Άρα ισχύει

$$C_{\beta\pi}(n) = C_{\chi\pi}(n) = C_{\mu o}(n) = n$$

Όπως θα δούμε παρακάτω, δεν χρειάζεται να κάνουμε καταμέτρηση όλων των διαφορετικών τύπων πράξεων σε έναν αλγόριθμο, αλλά αρκεί η υπολογιστικά βαρύτερη πράξη. Ακόμη πιο απλά η καταμέτρηση των εκτελούμενων εντολών αρκεί για τον υπολογισμό της πολυπλοκότητας του αλγορίθμου.

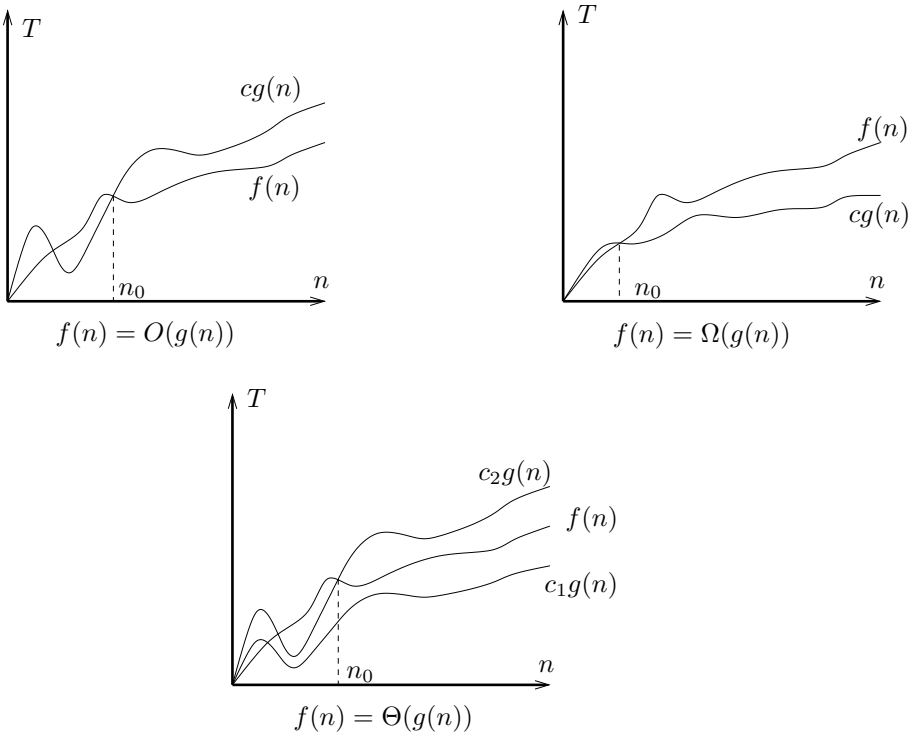
1.2 Ασυμπτωτική πολυπλοκότητα

Έστω ένα πρόβλημα με διάσταση n (π.χ. η αναζήτηση σε μια ακολουθία) και έστω και δύο αλγόριθμοι που το λύνουν, ο A με πολυπλοκότητα $100n$ και ο B με πολυπλοκότητα n^2. Θεωρώντας σαν καλύτερο αυτόν που κάνει λιγότερες πράξεις για την ίδια είσοδο μπορούμε να διακρίνουμε τρεις περιπτώσεις ανάλογα με το n:

$$n : \begin{cases} = 100 & \text{Οι A,B έχουν ίδια πολυπλοκότητα} \\ < 100 & \text{Ο B είναι αποδοτικότερος από τον A} \\ > 100 & \text{ο A είναι αποδοτικότερός από τον B} \end{cases}$$

Παρατηρούμε ότι μετά από ένα κατώφλι, όσο το n γίνεται μεγαλύτερο τόσο ο αλγόριθμος A γίνεται καλύτερος του B. Ακόμη περισσότερο ο ρυθμός με τον οποίο γίνεται ο A καλύτερος του B αυξάνει γραμμικά σε σχέση με το n. Για παράδειγμα, έστω μια μηχανή που εκτελεί 10^6 εντολές το δευτερόλεπτο. Για $n = 1000$ ο αλγόριθμος A εκτελείται σε $100n \times 10^{-6} = 0.1$ δευτερόλεπτα και ο B σε $n^2 \times 10^{-6} = 1$ δευτερόλεπτο, δηλαδή είναι 10 φορές πιο αργός. Για $n = 10000$ ο B είναι 100 φορές πιο αργός, κ.ο.κ.

Άρα όσο αυξάνει το n τόσο η διαφορά της αποδοτικότητας γίνεται μεγάλη. Λαμβάνοντας υπ' όψιν ότι μας ενδιαφέρει περισσότερο τι γίνεται για μεγάλες διαστάσεις δεδομένων και ότι μετά από ένα κατώφλι η τάξη της συνάρτησης πολυπλοκότητας



Σχήμα 1.3: Γραφική Απεικόνιση των συμβολισμών O, Ω, Θ

είναι αυτή που καθορίζει την αποδοτικότητα του αλγορίθμου[1] οδηγούμαστε στην εισαγωγή συμβολισμών, που θα ορίζουν την συμπεριφορά συναρτήσεων (πολυπλοκότητας) ασυμπτωτικά. Οι συμβολισμοί αυτοί διαβάζονται $f(n)$ ίσον (μεγάλο) όμικρον του $g(n)$, ωμέγα του $g(n)$ και θήτα του $g(n)$ και ορίζονται ως εξής:

- $f(n) = O(g(n))$ αν υπάρχουν $c, n_0 > 0$ έτσι ώστε $f(n) \leq cg(n)$ για κάθε $n > n_0$
- $f(n) = \Omega(g(n))$ αν υπάρχουν $c, n_0 > 0$ έτσι ώστε $f(n) \geq cg(n)$ για κάθε $n > n_0$
- $f(n) = \Theta(g(n))$ αν υπάρχουν $c_1, c_2, n_0 > 0$ έτσι ώστε $0 \leq c_1 g(n) \leq f(n) \leq$

[1] Θεωρητικά, ένας αλγόριθμος πολυπλοκότητας $10^{100}n$ είναι καλύτερος από έναν n^2; Ωστόσο τέτοιες συναρτήσεις πολυπλοκότητας δεν εμφανίζονται στην πράξη σε μη τεχνητούς αλγόριθμους, ενώ ακόμη κι αν υπάρχουν μεγάλες σταθερές, αυτές με έρευνα μειώνονται δραστικά.

$c_2 g(n)$ για κάθε $n > n_0$. Εναλλακτικά, αν $f(n) = O(g(n))$ και $f(n) = \Omega(g(n))$, τότε $f(n) = \Theta(g(n))$

Τα $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$ είναι σύνολα συναρτήσεων, δηλαδή είναι όλες οι συναρτήσεις για τις οποίες ισχύει μετά από κάποιο κατώφλι, ότι η f φράσσεται αντίστοιχα από πάνω, από κάτω και είναι ίδιας τάξης με την g. Στην πραγματικότητα δηλαδή, έχουμε ότι $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$ ή $f(n) \in \Theta(g(n))$ αλλά έχει επικρατήσει ο συμβολισμός της ισότητας. Η ισότητα είναι καθαρά συμβολισμός και δεν θα πρέπει να εξάγουμε εσφαλμένα συμπεράσματα όπως παραδείγματος χάριν: $f(n) = O(n)$ και $f(n) = O(n^2)$ άρα $O(n) = O(n^2)$. Μπορούμε να δούμε την σχέση φράγματος που προκύπτει από τους ορισμούς, στις γραφικές παραστάσεις του σχήματος 1.3.

Ας δούμε την ασυμπτωτική συμπεριφορά της συνάρτησης πολυπλοκότητας $f(n) = 3n^2 - 100n + 6$:

- $3n^2 - 100n + 6 = O(n^2)$ αφού $3n^2 - 100n + 6 < 3n^2$ για $n > 0$.
- $3n^2 - 100n + 6 = O(n^3)$ αφού $3n^2 - 100n + 6 < n^3$ για $n > 0$.
- $3n^2 - 100n + 6 \neq O(n)$ αφού $\nexists c : 3n^2 - 100n + 6 < cn$ για $n > c$.
- $3n^2 - 100n + 6 = \Omega(n^2)$ αφού $3n^2 - 100n + 6 > 2.99n^2$ για $n > 10^4$.
- $3n^2 - 100n + 6 = \Omega(n)$ αφού $3n^2 - 100n + 6 > (10^{10^{10}})n$ για $n > 10^{10^{10}}$.
- $3n^2 - 100n + 6 \neq \Omega(n^3)$ αφού $\nexists c : 3n^2 - 100n + 6 > cn^3$ για $n > n_0$.
- $3n^2 - 100n + 6 = \Theta(n^2)$ αφού $3n^2 - 100n + 6 = \Omega(n^2)$ και $3n^2 - 100n + 6 = O(n^2)$

Ορίζουμε και ασυμπτωτικά σύμβολα που ορίζουν ότι τα φράγματα είναι αυστηρά:

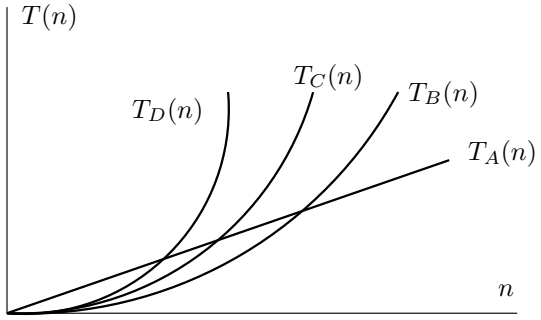
- $f(n) = o(g(n))$ αν για κάθε $c > 0$, υπάρχει n_0 έτσι ώστε $f(n) < cg(n)$ για κάθε $n > n_0$
- $f(n) = \omega(g(n))$ αν για κάθε $c > 0$, υπάρχει n_0 έτσι ώστε $f(n) > cg(n)$ για κάθε $n > n_0$

Η διαφορά του συμβολισμού o από το συμβολισμό O είναι λεπτή. Στον συμβολισμό o υπάρχει η έννοια της αυστηρότητας, δηλαδή αν $f(n) = o(g(n))$, τότε στον ορισμό ισχύει αυστηρή ανισότητα. Αντίθετα όταν $f(n) = O(g(n))$ ισχύει ανισοισότητα. Επίσης, στο συμβολισμό O υπάρχουν κάποιες σταθερές c για την οποία να ισχύει η ανισότητα. Ενώ στον o για κάθε σταθερά c ισχύει η ανισότητα. Για παράδειγμα $n^2 + n - 1 = O(n^2) = o(n^3) \neq o(n^2)$. Στην πραγματικότητα δηλαδή, όταν $f(n) = o(g(n))$ έχουμε ότι η $f(n)$ είναι αμελητέα σε σχέση με την $g(n)$.

1.2.1 Κλάσεις Πολυπλοκότητας

Η κατάταξη των συναρτήσεων πολυπλοκότητας σε γενικότερα σύνολα συναρτήσεων μας παρέχει μία ομαδοποίηση υψηλού επιπέδου, μέσω της οποίας μπορούμε να κρίνουμε πότε ένας αλγόριθμος είναι καλύτερος από έναν άλλον, αν είναι αποδοτικός κ.λ.π.

Ας δούμε ένα παράδειγμα. Έστω ένα πρόβλημα Π και τέσσερις αλγόριθμοι A, B, C, D που το λύνουν με αντίστοιχες συναρτήσεις πολυπλοκότητας που ασυμπτωτικά φράσσονται ως εξής: $T_A(n) = O(n), T_B(n) = O(n^2), T_C(n) = O(n^3)$ και $T_D(n) = 2^n$. Καθώς η διάσταση του προβλήματος αυξάνει, οι απαιτούμενες πράξεις αυξάνουν, όπως φαίνεται στην γραφική παράσταση του σχήματος 1.4.



Σχήμα 1.4: Αύξηση πολυπλοκότητας με την αύξηση του μεγέθους του προβλήματος για 4 διαφορετικούς αλγορίθμους

Καθώς το n αυξάνει η απόκλιση των απαιτούμενων πράξεων γίνεται περισσότερο φανερή. Ο ρυθμός με τον οποίο αυξάνουν οι πράξεις ακολουθεί την κατάταξη: $T_A(n) < T_B(n) < T_C(n) < T_D(n)$ ιδιαίτερα δε ο αλγόριθμος D παρουσιάζει πολύ μεγαλύτερο ρυθμό αύξησης από τους άλλους. Η μεγάλη αυτή πολυπλοκότητα, καθιστά τον αλγόριθμο D τόσο αδύναμο, ώστε να μπορεί να χρησιμοποιηθεί για την επίλυση μόνο μικρών στιγμιοτύπων του προβλήματος.

Για να δούμε όμως πόσο πραγματικά μικρά είναι τα στιγμιότυπα που μπορεί αυτός να λύσει. Έστω μια μηχανή που εκτελεί 10^7 πράξεις το δευτερόλεπτο (10 MIPS) και έστω ότι αφήνουμε τους αλγορίθμους A, B, C, D να τρέξουν για 0.1 δευτερόλεπτα, 10 λεπτά και 1 ημέρα. Η διάσταση που θα λύσουν φαίνεται στον πίνακα 1.1, όπου γίνεται εμφανής η αδυναμία του αλγορίθμου D να λύσει προβλήματα μεγάλης διάστασης. Αλγόριθμοι σαν τον D είναι συχνά οι καλύτεροι δυνατοί για έναν πρόβλημα, ιδίως στα αποκαλούμενα NP-δύσκολα προβλήματα (που θα δούμε αργότερα), στα οποία δεν μπορούμε μέχρι σήμερα να βρούμε καλούς(πολυωνυμικούς) αλγορίθμους που τα επιλύουν βέλτιστα.

Αλγόριθμος	0.1 sec	10 min	1 day
$T_A(n) = O(n)$	10^6	6×10^9	864×10^9
$T_B(n) = O(n^2)$	10^3	7.7×10^4	9.3×10^4
$T_C(n) = O(n^3)$	10^2	1800	9254
$T_D(n) = O(2^n)$	20	32	39

Πίνακας 1.1: Διάσταση προβλημάτων που εκτελούνται από τους 4 αλγόριθμους σε σταθερό χρόνο από μια μηχανή 10 MIPS

Για παράδειγμα έστω το εξής $NP-$δύσκολο πρόβλημα: Μία επιχείρηση έχει επιλέξει n τοποθεσίες για να χτίσει n εργοστάσια. Κάθε εργοστάσιο θα έχει έναν κύκλο παραγωγής υλικών και θα πρέπει να λαμβάνει και να στέλνει υλικά στα άλλα εργοστάσια πληρώνοντας το ανάλογο κόστος μεταφοράς. Αν σε κάθε τοποθεσία μπορεί να χτιστεί ακριβώς ένα εργοστάσιο, σε ποια τοποθεσία πρέπει να χτιστεί κάθε εργοστάσιο, ώστε να ελαχιστοποιείται το κόστος της μεταφοράς των υλικών;

Αν δημιουργήσουμε όλους τους δυνατούς συνδυασμούς τοποθεσιών - εργοστασίων (πολυπλοκότητας $n!$) και μετρήσουμε το συνολικό κόστος για κάθε συνδυασμό, ώστε να επιλέξουμε αυτόν που έχει ελάχιστο κόστος, τότε ακόμη και για μικρά στιγμιότυπα του προβλήματος, για παράδειγμα με διάσταση 80 και με μία μηχανή που εξετάζει 10^7 συνδυασμούς το δευτερόλεπτο, θα χρειαστούν περίπου 10^{112} χρόνια, ένας αριθμός πολύ μεγαλύτερος από τα μόρια του γαλαξία μας!

Σε άλλα πάλι προβλήματα (που δεν είναι $NP-$δύσκολα), η εύρεση ενός αποτελεσματικού αλγορίθμου μπορεί να βελτιώσει θεαματικά τον χρόνο επίλυσης. Για παράδειγμα έστω το εξής πρόβλημα ανάθεσης εργασιών:

Δίνεται ένα σύνολο υπαλλήλων και ένα σύνολο εργασιών οι οποίες πρέπει να ανατεθούν στους υπαλλήλους. Κάθε εργασία i στοιχίζει c_{ij} όταν ανατίθεται στον υπάλληλο j. Μία εργασία θα πρέπει να ανατεθεί σε έναν μόνο υπάλληλο και κάθε υπάλληλος θα εκτελέσει ακριβώς μία εργασία, ενώ έχουμε ίδιο πλήθος εργασιών και υπαλλήλων. Να βρεθεί η ανάθεση που ελαχιστοποιεί το συνολικό κόστος ανάθεσης.

Και εδώ, η δημιουργία όλων των δυνατών συνδυασμών εργασιών - υπαλλήλων (εκθετικός αλγόριθμος) ακόμη και για μικρά στιγμιότυπα του προβλήματος, για παράδειγμα με 20 υπαλλήλους και 20 εργασίες, θα χρειαστεί χιλιετίες για να ολοκληρωθεί. Εντούτοις, ο Hungarian αλγόριθμος, ο οποίος είναι πολυωνυμικός αλγόριθμος, θα χρειαστεί περίπου μόνο 1.95 sec σε μία μηχανή που ελέγχει 10^6 συνδυασμούς το δευτερόλεπτο.

Γίνεται λοιπόν εμφανές, ότι η ανάπτυξη αποδοτικών αλγορίθμων είναι μία αναγκαιότητα παρά την ραγδαία ανάπτυξη της ταχύτητας των υπολογιστών. Σύμφωνα με τον νόμο του Moore (που ακολουθείται με θαυμαστή συνέπεια τις τελευταίες δεκαετίες) η ταχύτητα των επεξεργαστών διπλασιάζεται κάθε 18 μήνες. Αυτό σημαίνει ότι σε 15 χρόνια από σήμερα, οι επεξεργαστές θα έχουν γίνει 1000 φορές πιο γρήγοροι. Αν σήμερα σε σταθερό χρόνο λύνουμε μέχρι διάσταση n με αλγορίθμους δεδομένης πολυπλοκότητας, τότε μετά από 15 χρόνια θα λύνουμε στον ίδιο χρόνο τις διαστάσεις που φαίνονται στον πίνακα 1.2.

Πολυπλοκότητα	Σημερινή Διάσταση	Μελλοντική Διάσταση
$O(n)$	n	$1000 \times n$
$O(n^2)$	n	$32 \times n$
$O(n^3)$	n	$10 \times n$
$O(n^4)$	n	$6 \times n$
$O(2^n)$	n	$n + 10$
$O(3^n)$	n	$n + 8$
$O(n!)$	n	$n + 7$

Πίνακας 1.2: Αύξηση της διάστασης του προβλήματος που επιλύεται σε σταθερό χρόνο από έναν αλγόριθμο δεδομένης πολυπλοκότητας σε μια μηχανή 1000 φορές ταχύτερη.

1	Σταθερή πολυπλοκότητα
$\log(n)$	Λογαριθμική πολυπλοκότητα
$\log^k(n)$	Πολυλογαριθμική πολυπλοκότητα όταν k μία σταθερά
n	Γραμμική πολυπλοκότητα (πχ, $f(n) = an + b$, με a, b σταθερές)
$nlogn$	
n^2	Τετραγωνική πολυπλοκότητα
n^3	Κυβική πολυπλοκότητα
n^k	Πολυωνυμικη πολυπλοκότητα, με k μία σταθερά, δηλ. $f(x) = a_k n^k + \ldots + a_0$
a^n	Εκθετική πολυπλοκότητα $1 < a < 2$
2^n	Εκθετική πολυπλοκότητα
a^n	Εκθετική πολυπλοκότητα $a > 2$
$n!$	Παραγοντική πολυπλοκότητα
n^n	Υπερεκθετική πολυπλοκότητα

Πίνακας 1.3: Χαρακτηριστικές κλάσεις πολυπλοκότητας

Άρα ο ισχυρισμός ότι η ανάγκη ανάπτυξης αποδοτικών αλγορίθμων θα εκλείψει μπροστά στις αυξητικές αποδόσεις των αυριανών υπολογιστών, αποτελεί μία πλήρως εσφαλμένη θέση.

Κατάταξη κλάσεων πολυπλοκότητας

Με βάση την παραπάνω συζήτηση, είναι δυνατή η κατηγοριοποίηση των συναρτήσεων πολυπλοκότητας σε κλάσεις, οι οποίες μας παρέχουν ένα μέτρο αποδοτικότητας αλγορίθμων (πίνακας 1.3). Η διαχωριστική γραμμή χωρίζει τους αποδοτικούς αλγόριθμους (πολυωνυμικούς) από τους μη αποδοτικούς (εκθετικούς). Οι εκθετικοί αλγόριθμοι χρειάζονται υπερβολικό χρόνο και δεν μπορούμε να τους εμπιστευθούμε ούτε για μικρές τάξεις δεδομένων για πολλά προβλήματα.

Γενικά, οι κλάσεις πολυπλοκότητας έχουν σαν μέλη τους συναρτήσεις. Ω-

στόσο ανάλογα με ποιον ασυμπτωτικό συμβολισμό χρησιμοποιούμε, υπάρχει και μια συνολοθεωρητική σχέση.

Η κλάση πολυπλοκότητας $O(n)$ είναι υποσύνολο της $O(n^2)$ που είναι υποσύνολο της $O(n^3)$ κ.ο.κ. Η αξίωση αυτή είναι λογική: ένας αλγόριθμος που φράσσεται ασυμπτωτικά από μία γραμμική συνάρτηση, φράσσεται και από μία τετραγωνική, η οποία φράσσεται από μία κυβική κ.ο.κ.

Αντίστροφα, η κλάση πολυπλοκότητας $\Omega(n)$ είναι υπερσύνολο της $\Omega(n^2)$, δηλαδή αν μια συνάρτηση φράσσεται ασυμπτωτικά από κάτω από μία τετραγωνική συνάρτηση, τότε φρασσεται και από μία γραμμική.

Αυτή η συνολοθεωρητική σχέση υποσυνόλου, δεν ισχύει για το συμβολισμό Θ. Εδώ, κάθε κλάση πολυπλοκότητας είναι ξένη με οποιανδήποτε άλλη. Δηλαδή, αν μια συνάρτηση είναι $\Theta(n)$ τότε δεν ανήκει σε καμία άλλη κλάση με συμβολισμό Θ.

Η έννοια του βέλτιστου αλγορίθμου

Ας θεωρήσουμε ένα οποιοδήποτε πρόβλημα και έστω ότι υπάρχουν γι΄ αυτό τρεις αλγόριθμοι που το λύνουν. Τότε ο καλύτερος από αυτούς, θα είναι εκείνος που έχει την καλύτερη πολυπλοκότητα στην χείριστη περίπτωση.

Πως όμως θα ξέρουμε αν υπάρχει καλύτερος αλγόριθμος από αυτούς, δηλαδή ένας αλγόριθμος που έχει καλύτερη πολυπλοκότητα και δεν έχει ακόμη ανακαλυφθεί, αν δηλαδή αξίζει τον κόπο να καταβάλλουμε προσπάθεια για την εύρεση ενός ακόμη καλύτερου αλγόριθμου;

Αν για ένα πρόβλημα, οποιοσδήποτε αλγόριθμος που το επιλύει έχει ασυμπτωτική πολυπλοκότητα $\Omega(f(n))$ και έχουμε ήδη έναν αλγόριθμο με ασυμπτωτική πολυπλοκότητα $O(f(n))$ τότε ο αλγόριθμος αυτός είναι βέλτιστος(ασυμπτωτικά).

Αυτό σημαίνει ότι αν για κάποιο πρόβλημα έχει αποδειχθεί ότι χρειάζεται το λιγότερο $f(n)$ πραξεις και έχουμε κατασκευάσει αλγόριθμο που κάνει το πολύ αυτές τις πράξεις, τότε ο αλγόριθμος είναι βέλτιστος.

Ένα παράδειγμα προβλήματος στο οποίο έχει βρεθεί ο βέλτιστος, ως προς την πολυπλοκότητα, αλγόριθμος είναι το πρόβλημα της ταξινόμησης με συγκρίσεις. Αποδεικνύεται εύκολα, ότι οποιοσδήποτε αλγόριθμος που βασίζεται σε συγκρίσεις πρέπει να κάνει το λιγότερο $n \log n$ συγκρίσεις. Ο αλγόριθμος της ταξινόμησης με σωρό (heapsort) και της γρήγορης ταξινόμησης (quicksort) επιτυγχάνουν πολυπλοκότητα $O(n \log n)$ κατά μέσο όρο, επομένως είναι βέλτιστοι ως προς την πολυπλοκότητα κατά μέσο όρο. Ο αλγόριθμος ταξινόμησης με σωρό έχει πολυπλοκότητα $O(n \log n)$ και στην χείριστη περίπτωση, άρα είναι βέλτιστος ασυμπτωτικά και στην χείριστη περίπτωση. Αντίθετα ο αλγόριθμος της γρήγορης ταξινόμησης, στην χείριστη περίπτωση (π.χ. ήδη ταξινομημένη ακολουθία) έχει πολυπλοκότητα $O(n^2)$ άρα δεν είναι βέλτιστος αλγόριθμος.

Ωστόσο η απόδειξη ύπαρξης μη τετριμμένων κάτω φραγμάτων για τις πράξεις ενός αλγορίθμου, είναι μια όχι εύκολη διαδικασία. Θα δούμε τώρα ένα δεύτερο παράδειγμα μη βέλτιστου αλγορίθμου.

Θεωρούμε το πρόβλημα πολλαπλασιασμού τετραγωνικών πινάκων, στο οποίο μας δίνονται δύο πίνακες $A_{n\times n}$ και $B_{n\times n}$ και μας ζητείται να υπολογίσουμε το

γινόμενο τους $C_{n\times n}$:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, 1 \leq i, j \leq n$$

Ένας προφανής αλγόριθμος που υπολογίζει το γινόμενο είναι αυτός του σχήματος 1.5.

ΠΟΛΛΑΠΛΑΣΙΑΣΜΟΣ ΠΙΝΑΚΩΝ (A, B)

```
1.  for i = 1 to n do
2.      for j = 1 to n do
3.          c_ij = 0
4.          for k = 1 to n do
5.              c_ij = c_ij + a_ik * b_kj
6.          end for
7.      end for
8.  end for
9.  return C
```

Σχήμα 1.5: Πολλαπλασιασμός δύο τετραγωνικών πινάκων διάστασης n.

Ο αλγόριθμος αυτός κάνει n^3 πολλαπλασιασμούς (που είναι η στοιχειώδης πράξη) άρα έχει πολυπλοκότητα $O(n^3)$. Ωστόσο επιστήμονες παρατήρησαν ότι μπορούσαν να ελαττώσουν το πλήθος των πολλαπλασιασμών με κάποιες έξυπνες πράξεις και έτσι το 1969 ο Strassen παρουσίασε αλγόριθμο που έκανε $O(n^{2.81})$ πολλαπλασιασμούς και ο καλύτερος αλγόριθμος μέχρι σήμερα είναι των Coopersmith & Winograd του 1986 με $O(n^{2.379})$ πολλαπλασιασμούς. Και οι 3 παραπάνω αλγορίθμοι για τον πολλαπλασιασμό πινάκων δεν είναι βέλτιστοι.

Είναι λογικό ότι οποιοσδήποτε αλγόριθμος που λύνει το πρόβλημα αυτό, πρέπει να κάνει $\Omega(n^2)$ βήματα, αφού τόσος χρόνος απαιτείται μόνο για το σάρωμα των δύο πινάκων εισόδου. Άρα ο αλγόριθμος των Coopersmith & Winograd δεν είναι βέλτιστος ως προς την πολυπλοκότητα.

1.2.2 Ιδιότητες ασυμπτωτικών συμβολισμών

Θεωρώντας ότι οι συναρτήσεις πολυπλοκότητας $f(n), g(n), h(n)$ είναι μονοτονικές μη φθίνουσες θετικές ακολουθίες, ισχύουν οι παρακάτω ιδιότητες:

Μεταβατικότητα

- Αν $f(n) = \Theta(g(n))$ και $g(n) = \Theta(h(n))$ τότε $f(n) = \Theta(h(n))$,
- Αν $f(n) = O(g(n))$ και $g(n) = O(h(n))$ τότε $f(n) = O(h(n))$,
- Αν $f(n) = \Omega(g(n))$ και $g(n) = \Omega(h(n))$ τότε $f(n) = \Omega(h(n))$,
- Αν $f(n) = o(g(n))$ και $g(n) = o(h(n))$ τότε $f(n) = o(h(n))$,

- Αν $f(n) = \omega(g(n))$ και $g(n) = \omega(h(n))$ τότε $f(n) = \omega(h(n))$

Ανακλαστικότητα

- $f(n) = \Theta(f(n))$,
- $f(n) = O(f(n))$,
- $f(n) = \Omega(f(n))$

Συμμετρία

- $f(n) = \Theta(g(n))$ αν και μόνο αν $g(n) = \Theta(f(n))$

Ανάστροφη Συμμετρία

- $f(n) = O(g(n))$ αν και μόνο αν $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ αν και μόνο αν $g(n) = \omega(f(n))$

Πράξεις

- $cO(f(n)) = O(f(n))$
- $O(g(n)) + O(g(n)) = O(g(n))$
- $O(g_1(n) + O(g_2(n)) = O(\max\{g_1(n), g_2(n)\})$
- $O(g_1(n)) * O(g_2(n)) = O(g_1(n) * g_2(n))$

Η εξαγωγή της ασυμπτωτικής συμπεριφοράς μίας συνάρτησης μπορεί να ορισθεί ισοδύναμα χρησιμοποιώντας την έννοια του όριου συναρτήσεων και αυτό γιατί στην ουσία μας ενδιαφέρει η σύγκριση της συμπεριφοράς μιας συνάρτησης σε σχέση με ένα φράγμα, καθώς η διάσταση του προβλήματος τείνει στο άπειρο. Έτσι, έχουμε τους ακόλουθους ισοδύναμους ορισμούς για τους ασυμπτωτικούς συμβολισμούς:

1. Αν
$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = a \neq 0$$
τότε $f(n) = \Theta(g(n))$
2. Αν
$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$$
τότε $f(n) = O(g(n))$ και $g(n) = \Omega(f(n))$ (με $f(n) \neq \Theta(g(n))$)
3. Αν
$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \infty$$
τότε $g(n) = O(f(n))$ και $f(n) = \Omega(g(n))$ (με $g(n) \neq \Theta(f(n))$)

Παράδειγμα: Να δειχθεί ότι η συνάρτηση πολυπλοκότητας

$$f(n) = \frac{(n^2 + logn)(n-1)}{n+n^2}$$

είναι $\Theta(n)$.
Έχουμε:

$$\lim_{n\to+\infty} \frac{\frac{(n^2+logn)(n-1)}{n+n^2}}{n} = \lim_{n\to+\infty} \frac{n^3 - n^2 + n\log n - \log n}{n^3 + n^2} = 1$$

Άρα $f(n) = \Theta(n)$. □

1.2.3 Ασκήσεις

Στα παρακάτω για τη συνάρτηση $\log n$ θα θεωρούμε βάση το 2, δηλαδή $\log n = \log_2 n$.

Άσκηση 1. Έστω $T_1(n) = O(f(n))$ και $T_2(n) = O(g(n))$ οι πολυπλοκότητες δύο τμημάτων P_1 και P_2 ενός προγράμματος P. Αν το P_2 εκτελείται αμέσως μετά το P_1 ποια είναι η πολυπλοκότητα του προγράμματος P;

Λύση: Επειδή γίνεται ακολουθία από βήματα, η πολυπλοκότητα του προγράμματος είναι η μέγιστη από τις πολυπλοκότητες των δύο κομματιών, δηλαδή:

$$T(n) = \max\{T_1(n), T_2(n)\} = \max\{O(f(n)), O(g(n))\}$$

Εφαρμογή: Αν $T_1(n) = O(n^2)$, $T_2(n) = O(n^3)$, $T_3(n) = O(n\log n)$ τότε

$$T(n) = \max\{O(n^2), O(n^3), O(n\log n)\} = O(n^3)$$

□

Άσκηση 2. Να δοθεί ο καλύτερος O συμβολισμός για τις ακόλουθες συναρτήσεις:

1. $2\log n - 4n + 3n\log n$
2. $2 + 4 + \ldots + 2n$
3. $2 + 4 + 8 + \ldots + 2^n$

Με τον όρο 'καλύτερος συμβολισμός' εννοείται το πιο μικρό άνω φράγμα για κάθε συνάρτηση. Άρα έχουμε:

1. $2\log n - 4n + 3n\log n = O(n\log n)$
2. $2 + 4 + \ldots + 2n = 2(1 + 2 + \ldots + n) = 2\sum_{i=1}^{n} i = n(n+1) = O(n^2)$
3. $2 + 4 + 8 + \ldots + 2^n = 2\frac{2^n-1}{2-1} = 2 * 2^n - 2 = O(2^n)$

□

Άσκηση 3. Έστω $a > 1$ και $f(n) = O(\log_a(n))$. Δείξτε ότι $f(n) = O(\log n)$.
Αφού $f(n) = O(\log_a(n))$ με βάση τον ορισμό, υπάρχει c που για κάθε $n \geq n_0$: $f(n) \leq c\log_a(n)$. Όμως επειδη $\log_a(n) = \frac{\log n}{\log a}$ έχουμε $f(n) \leq \frac{c}{\log a}\log n$ ή $f(n) \leq c' \log n$ με $c' = \frac{c}{\log a}$ για κάθε $n \geq n_0$. Άρα $f(n) = O(\log n)$. □

Άσκηση 4. Δείξτε ότι $\log(n!) = O(n \log n)$.
Επειδή $n! = n(n-1)\ldots 2 < n \ldots n = n^n$ και επειδή η λογαριθμική συνάρτηση είναι γνησίως αύξουσα, έχουμε $\log(n!) < n \log n$, άρα $\log(n!) = O(n \log n)$. □

Άσκηση 5. Ποια είναι η πολυπλοκότητα του παρακάτω τμήματος προγράμματος:

```
1.  for i = 1 to n do
2.      for j = 1 to (i+1)/2 do
3.          x = x + 1
4.      end for
5.  end for
```

Όπως είδαμε στα παραδείγματα πολλαπλασιασμού πινάκων και αναζήτησης σε πίνακα, κάθε βρόγχος εκφράζει έναν όρο του γινομένου που εμφανίζεται στην πολυπλοκότητα. Ετσι συμβαίνει και εδώ για τον εξωτερικό βρόγχο (n επαναλήψεων), αλλά πρέπει να είμαστε προσεκτικοί με τον εσωτερικό βρόγχο στον οποίο ο αριθμός επαναλήψεων είναι μεταβλητός. Άρα:

$$T(n) = \sum_{i=1}^{n} \frac{i+1}{2} = \frac{1}{2}\Big(\sum_{i=1}^{n} i + \sum_{i=1}^{n} 1\Big) = \frac{1}{2}\Big(\frac{n(n+1)}{2} + n\Big) = O(n^2)$$

□

Άσκηση 6. Να δειχθεί ότι η $3^n \neq O(2^n)$
Έχουμε

$$\lim_{n\to+\infty} \frac{2^n}{3^n} = 0$$

Άρα $3^n = \Omega(2^n)$ □

Άσκηση 7. Να υπολογιστεί η πολυπλοκότητα του αλγόριθμου του Ευκλείδη (σχήμα 1.6) για τον υπολογισμό του Μέγιστου Κοινού Διαιρέτη (Μ.Κ.Δ.) δύο θετικών ακέραιων αριθμών.

Ας θεωρήσουμε χωρίς βλάβη της γενικότητας ότι $m \geq n$. Σε κάθε βήμα ο αλγόριθμος του Ευκλείδη εφαρμόζει την ταυτότητα της διαίρεσης ώστε να βρεθούν αριθμοί q, r ώστε $m = qn + r$, $q \geq 1$, $r \geq 0$. Επειδή ΜΚΔ(m,n)=ΜΚΔ(n,r),

ΜΚΔ - ΕΥΚΛΕΙΔΗΣ $(m, n \in \mathbb{Z})$
1. repeat
2. $\quad r = m \mod n$
3. $\quad$ if $r \neq 0$ then
4. $\quad\quad m = n$
5. $\quad\quad n = r$
6. $\quad$ end if
7. until $r = 0$
8. return n

Σχήμα 1.6: Αλγόριθμος του Ευκλείδη για την εύρεση του Μ.Κ.Δ. δύο ακεραίων m και n.

έχουμε την εξής ακολουθία πράξεων μέχρι να τερματίσει ο αλγόριθμος:

$$\begin{aligned} m &= q_0 n + r_0 \quad , \quad 0 \leq r_0 < n, \quad q_0 = q, \quad r_0 = r \\ n &= q_1 r_0 + r_1 \quad , \quad 0 \leq r_1 < r_0 \\ r_0 &= q_2 r_1 + r_2 \quad , \quad 0 \leq r_2 < r_1 \\ r_1 &= q_3 r_2 + r_3 \quad , \quad 0 \leq r_3 < r_2 \\ &\dots \qquad\quad \dots \end{aligned}$$

Έστω k ο αριθμός των βημάτων που θέλει ο αλγόριθμος για να τερματίσει. Τότε δημιουργείται μία ακολουθία από αριθμούς $m, n, r_0, r_1, \dots r_k$ με $r_k = 0$ και $r_{k-1} = \text{ΜΚΔ}(m, n)$. Θεωρώντας ότι σε κάθε βήμα του αλγορίθμου γίνονται $O(1)$ πράξεις, η πολυπλοκότητα του αλγορίθμου είναι $O(k)$.

Αποδεικνύται (άσκηση) ότι $r_i < \frac{1}{2} r_{i-2}$, οπότε έχουμε ότι

$$r_k < \frac{1}{2} < \frac{1}{2} r_{k-2} < \frac{1}{4} r_{k-4} < \ldots < \frac{1}{2^k} r_0 < \frac{1}{2^k} n$$

Άρα $n > 2^{k-1}$, δηλαδή $k \leq \log n$, οπότε η πολυπλοκότητα του αλγορίθμου είναι $O(\log n)$. □

Άσκηση 8. Δείξτε ότι για πραγματικούς αριθμούς a, b, $b > 0$ έχουμε $(n + a)^b = \Theta(n^b)$.
Από τον τύπο που δίνει το διώνυμο του Newton παίρνουμε:

$$(n + a)^b = n^b + \binom{b}{1} n^{b-1} a + \ldots + \binom{b}{k} n^{b-k} a^k + \ldots + a^b = \Theta(n^b)$$

□

Άσκηση 9.

- Δείξτε ότι $2^{n+1} = O(2^n)$.
  Έχουμε $2^{n+1} = 2 * 2^n = O(2^n)$

- Είναι $2^{2n} = O(2^n)$;
  Επειδή $2^{2n} = (2^2)^n = 4^n$, το 2^{2n} δεν είναι $O(2^n)$. Ισοδύναμα:

$$\lim_{n \to +\infty} \frac{2^{2n}}{2^n} = +\infty$$

  και όχι μηδέν.

- Αν $f(n) = O(n)$ τότε $2^{f(n)}$ δεν είναι $O(2^n)$.
  Έστω $f(n) = 2n$. Τότε $f(n) = O(n)$, αλλα $2^{2n} \neq O(2^n)$.

Άσκηση 10. Να μελετηθούν και να συγκριθούν οι αλγόριθμοι ταξινόμησης Φυσαλίδας, ταξινόμησης με Επιλογή και ταξινόμησης με Εισαγωγή, όσον αφορά την πολυπλοκότητά τους.

Ταξινόμηση Φυσαλίδας (Bubble Sort)

Ο αλγόριθμος αυτός, σε κάθε επανάληψη ανυψώνει στο τέλος του πίνακα το ελαφρύτερο (μικρότερο) στοιχείο. Η υλοποίηση του σε ψευδογλώσσα φαίνεται στο σχήμα 1.7.

```
Bubble Sort (A)
1.  for i = n to 1 do
2.      for j = 1 to i − 1 do
3.          if a_j < a_{j+1} then
4.              swap(a_j, a_{j+1})
5.          end if
6.      end for
7.  end for
```

Σχήμα 1.7: Ταξινόμηση μιας ακολουθίας στοιχείων κατά φθίνουσα σειρά με τον αλγόριθμο της φυσαλίδας.

Παρατηρούμε ότι ανεξάρτητα από την μορφή της ακολουθίας εισόδου, ο αλγόριθμος θα κάνει τις ίδιες συγκρίσεις και καταχωρήσεις. Θεωρώντας σαν σταθερές ($O(1)$) τις πράξεις που γίνονται σε κάθε βήμα του εσωτερικού βρόγχου, έχουμε:

$$\begin{aligned} C_{\beta\pi}(n) = C_{\chi\pi}(n) = C_{\mu o}(n) &= \sum_{i=n}^{1} \sum_{j=1}^{i-1} O(1) = O(1) \sum_{i=1}^{n} (i-1) \\ &= O(1)[\frac{n(n+1)}{2} - n] = O(n^2) \end{aligned}$$

Ταξινόμηση με εισαγωγή (Insertion Sort)

Ο αλγόριθμος ταξινόμησης με εισαγωγή θεωρεί ότι στο βήμα i, η υπακολουθία $a_1, \ldots, a_{i-1}$ είναι ταξινομημένη και η $a_{i+1}, \ldots, a_n$ αταξινόμητη. Οπότε εισάγει το στοιχείο i στην σωστή θέση της πρώτης υπακολουθίας και συνεχίζει επαναληπτικά για τα υπόλοιπα στοιχεία.

INSERTION SORT (A)
1. for $i = 1$ to n do
2. $\quad val = a_i$
3. $\quad j = i - 1$
4. $\quad$ while $j \geq 1$ and $a_j < val$ do
5. $\quad\quad a_{j+1} = a_j$
6. $\quad\quad j = j - 1$
7. $\quad$ end while
8. $\quad a_{j+1} = val$
9. end for

Σχήμα 1.8: Ταξινόμηση μιας ακολουθίας στοιχείων κατά φθίνουσα σειρά με τον αλγόριθμο της εισαγωγής.

Στην βέλτιστη περίπτωση (ήδη ταξινομημένη φθίνουσα ακολουθία) ο αλγόριθμος δεν μπαίνει ποτέ στο εσωτερικό (while) βρόγχο, άρα η πολυπλοκότητα του είναι: $C_{\beta\pi}(n) = O(n)$. Στη χείριστη περίπτωση (αύξουσα ακολουθία), μπαίνει πάντα στο εσωτερικό βρόγχο (όσο $j \geq 1$). Άρα η πολυπλοκότητα είναι:

$$C_{\chi\pi}(n) = \sum_{i=1}^{n} \sum_{j=i-1}^{1} O(1) = O(1) \sum_{i=1}^{n} (i-1) = O(n^2)$$

Αποδεικνύεται ωστόσο, ότι η μέση πολυπλοκότητά του είναι: $C_{\mu o} = O(n^2)$.

Ταξινόμηση με Επιλογή (Selection Sort)

Η ταξινόμηση με επιλογή, βρίσκει στο βήμα i το στοιχείο που είναι το μεγαλύτερο στην υπακολουθία $a_i, \ldots, a_n$. Στην συνέχεια θέτει στην θέση i το μέγιστο αυτό στοιχείο και συνεχίζει επαναληπτικά.

Η πολυπλοκότητα όπως και στην Ταξινόμηση Φυσαλίδας δεν εξαρτάται από την ακολουθία που έχουμε σαν είσοδο. Άρα η πολυπλοκότητα και για τις τρεις περιπτώσεις είναι:

$$\begin{aligned} C_{\beta\pi}(n) = C_{\chi\pi}(n) = C_{\mu o}(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} O(1) = O(1) \sum_{i=1}^{n-1} (n-i) = \\ &= O(1)\big[n(n-1) - \frac{(n-1)(n-2)}{2}\big] = O(n^2) \end{aligned}$$

```
Selection Sort (A)
1.  for i = 1 to n − 1 do
2.      max = i
3.      for j = i + 1 to n do
4.          if a_j > a_max then
5.              max = j
6.          end if
7.      end for
8.      swap(a_max, a_i)
9.  end for
```

Σχήμα 1.9: Ταξινόμηση μιας ακολουθίας στοιχείων σε φθίνουσα σειρά με τον αλγόριθμο της επιλογής.

Παρατηρούμε ότι ο καλύτερος αλγόριθμος είναι ο αλγόριθμος της Εισαγωγής γιατί στην βέλτιστη περίπτωση είναι $O(n)$ σε αντίθεση με τους άλλους δύο που είναι πάντα $O(n^2)$. Ωστόσο και οι τρεις αλγόριθμοι παρουσιάζουν τετραγωνική πολυπλοκότητα κατά μέσο όρο, που κρίνεται μη ικανοποιητική για το πρόβλημα αυτό. Στις επόμενες ενότητες θα δούμε αλγόριθμους ταξινόμησης (HeapSort, MergeSort, QuickSort) που έχουν μέση πολυπλοκότητα $O(n \log n)$ και χείριστη πολυπλοκότητα $O(n \log n)$ (εκτός της Quicksort) και είναι καλύτεροι από αυτούς που μελετήσαμε.

Κεφάλαιο 2

Αναδρομή

2.1 Divide and Conquer

Πολλοί χρήσιμοι αλγόριθμοι οι οποίοι θα εξεταστούν στη συνέχεια έχουν αναδρομική δομή. Η *αναδρομή* είναι μια ισχυρή αλγοριθμική μέθοδος, πιο αισθητική (όσο αφορά στο μέγεθος των αλγορίθμων και κατά συνέπεια των προγραμμάτων που γράφουμε), πιο φυσική (αφού συμβαδίζει με την διαίσθησή μας) και πιο ισχυρή (μιας και μας δίδει την δυνατότητα να περιγράψουμε με εύκολο και εύληπτο τρόπο δύσκολα προβλήματα).

Για την επίλυση ενός δεδομένου προβλήματος ένας αναδρομικός αλγόριθμος καλεί τον εαυτό του αναδρομικά μία ή περισσότερες φορές επιλύοντας διάφορα υποπροβλήματα του αρχικού προβλήματος. Με τον όρο υποπρόβλημα εννοούμε ένα πρόβλημα της ίδιας φύσης με το αρχικό, αλλά μικρότερου μεγέθους. Ένας αλγόριθμος τέτοιου είδους τυπικά ακολουθεί μια **διαίρει και βασίλευε (divide and conquer)** προσέγγιση, κατά την οποία το αρχικό πρόβλημα διασπάται σε μια σειρά από υποπροβλήματα τα οποία επιλύονται ένα προς ένα αναδρομικά και έπειτα συνδυάζονται οι λύσεις αυτών για να δημιουργηθεί η λύση του αρχικού προβλήματος. Σε κάθε αναδρομικό επίπεδο, η τεχνική **divide and conquer** περιλαμβάνει τρία βήματα:

Divide: Διαίρεση του προβλήματος σε μικρότερα προβλήματα (υποπροβλήματα).

Conquer: Επίλυση των μικρότερων προβλημάτων (είτε αναδρομικά, είτε με άμεσο τρόπο εάν έχουν ικανοποιητικά μικρό μέγεθος).

Combine: Συνδυασμός των λύσεων για την εύρεση της αρχικής λύσης.

2.1.1 Merge-Sort

Ένα κλασσικό παράδειγμα **divide and conquer** αλγορίθμου, είναι ο αλγόριθμος ταξινόμησης Merge-Sort. Διαισθητικά, ο αλγόριθμος εξελίσσεται ως εξής:

Divide: Διαίρεση του πίνακα n στοιχείων σε δύο υποπίνακες $\frac{n}{2}$ στοιχείων ο καθένας.

Conquer: Αναδρομική ταξινόμηση των δύο υποπινάκων με χρήση της MERGE-SORT.

Combine: Συγχώνευση των δύο ταξινομημένων υποπινάκων.

Η αναδρομή στο βήμα **Conquer** σταματά όταν οι υποπίνακες οι οποίοι προκύπτουν μετά τη διάσπαση έχουν μόνο ένα στοιχείο, διότι σε αυτή την περίπτωση ο πίνακας είναι ήδη ταξινομημένος. Η βασική λειτουργία του αλγορίθμου MERGE-SORT είναι η συγχώνευση δύο ταξινομημένων υποπινάκων η οποία γίνεται στο βήμα **Combine** χρησιμοποιώντας τον αλγόριθμο MERGE.

Ο αλγόριθμος MERGE (σχήμα 2.1) ξεκινάει συγκρίνοντας το πρώτο στοιχείο του πίνακα a με το πρώτο στοιχείο του πίνακα b και επιλέγει το μικρότερο από τα δύο για να το τοποθετήσει πρώτο στον ταξινομημένο πίνακα c ο οποίος τελικά θα περιέχει τα στοιχεία και των δύο πινάκων. Εν συνεχεία ο μετρητής του πίνακα από τον οποίο επελέγει το πρώτο στοιχείο αυξάνεται κατά ένα και η διαδικασία σύγκρισης συνεχίζεται έτσι ώστε τελικά όλα τα στοιχεία των πινάκων a και b να τοποθετηθούν ταξινομημένα στον πίνακα c.

MERGE (a, b, c)

```
1.  i = 1
2.  j = 1
3.  for k = 1 to m + n do
4.      if a_i ≤ b_j then
5.          c_k = a_i
6.          i = i + 1
7.      else
8.          c_k = b_j
9.          j = j + 1
10.     end if
11. end for
```

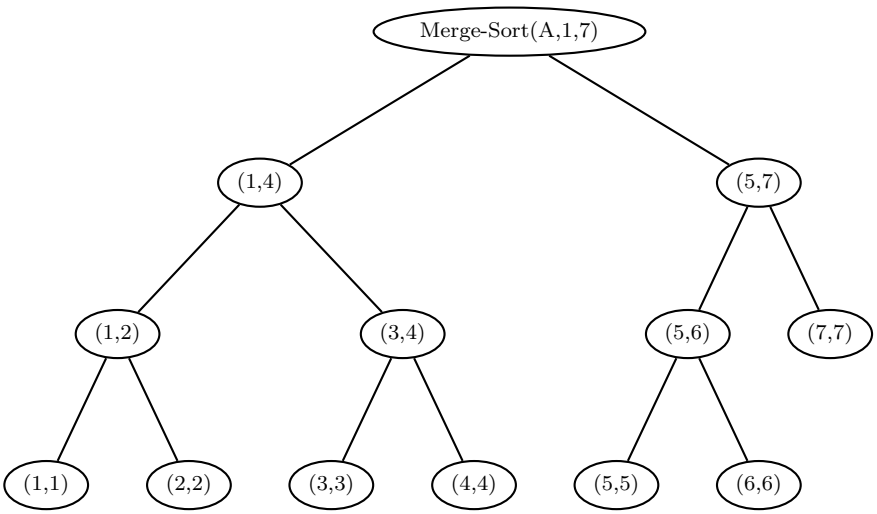
Σχήμα 2.1: Αλγόριθμος MERGE για την συγχώνευση των ταξινομημένων πινάκων a (μήκους m) και b (μήκους n) στον πίνακα c (μήκους $m + n$).

Για παράδειγμα, εάν ο πίνακας a περιέχει τα στοιχεία $2, 6, 7, 9$ και ο πίνακας b τα στοιχεία $4, 5, 8$ τότε ο πίνακας c ο οποίος θα προκύψει μετά την εκτέλεση του αλγορίθμου MERGE θα έχει ως εξής: $2, 4, 5, 6, 7, 8, 9$. Η πολυπλοκότητα του αλγορίθμου είναι $O(m + n)$.

Τον αλγόριθμο MERGE μπορούμε να τον χρησιμοποιήσουμε ως υπορουτίνα του αλγορίθμου MERGE-SORT (σχήμα 2.2), όπου A είναι ο πίνακας στοιχείων και l, r δείκτες που καθορίζουν τα άκρα του προς ταξινόμηση υποπίνακα του A.

MergeSort (A, l, r)
1. **if** $l < r$ **then**
2. $q \leftarrow \lfloor (l + r)/2 \rfloor$
3. MergeSort(A, l, q)
4. MergeSort$(A, q + 1, r)$
5. Merge /* Τους δύο υποπίνακες */
6. **end if**

Σχήμα 2.2: Αναδρομικός αλγόριθμος MergeSort για την ταξινόμηση των στοιχείων ενός πίνακα.



Σχήμα 2.3: Το δένδρο αναδρομικών κλήσεων του αλγορίθμου Merge-Sort.

Θα δούμε την λειτουργία του αλγορίθμου Merge-Sort με ένα παράδειγμα. Έστω ο πίνακας $A = 6, 9, 2, 7, 4, 5, 8$. Το δένδρο των αναδρομικών κλήσεων φαίνεται στο σχήμα 2.3. Ο αλγόριθμος υλοποιημένος σε μια γλώσσα τύπου C είναι ο εξής:

```
Merge-Sort int b[N] void MergeSort (int p, int r) {
    int i, j, k, m;
    if (p < r) {
      q = (p + r) / 2;
      MergeSort (p, q);
```

```
      MergeSort (q + 1, r);
      for (i = q; i >= p; --i)
        b[i] = a[i];
      for (j = q+1; j <= r; ++j)
        b[r+q+1-j] = a[j];
      i = p; j = r;
      for (k = p; k <= r; ++k)
        if (b[i] < b[j] {
           a[k] = b[i];
           ++i;
        } else {
           a[k] = b[j];
           --j;
        }
    }
}
```

Όταν ένας αλγόριθμος περιέχει μία ή περισσότερες αναδρομικές κλήσεις στον εαυτό του, ο συνολικός χρόνος εκτέλεσής του μπορεί συνήθως να περιγραφεί από μια *αναδρομική εξίσωση* ή απλά *αναδρομή*, μια συνάρτηση δηλαδή η οποία περιγράφει το συνολικό χρόνο εκτέλεσης του αλγορίθμου σε ένα πρόβλημα μεγέθους n χρησιμοποιώντας το χρόνο εκτέλεσης σε προβλήματα μικρότερου μεγέθους. Εν συνεχεία μπορούν να χρησιμοποιηθούν διάφορες μέθοδοι τις οποίες θα δούμε εκτενώς στη συνέχεια, για την επίλυση της αναδρομικής εξίσωσης η οποία έχει προκύψει.

Πιο συγκεκριμένα, ας συμβολίσουμε με $T(n)$ το χρόνο εκτέλεσης του αλγορίθμου MERGE-SORT για ένα πρόβλημα μεγέθους n. Εάν $n = 1$ τότε ο υποπίνακας με ένα στοιχείο είναι ήδη ταξινομημένος και ο αλγόριθμος παίρνει σταθερό χρόνο τον οποίο γράφουμε με $\Theta(1)$. Επίσης ο αλγόριθμος MERGE ο οποίος επιτυγχάνει τη συγχώνευση χρειάζεται χρόνο $\Theta(n)$. Συνολικά λοιπόν μπορούμε να γράψουμε την εξής αναδρομική εξίσωση:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Η επίλυση της παραπάνω αναδρομικής εξίσωσης μας δίδει $T(n) = \Theta(n \log n)$ όπως θα δούμε παρακάτω.

Γενικά, η εύρεση της αναδρομικής εξίσωσης η οποία περιγράφει το χρόνο εκτέλεσης ενός **divide and conquer** αλγορίθμου βασίζεται στα τρία βήματα (divide, conquer, combine) της μεθόδου. Αν υποθέσουμε ότι η υποδιαίρεση του αρχικού προβλήματος σε υποπροβλήματα οδηγεί στη δημιουργία a υποπροβλημάτων το καθένα από τα οποία έχει μέγεθος $1/b$ του αρχικού και ότι το βήμα combine χρειάζεται χρόνο $d(n)$, τότε λαμβάνουμε την παρακάτω αναδρομική εξίσωση:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ aT(\frac{n}{b}) + d(n) & \text{if } n > 1. \end{cases}$$

Μπορούμε να χρησιμοποιήσουμε την μέθοδο της *αντικατάστασης (substitution)* για να βρούμε τη λύση στην εξίσωση αυτή, ως εξής:

$$\begin{aligned}
T(n) &= aT(n/b) + d(n) \\
&= a[aT(n/b^2) + d(n/b)] + d(n) \\
&= a^2T(n/b^2) + ad(n/b) + d(n) \\
&= a^2[aT(n/b^3) + d(n/b^2)] + ad(n/b) + d(n) \\
&= a^3T(n/b^3) + a^2d(n/b^2) + ad(n/b) + d(n) \\
&= \dots \\
&= a^iT(n/b^i) + \sum_{j=0}^{i-1} a^j d(n/b^j)
\end{aligned}$$

Αν υποθέσουμε τώρα ότι $n = b^k$ τότε θα έχουμε ότι $T(n/b^k) = T(1) = 1$ και για $i = k$ θα λάβουμε:

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Επίσης $k = \log_b n$ και επομένως $a^k = a^{\log_b n} = n^{\log_b a}$. Στην περίπτωση της MERGE-SORT έχουμε $a = b = 2$ και $d(n) = n - 1 (= \Theta(n))$. Κάνοντας τις αντικαταστάσεις έχουμε:

$$\begin{aligned}
T(n) &= n + \sum_{j=0}^{k-1} 2^j(2^{k-j} - 1) \\
&= n + \sum_{j=0}^{k-1} (2^k - 2^j) \\
&= n + 2^k k - \frac{(2^k - 1)}{(2 - 1)} \\
&= n + n\log n - n + 1 \\
&= n\log n + 1
\end{aligned}$$

2.1.2 Binary Search

Ο αλγόριθμος της δυαδικής αναζητησης έχει ως είσοδο έναν ταξινομημένο πίνακα Α και ψάχνει να βρει αν υπάρχει ένα στοιχείο a μέσα σ΄ αυτον τον πίνακα. Παρακάτω περιγράφουμε πως ο αλγόριθμος εκμεταλλεύεται την τεχνική Divide-and-Conquer:

Divide: Βρίσκουμε το μεσαίο στοιχείο του πίνακα και τον χωρίζουμε σε δύο υποπίνακες. Αν το μεσαίο στοιχείο είναι μικρότερο του στοιχείου που ψάχνουμε, τότε κρατάμε τον υποπίνακα με τα μικρότερα από το μεσαίο στοιχεία,

διαφορετικά κρατάμε τον υποπίνακα με τα μεγαλύτερα από το μεσαίο στοιχεία.

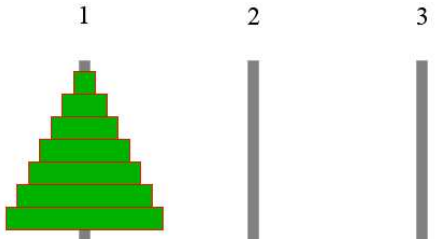
Conquer: Επαναλαμβάνουμε την παραπάνω διαδικασία στον επιλεγμένο υποπίνακα, μεχρίς ότου βρούμε το στοιχείο σαν μεσαίο η μεχρι να εχουμε κενούς υποπίνακες.

Combine: Δεν Υπάρχει βήμα Combine αφού με το πέρας όλων των αναδρομικών εκτελέσεων έχουμε τη λύση.

Η αναδρομική εξίσωση του αλγοριθμου είναι $T(n) = T(n/2) + c = O(logn)$. Οπότε, αντί να ψάχνουμε σειριακά σε όλο τον πίνακα σε $O(n)$ χρόνο χρησιμοποιώντας έναν αλγόριθμο Divide and Conquer ο χρόνος μειώνεται σε $O(logn)$!

2.1.3 Οι πύργοι του Hanoi

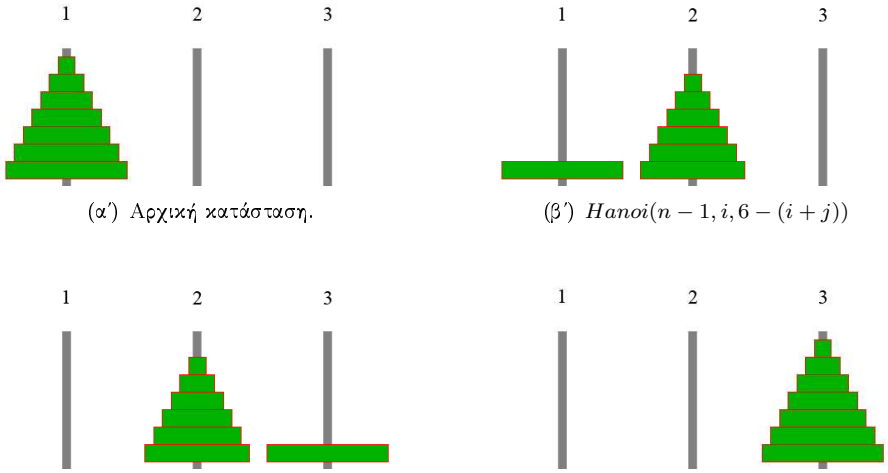
Ας εξετάσουμε ένα άλλο κλασσικό πρόβλημα στο οποίο χρησιμοποιείται η έννοια της αναδρομής. Πρόκειται για το πρόβλημα των πύργων του Hanoi το οποίο επινοήθηκε από τον γάλλο μαθηματικό Edouard Lucas το 1883. Στο πρόβλημα αυτό μας δίδονται 3 στύλοι 1,2,3 αντίστοιχα καθώς επίσης και n δίσκοι διαφορετικών μεγεθών οι οποίοι αρχικά είναι τοποθετημένοι στο στύλο 1, σχηματίζοντας έναν κώνο με τον πιο μεγάλο δίσκο κάτω και τον πιο μικρό πάνω, όπως φαίνεται στο σχήμα 2.4. Το ζητούμενο είναι να μεταφερθούν όλοι οι δίσκοι από το στύλο



Σχήμα 2.4: Οι πύργοι του Hanoi

1 στον στύλο 3, μετακινώντας μόνο ένα δίσκο τη φορά και προσέχοντας ποτέ να μην είναι ένας μικρός δίσκος κάτω από έναν μεγαλύτερο.

Ο αλγόριθμος ο οποίος επιλύει το πρόβλημα έχει ως εξής:



(α′) Αρχική κατάσταση.

(β′) $Hanoi(n-1, i, 6-(i+j))$

(γ′) Μετακίνηση του μεγάλου δίσκου από το στύλο i στον στύλο j.

(δ′) $Hanoi(n-1, 6-(i+j), j)$

Σχήμα 2.5: Σχηματική λειτουργία του αλγορίθμου.

HANOI (n, i, j)
1. **if** $n \geq 1$ **then**
2. HANOI$(n-1, i, 6-(i+j))$
3. Μεταφορά του μεγάλου δίσκου από τον στύλο i στον στύλο j
4. HANOI$(n-1, 6-(i+j), j)$
5. **end if**

Στο σχήμα 2.5 φαίνεται ο τρόπος λειτουργίας του αλγορίθμου.

Για να εξετάσουμε την πολυπλοκότητα του αλγορίθμου, γράφουμε την αναδρομική εξίσωση η οποία εκφράζει το χρόνο εκτέλεσης:

$$T(n) = \begin{cases} 0 & \text{if } n = 0, \\ 2T(n-1) + 1 & \text{if } n \geq 1. \end{cases}$$

Η επίλυση της παραπάνω αναδρομικής εξίσωσης θα γίνει με τη μέθοδο των *αθροιζομένων*

παραγόντων ως εξής:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \quad (\times 2^0) \\ T(n-1) &= 2T(n-2)+1 \quad (\times 2^1) \\ T(n-2) &= 2T(n-3)+1 \quad (\times 2^2) \\ &\dots \\ T(2) &= 2T(1)+1 \quad (\times 2^{n-2}) \\ T(1) &= 2T(0)+1 \quad (\times 2^{n-1}) \end{aligned}$$

Αθροίζοντας κατά μέλη λαμβάνουμε

$$T(n) = 2^n T(0) + (1 + \ldots + 2^{n-1}) = 2^n 0 + \frac{2^n - 1}{2-1}$$

και τελικά

$$T(n) = 2^n - 1, \qquad n \geq 0$$

Η γενική μορφή μιας αναδρομικής εξίσωσης όπως αυτή η οποία περιγράφει το χρόνο εκτέλεσης του αλγορίθμου Hanoi είναι η εξής:

$$a(n)T_n = b(n)T_{n-1} + c(n)$$

με T_0 σταθερά και $a(n), b(n), c(n)$ συναρτήσεις του n.

Άσκηση: Να επιλυθεί η αναδρομική εξίσωση $T(n) = T(n-1)+2$ με $T(1) = 1$.
Λύση:

$$\begin{aligned} T(n) &= 2 + T(n-1) \\ &= 2 + 2 + T(n-2) \\ &\dots \\ &= 2 + \ldots + 2 + T(1) \\ &= (n-1) \times 2 + 1 \\ &\leq 2n \end{aligned}$$

$$T(n) = \Theta(n)$$

2.1.4 Η ακολουθία Fibonacci

Ένα άλλο κλασσικό μαθηματικό πρόβλημα το οποίο εμπεριέχει την έννοια της αναδρομής είναι η ακολουθία Fibonacci. Η ακολουθία αυτή ορίζεται ως εξής:

$$T(n) = \begin{cases} 1 & \text{if } n = 0,\ n = 1, \\ T(n-1) + T(n-2) & \text{if } n > 1. \end{cases}$$

Πρακτικά κάθε αριθμός της ακολουθίας Fibonacci προκύπτει ως το άθροισμα των προηγουμένων δύο αριθμών της ακολουθίας. Για παράδειγμα οι 13 πρώτοι αριθμοί της ακολουθίας είναι: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233. Ένας επαναληπτικός αλγόριθμος ο οποίος μπορεί να χρησιμοποιηθεί για την εύρεση του $n-$οστού αριθμού της ακολουθίας είναι ο εξής:

FIBONACCI (n)
1. $f0 = 1$
2. $f1 = 1$
3. **for** $i = 3$ **to** n **do**
4. $\quad temp = f0 + f1$
5. $\quad f0 = f1$
6. $\quad f1 = temp$
7. **end for**
8. **return** f

Σχήμα 2.6: Επαναληπτικός Αλγόριθμος εύρεσης του n-οστού αριθμού Fibonacci

Για την εύρεση της ασυμπτωτικής πολυπλοκότητας του παραπάνω αλγορίθμου μπορούμε να παρατηρήσουμε ότι ο κύριος βρόγχος εκτελείται $n-2$ φορές, άρα η πολυπλοκότητα είναι της τάξης $O(n)$.

Μπορεί όμως να χρησιμοποιηθεί και ένας περισσότερο προφανής και πιο απλός αλγόριθμος ο οποίος βασίζεται στον αναδρομικό ορισμό της ακολουθίας.

Ο αλγόριθμος αυτός έχει ως εξής:

FIBONACCI (n)
1. $u = 1$
2. $f = 1$
3. **if** $n \leq 1$ **then**
4. $\quad$ **return** 1
5. **else**
6. $\quad$ **return** FIBONACCI$(n-1)$ + FIBONACCI$(n-2)$
7. **end if**

Σχήμα 2.7: Αναδρομικός Αλγόριθμος εύρεσης του n-οστού αριθμού Fibonacci

Ας θεωρήσουμε ένα παράδειγμα υπολογισμού κατά το οποίο ζητάμε να βρούμε τον 5ο αριθμό της ακολουθίας. Η εκτέλεση του αναδρομικού αλγορίθμου μπορεί να ιδωθεί ως ένα δένδρο αναδρομικών κλήσεων (άσκηση). Για την εύρεση της πολυπλοκότητας έχουμε:

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &= 1 + 1 + T(n-2) + T(n-3) + T(n-2) \quad (n \geq 3) \\ T(n) &\geq 2T(n-2) \\ &\geq 2^2 T(n-4) \\ &\dots \\ &\geq 2^{\frac{n}{2}} T(n - 2\frac{n}{2}) \end{aligned}$$

Άρα θα έχουμε

$$T(n) \geq 2^{\frac{n}{2}} T(0)$$

εάν n άρτιος και

$$T(n) \geq 2^{\frac{n-1}{2}} T(1)$$

εάν n περιττός, οπότε τελικά

$$T(n) \geq 2^{\frac{n}{2}},$$

δεδομένου ότι $T(0) = T(1) = 1$.

Παρατηρούμε λοιπόν ότι ο εκθετικός αλγόριθμος, αν και πολύ φυσικός στην σκέψη και εύκολος στην υλοποίηση έχει εκθετική πολυπλοκότητα. Το γεγονός αυτό οφείλεται στο ότι υπάρχει επανυπολογισμός αρκετών από τους ενδιάμεσους όρους, όπως μπορεί να φανεί και στο δένδρο υπολογισμών.

2.1.5 Ο αλγόριθμος Min-Max

Το τελευταίο πρόβλημα το οποίο θα θεωρήσουμε είναι το πρόβλημα της εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα A με n στοιχεία. Ένας απλός επαναληπτικός αλγόριθμος για την επίλυση του προβλήματος είναι αυτός του σχήματος 2.8.

MinMax (A)
1. $min = a_1$
2. $max = a_1$
3. **for** $i = 2$ **to** n **do**
4. **if** $a_i > max$ **then**
5. $max = a_i$
6. **else if** $a_i < min$ **then**
7. $min = a_i$
8. **end if**
9. **end for**

Σχήμα 2.8: Αλγόριθμος εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα

Ο παραπάνω αλγόριθμος στη χείριστη περίπτωση, όπου τα στοιχεία του πίνακα θα είναι ταξινομημένα σε φθίνουσα σειρά θα έχει ακριβή πολυπλοκότητα χρόνου $3(n-1)$, αν συμπεριλάβουμε και τη σύγκριση για να υλοποιήσουμε την επανάληψη for. Στην περίπτωση όπου τα στοιχεία του πίνακα είναι ταξινομημένα σε αύξουσα σειρά η ακριβής πολυπλοκότητα θα είναι $n-1$, άρα συνολικά θα έχουμε $\Theta(n)$.

Ας θεωρήσουμε τώρα όμως τον αναδρομικό αλγόριθμο του σχήματος 2.9 για την επίλυση του προβλήματος.

MinMax (A, i, j, min, max)

```
 1.  if i ≥ j − 1 then
 2.      if a_i < a_j then
 3.          max = a_j
 4.          min = a_i
 5.      else
 6.          max = a_i
 7.          min = a_j
 8.      end if
 9.  else
10.      m = ⌊(i + j)/2⌋
11.      MinMax(A, i, m, min_1, max_1)
12.      MinMax(A, m, j, min_2, max_2)
13.      min = fmin(min_1, min_2)
14.      max = fmax(max_1, max_2)
15.  end if
```

Σχήμα 2.9: Αναδρομικός Αλγόριθμος εύρεσης του μέγιστου και του ελάχιστου στοιχείου ενός πίνακα

Για την εύρεση της πολυπλοκότητας του αναδρομικού αλγορίθμου ας σημειώσουμε τα εξής:

- Ο έλεγχος $i \geq j-1$ καλύπτει τις δύο περιπτώσεις $i = j$ και $i = j-1$
- Οι συναρτήσεις fmax() και fmin() απαιτούν μία σύγκριση η κάθε μία για την εύρεση του μέγιστου ή του ελάχιστου (αντίστοιχα) μεταξύ δύο στοιχείων.

Η πολυπλοκότητα δίδεται από την παρακάτω αναδρομική σχέση:

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \text{ or } n = 1, \\ 2T(\frac{n}{2}) + 3 & \text{if } n > 2. \end{cases}$$

Για την επίλυσή της θα έχουμε (βλ. 2.1):

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + 3 \\
&= 2[2T(\frac{n}{2^2}) + 3] + 3 = 2^2T(\frac{n}{2^2}) + 2 \times 3 + 3 \\
&= 2^2[2T(\frac{n}{2^3}) + 3] + 2 \times 3 + 3 = 2^3T(\frac{n}{2^3}) + 2^2 \times 3 + 2 \times 3 + 3 \\
&= \ldots \\
&= 2^{k-1}T(\frac{n}{2^{k-1}}) + 2^{k-2} \times 3 + \ldots + 2 \times 3 + 3
\end{aligned}$$

και επομένως αν $n = 2^k$ έχουμε

$$\begin{aligned}
T(n) &= \frac{n}{2} \times T(2) + [\frac{2^{k-1} - 1}{2 - 1}] \times 3 \\
&= n + 3 \times 2^{k-1} - 3 \\
&= n + 3\frac{n}{2} - 3 \\
&= 5\frac{n}{2} - 3
\end{aligned}$$

Παρατηρούμε δηλαδή ότι στο συγκεκριμένο πρόβλημα η ακριβής πολυπλοκότητα του αναδρομικού αλγορίθμου είναι μικρότερη από την αντίστοιχη του επαναληπτικού, σε αντίθεση με το πρόβλημα των αριθμών Fibonacci που είδαμε προηγούμενα.

2.1.6 QuickSort

Ο Αλγόριθμος Quicksort είναι ένας αλγόριθμος ταξινόμησης που δημιουργήθηκε απο τον C. A. R. Hoare, με πολυπλοκότητα στη χείριστη περίπτωση $\Theta(n^2)$ σε είσοδο ενός πίνακα n αριθμών. Παρόλο που η πολυπλοκότητά του στη χείριστη περίπτωση είναι μεγάλη, ο Quicksort είναι συχνά η καλύτερη πρακτική επιλογή, γιατί είναι εκπληκτικά αποδοτικός στη μέση περίπτωση: Ο αναμενόμενος χρόνος είναι $\Theta(nlogn)$ και οι σταθεροί όροι που κρύβονται στο $\Theta(nlogn)$ είναι αρκετά μικροί. Ο αλγόριθμος έχει επίσης το πλεονέκτημα να ταξινομεί στον ίδιο πίνακα, χωρίς να χρειάζεται παραπάνω χώρο απ΄ οσο χρειάζεται η είσοδος.

Η QuickSort, οπώς και η MergeSort, ακολουθεί το παράδειγμα Divide-and-Conquer. Η διαδικασία Divide-and-Conquer 3 βημάτων για την ταξινόμηση ένος πίνακα $A[p \ldots r]$ είναι η εξής:

Divide: Διαμέριση του πίνακα $A[p \ldots r]$ σε δύο, πιθανώς κενούς, υποπίνακες $A[p \ldots q-1]$ και $A[q+1 \ldots r]$, τέτοιους ώστε κάθε στοιχείο του $A[p \ldots q-1]$ είναι μικρότερο ή ίσο από το $A[q]$ το οποίο με τη σειρά του είναι μικρότερο ή ίσο από κάθε στοιχείο του πίνακα $A[q+1 \ldots r]$. Διαμερίζοντας, βρίσκουμε τη θέση του $A[q]$ στον πίνακα $A[p \ldots r]$.

Conquer: Ταξινόμηση των δύο υποπινάκων $A[p \ldots q-1]$ και $A[q+1 \ldots r]$ με αναδρομικές κλήσεις της QuickSort.

Combine: Αφού οι υποπίνακες ταξινομούνται στον ίδιο πίνακα δε χρειάζεται παραπάνω εργασία για να συνδυάσουμε τις λύσεις με τους υποπίνακες. Ο συνολικός πίνακας $A[p \dots r]$ είναι ταξινομημένος μετά και την τελευταία αναδρομική κλήση της QuickSort.

Η διαδικασία του σχήματος 2.10 υλοποιεί την QuickSort.

QUICKSORT (A, p, r)
1. **if** $p < r$ **then**
2. $\quad q =$PARTITION(A, p, r)
3. $\quad$QUICKSORT $(A, p, q-1)$
4. $\quad$QUICKSORT $(A, q+1, r)$
5. **end if**

Σχήμα 2.10: Αλγόριθμος QUICKSORT

Το κλειδί του αλγορίθμου είναι η διαδικασία της διαμέρισης (Partition) η οποία επανατοποθετεί τα στοιχεία του πίνακα σύμφωνα με το στοιχείο περιστροφής (pivot) που έχει επιλεγεί: Τα μεγαλύτερα από αυτο μπαίνουν στα δεξιά του και τα μικρότερα στα αριστερά του, όπως φαίνεται στην υλοποίησή του στο σχήμα 2.11.

PARTITION (A, p, r)
1. $x = A[r]$
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$ **do**
4. $\quad$ **if** $A[j] \leq x$
5. $\quad\quad i = i + 1$
6. $\quad\quad swap(A[i],\ A[j])$
7. $\quad$ **end if**
8. **end for**
9. $swap(A[i+1],\ A[r])$
10. return $i + 1$

Σχήμα 2.11: Διαδικασία διαμέρισης με βάση το οδηγό στοιχείο.

Ο χρόνος εκτέλεσης της QuickSort εξαρτάται από το αν η διαμέριση είναι ισορροπημένη ή όχι. Αυτό εξαρτάται από το στοίχειο που θα επιλεγεί ως οδηγό στοιχείο κατά τη διαμέριση. Αν η διαμέριση είναι ισορροπημένη, τότε η QuickSort εκτελείται ασυμπτωτικά το ίδιο γρήγορα με τη MergeSort. Αν η διαμέριση δεν είναι ισορροπημένη, τότε η QuickSort θα εκτελεστεί αργά.

Διαμέριση Χείριστης Περίπτωσης

Η χείριστη περίπτωση προκύπτει όταν η διαμέριση χωρίζει τον πίνακα μεγέθους n σε 2 υποπίνακες μεγέθους $n-1$ και 0 στοιχείων. Ας υποθέσουμε ότι αυτο συμβαίνει σε κάθε αναδρομική κλήση. Η διαμέριση κοστίζει $\Theta(n)$ χρόνο. Έτσι η αναδρομική εξίσωση για αυτή την περίπτωση είναι

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Επομένως στην χείριστη περίπτωση, όταν δηλαδή η διαμέριση είναι η χειρότερη δυνατή, ο χρόνος είναι $T(n) = \Theta(n^2)$. Αξίζει να σημειωθεί ότι η παραπάνω περίπτωση θεωρείται εκφυλισμένη και συμβαίνει μόνο όταν η είσοδος είναι ένας ήδη ταξινομημένος πίνακας.

Διαμέριση Βέλτιστης Περίπτωσης

Στον πιο ευμενή διαχωρισμό, η διαδικασία Partition δημιουργεί δύο υποπροβλήματα κάθε ένα με μέγεθος όχι μεγαλύτερο από $n/2$, αφού το ένα υποπρόβλημα θα είναι μεγέθους $\lfloor n/2 \rfloor$ και το άλλο μεγέθους $\lceil n/2 \rceil - 1$. Σ′ αυτη την περίπτωση η QuickSort εκτελείται πολύ πιο γρήγορα. Η αναδρομική εξίσωση είναι $T(n) \leq 2T(n/2) + \Theta(n)$, η οποία ανήκει στην 2η περίπτωση του Master Theorem και έχει λύση την $T(n) = O(nlogn)$.

Διαμέριση Μέσης Περίπτωσης

Θεωρούμε όλες τις θέσεις για τοποθέτηση του οδηγού στοιχείου v ισοπίθανες ($= \frac{1}{n}$). Ο μέσος αριθμός συγκρίσεων T_n για $n \geq 2$ υπολογίζεται ως εξής:

$$T_0 = T_1 = 0$$

και

$$T_n = n + 1 + \frac{1}{n} \sum_{k=1}^{n} (T_{k-1} + T_{n-k})$$

θεωρώντας τις συγκρίσεις μεταξύ των στοιχείων μόνο.
Λόγω συμμετρίας έχουμε:

$$T_n = n + 1 + \frac{2}{n} \sum_{k=1}^{n} T_{k-1} \Rightarrow nT_n = n^2 + n + 2 \sum_{k=1}^{n} T_{k-1}$$

Για $n-1$ έχουμε:

$$T_{n-1} = n + \frac{2}{n-1} \sum_{k=1}^{n-1} T_{k-1} \Rightarrow (n-1)T_{n-1} = n^2 - n + 2 \sum_{k=1}^{n-1} T_{k-1}$$

Αφαιρώντας έχουμε μετά την απλοποίηση:

$$nT_n = (n+1)T_{n-1} + 2n$$

Διαιρώντας με $n(n+1)$ παίρνουμε:

$$\frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2}{n+1} = \frac{c}{3} + \sum_{k=3}^{n} \frac{2}{k+1}$$

Χρησιμοποιώντας την προσέγγιση:

$$\frac{T_n}{n+1} \simeq 2\sum_{k=1}^{n} \frac{1}{k} \simeq 2\int_1^n \frac{1}{x}dx = 2\ln n$$

παίρνουμε το αποτέλεσμα:

$$T_n = 1.38n\log n$$

2.2 Ασυμπτωτική προσέγγιση μερικών αθροισμάτων

Όταν μια συνάρτηση είναι μονότονη, τότε έχουμε τη δυνατότητα να φράξουμε το άθροισμά της από το ολοκλήρωμά της. Εάν επιπλέον η συνάρτηση μεταβάλλεται αργά, τότε το άθροισμά της και το ολοκλήρωμά της έχουν την *ίδια* τάξη μεγέθους ασυμπτωτικά.
Για κάθε μονότονη φθίνουσα συνάρτηση ισχύει η ακόλουθη σχέση (βλ. σχήμα 2.12):

$$\int_p^{q+1} f(x)dx \leq \sum_{i=p}^{q} f(i) \leq \int_{p-1}^{q} f(x)dx$$

Για να δούμε την χρησιμότητα της παραπάνω σχέσης ας θεωρήσουμε την συνάρτηση

$$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$$

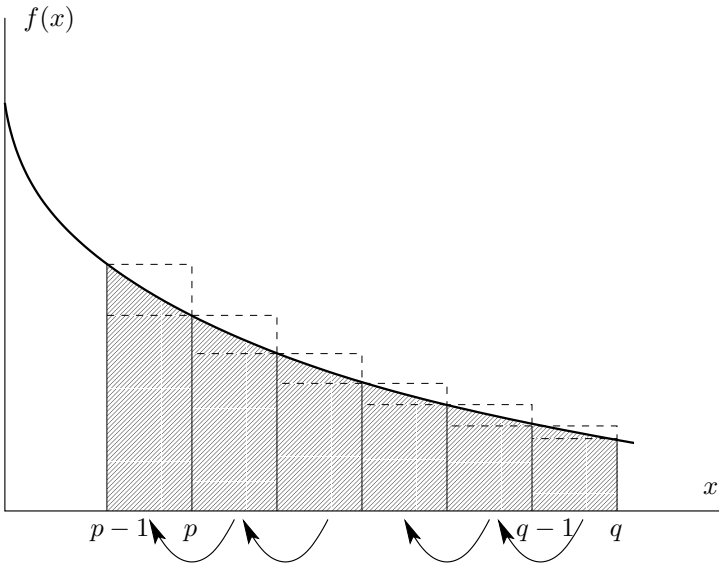
η οποία μας δίδει τον n-οστό αρμονικό αριθμό και ας εφαρμόσουμε την πιο πάνω σχέση. Εάν θέσουμε $f(x) = \frac{1}{x}$ τότε θα έχουμε

$$\int_2^{n+1} \frac{1}{x}dx \leq \sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x}dx$$

και επομένως, για τον n-οστό αρμονικό αριθμό θα ισχύει

$$\ln(n+1) - \ln(2) + 1 \leq H_n \leq \ln(n) + 1$$

δηλαδή $H_n = \Theta(\ln n)$.



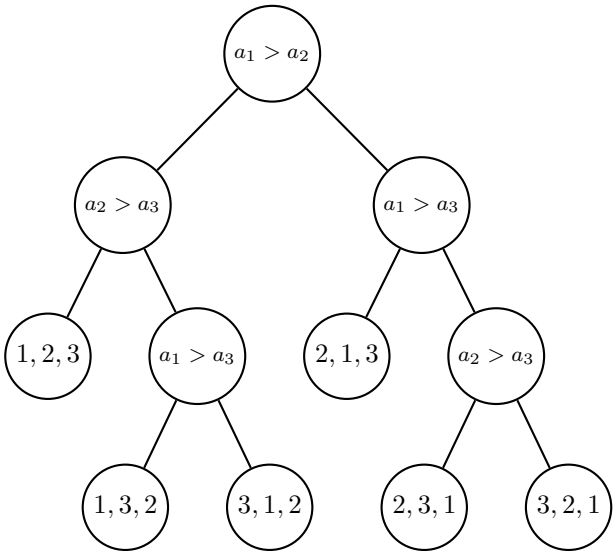
Σχήμα 2.12: Προσέγγιση μονότονης συνάρτησης από το ολοκλήρωμά της.

2.3 Δένδρα Απόφασης

Τα δένδρα απόφασης είναι κλασσικά δυαδικά δένδρα στα οποία ισχύουν μερικές ενδιαφέρουσες ιδιότητες. Κάθε εσωτερικός κόμβος ενός δένδρου απόφασης έχει ως ρόλο να θέτει μια ερώτηση όσο αφορά στη σύγκριση δύο στοιχείων. Το αριστερό παιδί του κόμβου αντιστοιχεί σε *αρνητική* απάντηση της ερώτησης. Το δεξιό παιδί του κόμβου αντιστοιχεί σε *θετική* απάντηση. Τα φύλλα ενός δένδρου απόφασης αντιπροσωπεύουν την μετάθεση που πρέπει να γίνει έτσι ώστε να επιτύχουμε τον ταξινομημένο πίνακα. Στο σχήμα 2.13 φαίνεται ένα παράδειγμα δένδρου απόφασης.

Κάθε δένδρο απόφασης το οποίο προορίζεται για την ταξινόμηση n στοιχείων έχει $n!$ φύλλα, όσες δηλαδή και οι δυνατές μεταθέσεις των n στοιχείων. Το ύψος h ενός τέτοιου δένδρου είναι $h \geq \lceil \log_2(n!) \rceil$. Αναπτύσσοντας ασυμπτωτικά το $n!$ από τον τύπο του Stirling θα έχουμε

$$n! = \sqrt{(2\pi n)} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + o(\frac{1}{n})\right)$$



Σχήμα 2.13: Παράδειγμα ενός δένδρου απόφασης.

και επομένως

$$\log_2(n!) = \log_2\left(\sqrt{(2\pi n)}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}+o(\frac{1}{n})\right)\right) =$$

$$n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n + \frac{1}{2}\log_2 2\pi + \log_2\left(1+\frac{1}{12n}+o\left(\frac{1}{n}\right)\right)$$

δηλαδή τελικά $h = \Omega(n \log n)$.

Όπως είναι προφανές, ακολουθώντας μια διαδρομή στο δένδρο και απαντώντας στις ερωτήσεις μπορούμε να έχουμε μια ταξινόμηση των n στοιχείων. Επομένως η ταξινόμηση n στοιχείων βασισμένη σε συγκρίσεις των στοιχείων ανά δύο, απαιτεί $\Omega(n \log n)$ συγκρίσεις. Συμπεραίνουμε ότι οι αλγόριθμοι ταξινόμησης με σωρό, με συγχώνευση και με διχοτομική εισαγωγή είναι βέλτιστοι αλγόριθμοι στην χείριστη περίπτωση. Αντίθετα ο αλγόριθμος QuickSort δεν είναι βέλτιστος στην χείριστη περίπτωση, αλλά είναι βέλτιστος προς την κατά μέσο όρο πολυπλοκότητα.

2.4 The Master Theorem

Στην παράγραφο αυτή θα περιγράψουμε μια μέθοδο γνωστή και ως *Master* μέθοδο ή αλλιώς *Master Theorem* η οποία μας βοηθά στο να επιλύουμε γρήγορα

αναδρομικές εξισώσεις της μορφής

$$T(n) = aT(\frac{n}{b}) + f(n)$$

όπου $a \geq 1$ και $b > 1$. Πριν υπεισέλθουμε στην περιγραφή και τις λεπτομέρειες της μεθόδου θα κάνουμε μερικές υπενθυμίσεις όσο αφορά σε βασικές σχέσεις που ισχύουν στις αναδρομικές εξισώσεις καθώς και σε θέματα συμβολισμού.

Για τη συνέχεια θα χρησιμοποιήσουμε τον συμβολισμό του παρακάτω πίνακα.

$$\begin{aligned}
\lg n + k &= (\lg n) + k \\
\lg(n) &= \log_2(n) \quad \text{(δυαδικός λογάριθμος)} \\
\ln n &= \log_e(n) \quad \text{(φυσικός λογάριθμος)} \\
\lg^k n &= (\lg n)^k \quad \text{(πολυλογάριθμος)} \\
\lg \lg n &= \lg(\lg n) \quad \text{(σύνθεση)}
\end{aligned}$$

Για όλους τους πραγματικούς αριθμούς $a > 0,\ b > 0,\ c > 0$ ισχύουν οι παρακάτω σχέσεις:

$$\begin{aligned}
a &= b^{\log_b a} \\
n &= 2^{\log_2 n} \\
\log_c(ab) &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b \frac{1}{a} &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}$$

Τέλος, υπενθυμίζουμε μια σειρά από σχέσεις - ιδιότητες οι οποίες ισχύουν και οι οποίες θα φανούν χρήσιμες στη συνέχεια.

- Μία συνάρτηση $f(n)$ είναι πολυωνυμικά φραγμένη αν $f(n) = O(n^k)$ για κάποια σταθερά k.

- Κάθε εκθετική συνάρτηση με βάση μεγαλύτερη του 1 αυξάνεται γρηγορότερα από κάθε πολυωνυμική συνάρτηση. Δηλαδή, έχουμε για a και b σταθερές, με $a > 1$, ότι:
$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n).$$

- Η $f(n)$ είναι πολυλογαριθμικά φραγμένη αν $f(n) = O(\log^k n)$ για κάποια σταθερά k.

- Κάθε θετική πολυωνυμική συνάρτηση αυξάνεται γρηγορότερα από κάθε πολυλογαριθμική συνάρτηση. Δηλαδή,έχουμε για k και b θετικές σταθερές ότι:

$$\lim_{n\to\infty} \frac{\log^k n}{n^b} = 0 \Rightarrow \log^k(n) = o(n^b).$$

2.4.1 Το Master θεώρημα

Ας θεωρήσουμε την αναδρομική εξίσωση

$$T(n) = aT(\frac{n}{b}) + f(n)$$

όπου $a \geq 1$ και $b > 1$ και $f(n)$ μια ασυμπτωτική θετική συνάρτηση. Για να είναι καλώς ορισμένη η παραπάνω αναδρομική εξίσωση και επειδή ο λόγος $\frac{n}{b}$ μπορεί να μην είναι ακέραιος, με το $\frac{n}{b}$ εννοούμε $\lfloor \frac{n}{b} \rfloor$ ή $\lceil \frac{n}{b} \rceil$.

Θεώρημα 1. Η $T(n)$ φράσσεται ασυμπτωτικά ως εξής:

1. Εάν $f(n) = O(n^{\log_b a - \epsilon})$ για κάποια σταθερά $\epsilon > 0$ τότε $T(n) = \Theta(n^{\log_b a})$.
2. Εάν $f(n) = \Theta(n^{\log_b a})$ τότε $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Εάν $f(n) = \Omega(n^{\log_b a + \epsilon})$ για κάποια σταθερά $\epsilon > 0$ και εάν $af(\frac{n}{b}) \leq cf(n)$ για κάποια σταθερά $c < 1$ και όλα τα αρκετά μεγάλα n τότε $T(n) = \Theta(f(n))$.

Σε κάθε μία από τις τρεις περιπτώσεις του θεωρήματος συγκρίνουμε τη συνάρτηση $f(n)$ με τη συνάρτηση $n^{\log_b a}$. Διαισθητικά, η λύση της αναδρομικής εξίσωσης καθορίζεται από την μεγαλύτερη από τις δύο αυτές συναρτήσεις. Εάν, όπως στην περίπτωση 1, η συνάρτηση $n^{\log_b a}$ είναι η μεγαλύτερη τότε η λύση είναι $T(n) = \Theta(n^{\log_b a})$. Εάν συμβαίνει το αντίθετο, όπως στην περίπτωση 3, η λύση είναι $T(n) = \Theta(f(n))$. Τέλος, εάν όπως στην περίπτωση 2, οι δύο συναρτήσεις είναι της ίδιας τάξης μεγέθους, τότε απλά πολλαπλασιάζουμε με ένα λογαριθμικό παράγοντα και η λύση είναι $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Πέρα από την παραπάνω διαίσθηση αξίζει να παρατηρήσουμε ορισμένες τεχνικές λεπτομέρειες. Στην πρώτη περίπτωση του θεωρήματος, η συνάρτηση $f(n)$ δεν πρέπει απλά να είναι μικρότερη από την $n^{\log_b a}$, αλλά πρέπει να είναι *πολυωνυμικά* μικρότερη. Δηλαδή η $f(n)$ πρέπει να είναι ασυμπτωτικά μικρότερη από την $n^{\log_b a}$ κατά ένα παράγοντα n^ϵ για κάποιο σταθερό $\epsilon > 0$. Επίσης, στην τρίτη περίπτωση του θεωρήματος, η $f(n)$ πρέπει επίσης να είναι *πολυωνυμικά* μεγαλύτερη από την $n^{\log_b a}$, αλλά επίσης πρέπει να ικανοποιεί και την συνθήκη κανονικότητας $af(\frac{n}{b}) \leq cf(n)$. Αξίζει να σημειώσουμε ότι η τελευταία συνθήκη ικανοποιείται από τις περισσότερες πολυωνυμικά φραγμένες συναρτήσεις.

Πριν μελετήσουμε κάποια παραδείγματα εφαρμογής του θεωρήματος, είναι σημαντικό να κατανοήσουμε ότι οι τρεις περιπτώσεις δεν καλύπτουν όλες τις πιθανές εκδοχές για τη συνάρτηση $f(n)$. Μεταξύ των περιπτώσεων 1 και 2 υπάρχει η περίπτωση κατά την οποία η $f(n)$ είναι μικρότερη από την $n^{\log_b a}$, αλλά όχι πολυωνυμικά μικρότερη. Ομοίως, μεταξύ των περιπτώσεων 2 και 3 υπάρχει και η περίπτωση κατά την οποία η $f(n)$ είναι μεγαλύτερη από την $n^{\log_b a}$, αλλά όχι

πολυωνυμικά μεγαλύτερη. Εάν λοιπόν η συνάρτηση $f(n)$ συμπίπτει με κάποια από αυτές τις ενδιάμεσες περιπτώσεις, ή εάν δεν ικανοποιείται η συνθήκη κανονικότητας της περίπτωσης 3, το θεώρημα προφανώς δεν μπορεί να εφαρμοστεί.

2.4.2 Παραδείγματα

Για να χρησιμοποιήσουμε τη Master μέθοδο απλά αποφασίζουμε σε ποια από τις τρεις περιπτώσεις εμπίπτει η αναδρομική εξίσωση που καλούμαστε να επιλύσουμε.

1. $T(n) = 9T(\frac{n}{3}) + n$.
   Σε αυτή την εξίσωση έχουμε $a = 9$, $b = 3$ και $f(n) = n$. Συνεπώς $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Αφού λοιπόν $f(n) = O(n^{\log_3 9-\epsilon})$ όπου $\epsilon = 1$, μπορούμε να εφαρμόσουμε την περίπτωση 1 του θεωρήματος και να συμπεράνουμε ότι η λύση είναι $T(n) = \Theta(n^2)$.

2. $T(n) = T(\frac{2n}{3}) + 1$.
   Εδώ έχουμε $a = 1$, $b = \frac{3}{2}$ και $f(n) = 1$. Συνεπώς $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Οπότε αφού $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, εφαρμόζουμε την περίπτωση 2 και η λύση είναι $T(n) = \Theta(\lg n)$.

3. $T(n) = 3T(\frac{n}{4}) + n \log n$. Σε αυτή την εξίσωση έχουμε $a = 3$, $b = 4$ και $f(n) = n \log n$ και $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Δηλαδή $f(n) = \Omega(n^{\log_4 3+\epsilon})$ με $\epsilon \approx 0.2$. Επιπρόσθετα $af(\frac{n}{b}) = 3(\frac{n}{4}) \log(\frac{n}{4}) \leq \frac{3}{4} n \log n = cf(n)$ με $c = \frac{3}{4}$. Συνεπώς μπορούμε να εφαρμόσουμε την περίπτωση 3 και άρα η λύση είναι $T(n) = \Theta(n \log n)$.

4. $T(n) = 2T(\frac{n}{2}) + n \log n$.
   Εδώ έχουμε $a = 2$, $b = 2$ και $f(n) = n \log n$. Συνεπώς $n^{\log_b a} = n$. Αλλά ενώ $n \log n > n$ ο λόγος $\frac{n \log n}{n} = \log n$ είναι ασυμπτωτικά μικρότερος από το n^ϵ για κάθε θετική σταθερά ϵ. Κατά συνέπεια η $f(n)$ δεν είναι πολυωνυμικά μεγαλύτερη, κατά ένα παράγοντα n^ϵ, από την συνάρτηση $n^{\log_b a} = n$ και η περίπτωση 3 του θεωρήματος δεν μπορεί να εφαρμοστεί.

2.5 Αναδρομικές γραμμικές εξισώσεις

Οι αναδρομικές εξισώσεις χωρίζονται σε κατηγορίες ανάλογα με

1. τον τύπο της συνάρτησης f. Η συνάρτηση f μπορεί να είναι γραμμικός συνδυασμός των $T(p)$, με συντελεστές σταθερούς ή μεταβλητούς ή πωλυώνυμα $T(p)$ κ.τ.λ.

2. το σύνολο τιμών p που εμπλέκονται για τον υπολογισμό του $T(n)$. Πιο συγκεκριμένα έχουμε:

   - $T(n) = f(T(n-1))$ εξίσωση τάξης 1
   - $T(n) = f(T(n-1), \ldots, T(n-k))$ εξίσωση τάξης k για k φιξαρισμένο
   - $T(n) = f(\{T(p); \forall p < n\})$ πλήρης εξίσωση

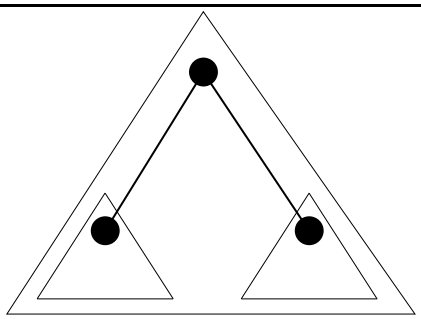
2.5.1 Γραμμικές αναδρομές τάξης k

Οι γραμμικές αναδρομές τάξης k είναι την μορφής

$$T(n) = f(n, T(n-1), \ldots, T(n-k)) + g(n)$$

όπου $k \geq 1$ σταθερά, f γραμμικός συνδυασμός των $T(i)$ για $n-k \leq i \leq n-1$ και g οποιαδήποτε συνάρτηση του n.
Παράδειγμα: Πλήρες δυαδικό δένδρο. Ο αριθμός κόμβων a_n του πλήρους δυαδικού δένδρου ύψους n είναι ίσος με τον αριθμό των κόμβων των δύο υποδένδρων ύψους $n-1$ συν 1 κόμβο ο οποίος είναι η ρίζα. Άρα (βλ. σχήμα 2.14)

$$a_n = 2a_{n-1} + 1 \ (n \geq 1, k = 1, a_0 = 1)$$



Σχήμα 2.14: Πλήρες δυαδικό δένδρο

2.5.2 Αναδρομές διαμερίσεων

Ο γενικός τύπος των αναδρομικών εξισώσεων αυτής της κατηγορίας είναι:

$$T(n) = aT(\frac{n}{b}) + d(n)$$

όπου a, b ακέραιες σταθερές και $d(n)$ οποιαδήποτε συνάρτηση. Τέτοιου τύπου αναδρομικές εξισώσεις προκύπτουν συνήθως από αναδρομικούς αλγορίθμους οι οποίοι χρησιμοποιούν διαμερίσεις.
Παράδειγμα: Ο αλγόριθμος Mergesort. Ο αριθμός των συγκρίσεων του αλγορίθμου Mergesort $T(n)$ δίδεται από την εξίσωση

$$T(n) = 2T(\frac{n}{2}) + n - 1 \quad n \geq 2$$
$$T(1) = 0$$

όπου $n-1$ είναι το κόστος της συγχώνευσης των δύο υποπινάκων.

2.5.3 Αναδρομές πλήρεις, γραμμικές ή πολυωνυμικές

Ο γενικός τύπος των αναδρομικών εξισώσεων αυτής της κατηγορίας είναι:

$$T(n) = f(n, T(n-1), T(n-2), \ldots, T(0)) + g(n)$$

όπου f μια συνάρτηση γραμμική ή πολυωνυμική των $T(i)$ και $g(n)$ οποιαδήποτε συνάρτηση.

Παράδειγμα: Αναδρομή πλήρης πολυωνυμική. Ο αριθμός δυαδικών δένδρων με n κόμβους δίδεται από την παρακάτω αναδρομική εξίσωση

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$
$$b_0 = 1$$

Παράδειγμα: Αναδρομή πλήρης γραμμική. Η κατά μέσο όρο πολυπλοκότητα της Quicksort δίδεται από την αναδρομική εξίσωση

$$C_n = (n+1) + \frac{2}{n} \sum_{k=1}^{n-1} C_k, \quad n \geq 2$$
$$C_0 = C_1 = 0$$

2.5.4 Παραδείγματα Γραμμικών Αναδρομικών Εξισώσεων

1. Να επιλυθεί η παρακάτω αναδρομική εξίσωση, (αριθμός κόμβων ενός πλήρους δυαδικού δέντρου ύψους n)

$$a_n = a_{n-1} + 2^n, \quad n \geq 1$$
$$a_0 = 1$$

Λύση:

$$a_n = a_{n-1} + 2^n$$
$$a_{n-1} = a_{n-2} + 2^{n-1}$$
$$a_{n-2} = a_{n-3} + 2^{n-2}$$
$$\ldots$$
$$a_2 = a_1 + 2^2$$
$$a_1 = a_0 + 2^1$$

Αθροίζοντας κατά μέλη λαμβάνουμε:

$$a_n = a_0 + \sum_{i=1}^{n} 2^i, \quad n \geq 1$$
$$a_n = 1 + \frac{2^{n+1} - 2}{2 - 1}$$
$$a_n = 2^{n+1} - 1$$

2. Να επιλυθεί η προηγούμενη αναδρομική εξίσωση με τη μέθοδο αθροιζόμενων παραγόντων.

Λύση:

$$\begin{aligned} a_n &= 2a_{n-1} + 1 \quad (\times 2^0) \\ a_{n-1} &= 2a_{n-2} + 1 \quad (\times 2^1) \\ a_{n-2} &= 2a_{n-3} + 1 \quad (\times 2^2) \\ &\dots \\ a_2 &= 2a_1 + 1 \quad (\times 2^{n-2}) \\ a_1 &= 2a_0 + 1 \quad (\times 2^{n-1}) \end{aligned}$$

Χρησιμοποιώντας τη μέθοδο των αθροιζομένων παραγόντων λαμβάνουμε:

$$\begin{aligned} a_n &= 2^n a_0 + \sum_{i=0}^{n-1} 2^i \\ a_n &= 2^n + \frac{2^n - 1}{2 - 1} \\ a_n &= 2^{n+1} - 1, \quad n \geq 0 \end{aligned}$$

2.5.5 Γραμμικές Αναδρομικές Εξισώσεις με σταθερούς συντελεστές

Η γραμμική εξίσωση τάξης k με σταθερούς συντελεστές είναι της μορφής:

$$u_n + a_1 u_{n-1} + a_2 u_{n-2} + \ldots + a_k u_{n-k} = b(n)$$

με a_i σταθερές. Η εξίσωση

$$u_n + a_1 u_{n-1} + a_2 u_{n-2} + \ldots + a_k u_{n-k} = 0$$

καλείται αντίστοιχη ομογενής εξίσωση της αναδρομικής.

Το σύνολο των λύσεων της ομογενούς εξίσωσης σχηματίζει έναν διανυσματικό χώρο διάστασης μικρότερης είτε ίσης του k και κάποιες λύσεις είναι της μορφής:

$$u_n = r^n.$$

Αν αντικαταστήσουμε στην εξίσωση παίρνουμε την χαρακτηριστική εξίσωση της ομογενούς εξίσωσης:

$$r^k + a_1 r^{k-1} + \ldots + a_k = 0$$

Αν οι k ρίζες της εξίσωσης είναι διαφορετικές $r_1, r_2, \ldots, r_k$ τότε η λύση της ομογενούς εξίσωσης είναι:

$$u_n = \lambda_1 r_1^n + \lambda_2 r_2^n + \ldots + \lambda_k r_k^n$$

με λ_i σταθερές που βρίσκουμε με την βοήθεια των αρχικών τιμών $u_1, \ldots, u_{k-1}$.

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση:

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$
$$F_0 = 0$$
$$F_1 = 1$$

Λύση: Έχουμε ότι:

$$F_n - F_{n-1} - F_{n-2} = 0.$$

Αν θεωρήσουμε ότι $u_n = r^n$ τότε η χαρακτηριστική εξίσωση της ομογενούς εξίσωσης γράφεται:

$$r^n - r^{n-1} - r^{n-2} = 0 \Rightarrow$$
$$r^{n-2}(r^2 - r - 1) = 0 \Rightarrow$$
$$r^2 - r - 1 = 0.$$

Οι λύσεις της εξίσωσης αυτής είναι:

$$r_1 = \frac{1+\sqrt{5}}{2}$$
$$r_2 = \frac{1-\sqrt{5}}{2}$$

Συνεπώς η λύση της ομογενούς εξίσωσης είναι της μορφής:

$$F_n = \lambda_1 r_1^n + \lambda_2 r_2^n$$

από: $F_0 = 0 \Leftrightarrow \lambda_1 = -\lambda_2$ και $F_1 = 1 \Leftrightarrow \lambda_1 r_1 + \lambda_2 r_2 = 1 \Leftrightarrow \lambda_1 r_1 - \lambda_1 r_2 = 1$ άρα τελικά $\lambda_1 = \frac{1}{r_1 - r_2}$ και $\lambda_2 = \frac{1}{r_2 - r_1}$ δηλαδή

$$\lambda_1 = \frac{1}{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}} = \frac{1}{\sqrt{5}}$$

$$\lambda_2 = -\frac{1}{\sqrt{5}}$$

Τελικά, η λύση είναι:

$$F_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

2.5.6 Εξισώσεις μετατρεπόμενες σε γραμμικές αναδρομές

Για την μετατροπή εξισώσεων σε γραμμικές αναδρομικές εξισώσεις και την επιτυχή επίλυσή τους εργαζόμαστε ως εξής: Ας υποθέσουμε την αναδρομική εξίσωση

$$T(n) = aT(\frac{n}{b}) + d(n)$$

με $n \geq 2$, a, b σταθερές και $T(1) = 1$. Το αρχικό πρόβλημα διάστασης n προκύπτει από το συνδυασμό a υποπροβλημάτων διάστασης $\frac{n}{b}$, ενώ $d(n)$ είναι το κόστος δημιουργίας των υποπροβλημάτων και το κόστος κατασκευής της λύσης από τις υπολύσεις. Επιπρόσθετα, ας υποθέσουμε ότι $n = b^k$ και άρα η αρχική εξίσωση γίνεται

$$T(b^k) = aT(b^{k-1}) + d(b^k).$$

Εάν τώρα θέσουμε $t_k = T(b^k)$ η εξίσωση η οποία προκύπτει από την αρχική με αντικατάσταση είναι γραμμική:

$$t_k = at_{k-1} + d(b^k)$$

με $t_0 = 1$.

Ας θεωρήσουμε, επιπρόσθετα την εξίσωση:

$$a(n)u_n = b(n)u_{n-1} + c(n).$$

Αν ισχύει

$$f(n) = \frac{\prod_{i=1}^{n-1} a(i)}{\prod_{i=1}^{n} b(i)}$$

τότε

$$v_n = v_{n-1} + f(n)c(n)$$

με $v_n = a(n)f(n)u_n$ και χρησιμοποιώντας τη μέθοδο των αθροιζομένων παραγόντων θα έχουμε

$$u_n = \frac{1}{a(n)f(n)}(a(0)f(0)u_0 + \sum_{i=1}^{n} f(i)c(i))$$

και άρα

$$t_k = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Τελικά, αφού $k = \log_b n$ και $a^{\log_b n} = n^{\log_b a}$ η αρχική εξίσωση γίνεται

$$T(n) = n^{\log_b a} + \sum_{j=0}^{(\log_b n)-1} a^j d(\frac{n}{b^j}).$$

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση (βλ. 2.1.1 πολυπλοκότητα Merge Sort):

$$T(n) = 2T(\frac{n}{2}) + n - 1 \quad n \geq 2$$
$$T(1) = 1$$

Λύση: Έχουμε $a = b = 2$ και $d(n) = n - 1$ άρα

$$\begin{aligned} T(n) &= n + \sum_{j=0}^{k-1} 2^j((\frac{n}{2^j}) - 1) \\ &= n + kn - (2^k - 1) \end{aligned}$$

και επειδή $k = \log_2 n$ τελικά

$$T(n) = n \log_2 n + 1.$$

Μία άλλη μέθοδος για την μετατροπή μιας αναδρομικής εξίσωσης σε γραμμική είναι ο πολλαπλασιασμός αυτής με ένα αθροιστικό παράγοντα s_n. Ας υποθέσουμε την εξίσωση

$$a_n T_n = b_n T_{n-1} + c_n.$$

Τότε θα έχουμε:

$$s_n a_n T_n = s_n b_n T_{n-1} + s_n c_n.$$

Διαλέγοντας το s_n έτσι ώστε $s_n b_n = s_{n-1} a_{n-1}$ και θέτοντας $u_n = s_n a_n T_n$ η εξίσωση γίνεται

$$u_n = u_{n-1} + s_n c_n$$

και συνεπώς

$$u_n = s_0 a_0 T_0 + \sum_{k=1}^{n} s_k c_k = s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k$$

και η λύση είναι

$$T_n = \frac{u_n}{s_n a_n} = \frac{s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k}{s_n a_n}$$

δηλαδή τελικά

$$T_n = \frac{1}{s_n a_n (s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k)}.$$

Παράδειγμα: Να επιλυθεί η παρακάτω αναδρομική εξίσωση (βλ. 2.1.3 πύργοι του Hanoi):

$$T_n = 2T_{n-1} + 1$$

Λύση: Έχουμε $u_n = 1$, $b_n = 2$ και $s_n = 2^{-n}$ άρα

$$\begin{aligned} T_n &= \frac{1}{2^{-n}(0 + \sum_{k=1}^{n} s_k c_k)} = 2^n(s_1 + \ldots + s_n) \\ &= 2^n - 1 \end{aligned}$$

Κεφάλαιο 3

Σωροί

3.1 Δένδρα

Ένα δένδρο είναι μια δομή δεδομένων η οποία μπορεί να είναι είτε ένα ατομικό δένδρο (ένα φύλλο), είτε ένας κόμβος και μια ακολουθία από υποδένδρα. Οι κόμβοι ενός δένδρου διακρίνονται σε *εσωτερικούς* κόμβους και *φύλλα*. Τα φύλλα είναι κόμβοι οι οποίοι δεν έχουν παιδιά. Επιπρόσθετα, το βάθος ενός κόμβου είναι το μήκος του μονοπατιού το οποίο ενώνει τον κόμβο αυτό με τη ρίζα. Κατά σύμβαση θεωρούμε πάντα ότι η ρίζα έχει βάθος 0. Το ύψος ενός κόμβους είναι το μήκος του μεγαλύτερου μονοπατιού που ενώνει τον κόμβο με τα φύλλα. Το ύψος του δένδρου είναι το ύψος της ρίζας.

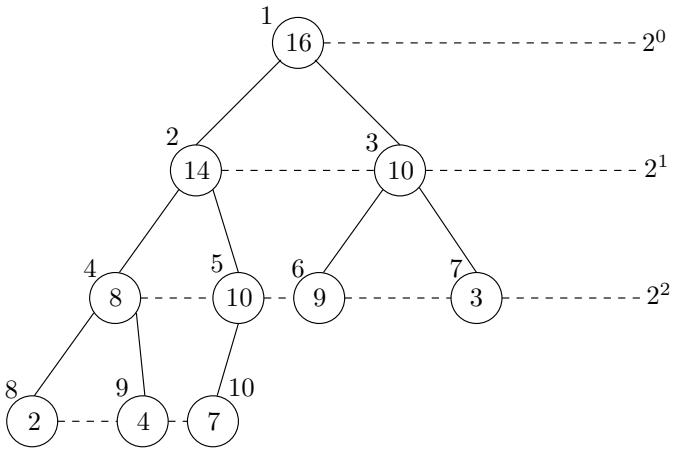
Μια ειδική κατηγορία δένδρων είναι τα *δυαδικά δένδρα* στα οποία κάθε κόμβος έχει το πολύ δύο παιδιά. Αν διαιρέσουμε το σύνολο των κόμβων ενός τέτοιου δένδρου σε γραμμές ακολουθώντας τα βάθη τους, θα έχουμε μια γραμμή (τη γραμμή 0) η οποία περιέχει τη ρίζα (2^0), μια γραμμή (τη γραμμή 1) η οποία περιέχει το πολύ 2 κόμβους (2^1) κ.ο.κ. Γενικά η γραμμή i περιέχει το πολύ 2^i κόμβους.

Σχεδόν πλήρες καλείται ένα δυαδικό δένδρο όταν όλες οι γραμμές, εκτός ίσως από την τελευταία, περιέχουν το μέγιστο αριθμό κόμβων (δηλαδή 2^i). Επιπλέον ισχύει μια σειρά από ιδιότητες όπως:

- Τα φύλλα της τελευταίας γραμμής είναι όλα αριστερά
- Τα φύλλα βρίσκονται όλα στην τελευταία και ενδεχομένως στην προτελευταία γραμμή
- Οι εσωτερικοί κόμβοι είναι όλοι δυαδικοί, εκτός από το δεξιότερο της προτελευταίας γραμμής, ο οποίος μπορεί να μην έχει δεξιό παιδί.

Για την αρίθμηση των κόμβων χρησιμοποιούμε τους παρακάτω κανόνες: Κάθε κόμβος έχει τον πατέρα του στη θέση $\lfloor i/2 \rfloor$, το αριστερό παιδί του κόμβου i, είναι ο κόμβος $2i$ και το δεξιό παιδί του κόμβου i είναι το $2i+1$. Επίσης σε ένα δυαδικό δένδρο με m κόμβους και ύψος n ισχύει

$$\lfloor \log_2 m \rfloor \leq n \leq m - 1$$



Σχήμα 3.1: Σχεδόν πλήρες δυαδικό δένδρο

.

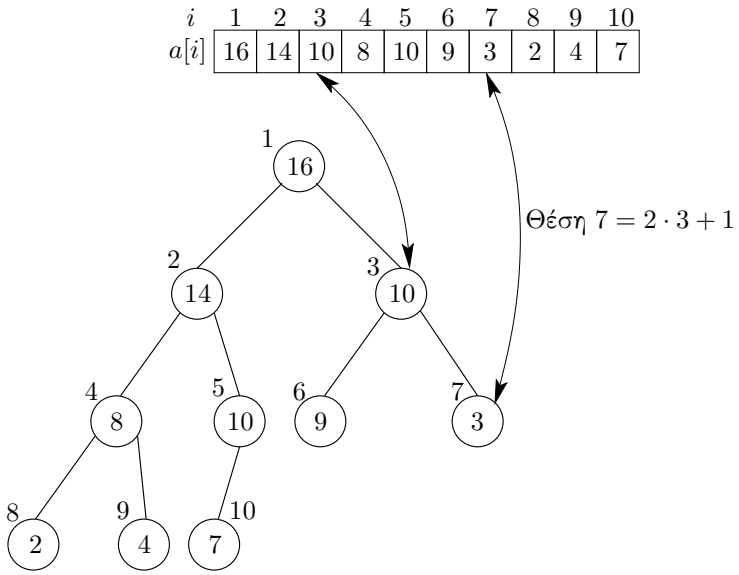
3.2 Η δομή σωρού

Ας θεωρήσουμε το εξής σενάριο: Πελάτες παρουσιάζονται στο ταμείο μιας τράπεζας με ένα νούμερο σε ένα χαρτί αντιπροσωπεύοντας τον αριθμό προτεραιότητας του καθενός. Όσο μεγαλύτερος είναι ο αριθμός ενός πελάτη τόσο πιο γρήγορα πρέπει αυτός ο πελάτης να εξυπηρετηθεί. Επιπρόσθετα η τράπεζα έχει ένα μόνο ταμείο ανοικτό. Σε αυτό το σενάριο ο υπάλληλος της τράπεζας πρέπει να μπορεί να εκτελέσει γρήγορα τις παρακάτω ενέργειες:

- Αναζήτηση του μέγιστου αριθμού στην ουρά
- Διαγραφή αυτού του στοιχείου από την ουρά
- Εισαγωγή ενός νέου στοιχείου στην ουρά

Μια πιθανή λύση θα ήταν η εισαγωγή των αριθμών της ουράς σε ένα πίνακα και έπειτα η ταξινόμηση κατά αύξουσα σειρά των στοιχείων της ουράς. Με τον τρόπο αυτό γίνεται σε σταθερό χρόνο η διαγραφή καθώς και η αναζήτηση του μεγίστου, αλλά η εισαγωγή απαιτεί γραμμικό χρόνο αφού πρέπει να διατρέξουμε όλο το μήκος της ουράς. Μια άλλη λύση θα ήταν απλά η διαχείριση των πελατών σαν μια ουρά, αλλά τότε ενώ η εισαγωγή θα γινόταν πολύ γρήγορα, η διαγραφή και η αναζήτηση στην ουρά θα χρειάζονταν επίσης γραμμικό χρόνο.

Μια κομψή μέθοδος είναι να διαχειρισθούμε μια δομή μερικής διάταξης με τη βοήθεια ενός δένδρου (δομή σωρού (heap)). Η ουρά n στοιχείων παριστάνεται από ένα δυαδικό δένδρο το οποίο σε κάθε κόμβο περιέχει ένα στοιχείο της ουράς. Το δένδρο αυτό θα πληρεί δύο βασικές ιδιότητες:



Σχήμα 3.2: Ένας σωρός και η υλοποίησή του με πίνακα

- Η τιμή κάθε κόμβου είναι μεγαλύτερη ή ίση της τιμής των παιδιών του κόμβου
- Το δέντρο είναι σχεδόν πλήρες

Ένα παράδειγμα σωρού φαίνεται στο σχήμα 3.2.

Η αναπαράσταση με σωρό μας επιτρέπει να κάνουμε τις πράξεις της αναζήτησης του μέγιστου στοιχείου, της εισαγωγής και της διαγραφής σε χρόνο $O(\log n)$.

3.2.1 Υλοποίηση με πίνακα ενός σωρού

Για την υλοποίηση του σωρού με πίνακα ακολουθούμε τον εξής βασικό κανόνα: *Κάνουμε μία κατά πλάτος αρίθμηση των κόμβων του δέδρου και το νούμερο κάθε κόμβου του δένδρου δίνει τον δείκτη του πίνακα που περιέχει την τιμή του κόμβου.* Η διαδικασία φαίνεται στο παράδειγμα σχήματος 3.2.

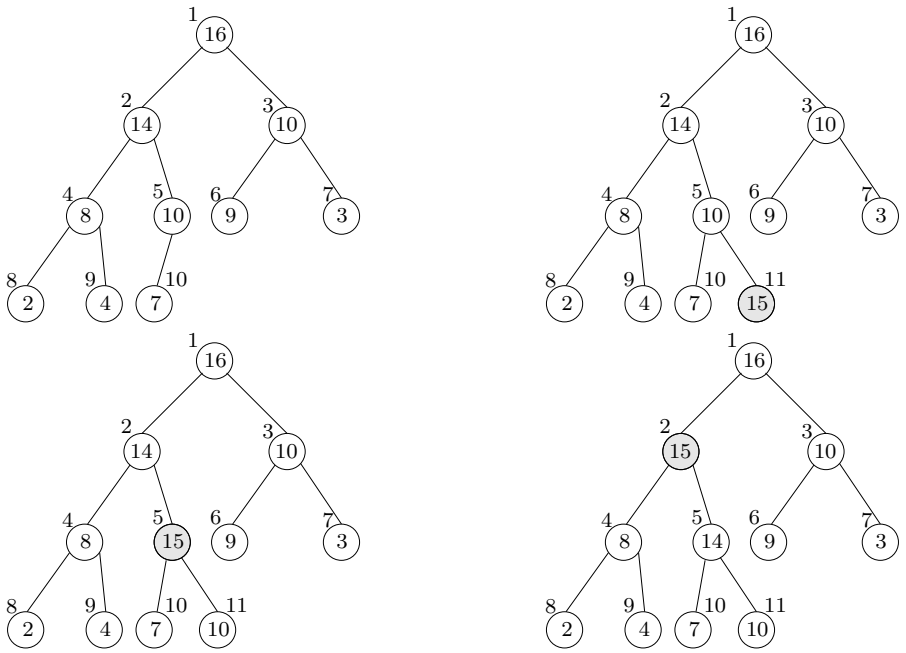
3.2.2 Εισαγωγή ενός νέου στοιχείου στο σωρό

Για την εισαγωγή ενός νέου στοιχείου στο σωρό, εισάγουμε το στοιχείο σαν τελευταίο φύλλο στο πλήρες δυαδικό δένδρο. Έπειτα συγκρίνουμε την τιμή του με αυτήν του πατέρα του και αν έχει μεγαλύτερη τιμή, τις αντιμεταθέτουμε. Η διαδικασία επαναλαμβάνεται εωσότου ικανοποιείται η συνθήκη του σωρού. Ο αλγόριθμος αυτός φαίνεται στο σχήμα 3.3

Η μέθοδος στην χείριστη περίπτωση (εισάγεται ένα στοιχείο με τιμή μεγαλύτερη από όλα τα άλλα), θα κάνει τόσα βήματα, όσο και το το ύψος του δένδρου, δηλαδή

ΣΩΡΟΣ-ΕΙΣΑΓΩΓΗ (u : στοιχείο)
1. $n = n + 1$
2. $k = n,\ a[k] = u$
3. **while** $a[k/2] < a[k]$
4. $\quad$ *swap* $(a[k],\ a[k/2])$
5. $\quad$ $k = \lfloor k/2 \rfloor$
6. **end while**

Σχήμα 3.3: Αλγόριθμος εισαγωγής στοιχείου στον σωρό



Σχήμα 3.4: Εισαγωγή ενός νέου στοιχείου στο σωρό

$O(\log n)$. Στο σχήμα 3.4 φαίνεται η εφαρμογή του παραπάνω αλγορίθμου σε έναν σωρό με 10 στοιχεία.

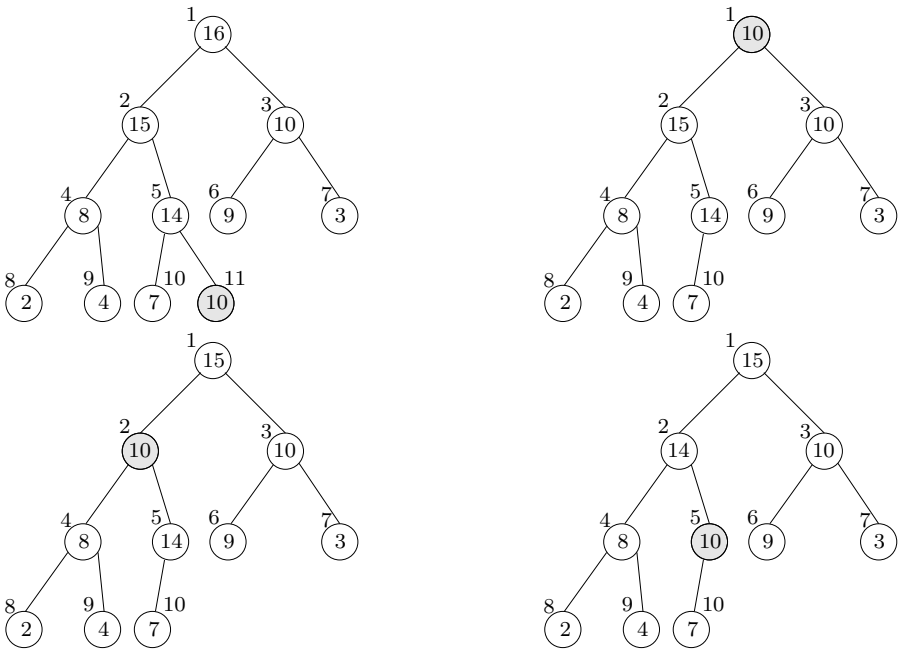
3.2.3 Διαγραφή ενός στοιχείου από το σωρό

Η διαδικασία διαγραφής του 1ου στοιχείου του σωρού αντικαθιστά τη ρίζα του δέντρου που παριστάνει τον σωρό, με το δεξιότερο φύλλο του τελευταίου επιπέδου του δένδρου. Έπειτα συγκρίνεται η τιμή της νέας ρίζας με των παιδιών της και αν είναι μικρότερη από κάποια, την αντιμεταθέτουμε με την μεγαλύτερη. Η διαδικασία επαναλαμβάνεται για το νέο στοιχείο, μέχρι να ικανοποιηθεί η συνθήκη του σωρού,

όπως φαίνεται στο σχήμα 3.5.

ΣΩΡΟΣ-ΔΙΑΓΡΑΦΗ ( )
1. $a[1] = a[n]$
2. $n = n - 1$, $i = 1$
3. **do**
4. $\quad l = 2 \cdot i$, $r = 2 \cdot i + 1$
5. $\quad$ **if** $l \leq n$ **and** $a[l] > a[i]$ **then** $max = l$
6. $\quad$ **else** $max = i$
7. $\quad$ **if** $r \leq n$ **and** $a[r] > a[max]$ **then** $max = r$
8. $\quad$ **if** $i \neq max$ **then** $swap\ (a[i], a[max])$, $i = max$
9. $\quad$ **else** break
10. **while** $(i < n)$

Σχήμα 3.5: Αλγόριθμος διαγραφής ρίζας από τον σωρό

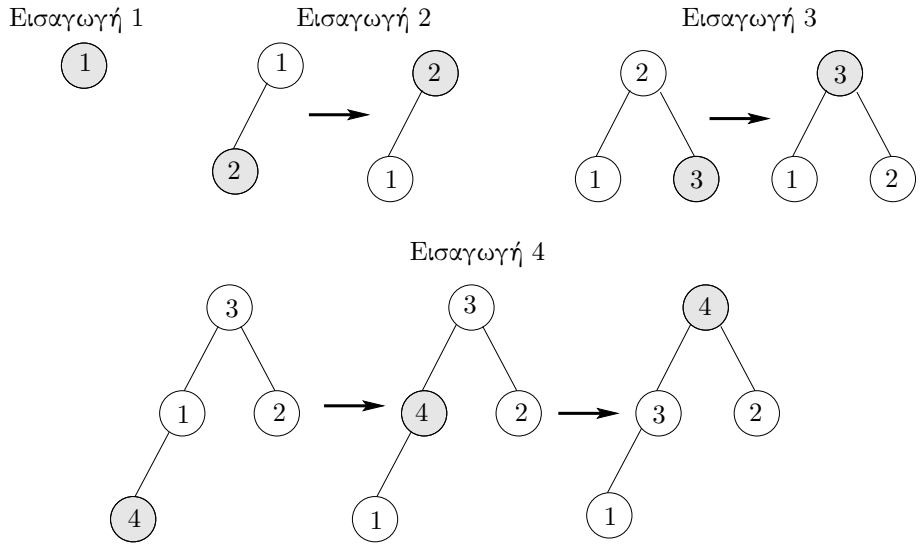


Σχήμα 3.6: Διαγραφή της ρίζας από το το σωρό

Η μέθοδος στην χείριστη περίπτωση (το στοιχείο που αντικατέστησε την ρίζα είναι το μικρότερο από όλα τα άλλα), θα κάνει τόσα βήματα, όσο και το το ύψος του δένδρου, δηλαδή $O(\log n)$. Στο σχήμα 3.6 φαίνεται η εφαρμογή του παραπάνω αλγορίθμου για την διαγραφή της ρίζας από έναν σωρό με 11 στοιχεία.

3.2.4 Κατασκευή σωρού

Σε αυτή την παράγραφο περιγράφουμε τη διαδικασία κατασκευής σωρού. Ας ξεκινήσουμε με ένα παράδειγμα για να δούμε διαισθητικά πώς δουλεύει η μέθοδος. Ας υποθέσουμε ότι έχουμε την ακολουθία στοιχείων $1, 2, 3, 4$. Η διαδικασία κατασκευής του σωρού φαίνεται στο παρακάτω σχήμα:



Σχήμα 3.7: Κατασκευή σωρού

Ο σωρός μπορεί να κατασκευαστεί χρησιμοποιώντας τον παρακάτω αλγόριθμο, ο οποίος καλεί τον αλγόριθμο εισαγωγής ενός στοιχείου. Υποθέτουμε ότι τα στοιχεία έρχονται με μια σειρά και δεν είναι γνωστά εκ των προτέρων.

ΚΑΤΑΣΚΕΥΗ-ΣΩΡΟΥ (πίνακας a με n στοιχεία)
1. **for** $i = 1$ **to** n **do**
2. ΣΩΡΟΣ-ΕΙΣΑΓΩΓΗ ($a[i]$)
3. **end for**

Σχήμα 3.8: Αλγόριθμος κατασκευής σωρού

Στην χείριστη περίπτωση (αύξουσα ακολουθία), κάθε στοιχείο που εισάγεται θα ξεκινήσει από φύλλο του σωρού και θα καταλήξει στην ρίζα, άρα αν εισάγεται στο βήμα i, τότε θα χρειαστεί $O(\log i)$ βήματα. Άρα με τη διαδοχική εισαγωγή n στοιχείων, έχουμε ότι ο χρόνος του αλγορίθμου φράσσεται ως:

$$O(\log 1) + O(\log 2) + \ldots + O(\log n) = O(n \log n)$$

Ωστόσο μπορούμε να επιτύχουμε καλύτερο χρόνο για την κατασκευή του σωρού αν γνωρίζουμε από την αρχή όλην την ακολουθία εισόδου και την έχουμε αποθηκευμένη σε έναν πίνακα. Τότε, θα θεωρήσουμε ότι ο αρχικός πίνακας εισόδου a είναι η αναπαράσταση ενός αταξινόμητου σωρού, και εμείς επιβάλλουμε την ιδιότητα του σωρού στον πίνακα αυτόν, ξεκινώντας από τους χαμηλότερους εσωτερικούς κόμβους και πηγαίνοντας προς την ρίζα. Για τον σκοπό αυτό, χρησιμοποιούμε την ακόλουθη συνάρτηση διατήρησης της ιδότητας του σωρού σε ένα υποδένδρο.

ΔΙΑΤΗΡΗΣΗ-ΣΩΡΟΥ (πίνακας a με n στοιχεία, θέση i)

```
1.  l = 2 · i
2.  r = 2 · i + 1
3.  if l ≤ n and a[l] > a[i]
4.        max = l
5.  else
6.        max = i
7.  if r ≤ n and a[r] > a[max]
8.        max = r
9.  if i ≠ max then
10.       swap (a[i], a[max])
11.       ΔΙΑΤΗΡΗΣΗ-ΣΩΡΟΥ (a, max)
```

Σχήμα 3.9: Αλγόριθμος διατήρησης ιδιότητας σωρού. Ο αλγόριθμος συγκρίνει το τρέχον στοιχείο με τα παιδιά του ανυψώνοντας το μεγαλύτερο, εωσότου και τα δύο παιδιά του έχουν μικρότερη τιμή.

ΚΑΤΑΣΚΕΥΗ-ΣΩΡΟΥ (πίνακας a με n στοιχεία)

```
1.  for i = ⌊n/2⌋ to 1 do
2.        ΔΙΑΤΗΡΗΣΗ-ΣΩΡΟΥ (a, i)
3.  end for
```

Σχήμα 3.10: Αλγόριθμος κατασκευής σωρού σε γραμμικό χρόνο

Θεωρούμε τώρα ότι στην αταξινόμητη ακολουθία όλα τα στοιχεία που είναι φύλλα του σωρού είναι υπο-σωροί που πληρούν την ιδιότητα του σωρού. Έπειτα για κάθε επόμενο εσωτερικό κόμβο (από κάτω προς τα πάνω, δηλαδή από $\lfloor n/2 \rfloor$ εώς 1) επιβάλλουμε την συνέπεια με την ιδιότητα σωρού χρησιμοποιώντας την ρουτίνα ΔΙΑΤΗΡΗΣΗ-ΣΩΡΟΥ όπως φαίνεται στο σχήμα 3.10.

Στο σχήμα 3.11 φαίνεται η εκτέλεση του αλγορίθμου κατασκευής του σωρού, όταν ο πίνακας εισόδου είναι ο $[5, 6, 1, 4, 8, 7, 3, 2, 9]$,

Η πολυπλοκότητα μίας κλήσης της συνάρτησης ΔΙΑΤΗΡΗΣΗ-ΣΩΡΟΥ είναι $O(h)$ όταν καλείται από έναν κόμβο που έχει ύψος h, άρα η συνολική πολυπλοκότητα της ΚΑΤΑΣΚΕΥΗ-ΣΩΡΟΥ φράσσεται από

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\Big(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\Big)$$

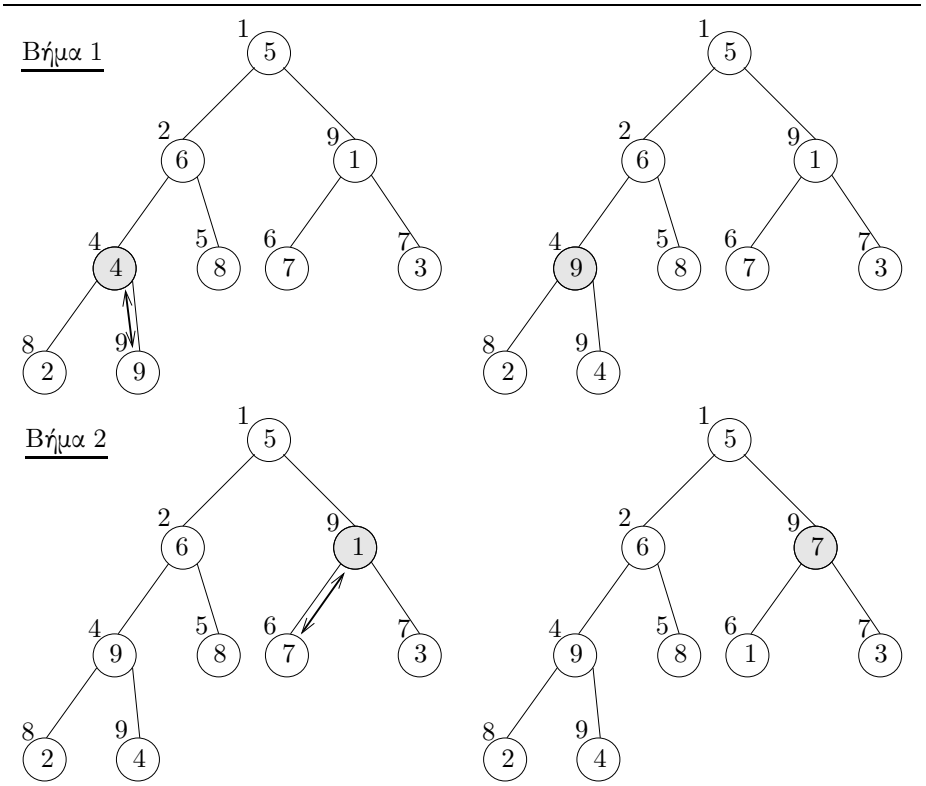
Επειδή τώρα

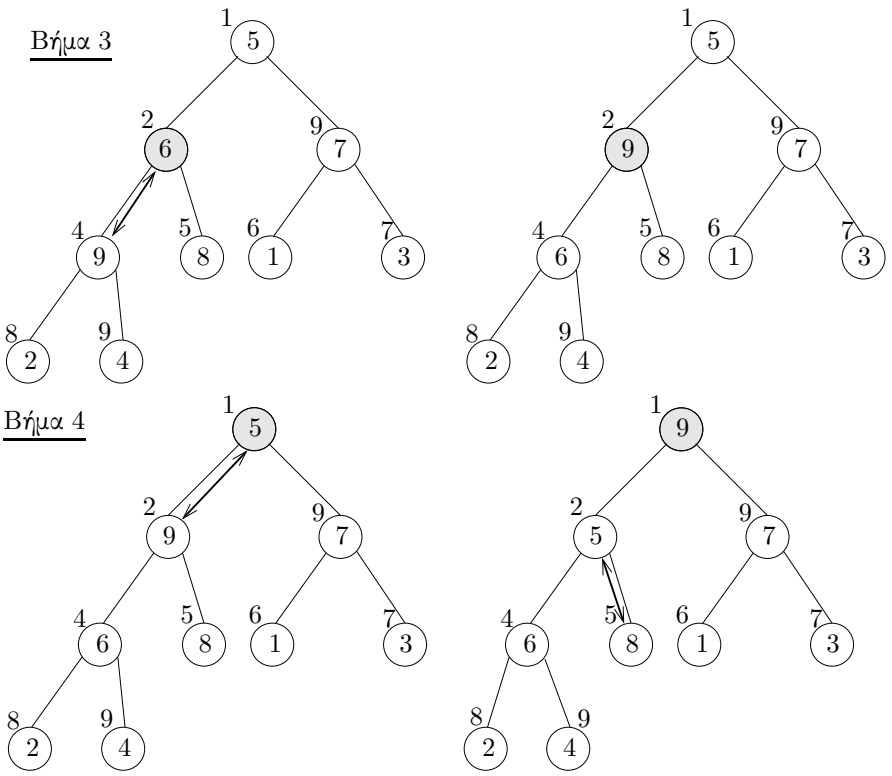
$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

για $x = 1/2$ έχουμε

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$$

Άρα η πολυπλοκότητα της κατασκευής του σωρού φράσσεται από $O(n \cdot 2) = O(n)$.





Σχήμα 3.11: Κατασκευή σωρού σε γραμμικό χρόνο

3.3 Ταξινόμηση με τη βοήθεια σωρού

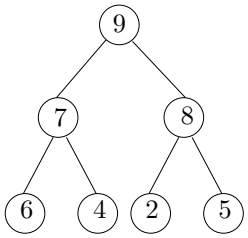
Έχοντας το παραπάνω υπόψη μας, μπορούμε να κατασκευάσουμε έναν αποδοτικό αλγόριθμο ταξινόμησης. Αρχικά, κατασκευάζουμε τον σωρό των προς ταξινόμηση στοιχείων και έπειτα κάνουμε διαδοχικές διαγραφές της ρίζας του δένδρου. Η σειρά με την οποία εξάγονται τα στοιχεία από τον σωρό είναι και η ταξινομημένη ακολουθία. Αυτός ο αλγόριθμος ταξινόμησης φαίνεται στο σχήμα 3.12

HeapSort (πίνακας a με n στοιχεία)
1. Κατασκευη-Σωρου (a)
2. **for** $i = 1$ **to** n
3. $\quad a[i] =$ Μεγιστος-Σωρου( )
4. $\quad$ Σωρος-Διαγραφη( )

Σχήμα 3.12: Αλγόριθμος ταξινόμησης με σωρό. Η συνάρτηση Μεγιστος-Σωρου( ) επιστρέφει την ρίζα του σωρού.

Παράδειγμα: Να ταξινομηθούν με τη βοήθεια σωρού τα παρακάτω στοιχεία $6, 9, 2, 7, 4, 5, 8$.

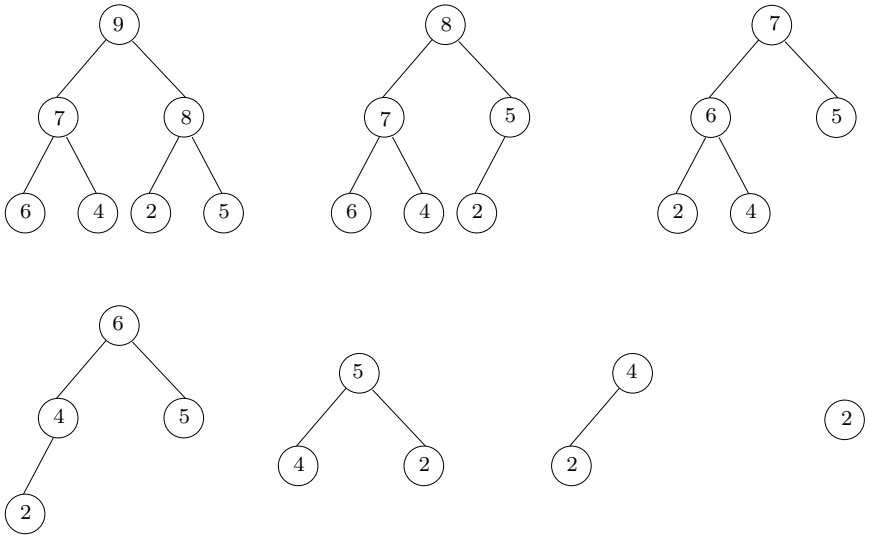
Λύση: Κατασκευάζουμε το σωρό (βλ. σχήμα 3.13).



Σχήμα 3.13: Ο σωρός

Έπειτα αφαιρούμε διαδοχικά το μεγαλύτερο στοιχείο με την βοήθεια του αλγορίθμου διαγραφής διατηρώντας το σωρό. Στο σχήμα 3.14 φαίνεται η μορφή του σωρού μετά από κάθε διαγραφή.

Το πρώτο βήμα του αλγορίθμου απαιτεί χρόνο $O(n)$ ενώ το δεύτερο απαιτεί χρόνο $O(n \log n)$, αφού το i-ο βήμα θέλει το πολύ $O(\log i)$ βήματα. Συνολικά επομένως ο αλγόριθμος ταξινόμησης με σωρό έχει πολυπλοκότητα $O(n \log n)$.



Σχήμα 3.14: Ο σωρός μετά τις διαδοχικές διαγραφές του μεγίστου κατά την εκτέλεση του αλγόριθμου ταξινόμησης.

Κεφάλαιο 4

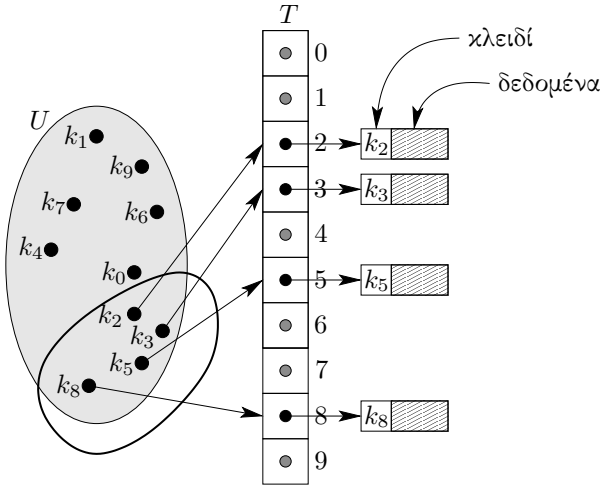
Κατακερματισμός (Hashing)

4.1 Εισαγωγή

Σε πολλές εφαρμογές απαιτείται η χρήση ενός δυναμικού συνόλου δεδομένων όπου οι ενέργειες *INSERT (εισαγωγή), SEARCH (αναζήτηση)* και *DELETE (διαγραφή)* να εκτελούνται πολύ αποδοτικά. Για παράδειγμα, ένας μεταγλωττιστής μιας γλώσσας προγραμματισμού πρέπει να διατηρεί έναν πίνακα συμβόλων, ο οποίος είναι σαν ένα λεξικό των μεταβλητών, σταθερών και άλλων στοιχείων του προγράμματος και στον οποίο οι ενέργειες *εισαγωγής, αναζήτησης* και *διαγραφής* εκτελούνται πολύ συχνά. Σε τέτοιες περιπτώσεις η λύση λέγεται ***κατακερματισμός***. Ένας ***πίνακας κατακερματισμού*** *(Hash table)* είναι μία αποδοτική δομή δεδομένων για υλοποίηση λεξικών. Παρόλο που η αναζήτηση ενός στοιχείου στη χείριστη περίπτωση απαιτεί χρόνο $\Theta(n)$, στην πράξη ο κατακερματισμός αποδίδει εξαιρετικά. Κάνοντας κάποιες εύλογες υποθέσεις ο αναμενόμενος χρόνος εισαγωγής, αναζήτησης και διαγραφής ενός στοιχείου από έναν πίνακα κατακερματισμού είναι $O(1)$.

4.2 Πίνακες Άμεσης Διευθυνσιοδότησης (Direct Address Tables)

Θεωρείστε ότι μία εφαρμογή χρειάζεται ένα δυναμικό σύνολο στο οποίο κάθε στοιχείο έχει ένα κλειδί από το σύμπαν $U = \{0, 1, \ldots, m-1\}$ και ότι το m δεν είναι μεγάλο. Επίσης θεωρείστε ότι το κλειδί είναι μοναδικό, δηλαδή ανά δύο τα στοιχεία δεν μπορούν να έχουν το ίδιο κλειδί. Στο Σχήμα 4.1 βλέπουμε πώς δουλεύει η *άμεση διευθυνσιοδότηση* σε τέτοιες απλές περιπτώσεις.

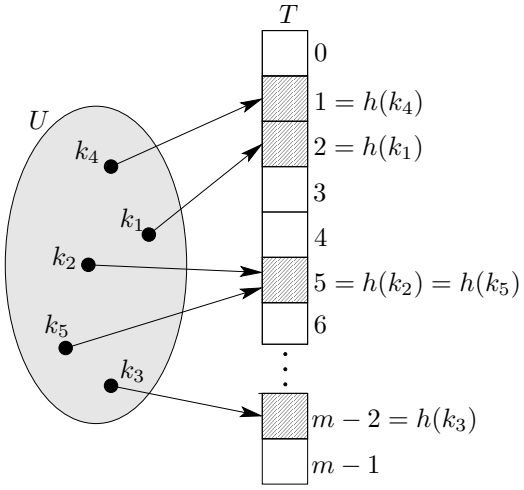


Σχήμα 4.1: Υλοποίηση ενός δυναμικού συνόλου με έναν πίνακα άμεσης διευθυνσιοδότησης Τ. Κάθε κλειδί του σύμπαντος $U = \{0, \ldots, 9\}$ αντιστοιχεί σε ένα στοιχείο του πίνακα. Το σύνολο $K = \{2, 3, 5, 8\}$ των πραγματικών κλειδιών καθορίζει τις θέσεις του πίνακα που περιέχουν δείκτες σε στοιχεία. Οι υπόλοιπες θέσεις περιέχουν την τιμή NIL.

Το δυναμικό σύνολο αναπαριστάται με έναν πίνακα, ο οποίος λέγεται *πίνακας άμεσης διευθυνσιοδότησης*, και συμβολίζεται με $T[0, \ldots, m-1]$. Η κάθε θέση του πίνακα αντιστοιχεί σε ένα κλειδί του σύμπαντος U. Αν κάποιο στοιχείο δεν υπάρχει τότε $T[k] = \text{NIL}$, αλλιώς η k θέση του πίνακα δείχνει στο στοιχείο του συνόλου με κλειδί k.

Οι ενέργειες λεξικού υλοποιούνται πολύ απλά και τρέχουν σε χρόνο $O(1)$.

```
DIRECT-ADDRESS-SEARCH(T,k)
   return T[k]

DIRECT-ADDRESS-INSERT(T,x)
   T[key[x]]←x

DIRECT-ADDRESS-DELETE(T,x)
   T[key[x]]←NIL
```



Σχήμα 4.2: Χρήση της συνάρτησης κατακερματισμού h για να απεικονιστούν τα κλειδιά στις θέσεις του πίνακα T. Τα κλειδιά k_2 και k_5 απεικονίζονται στην ίδια θέση, οπότε συγκρούονται.

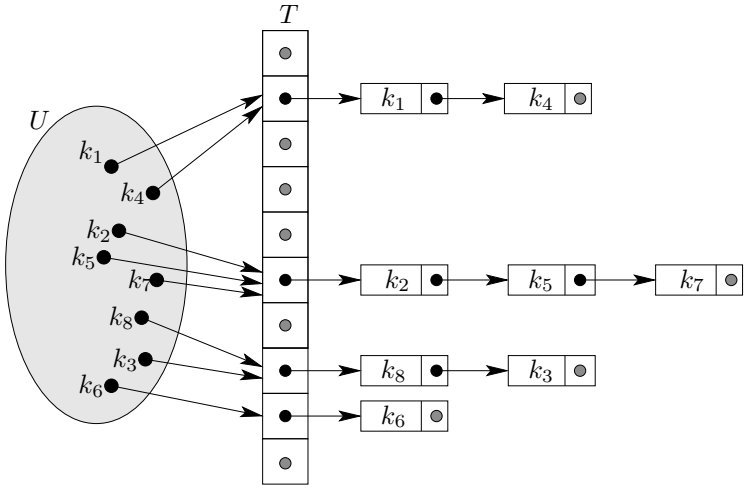
4.3 Πίνακες Κατακερματισμού (Hash Tables)

Το πρόβλημα με την άμεση διευθυνσιοδότηση είναι προφανές. Αν το σύμπαν U είναι μεγάλο τότε η αποθήκευση ενός πίνακα T μεγέθους $|U|$ μπορεί να είναι αδύνατη στη πράξη. Επιπλέον το σύνολο K των κλειδιών που αποθηκεύονται στον πίνακα μπορεί να είναι πολύ μικρό σε σχέση με το U επομένως να έχουμε μεγάλη σπατάλη χώρου.

Ένας πίνακας κατακερματισμού, αντίθετα, απαιτεί αποθηκευτικό χώρο μόλις $\Theta(|K|)$ ενώ διατηρεί τους χρόνους αναζήτησης, εισαγωγής και διαγραφής σε $O(1)$. Αυτό αφορά βέβαια μέσους χρόνους, ενώ στην άμεση διευθυνσιοδότηση είχαμε χρόνους στη χείριστη περίπτωση.

Στον κατακερματισμό ένα στοιχείο με κλειδί k αποθηκεύεται τώρα στη θέση $h(k)$ του πίνακα T. Η συνάρτηση $h : U \to \{0, 1, \ldots, m-1\}$ λέγεται *συνάρτηση κατακερματισμού (hash function)* και όπως βλέπουμε στο Σχήμα 4.2 απεικονίζει τα $|K|$ κλειδιά σε πίνακα m θέσεων.

Υπάρχει βέβαια και πάλι πρόβλημα όταν δύο διαφορετικά κλειδιά απεικονίζονται στην ίδια θέση του πίνακα. Αυτή η κατάσταση λέγεται *σύγκρουση* (collision). Η συνάρτηση h θα πρέπει να είναι έτσι φτιαγμένη ώστε να παράγει όσο το δυνατόν ''τυχαίες'' φαινομενικά αντιστοιχίσεις για να ελαχιστοποιούνται οι συγκρούσεις. Ωστόσο, όταν $|K| > m$, θα υπάρχουν σίγουρα συγκρούσεις. Για την επίλυσή



Σχήμα 4.3: Επίλυση συγκρούσεων με λίστες. Κάθε θέση του πίνακα κατακερματισμού $T[j]$ περιέχει μία συνδεδεμένη λίστα με όλα τα στοιχεία των οποίων τα κλειδιά απεικονίζονται στη θέση j.

τους χρησιμοποιούνται αποδοτικοί αλγόριθμοι όπως αυτός της *επίλυσης των συγκρούσεων με λίστες (collision resolution by chaining)*, που θα δούμε αμέσως μετά, αλλά και της *ανοιχτής διευθυνσιοδότησης (open addressing)* (Ενότητα 4.5).

4.3.1 Επίλυση Συγκρούσεων με Λίστες (Collision Resolution by Chaining)

Στο Σχήμα 4.3 βλέπουμε ότι όλα τα στοιχεία που το κλειδί τους απεικονίζεται στην ίδια θέση του πίνακα, τοποθετούνται σε αυτήν τη θέση του πίνακα, σε μία συνδεδεμένη λίστα.

Οι ενέργειες λεξικού τώρα έχουν ως εξής.

```
CHAINED-HASH-SEARCH(T,k)
    Ψάξε για ένα στοιχείο με κλειδί k στη λίστα T[h(k)]

CHAINED-HASH-INSERT(T,x)
    Βάλε το x στην κεφαλή της λίστας T[h(key[x])]
```

```
CHAINED-HASH-DELETE(T,x)
    Διάγραψε το x από τη λίστα T[h(key[x])]
```

4.3.2 Ανάλυση Κατακερματισμού με Λίστες

Καταρχήν, εύκολα βλέπουμε ότι η ΕΙΣΑΓΩΓΗ απαιτεί χρόνο $O(1)$ (εκτός αν πρέπει πρώτα να ελέγχουμε ότι δεν υπάρχει το κλειδί οπότε πρέπει να γίνεται κάθε φορά και μία αναζήτηση). Η ΔΙΑΓΡΑΦΗ, αν η υλοποίηση γίνει με διπλά συνδεδεμένες λίστες, απαιτεί πάλι χρόνο $O(1)$. Προσέξτε ότι το στοιχείο x που πρέπει να διαγραφεί, και όχι το κλειδί του, δίνεται σαν παράμετρος, οπότε δεν χρειάζεται να το αναζητήσουμε στη λίστα.

Στην ΑΝΑΖΗΤΗΣΗ παρατηρούμε ότι στη χείριστη περίπτωση, όταν υπάρχουν n κλειδιά και m θέσεις και όλα τα κλειδιά έχουν απεικονιστεί στην ίδια θέση, ο χρόνος είναι $\Theta(n)$. Ωστόσο, στην μέση περίπτωση η απόδοση του κατακερματισμού εξαρτάται από το πόσο καλά κατανέμει η h τα κλειδιά στον πίνακα T κατά μέσο όρο.

Αν υποθέσουμε ότι οποιοδήποτε στοιχείο έχει την ίδια πιθανότητα να πάει σε οποιαδήποτε από τις m θέσεις, ανεξάρτητα από το που πήγε κάθε άλλο στοιχείο τότε λέμε ότι έχουμε *Απλό Ομοιόμορφο Κατακερματισμό (Simple Uniform Hashing)*. Σε αυτή την περίπτωση θα υπολογίσουμε το μέσο χρόνο αναζήτησης.

Πρώτα ορίζουμε ως *παράγοντα φόρτου (load factor)* a για τον T το $a = n/m$. Επίσης για $j = 0, 1, \ldots, m-1$, έστω το μήκος της λίστας $T[j]$ να το συμβολίζουμε με n_j έτσι ώστε

$$n = n_0 + n_1 + \ldots + n_{m-1}, \tag{4.1}$$

και η μέση τιμή του n_j είναι $E[n_j] = a = n/m$.

Υποθέτουμε ότι ο χρόνος υπολογισμού της h είναι $O(1)$. Έτσι η αναζήτηση του k εξαρτατάται γραμμικά από το $n_{h(k)}$. Θα υπολογίσουμε τον αναμενόμενο αριθμό στοιχείων που ελέγχει ο αλγόριθμος της αναζήτησης ψάχνοντας ένα στοιχείο. Θεωρούμε δύο περιπτώσεις, μία για την περίπτωση ανεπιτυχούς αναζήτησης ενός στοιχείου με κλειδί k και μία για επιτυχή αναζήτηση.

Θεώρημα 2. Σε έναν πίνακα κατακερματισμού στον οποίο οι συγκρούσεις επιλύονται με λίστες, μία ανεπιτυχής αναζήτηση απαιτεί αναμενόμενο χρόνο $\Theta(1 + a)$, με την υπόθεση του απλού ομοιόμορφου κατακερματισμού.

Απόδειξη. Ο αναμενόμενος χρόνος της ανεπιτυχούς αναζήτησης ισούται με αυτόν της αναζήτησης στο τέλος της λίστας $T[h(k)]$, η οποία έχει αναμενόμενο μήκος $E[n_{h(k)}] = a$, αφού υποθέσαμε απλό ομοιόμορφο κατακερματισμό. Άρα το αναμενόμενο πλήθος στοιχείων που εξετάζονται σε μία ανεπιτυχή αναζήτηση είναι a και ο συνολικός χρόνος που απαιτείται, συμπεριλαμβανομένου και του χρόνου υπολογισμού του $h(k)$, είναι $\Theta(1 + a)$. □

Στην περίπτωση της επιτυχούς αναζήτησης η κατάσταση είναι λίγο διαφορετική, αφού κάθε λίστα δεν είναι ισοπίθανο να αναζητηθεί. Μάλιστα, η πιθανότητα μιας λίστας να αναζητηθεί εξαρτάται από το πλήθος των στοιχείων που περιέχει. Παρ΄ όλ΄ αυτά ο αναμενόμενος χρόνος αναζήτησης παραμένει $\Theta(1 + a)$.

Θεώρημα 3. Σε έναν πίνακα κατακερματισμού στον οποίο οι συγκρούσεις επιλύονται με λίστες, μία επιτυχής αναζήτηση απαιτεί αναμενόμενο χρόνο $\Theta(1+a)$, με την υπόθεση του απλού ομοιόμορφου κατακερματισμού.

Απόδειξη. Υποθέτουμε ότι είναι ισοπίθανο οποιοδήποτε από τα n στοιχεία του πίνακα να είναι το υπό αναζήτηση στοιχείο. Το πλήθος των στοιχείων που εξετάζονται σε μία επιτυχή αναζήτηση του στοιχείου x είναι κατά ένα περισσότερα του πλήθους των στοιχείων που βρίσκονται πριν από το x στη λίστα του. Όλα τα στοιχεία πριν το x εισήχθησαν στη λίστα μετά το x, γιατί τα νέα στοιχεία τοποθετούνται στην αρχή της λίστας. Για να βρούμε τον μέσο αριθμό των εξετασμένων στοιχείων παίρνουμε το μέσο όρο πάνω στα n στοιχεία x του πίνακα, του 1 συν το αναμενόμενο πλήθος των στοιχείων που εισήχθησαν στη λίστα του x μετά την εισαγωγή του ιδίου. Έστω ότι με x_i συμβολίζουμε το i-οστό στοιχείο που εισήχθει στον πίνακα, για $i = 1, 2, \ldots, n$ και $k_i = key[x_i]$. Για τα κλειδιά k_i και k_j, ορίζουμε την τυχαία μεταβλητή δείκτη (indicator) $X_{ij} = I\{h(k_i) = h(k_j)\}$. Υπό την προϋπόθεση του απλού ομοιόμορφου κατακερματισμού έχουμε $Pr\{h(k_i) = h(k_j)\} = 1/m$ επομένως και $E[X_{ij}] = 1/m$. Επομένως το αναμενόμενο πλήθος στοιχείων υπό εξέταση σε μία επιτυχή αναζήτηση είναι

$$\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = & \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E\left[X_{ij}\right]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{a}{2}-\frac{a}{2n}.
\end{aligned}$$

Άρα ο συνολικός χρόνος, συμπεριλαμβανομένου του χρόνου υπολογισμού της h είναι $\Theta(2+a/2-a/2n) = \Theta(1+a)$. □

Αυτό σημαίνει ότι αν το μέγεθος του πίνακα είναι τουλάχιστον ανάλογο του πλήθους των στοιχείων έχουμε $n = O(m)$ οπότε και $a = n/m = O(m)/m = O(1)$. Έτσι και η αναζήτηση θα απαιτεί σταθερό χρόνο κατά μέσο όρο.

4.4 Συναρτήσεις Κατακερματισμού (Hash Functions)

Η βασική μας προϋπόθεση για να οδηγηθούμε στα προηγούμενα θετικά συμπεράσματα ήταν ότι ο κατακερματισμός μας είναι απλός και ομοιόμορφος. Αυτό όμως εξαρτάται από την συνάρτηση h. Εδώ θα δούμε διάφορους τρόπους σχεδιασμού καλών, με την έννοια ότι ικανοποιούν (προσεγγιστικά) την προϋπόθεση μας, συναρτήσεων κατακερματισμού.

Απλός Ομοιόμορφος Κατακερματισμός συνεπάγεται ότι κάθε κλειδί έχει ίση πιθανότητα να αντιστοιχιθεί μέσω της συνάρτησης κατακερματισμού h σε οποιαδήποτε από τις m θέσεις του πίνακα. Δυστυχώς όμως κάτι τέτοιο είναι αδύνατο να επιβεβαιωθεί, αφού σπάνια γνωρίζουμε την κατανομή πιθανότητας ή τις εξαρτήσεις των κλειδιών.

Υπάρχουν περιπτώσεις όπου ξέρουμε την κατανομή των κλειδιών όπως στην περίπτωση που γνωρίζουμε ότι τα κλειδιά είναι τυχαίοι πραγματικοί αριθμοί k, ανεξάρτητοι και ομοιόμορφα κατανεμημένοι στην περιοχή του $0 \leq k < 1$. Τότε η συνάρτηση $h(k) = \lfloor km \rfloor$ ικανοποιεί τη συνθήκη του απλού ομοιόμορφου κατακερματισμού.

Στην πραγματικότητα, έχουμε αρκετές φορές κάποια μερική γνώση για την κατανομή των κλειδιών, και προσπαθούμε να κάνουμε το πεδίο τιμών της συνάρτησης κατακερματισμού να είναι ανεξάρτητοι από τα οποιαδήποτε μοτίβα μπορεί να υπάρχουν στα κλειδιά.

Στη συνέχεια θα δούμε δύο γενικές μεθόδους δημιουργίας καλών συναρτήσεων κατακερματισμού. Πρώτα, όμως, πρέπει να πούμε ότι οι περισσότερες συναρτήσεις θεωρούν το σύμπαν των κλειδιών ότι είναι το σύνολο $\{N\} = \{0, 1, 2 \ldots\}$ των φυσικών. Αν τα κλειδιά μας δεν είναι φυσικοί μπορούμε να τα μετατρέψουμε σε φυσικούς. Θεωρήστε για παράδειγμα μία γραμματοσειρά η οποία μπορεί να μετατραπεί σε έναν φυσικό εκφρασμένο στην κατάλληλη βάση. Έτσι η λέξη $\pi\tau$ μπορεί να μετατραπεί σε ένα ζεύγος φυσικών, το (112,116), αφού στον κώδικα ASCII έχουμε $p = 112$ και $t = 116$. Επομένως σε ακέραιο με βάση το 128 το $\pi\tau$ γίνεται $(112 \cdot 128) + 116 = 14452$. Στη συνέχεια θεωρούμε ότι τα κλειδιά είναι φυσικοί αριθμοί.

4.4.1 Η Μέθοδος της Διαίρεσης (Division Method)

Η συνάρτηση κατακερματισμού στη *μέθοδο της διαίρεσης* είναι

$$h(k) = k \mod m.$$

Για παράδειγμα αν ο πίνακας έχει μέγεθος $m = 4$ και το κλειδί είναι $k = 100$ τότε $h(k) = 4$. Ένα καλό m συνήθως είναι ένας πρώτος όχι πολύ κοντά σε ακριβή δύναμη του 2, γιατί αν ήταν δύναμη του 2 τότε η $h(k)$ θα ήταν τα p χαμηλότερης τάξης ψηφία του k.

Θεωρήστε το εξής παράδειγμα. Αν είχαμε $n = 2000$ στοιχεία και δεν μας πείραζε αν κατά μέσο όρο ελέγχαμε 3 στοιχεία στην αναζήτηση τότε μία καλή

επιλογή θα ήταν το $m = 701$ αφού είναι πρώτος, κοντά στο $2000/3$ και μακριά από δύναμη του 2.

Είμαστε έτοιμοι για να δούμε μία συνάρτηση κατακερματισμού, της μεθόδου της διαίρεσης, για γραμματοσειρές η οποία μας δίνει μία καλή κατανομή στο $[0, \ldots, N-1]$, εύκολα. Θεωρήστε τη συνάρτηση

$$h(x) = (x_1 B^{l-1} + x_2 B^{l-2} + \ldots + x_l) \mod N,$$

όπου N είναι η διάσταση του πίνακα κατακερματισμού, l είναι το μήκος της γραμματοσειράς x και B είναι μία δύναμη του 2 (128, 256 κλπ). Προσέξτε ότι αν π.χ. $B = N = 256$ τότε το $h(x) = x_1$ και έτσι δεν πετυχαίνουμε καλή κατανομή. Την υλοποίηση της συνάρτησης τη βλέπουμε παρακάτω

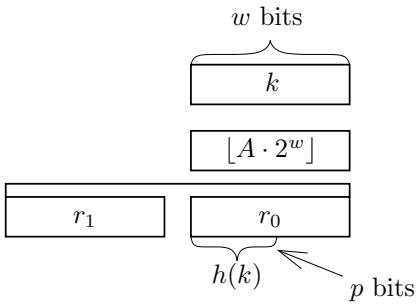
$h\ (x : string,\ l : integer)$
1. $r = x_l$, $p = 1$
2. **for** $i = l - 1$ **to** 1 **do**
3. $\quad p = p * B$
4. $\quad r = r + x_i * p$
5. **end for**
6. $r = r \mod N$
7. return r

4.4.2 Η Πολλαπλασιαστική Μέθοδος (Multiplication Method)

Η πολλαπλασιαστική μέθοδος λειτουργεί σε δύο βήματα. Πρώτα πολλαπλασιάζουμε το κλειδί k με μία σταθερά A όπου $0 < A < 1$ και παίρνουμε το δεκαδικό μέρος του kA. Έπειτα πολλαπλασιάζουμε αυτήν τη τιμή με m και παίρνουμε τη στρογγυλοποίηση προς τα κάτω του αποτελέσματος. Δηλαδή

$$h(k) = \lfloor m(kA \mod 1) \rfloor.$$

Ένα πλεονέκτημα αυτής της μεθόδου είναι ότι η τιμή του m δεν είναι τόσο σημαντική. Συνήθως τη διαλέγουμε να είναι μία δύναμη του 2, $m = 2^p$, για $p \in \mathbb{N}$, γιατί έτσι μπορούμε να την υλοποιήσουμε εύκολα στους υπολογιστές ως εξής. Έστω ότι το μέγεθος λέξης της μηχανής είναι w bits και ότι το k χωράει σε μία λέξη. Περιορίζουμε το A να είναι ένα κλάσμα της μορφής $s/2^w$, όπου το s είναι ένας ακέραιος στο διάστημα $0 < s < 2^w$. Όπως φαίνεται στο Σχήμα 4.4 πρώτα πολλαπλασιάζουμε το k με τον w-μπιτο ακέραιο $s = A \cdot 2^w$. Το αποτέλεσμα είναι ένα $2w$-μπιτος αριθμός $r_1 2^w + r_0$, όπου r_1 είναι η υψηλής τάξης λέξη του γινομένου και r_0 είναι η χαμηλής τάξης λέξη του γινομένου. Η επιθυμητή p-μπιτη τιμή της συνάρτησης κατακερματισμού αποτελείται από τα p πιο σημαντικά βιτς του r_0. Αν και η μέθοδος δουλεύει με κάθε A δουλεύει καλύτερα με κάποιες συγκεκριμένες τιμές, όπως όσες είναι κοντά στο $(\sqrt{5} - 1)/2$.



Σχήμα 4.4: Η πολλαπλασιαστική μέθοδος του κατακερματισμού. Η w-μπιτη αναπαράσταση του κλειδιού πολλαπλασιάζεται με την w-μπιτη τιμή $s = A \cdot 2^w$. Τα p υψηλότερης τάξης βιτς του χαμηλότερου w-μπιτου μισού του γινομένου είναι η τιμή της $h(k)$.

4.5 Ανοιχτή Διευθυνσιοδότηση (Open Addressing)

Η *ανοιχτή διευθυνσιοδότηση* χρησιμοποιείται σαν εναλλακτική μέθοδος για τη διαχείριση των συγκρούσεων στον πίνακα κατακερματισμού. Με αυτήν όλα τα στοιχεία αποθηκεύονται μέσα στον πίνακα και αποφεύγεται εντελώς η χρήση δεικτών. Έτσι υπάρχει όφελος, από την αξιοποίηση της απελευθερωμένης μνήμης για αύξηση των θέσεων του πίνακα, τόσο από πλευράς πιθανότητας συγκρούσεων όσο και από πλευράς μέσου χρόνου αναζήτησης. Ωστόσο ο πίνακας κατακερματισμού μπορεί κάποτε να γεμίσει και να μη δέχεται άλλες εισαγωγές. Ο παράγοντας φόρτου a, δηλαδή, στην ανοιχτή διευθυνσιοδότηση δεν μπορεί να υπερβεί το 1.

Για να εισάγουμε ένα στοιχείο με την ανοιχτή διευθυνσιοδότηση, ελέγχουμε διαδοχικά, ή *διερευνούμε (probe)*, τον πίνακα κατακερματισμού μέχρι να βρούμε μία άδεια θέση για να βάλουμε το κλειδί. Η σειρά με την οποία εξετάζονται οι θέσεις του πίνακα εξαρτάται από το κλειδί που εισάγουμε. Για τον καθορισμό της σειράς διερεύνησης, η συνάρτηση κατακερματισμού διευρύνεται ώστε να περιλαμβάνει και τον αριθμό διερεύνησης (ξεκινώντας από το 0) σαν δεύτερη είσοδο. Έτσι η συνάρτηση κατακερματισμού γίνεται

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Για κάθε k, η *σειρά διερεύνησης (probe sequence)* $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ πρέπει να είναι μία μετάθεση της $\langle 0, 1, \ldots, m-1\rangle$, έτσι ώστε κάθε θέση του πίνακα να εξετάζεται προοδευτικά για την τοποθέτηση ενός νέου κλειδιού. Παρακάτω

δίνεται ο κώδικας για την ΕΙΣΑΓΩΓΗ. Θεωρούμε ότι κάθε στοιχείο αποτελείται μόνο από το κλειδί του.

HASHINSERT (T, k)
1. $i = 0$
2. **repeat**
3. $\quad j = h(k, i)$
4. $\quad$ **if** $T[j] = NIL$ **then**
5. $\quad\quad T[j] = k$
6. $\quad\quad$ return j
7. $\quad$ **else**
8. $\quad\quad i = i + 1$
9. **until** $i = m$
10. error "hash table overflow"

Ο αλγόριθμος για την αναζήτηση ενός κλειδιού k διερευνά τις ίδιες θέσεις που εξέτασε ο αλγόριθμος εισαγωγής κατά την εισαγωγή του κλειδιού. Γι΄ αυτό και ο αλγόριθμος τερματίζει όταν βρει μία θέση με τιμή NIL, αφού το k θα είχε εισαχθεί εκεί και όχι σε κάποια επόμενη θέση της σειράς διερεύνησής του. Κάτι τέτοιο όμως προϋποθέτει ότι δεν γίνονται διαγραφές από τον πίνακα.

HASHSEARCH (T, k)
1. $i = 0$
2. **repeat**
3. $\quad j = h(k, i)$
4. $\quad$ **if** $T[j] = k$ **then**
5. $\quad\quad$ return j
6. $\quad$ **else**
7. $\quad\quad i = i + 1$
8. **until** $i = m$ **or** $T[j] = NIL$
9. return NIL

Η διαγραφή από τον πίνακα κατακερματισμού όταν χρησιμοποιείται η ανοιχτή διευθυνσιοδότηση μπορεί να υλοποιηθεί σχετικά εύκολα, απλά προσθέτοντας μία τιμή DELETED που θα παίρνουν τα κελιά μετά από μία διαγραφή και τροποποιώντας κατάλληλα την ΕΙΣΑΓΩΓΗ ώστε να τα αναγνωρίζει και αυτά ως κενά. Όμως έτσι οι χρόνοι αναζήτησης δεν εξαρτώνται πλέον από το a, οπότε αν πρέπει να διαγράφουμε στοιχεία είναι προτιμότερο να χρησιμοποιούμε τη διαχείριση με λίστες.

Στην ανάλυση που θα κάνουμε θεωρούμε ότι έχουμε *Ομοιόμορφο Κατακερματισμό (Uniform Hashing)*, μία γενίκευση του Απλού Ομοιόμορφου Κατακερματισμού. Δηλαδή υποθέτουμε ότι για κάθε κλειδί είναι ισοπίθανο να έχει οποιαδήποτε από της $m!$ μεταθέσεις της $\langle 0, 1, \ldots, m-1 \rangle$ σαν σειρά διερεύνησης. Ο

Ομοιόμορφος Κατακερματισμός αν και δύσκολο να επιτευχθεί μπορεί εύκολα να προσσεγγιστεί.

Τρεις είναι οι συνήθεις τεχνικές για τον υπολογισμό της σειράς διερεύνησης: *η γραμμική και τετραγωνική διερεύνηση και ο διπλός κατακερματισμός*. Και οι τρεις εγγυώνται ότι κάθε κλειδί έχει μια μετάθεση της $\langle 0, 1, \ldots, m-1 \rangle$ για σειρά διερεύνησης, όμως καμία δεν ικανοποιεί τον ομοιόμορφο κατακερματισμό αφού μπορούν μόνο να παράγουν m^2 από τις $m!$ μεταθέσεις. Ο διπλός κατακερματισμός μπορεί να παράγει τις περισσότερες σειρές διερεύνησης επομένως έχει και τα καλύτερα αποτελέσματα.

4.5.1 Γραμμική Διερεύνηση (Linear Probing)

Στη *γραμμική διερεύνηση* παίρνουμε μία απλή συνάρτηση κατακερματισμού $h' : U \to \{0, 1, \ldots, m-1\}$, που την λέμε *βοηθητική*, και παράγουμε τη σειρά διερεύνησης για $i = 0, 1, \ldots, m-1$ ως εξής

$$h(k, i) = (h'(k) + i) \mod m.$$

Η γραμμική διερεύνηση, αν και απλή, έχει ένα πρόβλημα που λέγεται *βασική ομαδοποίηση (primary clustering)*. Ο μέσος χρόνος αναζήτησης αυξάνεται σταδιακά, αφού σχηματίζονται όλο και μεγαλύτερες ομάδες από συνεχόμενες κατειλημμένες θέσεις. Αυτό συμβαίνει γιατί η πιθανότητα να καταληφθεί μία άδεια θέση, η οποία έχει i γεμάτες θέσεις πριν από αυτή, είναι $(i+1)/m$.

4.5.2 Τετραγωνική Διερεύνηση (Quadratic Probing)

Η *τετραγωνική διερεύνηση* χρησιμοποιεί μια συνάρτηση της μορφής

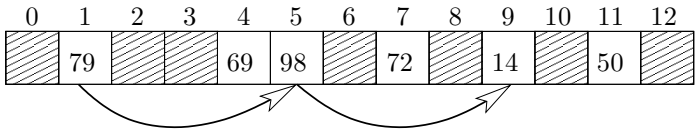
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

όπου h' είναι η βοηθητική συνάρτηση , $c_1, c_2 \neq 0$ είναι βοηθητικές σταθερές και $i = 0, 1, \ldots, m-1$. Αυτή η μέθοδος δουλεύει αρκετά καλύτερα από τη γραμμική διερεύνηση, αλλά για να κάνει πλήρη χρήση του πίνακα πρέπει να χρησιμοποιούνται συγκεκριμένες τιμές για τα c_1, c_2 και m. Επίσης αν δύο κλειδιά έχουν την ίδια αρχική θέση, δηλαδή $h(k_1, 0) = h(k_2, 0)$, τότε ολόκληρες οι σειρές διερεύνησής τους είναι ίδιες. Αυτό οδηγεί σε μία ηπιότερη μορφή ομαδοποίησης, τη *δευτερεύουσα ομαδοποίηση (secondary clustering)*, γιατί αφού η αρχική θέση καθορίζει όλη τη σειρά, μόνο m διαφορετικές σειρές διερεύνησης χρησιμοποιούνται.

4.5.3 Διπλός Κατακερματισμός (Double Hashing)

Ο *διπλός κατακερματισμός* είναι μία από τις καλύτερες μεθόδους για την ανοιχτή διευθυνσιοδότηση γιατί οι μεταθέσεις που παράγει έχουν πολλά από τα χαρακτηριστικά των τυχαία επιλεγμένων μεταθέσεων. Η συνάρτηση που χρησιμοποιεί είναι της μορφής

$$h'(k, i) = (h_1(k) + i h_2(k)) \mod m,$$



Σχήμα 4.5: Εισαγωγή με διπλό κατακερματισμό. Ο πίνακας κατακερματισμού έχει μέγεθος 13 και $h_1(k) = k \mod 13, h_2(k) = 1 + (k \mod 11)$. Εφόσον $14 \equiv 1(\mod 13)$ και $14 \equiv 3(\mu\text{οδ } 11)$, το κλειδί 14 εισάγεται στην άδεια θέση 9, αφού πρώτα βρεθούν κατειλημμένες οι θέσεις 1 και 5.

όπου οι h_1 και h_2 είναι βοηθητικές συναρτήσεις κατακερματισμού. Στο Σχήμα 4.5 βλέπουμε ένα παράδειγμα εισαγωγής με διπλό κατακερματισμό.

Το $h_2(k)$ πρέπει να είναι πρώτο σε σχέση με το μέγεθος m του πίνακα για να μπορεί να διερευνηθεί όλος ο πίνακας. Αυτή η συνθήκη μπορεί να εξασφαλιστεί αν θέσουμε το m σε μία δύναμη του 2 και να σχεδιάσουμε την h_2 έτσι ώστε πάντα να παράγει έναν περιττό αριθμό. Ένας άλλος τρόπος είναι να θέσουμε το m πρώτο και να σχεδιάσουμε την h_2 έτσι ώστε να παράγει πάντα έναν φυσικό μικρότερο του m. Για παράδειγμα, έστω m πρώτος και έστω

$$h_1(k) = k \mod m,$$

$$h_2(k) = 1 + (k \mod m'),$$

όπου το m' επιλέγεται να είναι λίγο μικρότερο του m (π.χ. $m - 1$). Οπότε αν $k = 123456, m = 701$ και $m' = 700$, τότε έχουμε $h_1(k) = 80$ και $h_2(k) = 257$. Άρα η πρώτη θέση προς διερεύνηση είναι η 80 και μετά κάθε 257η θέση (modulo m) εξετάζεται μέχρι να βρεθεί άδεια θέση ή να εξεταστούν όλες οι θέσεις.

Ο διπλός κατακερματισμός είναι καλύτερος από την γραμμική και την τετραγωνική διερεύνηση αφού χρησιμοποιεί $\Theta(m^2)$ σειρές διερεύνησης (μία από κάθε ζεύγος $(h_1(k), h_2(k))$) έναντι των $\Theta(m)$ των άλλων δύο και προσεγγίζει το 'ιδανικό' σχήμα του ομοιόμορφου κατακερματισμού.

4.5.4 Ανάλυση του Κατακερματισμού Ανοιχτής Διευθυνσιοδότησης

Η ανάλυση του κατακερματισμού ανοιχτής διευθυνσιοδότησης, όπως και αυτή με τις λίστες, εκφράζεται ως προς τον παράγοντα φόρτου $a = n/m$ καθώς το n και το m πάνε στο άπειρο. Εδώ βέβαια ισχύει $a \leq 1$ αφού κάθε θέση μπορεί να περιέχει το πολύ ένα στοιχείο. Επίσης βασιζόμαστε στην υπόθεση του ομοιόμορφου κατακερματισμού που ορίσαμε προηγουμένως.

Θα αναλύσουμε το αναμενόμενο πλήθος διερευνήσεων στον κατακερματισμό με ανοιχτή διευθυνσιοδότηση ξεκινώντας από την περίπτωση της ανεπιτυχούς αναζήτησης.

Θεώρημα 4. Δεδομένου ενός πίνακα κατακερματισμού ανοιχτής διευθυνσιοδότησης με παράγοντα φόρτου $a = n/m < 1$, το αναμενόμενο πλήθος διερευνήσεων σε μία ανεπιτυχή αναζήτηση είναι το πολύ $1/(1-a)$, υπό την προϋπόθεση του ομοιόμορφου κατακερματισμού.

Απόδειξη. Σε μία ανεπιτυχή αναζήτηση κάθε διερεύνηση, εκτός της τελευταίας, προσπελαύνει έναν κατειλημμένο κόμβο ο οποίος δεν περιέχει το επιθυμητό κλειδί, ενώ η θέση που διερευνάται είναι κενή. Έστω X η τυχαία μεταβλητή που δείχνει το πλήθος των διερευνήσεων που γίνονται σε μία ανεπιτυχή αναζήτηση και έστω A_i, για $i = 1, 2, \ldots$, το γεγονός ότι υπάρχει i-οστη διερεύνηση η οποία πέφτει πάνω σε κατειλημμένη θέση. Τότε το γεγονός $\{X \geq i\}$ είναι η τομή των γεγονότων $A_1 \cap A_2 \cdots \cap A_{i-1}$. Θα φράξουμε την $Pr\{X \geq i\}$ φράζοντας την $Pr\{A_1 \cap A_2 \cdots \cap A_{i-1}\}$. Ξέρουμε ότι

$$Pr\{A_1 \cap A_2 \cdots \cap A_{i-1}\} =$$

$$= Pr\{A_1\} \cdot Pr\{A_2|A_1\} \cdot Pr\{A_3|A_1 \cap A_2\} \cdots Pr\{A_{i-1}|A_1 \cap A_2 \cdots \cap A_{i-2}\}.$$

Εφόσον υπάρχουν n στοιχεία και m θέσεις ισχύει $Pr\{A_1\} = n/m$. Για $j > 1$, η πιθανότητα να υπάρχει j-οστή διερεύνηση σε κατειλημμένη μάλιστα θέση, δεδομένου ότι οι πρώτες $j-1$ διερευνήσεις έγιναν και αυτές σε κατειλημμένες θέσεις είναι $(n-j+1)/(m-j+1)$. Αυτό ισχύει γιατί βρίσκουμε ένα από τα εναπομείναντα $(n-(j-1))$ στοιχεία σε μία από της $(m-(j-1))$ ανεξερεύνητες θέσεις, και με την υπόθεση του ομοιόμορφου κατακερματισμού, η πιθανότητα θα είναι ο λόγος αυτών των ποσοτήτων. Παρατηρώντας ότι $n < m$ συνεπάγεται $(n-j)/(m-j) < n/m$ για κάθε j τέτοιο ώστε $0 \leq j < m$, έχουμε για κάθε i με $1 \leq i \leq m$,

$$Pr\{X \geq 1\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = a^{i-1}.$$

Οπότε

$$E[X] = \sum_{i=1}^{\infty} Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} a^{i-1} = \sum_{i=0}^{\infty} a^i = \frac{1}{1-a}.$$

□

Η ερμηνεία του φράγματος $1 + a + a^2 + a^3 + \cdots$ είναι η εξής. Αρχικά, πάντα γίνεται μία διερεύνηση. Έπειτα με πιθανότητα περίπου a η πρώτη διερεύνηση βρίσκει μία κατειλημμένη θέση και απαιτείται δεύτερη διερεύνηση. Με πιθανότητα περίπου a^2 οι πρώτες δύο θέσεις είναι κατειλημμένες οπότε απαιτείται και τρίτη διερεύνηση κ.ο.κ.

Αν το a είναι σταθερά, το παραπάνω θεώρημα μας λέει ότι μία ανεπιτυχής αναζήτηση τρέχει σε χρόνο $O(1)$. Για παράδειγμα αν ο πίνακας είναι μισογεμάτος, τότε ο μέσος χρόνος διερευνήσεων σε μία ανεπιτυχή αναζήτηση είναι το πολύ

$1/(1-0.5) = 2$, ενώ αν είναι κατά 90% το μέσο πλήθος διερευνήσεων είναι το πολύ $1/(1-0.9) = 10$.

Από το θεώρημα 4 προκύπτει εύκολα η απόδοση της ΕΙΣΑΓΩΓΗΣ.

Πόρισμα 1. Η εισαγωγή ενός στοιχείου σε ένα πίνακα κατακερματισμού ανοιχτής διευθυνσιοδότησης με παράγοντα φόρτου a απαιτεί το πολύ $1/(1-a)$ διερευνήσεις, υποθέτοντας ομοιόμορφο κατακερματισμό.

Απόδειξη. Ένα στοιχείο εισάγεται μόνο αν υπάρχει χώρος στον πίνακα, άρα $a < 1$. Η εισαγωγή ενός στοιχείου προϋποθέτει μία ανεπιτυχή αναζήτηση ακολουθούμενη από την τοποθέτηση του στοιχείου στην πρώτη άδεια θέση που θα βρεθεί. Έτσι το αναμενόμενο πλήθος διερευνήσεων είναι το πολύ $1/(1-a)$.□

Ο υπολογισμός του αναμενόμενου πλήθους διερευνήσεων σε μία επιτυχημένη αναζήτηση έχει ως εξής.

Θεώρημα 5. Δεδομένου ενός πίνακα κατακερματισμού ανοιχτής διευθυνσιοδότησης με παράγοντα φόρτου $a < 1$, το αναμενόμενο πλήθος διερευνήσεων σε μία επιτυχημένη αναζήτηση είναι το πολύ

$$\frac{1}{a} ln \frac{1}{1-a},$$

υποθέτοντας ομοιόμορφο κατακερματισμό και ότι κάθε κλειδί στον πίνακα έχει ίδια πιθανότητα να αναζητηθεί.

Απόδειξη. Η αναζήτηση ενός κλειδιού k ακολουθεί την ίδια σειρά διερεύνησης με αυτήν που ακολουθήθηκε όταν το στοιχείο με κλειδί το k εισήχθη. Από το προηγούμενο πόρισμα έχουμε ότι αν το k είναι το $(i+1)$-οστό κλειδί που εισήχθη στον πίνακα, τότε το αναμενόμενο πλήθος διερευνήσεων που γίνεται στην αναζήτηση του k είναι το πολύ $1/(1-i/m) = m/(m-i)$. Παίρνοντας το μέσο όρο των n κλειδιών του πίνακα κατακερματισμού παίρνουμε τον μέσο αριθμό διερευνήσεων μιας επιτυχημένης αναζήτησης:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{a}(H_m - H_{m-n}),$$

όπου το $H_i = \sum_{j=1}^{i} 1/j$ είναι ο i-οστός αρμονικός αριθμός. Επομένως έχουμε

$$\frac{1}{a}(H_m - H_{m-n}) = \frac{1}{a} \sum_{k=m-n+1}^{m} \frac{1}{k} \leq \frac{1}{a} \int_{m-n}^{m} \frac{1}{x} dx = \frac{1}{a} ln \frac{m}{m-n} = \frac{1}{a} ln \frac{1}{1-a}.$$□

Κεφάλαιο 5

Γράφοι

5.1 Γενικά

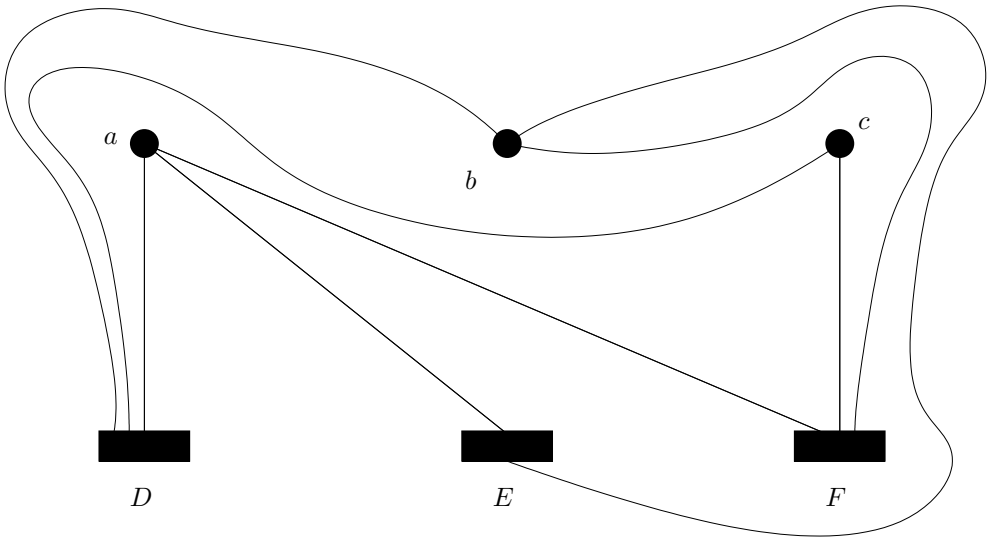
Η έννοια του γράφου είναι μια συνδυαστική δομή, η οποία μας επιτρέπει να αναπαραστήσουμε πολυάριθμες καταστάσεις, που συναντάμε σε εφαρμογές, οι οποίες εμπλέκουν τα διακριτά μαθηματικά και απαιτούν μια λύση πληροφορικής. Ένας γράφος είναι μια μαθηματική οντότητα και αποτελεί μια δομή δεδομένων ισχυρή για την πληροφορική. Οι γράφοι αποτελούν ένα ισχυρό εργαλείο για την αναπαράσταση προβλημάτων της καθημερινής ζωής, που θα ήταν δύσκολα προσεγγίσιμα με τις κλασικές μεθόδους των μαθηματικών. Οι γράφοι χρησιμοποιούνται ιδιαίτερα για την περιγραφή της δομής ενός πολύπλοκου συνόλου και για τις σχέσεις και τις εξαρτήσεις ανάμεσα στα στοιχεία του συνόλου.

Μερικοί από τους τομείς στους οποίους μπορούμε να χρησιμοποιήσουμε τους γράφους για την μοντελοποίηση και επίλυση προβλημάτων είναι:

- Ηλεκτρικά κυκλώματα
- Δίκτυα μεταφορών
  - Σιδηροδρομικά
  - Οδικά
  - Αεροπορικά
- Δίκτυα Υπολογιστών
- Χρονοπρογραμματισμός εργασιών
- Διάγραμμα διαδοχής εργασιών στη διαχείριση προγραμμάτων
- Αναπαράσταση συνδέσεων και δυνατότητες διάσχισης.

Παράδειγμα 1 (Τρεις κατοικίες - Τρία εργοστάσια)**.** Έχουμε τρεις κατοικίες a, b, c και θέλουμε να συνδέσουμε κάθε κατοικία για τροφοδοσία με ένα εργοστάσιο υδρεύσεως D, ένα εργοστάσιο παραγωγής φυσικού αερίου E, και ένα εργοστάσιο

παραγωγής ηλεκτρισμού F. Το πρόβλημα που έχουμε να αντιμετωπίσουμε είναι το εξής: *Μπορούμε να τοποθετήσουμε (πάνω στο επίπεδο) τις* 3 *κατοικίες, τα* 3 *εργοστάσια και τις τροφοδοσίες, έτσι ώστε δυο τροφοδοσίες να μην τέμνονται εκτός των άκρων τους;*

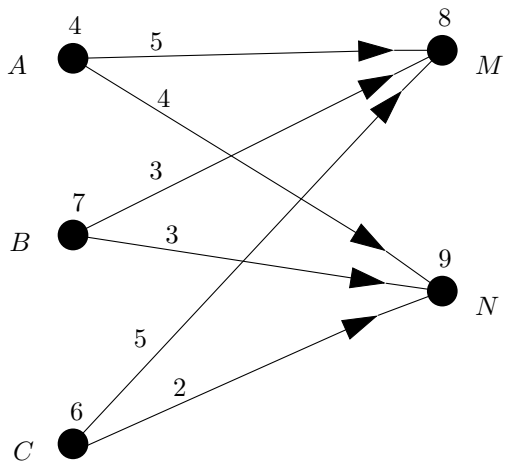


Σχήμα 5.1: Τροφοδοσία 3 κατοικιών με 3 εργοστάσια στο επίπεδο

Όπως βλέπουμε στο σχήμα 5.1 μπορούμε να τοποθετήσουμε 8 τροφοδοσίες (συνδέσεις) χωρίς κανένα πρόβλημα. Η ένατη όμως σύνδεση τέμνει πάντα μία από τις πρώτες οκτώ. Έτσι είναι αδύνατο να συνδεθούν και οι τρεις κατοικίες με όλα τα εργοστάσια. Αναγκαστικά λοιπόν μια κατοικία θα συνδεθεί μόνο με δύο εργοστάσια.

Παράδειγμα 2 (Δίκτυα μεταφορών)**.** Τρία εργοστάσια A, B, C είναι συνδεδεμένα με οδούς (σιδηροδρομικές, αεροπορικές) με δύο κέντρα αποθήκευσης M και N. Οι διαθέσιμες ποσότητες των εργοστασίων είναι: $A = 4$, $B = 7$, $C = 6$, ενώ οι αναμενόμενες ποσότητες των αποθηκών είναι: $M = 8$ και $N = 9$.

Θεωρούμε ότι το κόστος μεταφοράς ανάμεσα σε κάθε αφετηρία και κάθε προορισμό είναι γνωστό. Το πρόβλημα που θέλουμε να επιλύσουμε είναι το εξής: *Να καθοριστεί ποιο εργοστάσιο θα πρέπει να στείλει σε ποιο κέντρο, έτσι ώστε το συνολικό κόστος μεταφοράς να είναι ελάχιστο.*



Σχήμα 5.2: Δίκτυο Μεταφορών

Παράδειγμα 3 (Δίκτυο Τηλεπικοινωνιών)**.** Έχουμε ένα δίκτυο υπολογιστών που ανταλλάσσουν δεδομένα μεταξύ τους, όπως φαίνεται στο σχήμα 5.3.

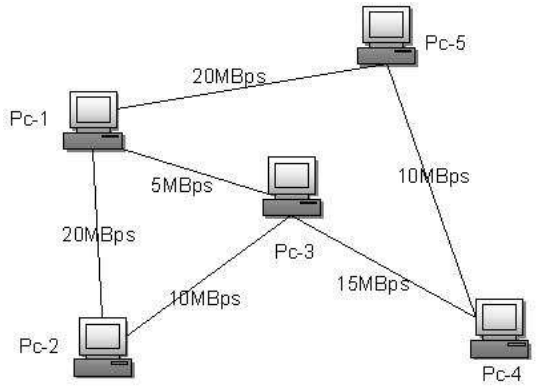
Θα μοντελοποιήσουμε το πρόβλημα με ένα γράφο (βλ. σχήμα 5.2). Αντιστοιχούμε κάθε εργοστάσιο και κάθε κέντρο αποθήκευσης σε ένα κόμβο του γράφου. Τα βάρη των κόμβων του γράφου αναπαριστούν το απόθεμα κάθε εργοστασίου και την ζήτηση κάθε αποθήκης. Ομοίως τα βάρη των ακμών αναπαριστούν το κόστος ανά μεταφερόμενη μονάδα αγαθού στον αντίστοιχο δρόμο.

Οι συνδέσεις του δικτύου έχουν κάποια χωρητικότητα. Το πρόβλημα που έχουμε να αντιμετωπίσουμε είναι το εξής: *Πώς θα δρομολογήσουμε τις παρακάτω ποσότητες δεδομένων:*

- $7MBps$ *από το* $pc-1$ *στο* $pc-4$
- $6MBps$ *από το* $pc-3$ *στο* $pc-5$
- $10MBps$ *από το* $pc-2$ *στο* $pc-3$
- $5MBps$ *από το* $pc-4$ *στο* $pc-1$

έτσι ώστε να μεταφερθούν όλα τα δεδομένα χωρίς να παραβιαστεί η χωρητικότητα των ακμών;

Το πρόβλημα μπορεί να μοντελοποιηθεί με την βοήθεια ενός γράφου. Κάθε υπολογιστής του δικτύου αντιστοιχεί σε ένα κόμβο του γράφου και η ποσότητα



Σχήμα 5.3: Δίκτυο Τηλεπικοινωνιών

που θέλει να στείλει ο υπολογιστής αυτός είναι το βάρος του αντίστοιχου κόμβου. Κάθε ακμή έχει ως βάρος την χωρητικότητά της.

5.2 Ορισμοί

Τυπικά γράφος είναι μια δομή που αποτελείται από ένα σύνολο κόμβων (nodes) που συνδέονται μεταξύ τους με ένα σύνολο ακμών (edges). Γενικά, ένας γράφος συμβολίζεται ως $G = (V, E)$, όπου V και E είναι τα σύνολα των κόμβων και των κορυφών αντίστοιχα.

Ορισμός 1. Ένας γράφος $G = (V, E)$ δίνεται από ένα *σύνολο κόμβων* V και από ένα υποσύνολο E του καρτεσιανού γινομένου $V \times V$, ονομαζόμενο *σύνολο ακμών* του G.

Το *πλήθος των κορυφών* του γράφου συμβολίζεται με $n = |V|$ και το *πλήθος των ακμών* του γράφου συμβολίζεται με $m = |E|$.

Ορισμός 2. Το πλήθος n των κορυφών του γράφου ονομάζεται *τάξη* του γράφου.

Ορισμός 3. *Πυκνότητα* ενός γράφου ορίζεται το πηλίκο $\rho(G) = \frac{|E|}{n^2}$

Ορισμός 4. Μία ακμή με ταυτόσημα τερματικά σημεία ονομάζεται *βρόγχος*. Δύο ή περισσότερες ακμές που ενώνουν το ίδιο ζεύγος κορυφών ονομάζονται *παράλληλες*. Κάθε γράφος χωρίς βρόγχους ή παράλληλες ακμές ονομάζεται *απλός γράφος*.

5.2.1 Κατευθυνόμενοι γράφοι

Ορισμός 5. *Κατευθυνόμενος γράφος* (directed graph) ονομάζεται ένας γράφος $G = (X, U)$, που αποτελείται από ένα μη κενό σύνολο κορυφών X και ένα σύνολο U από *διατεταγμένα* (κατευθυνόμενα) ζεύγη κορυφών, που ονομάζονται κατευθυνόμενες πλευρές (ή τόξα).

Τα τόξα σε ένα κατευθυνόμενο γράφο πρέπει να διασχίζονται σε μονόδρομη κατεύθυνση. Για παράδειγμα στο τόξο (i, j) μπορούμε να μεταβούμε από τον κόμβο i προς τον κόμβο j, αλλά όχι από τον κόμβο j προς τον κόμβο i.

Ένα τόξο (i, j) σε έναν κατευθυνόμενο γράφο έχει δύο άκρα i και j. Η κορυφή i λέγεται *ουρά* (tail) ενώ η κορυφή j λέγεται *κεφαλή* (head). Λέμε ότι το τόξο (i, j) *ξεκινάει* από την κορυφή i και *κατευθύνεται* στην κορυφή j. Το τόξο (i, j) είναι *εξερχόμενο* της κορυφής i και *εισερχόμενο* της κορυφής j.

Ορισμός 6. *Έσω βαθμός* (indegree) d_i^- ενός κόμβου i ονομάζεται το πλήθος εισερχόμενων τόξων του κόμβου. *Έξω βαθμός* (outdegree) d_i^+ ενός κόμβου i ονομάζεται το πλήθος εξερχόμενων τόξων του κόμβου. *Βαθμός* (degree) d_i μιας κορυφής i ενός κατευθυνόμενου γράφου λέγεται το άθροισμα του έσω και του έξω βαθμού της κορυφής i ($d_i = d_i^+ + d_i^-$).

5.2.2 Μη κατευθυνόμενοι γράφοι

Ορισμός 7. *Μη Κατευθυνόμενος γράφος* (undirected graph) ονομάζεται ένας γράφος $G = (V, E)$, που αποτελείται από ένα μη κενό σύνολο κορυφών V και ένα σύνολο E από μη κατευθυνόμενα ζεύγη κορυφών, που ονομάζονται ακμές.

Οι πλευρές στους μη κατευθυνόμενους γράφους, μπορούν να διασχίζονται και προς τις δυο κατευθύνσεις.

Αξίζει να σημειώσουμε πως οι αλγόριθμοι διάσχισης (που θα δούμε στη συνέχεια) για τους κατευθυνόμενους γράφους, ισχύουν ειδικότερα για τους μη κατευθυνόμενους γράφους. Αρκεί να κατασκευάσουμε από το μη κατευθυνόμενο γράφο, ένα νέο κατευθυνόμενο αντικαθιστώντας κάθε πλευρά του μη κατευθυνόμενου γράφου με δύο παράλληλες κατευθυνόμενες ακμές.

Ορισμός 8. *Βαθμός* (degree) d_i μιας κορυφής i ενός μη κατευθυνόμενου γράφου λέγεται το πλήθος των ακμών που προσπίπτουν στην κορυφή.

Ορισμός 9. *Γράφος με βάρη ή αντισταθμισμένος γράφος* (κατευθυνόμενος ή μη) ονομάζεται ο γράφος, που έχει σε κάθε ακμή του e μια χαρακτηριστική τιμή, συμβολιζόμενη $w(e)$ και ονομαζόμενη *βάρος*. Το *βάρος* του γράφου είναι το άθροισμα των βαρών των ακμών του. Πολλές φορές τα βάρη είναι στις κορυφές του γράφου μόνο ή στις κορυφές και στις πλευρές.

5.2.3 Κύκλοι και μονοπάτια

Σε μια κατευθυνόμενη πλευρά $\alpha = (x, y)$ η ουρά x λέγεται αρχή της α και η κεφαλή y άκρο της α. Θα συμβολίζουμε : $\text{αρ}(\alpha) = x$ και $\text{ακ}(\alpha) = y$, αντίστοιχα.

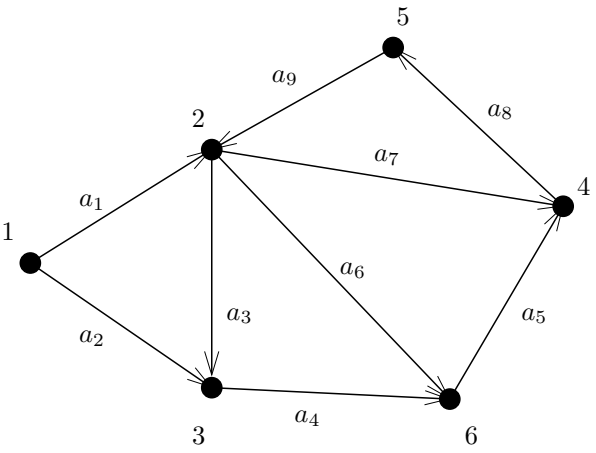
Ορισμός 10. Έστω γράφος $G = (X, U)$ με $X = (x_1, x_2, ..., x_n)$ και $U = (u_1, u_2, ..., u_m)$. Ένα *μονοπάτι* (path) f στο γράφο G είναι μια ακολουθία $x_1, u_1, x_2, u_2, \ldots, x_k$ εναλλασσόμενων κορυφών και ακμών ($u_i = [x_i, x_{i+1}]$), που ξεκινάει και τελειώνει σε κορυφή. Λέμε ότι το μονοπάτι συνδέει τον κόμβο x_1 με τον κόμβο x_k. Κάθε μονοπάτι έχει μήκος $k-1$, όπου k ο αριθμός των κόμβων του μονοπατιού. Εναλλακτικά μπορούμε να πούμε ότι *το μήκος του μονοπατιού* είναι p, όπου p ο αριθμός των ακμών από τις οποίες διέρχεται το μονοπάτι.

Εναλλακτικά ο ορισμός του μονοπατιού μπορεί να διατυπωθεί ως εξής:

Ορισμός 11. Ένα *μονοπάτι* f του γράφου $G = (X, U)$ είναι μια πεπερασμένη ακολουθία τόξων $\alpha_1, \alpha_2, \ldots, \alpha_p$ έτσι ώστε: $\forall i,\ 1 \leq i < p \quad \text{αρ}(\alpha_{i+1}) = \text{ακ}(\alpha_i)$.

Ένα μονοπάτι καλείται *απλό* αν διέρχεται από κάθε τόξο του μόνο μία φορά. Ένα μονοπάτι καλείται *στοιχειώδες* αν διέρχεται από κάθε κόμβο του μόνο μία φορά.

Παράδειγμα 4 (Μονοπάτια)**.**



Σχήμα 5.4: Ένας κατευθυνόμενος γράφος τάξης 6 με 9 κατευθυνόμενες πλευρές

Στο γράφο του σχήματος 5.4 διακρίνουμε τα εξής μονοπάτια:

Μονοπάτι: $\mu = \alpha_1\ \alpha_7\ \alpha_8\ \alpha_9\ \alpha_6\ \alpha_5\ \alpha_8$

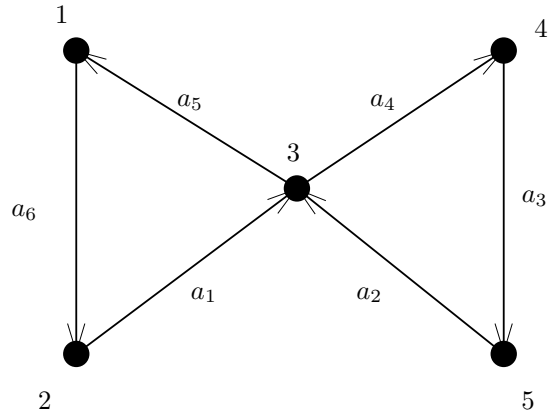
Απλό Μονοπάτι: $\mu = \alpha_1\ \alpha_7\ \alpha_8\ \alpha_9\ \alpha_6$

Στοιχειώδες Μονοπάτι: $\mu = \alpha_1\ \alpha_3\ \alpha_4\ \alpha_5$

Η αρχή ενός μονοπατιού f, συμβολιζόμενη αρ(f) είναι αυτή του πρώτου τόξου α_1 και το άκρο του, συμβολιζόμενο ακ(f) είναι αυτό του τελευταίου τόξου α_p.

Κύκλος (cycle) είναι ένα μονοπάτι μ, στο οποίο ο πρώτος και ο τελευταίος κόμβος ταυτίζονται $(x_1 = x_k)$ ή αρ$(\mu) =$ ακ(μ).
Απλός Κύκλος (simple cycle) είναι ένας κύκλος που διέρχεται από κάθε ακμή του μόνο μία φορά.
Στοιχειώδης Κύκλος είναι ένας κύκλος που διέρχεται από κάθε κόμβο του μόνο μία φορά.

Παράδειγμα 5 (Κύκλοι - Απλοί κύκλοι - Στοιχειώδεις Κύκλοι)**.**



Σχήμα 5.5: Γράφος παραδείγματος 5

Στο γράφο του σχήματος 5.5 διακρίνουμε τους εξής κύκλους:

Κύκλος: $\mu = \alpha_6\ \alpha_1\ \alpha_4\ \alpha_3\ \alpha_2\ \alpha_5 \alpha_6\ \alpha_1\ \alpha_5$

Απλός Κύκλος: $\mu = \alpha_6\ \alpha_1\ \alpha_4\ \alpha_3\ \alpha_2\ \alpha_5$

Στοιχειώδης Κύκλος: $\mu = \alpha_1\ \alpha_5\ \alpha_6$

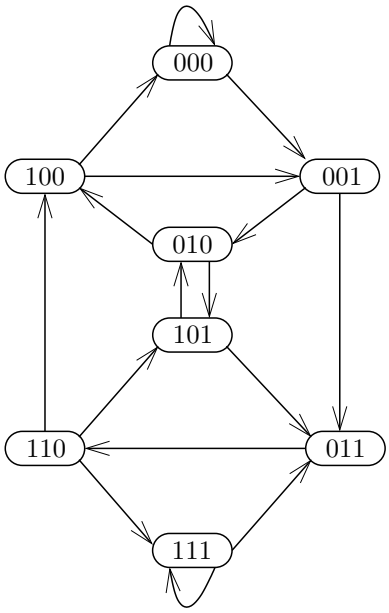
Παραδείγματα Κατευθυνόμενων γράφων

Γράφοι De Bruijn

Οι κόμβοι ενός γράφου De Bruijn είναι ακολουθίες μήκους κ σχηματιζόμενες από τα σύμβολα 0 και 1. Ένα τόξο ενώνει την ακολουθία f με την ακολουθία g αν:

- $f = xh$ και
- $g = hy$

όπου x και y είναι 0 ή 1 και h είναι μια οποιαδήποτε ακολουθία από $\kappa - 1$ σύμβολα (βλ. για παράδειγμα σχήμα 5.6).



Σχήμα 5.6: Γράφος De Bruijn με $\kappa = 3$

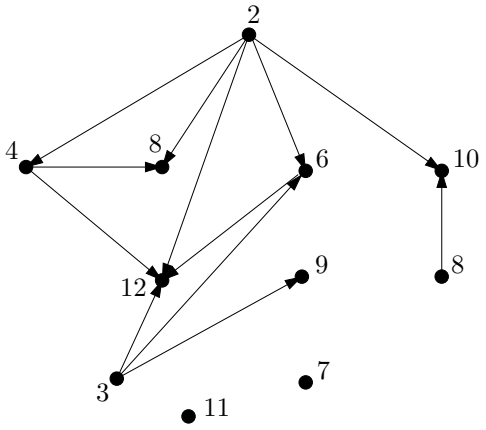
Γράφοι Διαιρετών

Κάθε κόμβος ενός γράφου διαιρετών είναι ένας αριθμός του συνόλου $\{2, 3, \ldots, n\}$. Ένα τόξο ενώνει ένα κόμβο p με τον κόμβο q αν ο p διαιρεί τον q (βλ. σχήμα 5.7).

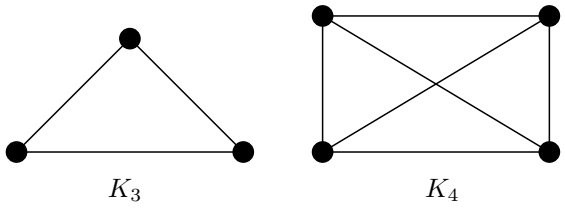
5.3 Ειδικές Κατηγορίες Γράφων

5.3.1 Πλήρεις Γράφοι

Ορισμός 12. *Πλήρης γράφος ή κλίκα τάξης* n (K_n) ονομάζεται ο γράφος, στον οποίο κάθε ζευγάρι κόμβων ενώνεται με μία ακμή.



Σχήμα 5.7: Γράφος Διαιρετών

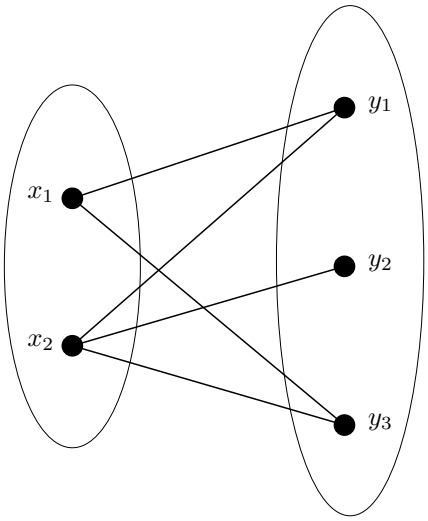


Σχήμα 5.8: Κλίκα 3 και Κλίκα 4

Στο σχήμα 5.8 παρουσιάζονται ο πλήρης γράφος με 3 κόμβους και ο πλήρης γράφος με 4 κόμβους.

5.3.2 Διμερείς Γράφοι

Ορισμός 13. *Διμερής* (bipartite) ονομάζεται ένας γράφος $G = (V, E)$ όταν μπορούμε να διαιρέσουμε τους κόμβους σε δύο υποσύνολα (V_1, V_2) έτσι ώστε $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$ και κάθε ακμή έχει το ένα άκρο στο V_1 και το άλλο άκρο στο V_2 (βλ. σχήμα 5.9).



Σχήμα 5.9: Διμερής Γράφος

Ορισμός 14. *Διμερής Πλήρης Γράφος* ονομάζεται ένας διμερής γράφος $G(V, E)$ στον οποίο κάθε ζευγάρι κόμβων ενώνεται με μία ακμή. Στον διμερή πλήρη γράφο κάθε κόμβος του συνόλου V_1 έχει βαθμό ίσο με την πληθυκότητα του συνόλου V_2, δηλαδή $\forall v \in V_1,\ d(v) = |V_2|$ (συμβολίζουμε $K_{a,b}$).

Στο σχήμα 5.10 παρατηρούμε τους πλήρεις διμερής γράφους $K_{2,3}$ και $K_{4,4}$.

5.3.3 Κανονικοί Γράφοι

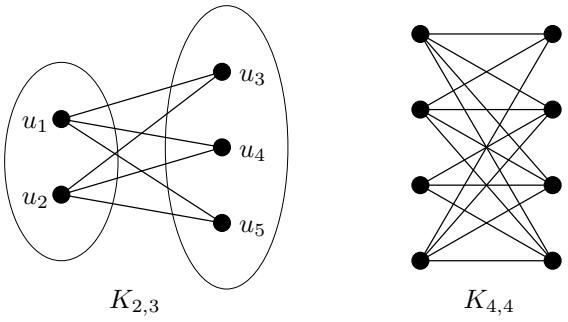
Ορισμός 15. *Κανονικός Γράφος* ή κ-κανονικός ονομάζεται ο γράφος $G = (V, E)$ στον οποίο όλοι οι κόμβοι έχουν τον ίδιο βαθμό κ, δηλ. $\forall v \in V\ d(v) = \kappa$.

5.3.4 Επίπεδοι γράφοι

Ορισμός 16. Ένας γράφος ονομάζεται *επίπεδος* αν μπορούμε να τον σχεδιάσουμε στο επίπεδο, χωρίς τμήση των πλευρών του. Στο σχήμα 5.1 ο γράφος $K_{3,3}$ των τριών κατοικιών και των τριών εργοστασίων δεν είναι επίπεδος.

5.3.5 Υπογράφοι (subgraph)

Έστω ένας γράφος $G = (X, U)$. Ένας *υπογράφος* του G είναι ένας γράφος $G' = (X', U')$, τέτοιος ώστε $X' \subseteq X$ και $U' \subseteq U$.



Σχήμα 5.10: Πλήρεις Διμερής Γράφοι

Παράγων υπογράφος του $G = (X, U)$ είναι ένας υπογράφος $G'' = (X, U'')$, όπου $U'' \subseteq U$. Δηλαδή ο παράγων υπογράφος G'' περιέχει τις ίδιες κορυφές με τον G και ένα υποσύνολο των ακμών του.

Στο σχήμα 5.11 παρουσιάζεται ένας υπογράφος $G^{'}$ του γράφου G και ένας παράγων υπογράφος $G^{''}$ του G.

5.3.6 Συνεκτικοί γράφοι

Δύο κόμβοι i, j ενός γράφου είναι *συνδεδεμένοι* όταν υπάρχει τουλάχιστον ένα μονοπάτι από τον κόμβο i στον κόμβο j.

Ορισμός 17. Ένας γράφος ονομάζεται *συνεκτικός* (connected) όταν κάθε ζευγάρι κορυφών του είναι συνδεδεμένο, αλλιώς ονομάζεται *μη συνεκτικός*.

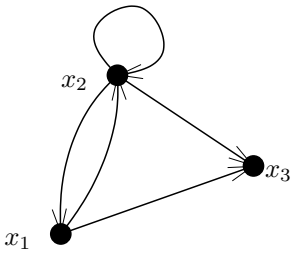
Αν ο γράφος δεν είναι *συνεκτικός* μπορούμε να βρούμε πολλούς υπογράφους συνεκτικούς, μέγιστους ως προς την έννοια της έγκλισης που ονομάζονται *συνεκτικές συνιστώσες*.

5.3.7 Χορδικοί Γράφοι (Chordal Graphs)

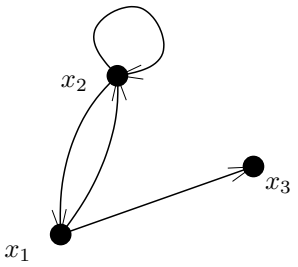
Οι χορδικοί γράφοι αποτελούν μια πολύ σημαντική κατηγορία γράφων. Είναι γνωστό ότι αρκετά δύσκολα προβλήματα λύνονται βέλτιστα σε πολυωνυμικό χρόνο σ' αυτή την κατηγορία γράφων.

Ένας γράφος $G = (V, E)$ καλείται *χορδικός* (chordal) αν κάθε απλός κύκλος του G, μήκους μεγαλύτερου του 3 έχει πάντα μια χορδή.

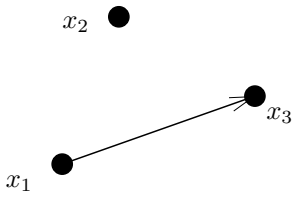
Ορισμός 18. Ένας γράφος G είναι χορδικός αν κάθε κύκλος με μήκος μεγαλύτερο από 3 περιέχει τουλάχιστον μία χορδή.



$G = (X, U)$



$G' = (X, U')$



$G'' = (X, U'')$

Σχήμα 5.11: $G^{'}$ υπογράφος του G και $G^{''}$ παράγων υπογράφος του G

5.3.8 Δέντρα (trees)

Ορισμός 19. Ένας *συνεκτικός* γράφος G είναι *δέντρο* αν και μόνο αν δύο οποιεσδήποτε κορυφές του ενώνονται με ένα μοναδικό μονοπάτι.

Ο G με βάση τον παραπάνω ορισμό είναι συνεκτικός και περιέχει ακριβώς ένα μονοπάτι μεταξύ δύο οποιονδήποτε κορυφών του γράφου. Αν μεταξύ των δύο κορυφών υπάρχουν παραπάνω από ένα μονοπάτια τότε αυτά σχηματίζουν κύκλο. Άρα ένα δέντρο δεν περιέχει κύκλους.

Κάθε δέντρο έχει μια ειδική κορυφή που ονομάζεται *ρίζα* (root) και μπορεί να είναι οποιοσδήποτε κόμβος του δέντρου. Επίσης κάθε δέντρο με περισσότερους από ένα κόμβο περιέχει τουλάχιστον δύο κόμβους βαθμού 1 , οι οποίοι ονομάζονται *φύλλα*. Οι κόμβοι με βαθμό μεγαλύτερο του 1 ονομάζονται *εσωτερικοί* κόμβοι.

Θεώρημα 6. Έστω T ένας γράφος με n κορυφές, τότε τα παρακάτω είναι ισοδύναμα

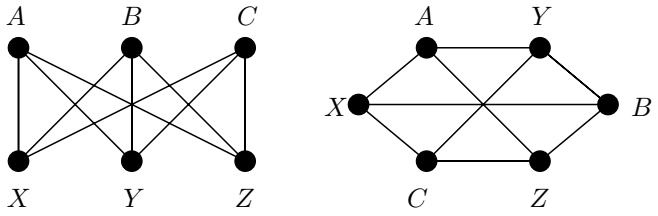
1. Ο T είναι δέντρο

2. Ο T είναι συνδεδεμένος και δεν έχει κύκλους
3. Ο T είναι συνδεδεμένος και έχει $n-1$ ακμές
4. Ο T δεν έχει κύκλους και έχει $n-1$ ακμές
5. Αν στον T προστεθεί μία καινούργια ακμή e τότε ο γράφος $T+e$ έχει ακριβώς ένα κύκλο
6. Οποιαδήποτε διαγραφή ακμής καταστρέφει τη συνεκτικότητα του T

5.4 Πράξεις επι των γράφων

5.4.1 Ισομορφισμός γράφων

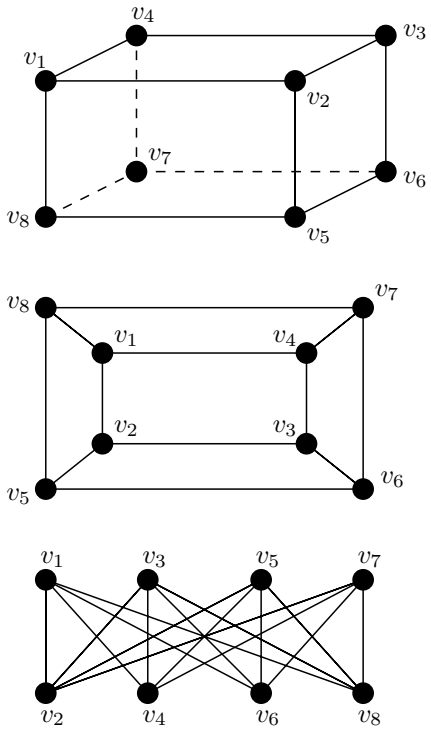
Δύο γράφοι $G_1 = (V_1, E_1)$ και $G_2 = (V_2, E_2)$ λέγονται ισομορφικοί αν υπάρχει μια αμφιμονοσήμαντη αντιστοιχία $f : V_1 \to V_2$ έτσι ώστε $[f(v_1), f(v_2)] \in E_2 \Leftrightarrow (v_1, v_2) \in E_1$. Δηλαδή αν οι κορυφές v_1, v_2 ενώνονται με μια ακμή στον γράφο G_1 τότε οι αντίστοιχες κορυφές $f(v_1), f(v_2)$ θα είναι γειτονικές στον γράφο G_2. Δύο ισομορφικοί γράφοι λοιπόν μπορεί να έχουν διαφορετική μορφή, αλλά έχουν τις ίδιες βασικές δομικές ιδιότητες. Για παράδειγμα οι γράφοι του σχήματος 5.12 είναι ισομορφικοί. Στο σχήμα 5.13 φαίνονται τρεις ισομορφικοί 3-κανονικοί γράφοι. Παρατηρούμε λοιπόν ότι ο 3-κανονικός κύβος είναι ισομορφικός με ένα 3-κανονικό επίπεδο γράφο και με ένα 3-κανονικό διμερή γράφο.



Σχήμα 5.12: Δύο Ισομορφικοί Γράφοι

5.4.2 Συμπλήρωμα ενός Γράφου

Το *συμπλήρωμα* ενός γράφου $G = (V, E)$ συμβολίζεται ως $\bar{G} = (\bar{V}, \bar{E}) = (V, V^2 - E)$ και είναι ένας γράφος που έχει σύνολο κορυφών $\bar{V} = V$, ενώ το σύνολο των ακμών του $\bar{E}$ αποτελείται από όλες τις δυνατές ακμές που δεν ανήκουν στο σύνολο E. Στην ουσία με την πράξη αυτή αγνοούνται όλες οι ακμές του G



Σχήμα 5.13: Τρεις Ισομορφικοί Γράφοι

και θεωρούνται όλες οι ακμές που αρχικά δεν υπήρχαν στον G. Αποδεικνύεται ότι αν ο γράφος G δεν είναι συνεκτικός τότε ο $\bar{G}$ είναι συνεκτικός.

5.4.3 Διαγραφή (deletion)

Διαγραφή κορυφής Έστω ένας γράφος $G = (V, E)$ και μια κορυφή $v \in V$. Εφαρμόζοντας την πράξη της διαγραφής στον γράφο G για την κορυφή v προκύπτει ένας νέος γράφος $G' = (V - v, E - adj(v))$, όπου με $adj(v)$ συμβολίζεται το σύνολο των ακμών που προσπίπτουν στην κορυφή v. Δηλαδή ο G' προκύπτει από τον γράφο G αγνοώντας την κορυφή v και τις αντίστοιχες προσπίπτουσες ακμές.

Διαγραφή ακμής Έστω ένας γράφος $G = (V, E)$ και μια ακμή $e \in E$. Κατά την διαγραφή της ακμής e από τον G προκύπτει ένας νέος γράφος $G' = (V, E - e)$. Δηλαδή ο γράφος G' προκύπτει από τον G αγνοώντας την ακμή

e.

5.5 Αποθήκευση Γράφων

Στις προηγούμενες ενότητες παρουσιάστηκαν μερικοί βασικοί ορισμοί των γράφων και κάποιες ιδιότητές τους. Στην παρούσα ενότητα θα μελετήσουμε τους τρόπους αναπαράστασης των γράφων στη μνήμη ενός υπολογιστή.

5.5.1 Πίνακας Γειτνίασης

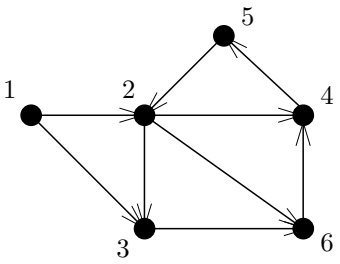
Έστω ένας απλός μη κατευθυνόμενος γράφος $G = (V, E), n = |V|, m = |E|$. Θεωρούμε ένα δισδιάστατο πίνακα M με n γραμμές και n στήλες, όπου κάθε γραμμή και κάθε στήλη του M αντιστοιχίζεται σε μια κορυφή του G. Το στοιχείο $M(i, j)$ ισούται με 1 αν η κορυφή v_i είναι γειτονική της v_j αλλιώς ισούται με 0.

$$M(i,j) = \begin{cases} 1 & \text{αν } (v_i, v_j) \in E \\ 0 & \text{αν } (v_i, v_j) \notin E \end{cases}$$

Ο πίνακας αυτός ονομάζεται *πίνακας γειτνίασης* (adjacency matrix) και είναι ένας συμμετρικός πίνακας $M(i, j) = M(j, i)$ με n^2 δυαδικά ψηφία που έχει μηδενικά στην κύρια διαγώνιο $M(i, i) = 0$. Αν ο γράφος $G = (V, E)$ είναι κατευθυνόμενος τότε ο πίνακας M δεν είναι απαραίτητα συμμετρικός ενώ αν ο γράφος $G = (V, E)$ δεν είναι απλός (περιέχει βρόγχους), τότε η διαγώνιος έχει και μη μηδενικές τιμές. Αν ο γράφος G είναι με βάρη ($G = (V, E, W)$) και $w_{ij} \in W$ είναι το βάρος της ακμής (v_i, v_j) τότε ο πίνακας γειτνίασης κωδικοποιείται ως εξής:

$$M(i,j) = \begin{cases} w_{ij} & \text{αν } (v_i, v_j) \in E \\ 0 & \text{αν } (v_i, v_j) \notin E \end{cases}$$

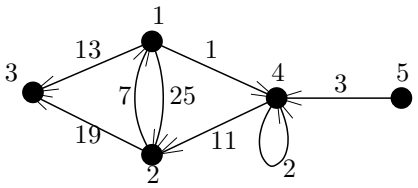
Παράδειγμα 6 (Πίνακας Γειτνίασης)**.** Δίνεται ο κατευθυνόμενος γράφος $G = (X, U)$, $|X| = 6$, $|U| = 9$ του σχήματος 5.14. Ο πίνακας γειτνίασης αυτού του γράφου φαίνεται στο ίδιο σχήμα.



$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Σχήμα 5.14: Ένας κατευθυνόμενος γράφος και ο πίνακας γειτνίασης M

Παράδειγμα 7 (Πίνακας Γειτνίασης Βεβαρημένου Γράφου)**.** Δίνεται ο κατευθυνόμενος γράφος $G(X, U, W)$, $|X| = 5$, $|U| = 8$, που φαίνεται στο σχήμα 5.15. Ο πίνακας γειτνίασης αυτού του γράφου φαίνεται στο σχήμα δεξιά του γράφου.



$$\mathbf{M} = \begin{pmatrix} 0 & 25 & 13 & 1 & 0 \\ 7 & 0 & 19 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{pmatrix}$$

Σχήμα 5.15: Ένας γράφος με βάρη και ο πίνακας γειτνίασης M

Ένα ιδιαίτερο χαρακτηριστικό του πίνακα γειτνίασης είναι ότι ο καθορισμός του πλήθους των μονοπατιών σε έναν μη κατευθυνόμενο γράφο G, γίνεται απλά υπολογίζοντας τις διαδοχικές δυνάμεις του πίνακα M. Έστω για παράδειγμα M^p, η p-ιοστή δύναμη του πίνακα M, τότε το στοιχείο $M^p(i, j)$ είναι ίσο προς τον αριθμό των μονοπατιών του G, μήκους p, των οποίων η αρχή είναι ο κόμβος x_i και το τέλος ο κόμβος x_j. Με βάση αυτή την παρατήρηση μπορούμε να σκιαγραφήσουμε έναν αλγόριθμο για να ελέγξουμε αν υπάρχει μονοπάτι μεταξύ δύο κόμβων x_i, x_j του γράφου. Σε κάθε βήμα του αλγορίθμου υψώνουμε τον πίνακα γειτνίασης M σε μια δύναμη p, $1 \leq p \leq n$ (στην ουσία πολλαπλασιάζουμε τον M με τον M^{p-1}) και εξετάζουμε αν υπάρχει αριθμός στη θέση (i, j) του πίνακα M^p. Δεδομένου ότι ο πολλαπλασιασμός δύο πινάκων έχει πολυπλοκότητα $\mathcal{O}(n^3)$ ($\mathcal{O}(n^{2.81})$ με τον αλγόριθμο του Strassen) και ότι θα κάνουμε το πολύ n τέτοιους πολλαπλασιασμούς, η συνολική πολυπλοκότητα του παραπάνω αλγορίθμου είναι $\mathcal{O}(n^4)$.

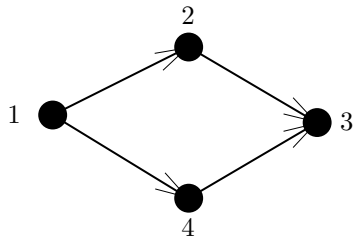
Παράδειγμα 8 (Ύπαρξη μονοπατιού)**.**

Δίνεται ο κατευθυνόμενος γράφος $G(X, U)$, $|X| = 4$, $|U| = 4$, του σχήματος 5.16. Αριστερά του γράφου φαίνεται ο πίνακας γειτνίασης του M και δεξιά του αποτέλεσμα του πολλαπλασιασμού του M με τον εαυτό του M^2.

Επειδή $M^2(1, 3) = 2$ υπάρχουν δύο μονοπάτια (όπως φαίνεται και στο σχήμα) μεταξύ των κόμβων 1 και 3.

Ο πίνακας γειτνίασης είναι μια δομή αναπαράστασης γράφων που απαιτεί n^2 bits για την αποθήκευση στην μνήμη ενός υπολογιστή. Η δομή αυτή μας επιτρέπει να ελέγξουμε την ύπαρξη ακμής μεταξύ δύο κόμβων σε σταθερό χρόνο ($\mathcal{O}(1)$). Επίσης μπορούμε σε χρόνο $\mathcal{O}(n)$ να εξετάσουμε αν ο γράφος περιέχει βρόγχους, ελέγχοντας αν κάποιο από τα στοιχεία της διαγωνίου είναι 1. Εξετάζοντας την συμμετρία του πίνακα μπορούμε να αναγνωρίσουμε αν ένας γράφος είναι κατευθυνόμενος ή όχι. Αξίζει να σημειώσουμε πως υπάρχουν τεχνικές συμπίεσης του πίνακα γειτνίασης, τόσο για μη κατευθυνόμενους όσο και για κατευθυνόμενους γράφους, οι οποίες επιτρέπουν την αποθήκευση σε χώρο μικρότερο από n^2 bits. Ειδικά σε αραιούς γράφους (π.χ. γράφοι με μέγιστο βαθμό 3) η ύπαρξη μιας

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



$$\mathbf{M}^2 = \begin{pmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Σχήμα 5.16: Ένας κατευθυνόμενος γράφος, ο πίνακας γειτνίασης M αριστερά και ο πίνακας M^2 δεξιά

τέτοιας τεχνικής συμπίεσης είναι αναγκαία καθώς τα περισσότερα στοιχεία του πίνακα είναι 0.

Ο υπολογισμός όλων των επόμενων κόμβων x_i ενός κόμβου x_j σε ένα γράφο, που αναπαρίσταται με πίνακα γειτνίασης, απαιτεί σάρωμα ολόκληρης της γραμμής j, δηλαδή γίνεται σε χρόνο $\mathcal{O}(n)$. Αντίστοιχα ο υπολογισμός όλων των προηγούμενων κόμβων x_i ενός κόμβου x_j απαιτεί σάρωμα ολόκληρης της στήλης j, δηλαδή και αυτός ο υπολογισμός γίνεται σε χρόνο $\mathcal{O}(n)$.

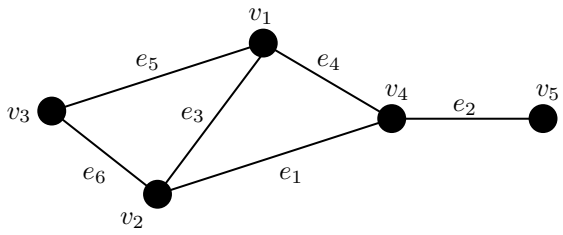
5.5.2 Πίνακας Πρόσπτωσης

Έστω ένας απλός γράφος $G = (V, E)$, με $n = |V|$ και $m = |E|$. Θεωρούμε ένα δισδιάστατο πίνακα B με n γραμμές και m στήλες, όπου κάθε γραμμή του B αντιστοιχίζεται σε μια κορυφή του G και κάθε στήλη του B αντιστοιχίζεται σε μια ακμή του G. Το στοιχείο $B(i, j)$ ισούται με 1 αν η ακμή e_j προσπίπτει στην κορυφή v_i αλλιώς ισούται με 0.

$$B(i,j) = \begin{cases} 1 & \text{αν η ακμή } e_j \text{ προσπίπτει στην κορυφή } v_j) \\ 0 & \text{αλλιώς} \end{cases}$$

Ο πίνακας αυτός ονομάζεται *πίνακας πρόσπτωσης* (incidence matrix) και έχει μέγεθος $n \times m$ δυαδικά ψηφία.

Παράδειγμα 9 (Πίνακας Πρόσπτωσης)**.** Δίνεται ο μη κατευθυνόμενος γράφος $G = (V, E)$, $|V| = 6$, $|E| = 9$, του σχήματος 5.17.



Σχήμα 5.17: Ένας γράφος όπου οι πλευρές του είναι αριθμημένες αυθαίρετα $e_1, e_2, \ldots, e_6$

Ο πίνακας πρόσπτωσης αυτού του γράφου είναι:

$$\mathbf{B} = \begin{array}{c|cccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \hline v_1 & 0 & 0 & 1 & 1 & 1 & 0 \\ v_2 & 1 & 0 & 1 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 & 1 & 1 \\ v_4 & 1 & 1 & 0 & 1 & 0 & 0 \\ v_5 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Ο πίνακας πρόσπτωσης απαιτεί για την αποθήκευσή του $\mathcal{O}(n \times m)$ bits και δεν είναι τετραγωνικός ή συμμετρικός. Στην περίπτωση των απλών γράφων (χωρίς βρόγχους) κάθε στήλη περιέχει δύο άσσους. Αν ο γράφος έχει παράλληλες ακμές τότε οι αντίστοιχες στήλες είναι ίδιες ενώ μια γραμμή με μηδενικά υποδηλώνει την ύπαρξη απομονωμένης κορυφής.

5.5.3 Λίστες Γειτνίασης

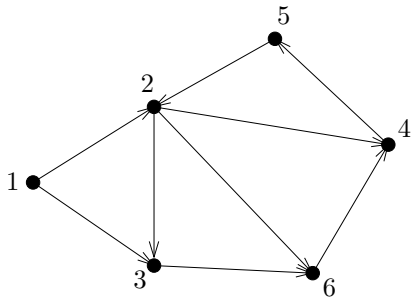
Όταν αναπαριστούμε ένα γράφο $G = (V, E), n = |V|, m = |E|$, με *λίστες γειτνίασης* η επιγραφή[1] κάθε κορυφής είναι η κεφαλή μίας *συνδεδεμένης λίστας* που αποτελείται από το σύνολο των γειτονικών κορυφών. Παρατηρούμε ότι η δομή αυτή δεν περιέχει πληροφορία για την μη γειτνίαση, όπως για παράδειγμα στην περίπτωση των μηδενικών των πινάκων στους δύο προηγούμενους τρόπους αποθήκευσης. Για την αποθήκευση των κορυφών που αποτελούν κεφαλές στις συνδεδεμένες λίστες χρησιμοποιούνται οι δύο παρακάτω μέθοδοι:

1. Αποθήκευση σε *γραμμικό πίνακα*. Σε κάθε θέση του πίνακα αποθηκεύεται η επιγραφή της κορυφής και ένας δείκτης προς τον πίνακα που περιέχει τους γείτονές της.

[1] Επιγραφή είναι μια τιμή συνήθως αριθμητική που προσδιορίζει μοναδικά κάθε κόμβο στο γράφο

2. Αποθήκευση σε *γραμμική συνδεδεμένη λίστα*. Αντί για στατικό πίνακα χρησιμοποιείται η δυναμική δομή της συνδεδεμένης λίστας και σε κάθε θέση της αποθηκεύεται η επιγραφή της κορυφής και ένας δείκτης προς τη λίστα που περιέχει τους γείτονές της.

Παράδειγμα 10 (Λίστα Γειτνίασης)**.** Δίνεται κατευθυνόμενος γράφος $G = (X, U)$, $|X| = 6$, $|U| = 9$ του σχήματος 5.18. Ο πίνακας γειτνίασης αυτού του γράφου φαίνεται στο ίδιο σχήμα.



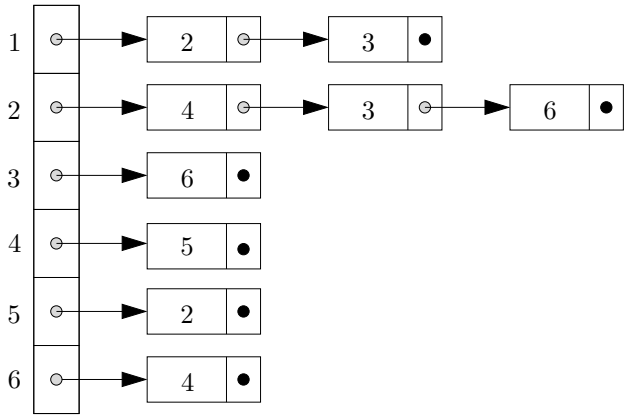
$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Σχήμα 5.18: Ένας κατευθυνόμενος γράφος και ο πίνακας γειτνίασης M

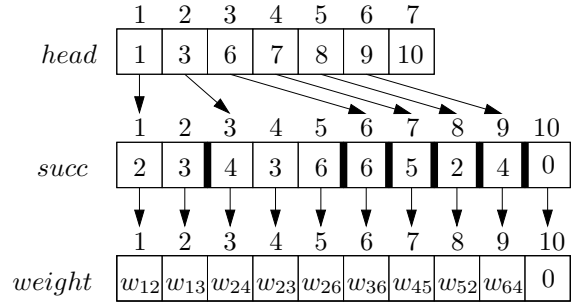
Η λίστα γειτνίασης του γράφου όταν αναπαρίσταται με συνδεδεμένες λίστες φαίνεται στο σχήμα 5.19 και όταν αναπαρίσταται με γραμμικούς πίνακες στο σχήμα 5.20.

Βασικό πλεονέκτημα αυτής της δομής είναι ότι είναι πολύ εύκολο να ανακτηθούν όλοι οι γείτονες μιας κορυφής πολύ γρήγορα. Για παράδειγμα σε έναν αραιό γράφο G με 20000 κορυφές και βαθμό κάθε κορυφής το πολύ 5, με αναπαράσταση πίνακα απαιτούνται 20000 έλεγχοι για να βρούμε τους διαδόχους μιας κορυφής. Αντίθετα με λίστα γειτνίασης απαιτούνται το πολύ 6 έλεγχοι.

Η μνήμη που απαιτείται για την αποθήκευση ενός γράφου με λίστα γειτνίασης είναι $\Theta(n+m)$ bits για κατευθυνόμενο γράφο και $\Theta(n+2m)$ για μη κατευθυνόμενο γράφο. Κύριο μειονέκτημα αυτής της αναπαράστασης είναι ότι για να ελεγχθεί η ύπαρξη μιας ακμής απαιτούνται $\mathcal{O}(m)$ πράξεις. Ο ίδιος χρόνος απαιτείται για την επεξεργασία όλων των ακμών του γράφου και για την πρόσθεση μιας ακμής.



Σχήμα 5.19: Λίστα Γειτνίασης με συνδεδεμένες λίστες



Σχήμα 5.20: Λίστα Γειτνίασης με γραμμικούς πίνακες

Κεφάλαιο 6

Αλγόριθμοι Γράφων

6.1 Εξερεύνηση Γράφων

Η εξερεύνηση (ή αναζήτηση) σε γράφους, αποσκοπεί στο να βρει όλους τους κόμβους ενός γράφου που ικανοποιούν μια συγκεκριμένη ιδιότητα. Μερικές από τις εφαρμογές των αλγορίθμων εξερεύνησης είναι οι παρακάτω

1. Εύρεση όλων των κόμβων ενός γράφου, στους οποίους καταλήγουν μονοπάτια που ξεκινούν από μια καθορισμένη κορυφή s.
2. Εύρεση όλων των κόμβων ενός γράφου, από τους οποίους ξεκινούν μονοπάτια που καταλήγουν σε μια καθορισμένη κορυφή. t
3. Εύρεση ελάχιστων μονοπατιών.
4. Αναγνώριση κύκλων σε ένα γράφο.
5. Αναγνώριση των συνεκτικών συνιστωσών ενός γράφου.
6. Αναγνώριση διμερών (bipartite) γράφων.

Για να δείξουμε τα βασικά χαρακτηριστικά των αλγορίθμων εξερεύνησης στη συγκεκριμένη ενότητα θα ασχοληθούμε κυρίως με τις δύο πρώτες κατηγορίες εφαρμογών. Στη συνέχεια θα θεωρούμε ότι οι γράφοι είναι κωδικοποιημένοι με λίστες γειτνίασης σε πίνακα. Επομένως οι γείτονες ενός κόμβου θα βρίσκονται σε $\times(d(v))$ χρόνο, όπου $d(v)$ ο βαθμός του κόμβου v. Αν ο γράφος είναι κατευθυνόμενος οι επόμενοι θα βρίσκονται σε $\times(d^+(v))$

6.1.1 Γενική μορφή ενός αλγορίθμου εξερεύνησης

Έστω ένας γράφος $G = (V, E)$ μια κορυφή $s \in V$, την οποία αποκαλούμε *πηγή* (source). Ο αλγόριθμος εξερεύνησης αναζητά τους διαδόχους του κόμβου s, δηλαδή όλους τους κόμβους του γράφου G που βρίσκονται σε μονοπάτια με αρχή τον κόμβο s.

Ο αλγόριθμος αναζήτησης (βλ. σχήμα 6.1) ξεκινά από την κορυφή s και προσδιορίζει έναν αυξανόμενο αριθμό κόμβων, που είναι διάδοχοί[1] του. Σε κάθε ενδιάμεσο βήμα του αλγορίθμου κάθε κόμβος είναι *μαρκαρισμένος (marked)* ή *όχι μαρκαρισμένος (unmarked)*. Οι μαρκαρισμένοι κόμβοι είναι διάδοχοι του s και οι όχι μαρκαρισμένοι κόμβοι είναι οι κόμβοι που δεν έχει προσδιοριστεί ακόμα η κατάστασή τους. Αν ο κόμβος i είναι μαρκαρισμένος, ο κόμβος j είναι μη μαρκαρισμένος και υπάρχει η ακμή (i, j) τότε μπορούμε να μαρκάρουμε τον κόμβο j (μπορούμε να φτάσουμε στον κόμβο j μέσω ενός μονοπατιού από τον s στον i προσθέτοντας την ακμή (i, j)). Η ακμή (i, j) ονομάζεται *αποδεκτή (admissible)* αλλιώς ονομάζεται *μη αποδεκτή (inadmissible)*. Ο αλγόριθμος ξεκινά μαρκάροντας τον κόμβο s και βρίσκοντας αποδεκτές ακμές μαρκάρει καινούριους κόμβους. Κάθε φορά που μαρκάρουμε ένα κόμβο j εξετάζοντας μια αποδεκτή ακμή (i, j) λέμε ότι ο i είναι *πρόγονος* (predecessor) του j.

Ο αλγόριθμος χρησιμοποιεί έναν πίνακα mark για να δείχνει όλες τις κορυφές που έχουμε επισκεφθεί, ένα πίνακα List που διατηρεί τους κόμβους των οποίων απομένουν διάδοχοι για ανίχνευση και ένα πίνακα order που φυλάσσει την σειρά επίσκεψης των κόμβων.

SEARCH (G, s)

1. Αρχικοποίησε τον πίνακα $mark$ σε $false$
2. $Mark[s] = True$
3. $List = \{s\}$
4. Όσο $List \neq \emptyset$
5. Διάλεξε ένα κόμβο i από τον $List$
6. Αν όλοι οι διάδοχοι του i έχουν εξεταστεί
7. $List = List - \{i\}$
8. Διαφορετικά
9. πάρε j επόμενο διάδοχο του i
10. Αν j δεν είναι μαρκαρισμένος τότε
11. $Mark[j] = True$
12. $List = List \cup \{j\}$

Σχήμα 6.1: Γενικός αλγόριθμος στο γράφο G ξεκινώντας από τον κόμβο s

6.1.2 Εξερεύνηση Πρώτα Κατά Πλάτος

Αν θεωρήσουμε ότι ο πίνακας $List$ είναι κωδικοποιημένος με τη δομή της ουράς, τότε στον αλγόριθμο αναζήτησης θα διαλέγουμε πάντα κόμβους από την αρχή της λίστας $List$ και θα προσθέτουμε στο τέλος της λίστας. Σ′ αυτή την περίπτωση ο αλγόριθμος (βλ. σχήμα 6.2) αναζήτησης επιλέγει τους μαρκαρισμένους κόμβους σε μια *πρώτος μπαίνει - πρώτος βγαίνει* σειρά. Κάθε κόμβος που μαρκάρεται δεν

[1] Διάδοχος του s λέγεται κάθε κόμβος που βρίσκεται σε μονοπάτι με αρχή τον κόμβο s

επιστρέφει στη λίστα $List$. Σε κάθε βήμα ο αλγόριθμος εξετάζει όλους του διαδόχους ενός κόμβου. Δηλαδή αρχικά εξερευνά τους γείτονες του s που βρίσκονται σε απόσταση 1 από αυτόν, μετά αυτούς που βρίσκονται σε απόσταση 2 κ.ο.κ.. Αυτή η έκδοση του αλγορίθμου αναζήτησης ονομάζεται *πρώτα κατά πλάτος αναζήτηση (Breadth-First Search)*. Επειδή σε κάθε βήμα εξετάζονται όλοι οι άμεσοι διάδοχοι ενός κόμβου η πολυπλοκότητα του αλγορίθμου είναι:

$$\sum_{i=1}^{n} \left(d^+(i)+1\right) = \sum_{i=1}^{n} \left(d^+(i)\right) + n = n + m.$$

Δηλαδή η πολυπλοκότητα είναι $\Theta(m+n)$. Επειδή γενικά ισχύει $m > n$ η πολυπλοκότητα είναι $\Theta(m)$.

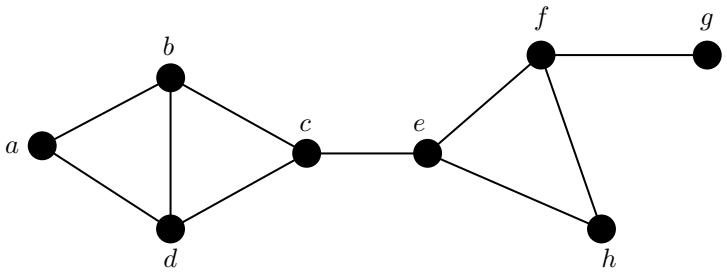
Breadth First Search (G, s)
1. Αρχικοποίησε τον πίνακα $mark$ σε $false$
2. $Mark[s] = True$
3. Θέσε s στο τέλος της ουράς $List$
4. Όσο $List \neq \emptyset$
5. Πάρε τον κόμβο i από την κορυφή της ουράς $List$
6. Για κάθε μη μαρκαρισμένο διάδοχο j του i
7. $Mark[j] = True$
8. Θέσε j στο τέλος της ουράς $List$

Σχήμα 6.2: Αλγόριθμος Breadth First Search ξεκινώντας από τον κόμβο s

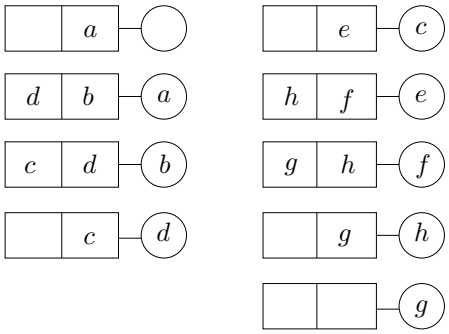
Παράδειγμα 11 (Αναζήτηση Κατά Πλάτος)**.** Στο σχήμα 6.5 βλέπουμε ένα παράδειγμα της εκτέλεσης του αλγορίθμου αναζήτησης πρώτα κατά πλάτος στο γράφο του σχήματος 6.3. Ο αλγόριθμος ξεκινά από τον κόμβο a, ο οποίος μαρκάρεται και τοποθετείται στην ουρά (βλ. σχήμα 6.4). Στη συνέχεια αφαιρείται ο a από την κεφαλή της ουράς και εξετάζονται οι γείτονες του. Οι κόμβοι b, d δεν είναι μαρκαρισμένοι, κατά συνέπεια μαρκάρονται και τοποθετούνται στο τέλος της ουράς. Στο επόμενο βήμα ο αλγόριθμος λαμβάνει τον κόμβο b από την κορυφή της ουράς. Αναλυτικά τα περιεχόμενα της ουράς κατά την διάρκεια της εκτέλεσης του αλγορίθμου φαίνονται στο σχήμα 6.4. Παρατηρούμε ότι στο τέλος του αλγορίθμου έχει δημιουργηθεί ένα κατευθυνόμενο δέντρο, του οποίου ρίζα είναι ο κόμβος από τον οποίο ξεκίνησε η αναζήτηση, δηλαδή ο κόμβος $s = a$ (βλ. σχήμα 6.5).

6.1.3 Εξερεύνηση Πρώτα Κατά Βάθος

Αν θεωρήσουμε ότι ο πίνακας $List$ είναι κωδικοποιημένος με τη δομή της στοίβας, τότε στον αλγόριθμο αναζήτησης θα διαλέγουμε πάντα κόμβους από την αρχή της λίστας $List$ και θα προσθέτουμε πάντα στην αρχή της λίστας. Σ′ αυτή την περίπτωση ο αλγόριθμος αναζήτησης επιλέγει τους μαρκαρισμένους κόμβους σε μια *τελευταίος μπαίνει - πρώτος βγαίνει* σειρά. Ο αλγόριθμος κάνει μια βαθιά



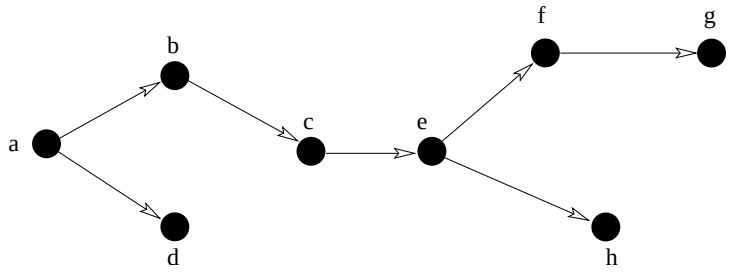
Σχήμα 6.3: Ένας μη κατευθυνόμενος γράφος τάξης 8



Σχήμα 6.4: Η κατάσταση της ουράς κατά την εκτέλεση του αλγορίθμου αναζήτησης κατά πλάτος

αναζήτηση δημιουργώντας το μεγαλύτερο δυνατό μονοπάτι με αρχή τον κόμβο s πριν επιστρέψει πίσω. Αυτή η έκδοση του αλγορίθμου αναζήτησης ονομάζεται *πρώτα κατά βάθος αναζήτηση (Depth First Search)* και περιγράφεται στο σχήμα 6.6. Η πολυπλοκότητα του αλγορίθμου είναι εύκολο να υπολογιστεί (άσκηση) και είναι $\Theta(m+n)$.

Παράδειγμα 12 (Αναζήτηση Κατά Βάθος)**.** Στο σχήμα 6.8 δείχνουμε την εφαρμογή της αναζήτησης πρώτα κατά βάθος στον γράφο του σχήματος 6.3. Ο αλγόριθμος ξεκινά από τον κόμβο a ο οποίος μαρκάρεται και τοποθετείται στην στοίβα. Στη συνέχεια επειδή όλοι οι γείτονές του δεν έχουν εξεταστεί (εδώ



Σχήμα 6.5: Το κατευθυνόμενο δέντρο που σχηματίζεται με την εκτέλεση της κατά πλάτος αναζήτησης

χρειαζόμαστε ένα μετρητή) επιλέγεται τυχαία ένας από τους δύο γείτονες (έστω ο b) για να μαρκαριστεί και να τοποθετηθεί στην κορυφή της στοίβας. Στο επόμενο βήμα λαμβάνεται ο κόμβος b από την κορυφή της στοίβας και ο μη μαρκαρισμένος γείτονάς του c τοποθετείται στη στοίβα. Στη συνέχεια εξετάζεται ο d γείτονας του c και τοποθετείται ο d στη στοίβα. Σ′ αυτό το σημείο ο d βρίσκεται στην κορυφή της στοίβας. Οι γείτονές του είναι όλοι μαρκαρισμένοι. Έτσι ο d διαγράφεται από την στοίβα και ο αλγόριθμος επιστρέφει στον κόμβο c, που βρίσκεται στην κορυφή της στοίβας. Ο αλγόριθμος συνεχίζει ομοίως μέχρι να μαρκαριστούν όλοι οι κόμβοι και να αδειάσει η στοίβα. Τα περιεχόμενα της στοίβας στο μελετούμενο παράδειγμα φαίνονται στο σχήμα 6.7.

Παρατηρούμε ότι και σε αυτή την περίπτωση δημιουργείται ένα κατευθυνόμενο δέντρο, το οποίο ωστόσο διαφέρει από το δέντρο που επιστρέφει η αναζήτηση κατά πλάτος (βλ. σχήμα 6.8).

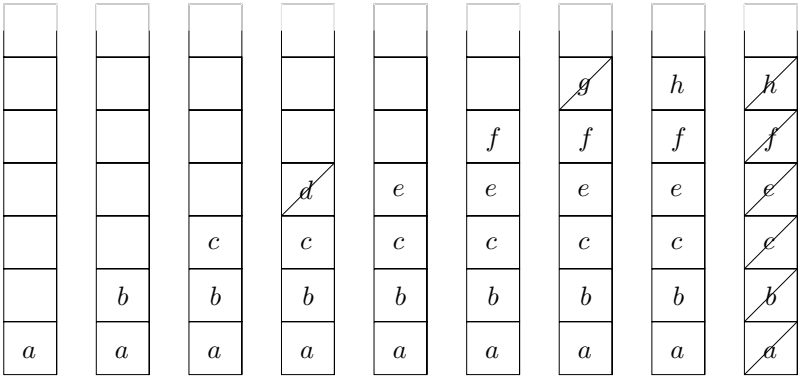
6.1.4 Εφαρμογές των αλγορίθμων εξερεύνησης

Υπολογισμός Κατευθυνόμενου Δέντρου Επίσκεψης

Για να κατασκευαστεί το κατευθυνόμενου δέντρο επίσκεψης, που παράγουν οι αλγόριθμοι αναζήτησης, χρησιμοποιούμε ένα πίνακα $father$ μεγέθους n. Ο πίνακας αυτός αρχικοποιείται πριν την εκκίνηση του αλγορίθμου αναζήτησης σε $father[i] = 0,\ \forall\ i \in [1, n]$ και $father[s] = s$. Θέτουμε $fathers[j] = i$ κάθε φορά, που εξερευνώντας τους γείτονες του κόμβου i, καταλήγουμε σε ένα μη μαρκαρισμένο κόμβο j, ο οποίος θα μαρκαριστεί και θα τοποθετηθεί στη χρησιμοποιούμενη δομή (ουρά ή στοίβα). Όταν τερματίσει ο αλγόριθμος αναζήτησης ο πίνακας $father$ περιέχει το πατέρα κάθε κόμβου στο δέντρο επίσκεψης. Διασχίζοντας αυτόν τον πίνακα από το τέλος ($father[n]$) μέχρι την αρχή ($father[1]$), μπορούμε να ανακατασκευάσουμε το δέντρο επίσκεψης.

```
Depth First Search (G, s)
 1.  Αρχικοποίησε τον πίνακα mark σε false
 2.  Mark[s] = True
 3.  Θέσε s στην αρχή της στοίβας List
 4.  Όσο List ≠ ∅
 5.      Πάρε τον κόμβο i από την κορυφή της στοίβας List
 6.      Αν όλοι οι διάδοχοι του i έχουν εξεταστεί
 7.          Διέγραψε το i από την στοίβα List
 8.      Διαφορετικά
 9.          πάρε j επόμενο διάδοχο του i
10.          Αν j όχι ακόμα επισκεφθείς τότε
11.              Mark[j] = True
12.              Θέσε j στην αρχή της στοίβας List
```

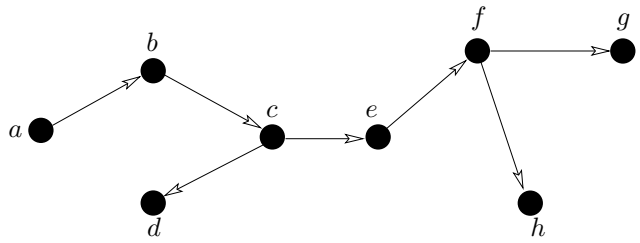
Σχήμα 6.6: Αλγόριθμος Depth First Search



Σχήμα 6.7: Η κατάσταση της στοίβας κατά την εκτέλεση της αναζήτησης κατά βάθος

Υπολογισμός ελάχιστων μονοπατιών σε αριθμό ακμών

Η πρώτα κατά πλάτος αναζήτηση μπορεί να υπολογίσει για κάθε προσβάσιμη κορυφή σε ένα γράφο $G = (V, E)$ την απόσταση της από μια δεδομένη κορυφή $s \in V$. Για να γίνει αυτό χρησιμοποιείται ένας πίνακας $distance$ μεγέθους n, ο οποίος αρχικοποιείται σε $distance[i] = \infty,\ \forall\ i \in [1, n]$, και $distance[s] = 0$. Θέτουμε $distance[j] = distance[i] + 1$ κάθε φορά που μαρκάρουμε ένα διάδοχο



Σχήμα 6.8: Το κατευθυνόμενο δέντρο αναζήτησης που σχηματίζεται από την κατά βάθος αναζήτηση

j του i. Στην ουσία η πρώτα κατά πλάτος αναζήτηση υπολογίζει το μήκος των ελάχιστων μονοπατιών, σε αριθμό ακμών.

Ορίζουμε $\delta(s, v)$ το μήκος του ελάχιστου μονοπατιού σε αριθμό πλευρών από τον κόμβο s στον κόμβο v. Αν δεν υπάρχει μονοπάτι από τον s στον v τότε $\delta(s, v) = \infty$. Πριν δείξουμε ότι η αναζήτηση κατά πλάτος υπολογίζει το μήκος των ελαχίστων μονοπατιών εξετάζουμε μια σημαντική ιδιότητα του μήκους των ελαχίστων μονοπατιών.

Λήμμα 1. Έστω $G = (V, E)$ ένας κατευθυνόμενος ή μη γράφος και έστω $s \in V$ μια κορυφή. Για κάθε πλευρά $(u, v) \in E$ ισχύει

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Απόδειξη. Αν ο κόμβος u είναι σε μονοπάτι με αρχή τον s τότε είναι και ο κόμβος v. Σ′ αυτή την περίπτωση το συντομότερο μονοπάτι από τον s στον v δεν μπορεί να είναι μεγαλύτερο από το συντομότερο μονοπάτι από τον s στον u ακολουθούμενο από την ακμή (u, v), και έτσι η ανισότητα ισχύει. Αν ο u δεν είναι προσβάσιμος από τον s τότε $\delta(s, u) = \infty$ και η ανισότητα ισχύει και πάλι. □

Θεώρημα 7. Έστω γράφος $G = (V, E)$ στον οποίο εκτελέσαμε τον αλγόριθμο Breadth First Search (BFS) με αρχή μια τυχαία κορυφή $s \in V$. Κατά τη διάρκεια της εκτέλεσης ο BFS ανακαλύπτει κάθε κορυφή $v \in V$ που είναι προσβάσιμη από την κορυφή s και κατά τον τερματισμό $distance[v] = \delta(s, v)$ για κάθε $v \in V$. Επιπρόσθετα για κάθε κορυφή $v \neq s$ που είναι προσβάσιμη από τον s ένα από τα ελάχιστα μονοπάτια από τον s στον v είναι ένα ελάχιστο μονοπάτι από τον s στον $father[v]$ ακολουθούμενο από την ακμή $(father[v], v)$.

Η απόδειξη του παραπάνω θεωρήματος παραλείπεται και ο αναγνώστης καλείται να την μελετήσει ως άσκηση.

Ανακάλυψη Κύκλων

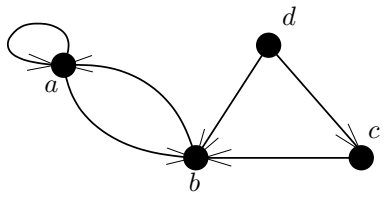
Ο αλγόριθμος αναζήτησης πρώτα κατά βάθος μπορεί να χρησιμοποιηθεί για τον εντοπισμό κλειστών μονοπατιών (κύκλων) σε ένα γράφο. Ο αλγόριθμος βρίσκει ένα κύκλο κάθε φορά που εντοπίζει ένα κόμβο j διάδοχο του i, που είναι ήδη μαρκαρισμένος.

Άσκηση: Να γραφεί ο αλγόριθμος, ο οποίος εντοπίζει ένα κύκλο στο γράφο $G = (V, E)$.

6.2 Συνεκτικότητα

Ορισμός 20. Ένας γράφος καλείται *συνεκτικός* αν υπάρχει μια αλυσίδα (ένα μονοπάτι) μεταξύ κάθε ζευγαριού κόμβων του γράφου (δηλαδή μπορούμε να πούμε ότι ο γράφος αποτελείται από ένα μόνο block). Αν ο γράφος είναι κατευθυνόμενος δεν λαμβάνουμε υπ'οψιν την κατεύθυνση.

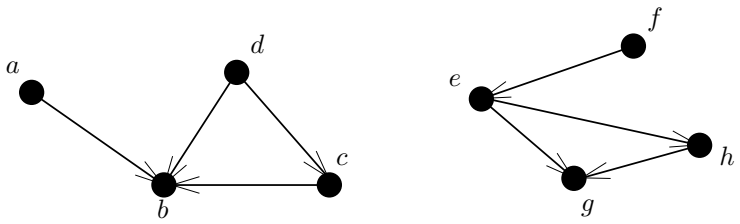
Αν ένας γράφος δεν είναι συνεκτικός, μπορούμε να βρούμε πολλούς συνεκτικούς υπογράφους, μέγιστους ως προς την έννοια της έγκλισης, οι οποίοι ονομάζονται συνεκτικές συνιστώσες. Στα σχήματα που ακολουθούν παρουσιάζουμε ένα συνεκτικό κατευθυνόμενο γράφο (σχ. 6.9), ένα γράφο αποτελούμενο από δύο συνεκτικές συνιστώσες (σχ. 6.10) και έναν απλό συνεκτικό γράφο (σχ. 6.11).



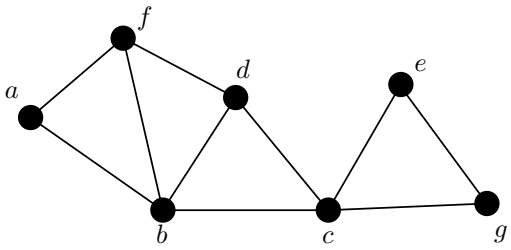
Σχήμα 6.9: Κατευθυνόμενος Συνεκτικός Γράφος

Ορισμός 21. Ένας κατευθυνόμενος γράφος είναι *ισχυρά συνεκτικός* αν για κάθε ζευγάρι διακεκριμένων κόμβων (u, v) υπάρχει ένα μονοπάτι από τον u στον v και από τον v στον u. Δηλαδή, οι κόμβοι u και v βρίσκονται πάνω σε ένα κλειστό μονοπάτι (κύκλο).

Σε αυτό το σημείο αξίζει να σημειώσουμε ότι ένας κατευθυνόμενος συνεκτικός γράφος δεν είναι αναγκαία ισχυρά συνεκτικός (π.χ. σχήμα 6.9).



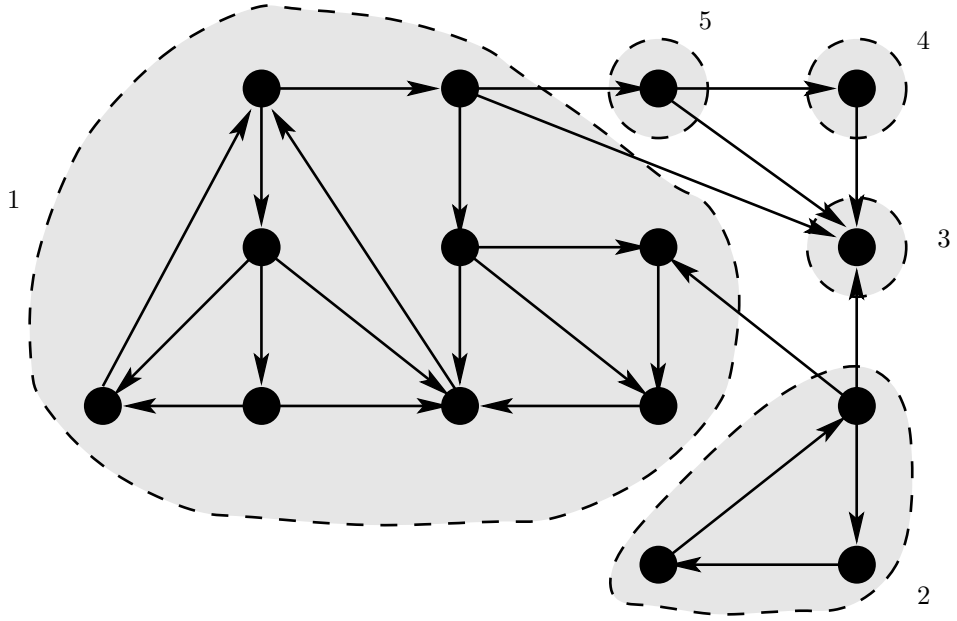
Σχήμα 6.10: Γράφος με 2 Συνεκτικές Συνιστώσες



Σχήμα 6.11: Απλός Συνεκτικός Γράφος

Αν ένας γράφος δεν είναι ισχυρά συνεκτικός, περιέχει $p > 1$ ισχυρά συνεκτικούς υπογράφους, μέγιστους ως προς την έγκλιση, ονομαζόμενοι *ισχυρά συνεκτικές συνιστώσες*. Στο σχήμα 6.12 παρουσιάζεται ένας κατευθυνόμενος γράφος και οι πέντε ισχυρά συνεκτικές συνιστώσες του.

Ένα από τα πιο γνωστά προβλήματα στη θεωρία γράφων είναι η εύρεση των συνεκτικών συνιστωσών ενός γράφου. Στο σχήμα 6.13 παρουσιάζουμε έναν αλγόριθμο, βασισμένο στην αναζήτηση κατά πλάτος, ο οποίος υπολογίζει τις συνεκτικές συνιστώσες ενός γράφου. Ο αλγόριθμος ξεκινά μαρκάροντας όλους τους κόμβους σε false. Στη συνέχεια εκτελούμε για κάθε μη μαρκαρισμένο κόμβο μια αναζήτηση κατά πλάτος. Προφανώς κάθε νέα αναζήτηση κατά πλάτος ξεκινά από ένα κόμβο που ανήκει σε μια συνεκτική συνιστώσα που δεν έχει ακόμα εξερευνηθεί. Έτσι κάθε αναζήτηση κατά πλάτος ανακαλύπτει μια ξεχωριστή συνεκτική συνιστώσα. Η πολυπλοκότητα του αλγορίθμου στην χείριστη περίπτωση είναι $\Theta(m + n)$ (άσκηση).



Σχήμα 6.12: Ισχυρά Συνεκτικές Συνιστώσες

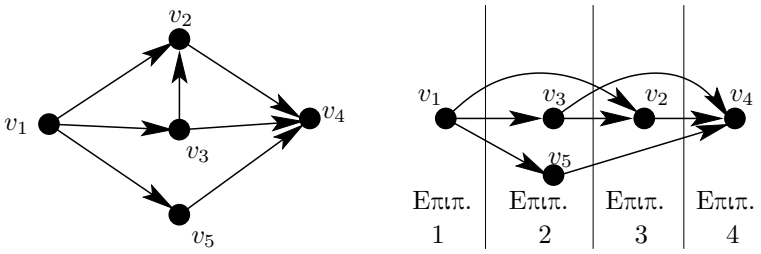
Breadth First Graph Traversal
1. Αρχικοποίησε τον πίνακα $mark$ σε $false$
2. Επανέλαβε για $i = 1$ μέχρι $i = n$
3. Αν $mark[v_i] = false$ τότε $BFS(v_i)$

Σχήμα 6.13: Αλγόριθμος Εύρεσης Συνεκτικών Συνιστωσών

6.3 Τοπολογική Ταξινόμηση

Έστω ένα πρόγραμμα που αποτελείται από ένα σύνολο διεργασιών X ίσου χρόνου που μπορούν να εκτελεστούν σε μία παράλληλη μηχανή, με τον περιορισμό ότι μία διεργασία $v \in X$ μπορεί να εκτελεστεί αφού έχουν ολοκληρωθεί πρώτα άλλες διεργασίες $u \in X$. Ζητείται η εύρεση ενός χρονοπρογράμματος (schedule) το οποίο θα αποφασίζει ποιες διεργασίες μπορούν να εκτελεστούν παράλληλα, τηρώντας τους περιορισμούς.

Το πρόβλημα μπορεί να μοντελοποιηθεί σαν ένας κατευθυνόμενος γράφος $G = (X, U)$ με X να είναι το σύνολο των διεργασιών και η ακμή $(u, v) \in U$ αν η διεργασία u πρέπει να έχει ολοκληρωθεί, για να εκτελεστεί η διεργασία v. Μία λύση στο πρόβλημα μας, είναι η διαίρεση του γράφου G σε επίπεδα, με τους κόμβους του κάθε επιπέδου να υποδηλώνουν ότι οι αντίστοιχες διεργασίες μπορούν να εκτελεστούν παράλληλα, ενώ η διάταξη των επιπέδων δείχνει την σειρά με την οποία μπορούν να εκτελεστούν τα πακέτα διεργασίων. Στο σχήμα 6.14 φαίνεται ένας γράφος διεργασιών με τους περιορισμούς εκτέλεσης και η τοποθέτηση των κόμβων σε επίπεδα τηρώντας τους περιορισμούς.



Σχήμα 6.14: Ένας ακυκλικός κατευθυνομενος γράφος αριστερά και η τοπολογική του ταξινόμηση δεξιά.

Το πρόβλημα της ταξινόμησης των κόμβων ενός κατευθυνόμενου γράφου σε επίπεδα (τοπολογική ταξινόμηση), συναντάται συχνά σαν υποπρόβλημα μεγαλύτερων προβλημάτων ή χρησιμοποιείται για την επιτάχυνση, μέσω της διάταξης των κόμβων, υπαρχόντων αλγορίθμων. Πρέπει να παρατηρήσουμε ότι ο γράφος πρέπει να είναι ακυκλικός.

Μία τοπολογική ταξινόμηση των κόμβων εκφράζεται μέσω ενός πίνακα $Layer$ που δείχνει σε ποιο επίπεδο έχει τοποθετηθεί κάθε κόμβος, ενώ οι κόμβοι τοποθετούνται στον τοπολογικά ταξινομημένο πίνακα $Sorted$. Στο παράδειγμα του σχήματος 6.14 ο πίνακας $Layer$ της τοπολογικής ταξινόμησης είναι ο $[1, 3, 2, 4, 2]$, ενώ ο πίνακας $Sorted$ είναι ο $[v_1, v_3, v_5, v_2, v_4]$.

Για να ισχύει ότι μία ταξινόμηση των κόμβων του γράφου σε επίπεδα, είναι τοπολογική ταξινόμηση, θα πρέπει να ισχύουν τα εξής:

- Για κάθε ακμή $(u, v) \in U$ πρέπει να ισχύει ότι $Layer[u] < Layer[v]$, δηλαδή κάθε κόμβος να έχει όλους του επόμενούς του σε επόμενα επίπεδα.
- Για κάθε κόμβο v που δεν ανήκει στο πρώτο επίπεδο, υπάρχει $u \in \Gamma^{-1}(v)$: $Layer[u] \leq Layer[v] - 1$.

Ένας απλός αλγόριθμος τοπολογικής ταξινόμησης έχει ως εξής: Θέτουμε σε μια ουρά Q, όλους τους κόμβους που δεν έχουν προηγούμενους. Στην συνέχεια

σε κάθε επανάληψη, αν η ουρά Q περιέχει s κόμβους, τους αφαιρούμε δίνοντας τους τον τρέχων αριθμό επιπέδου και τους ''διαγράφουμε'' από τον γράφο G. Η υλοποίηση αυτής της ιδέας φαίνεται στο σχήμα 6.15.

ΤΟΠΟΛΟΓΙΚΗ ΤΑΞΙΝΟΜΗΣΗ ($G = (X, U)$)

```
1.  Υπολογισμός πίνακα inDeg περιέχων τους έσω βαθμούς των κόμβων
2.  NLayer = 0 /* Τρέχων νούμερο επιπέδου */
3.  Θέσε i = 1 /* μετρητής πίνακα Sorted */
4.  Θέσε ουρά Q = Ø
5.  for x = 1 to n
6.      if inDeg[x] = 0 then EnQueue(Q, x)
7.  end for
8.  while Q ≠ Ø do
9.      NLayer = NLayer + 1
10.     for iter = 1 to |Q|
11.         Dequeue(Q, x) /* Διαγραφή x */
12.         Sorted[i] = x, i = i + 1
13.         Layer[x] = NLayer
14.         for k = head[x] to head[x + 1] − 1
15.             y = succ[k]
16.             inDeg[y] = inDeg[y] − 1
17.             if inDeg[y] = 0 then EnQueue(Q, y)
18.         end for
19.     end for
20. end while
```

Σχήμα 6.15: Αλγόριθμος Τοπολογικής Ταξινόμησης

Ο υπολογισμός του πίνακα $inDeg$ (βήμα 1) απαιτεί χρόνο $O(m)$, άρα η πολυπλοκότητα του αλγορίθμου χρησιμοποιώντας λίστες γειτνίασης είναι $O(m)$ (άσκηση).

6.4 Πράξεις σε ασύνδετα σύνολα (disjoint sets)

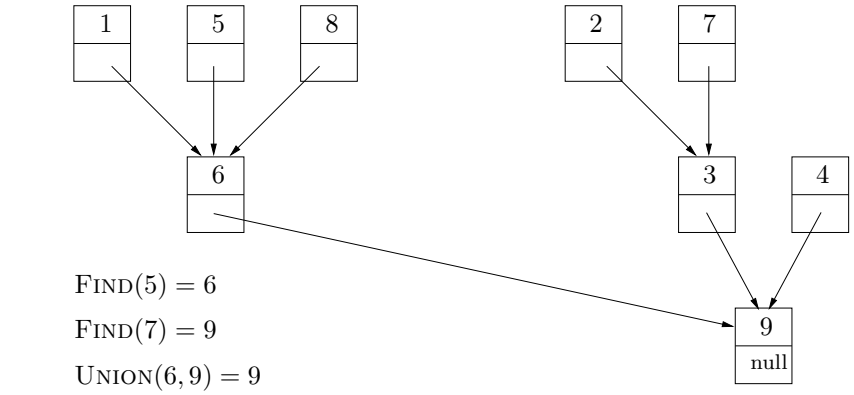
Δεδομένου ενός συνόλου στοιχείων είναι συχνά χρήσιμο (όπως θα δούμε με παραδείγματα στη συνέχεια) να χωρίσουμε τα στοιχεία αυτά σε διαφορετικές μη επικαλυπτόμενες ομάδες. Η δομή δεδομένων disjoint-set είναι μια κατάλληλη δομή για την διατήρηση ενός τέτοιου διαχωρισμού. Κάθε μία από τις ομάδες αναγνωρίζεται από ένα στοιχείο της το οποίο ονομάζεται αντιπρόσωπος (representative) της ομάδας, και το οποίο εν γένει μπορεί να είναι οποιοδήποτε στοιχείο της ομάδας. Οι βασικές πράξεις οι οποίες υποστηρίζονται από μια τέτοια δομή είναι οι εξής:

- FIND(x): Δεδομένου ενός στοιχείου x απαντά τον αντιπρόσωπο της ομάδας

στην οποία ανήκει το x.

- UNION(x, y): Ενώνει τις δύο ομάδες / σύνολα που έχουν αντιπροσώπους τα x και y αντίστοιχα, σε μία νέα ομάδα. Αντιπρόσωπος της νέας ομάδας μπορεί να είναι οποιοδήποτε στοιχείο των δύο ομάδων αν και συνήθως επιλέγεται ένας εκ των x και y.

- MAKE-SET(x): Δημιουργεί μια νέα ομάδα η οποία περιέχει το στοιχείο x (το οποίο γίνεται και αντιπρόσωπος της ομάδας).

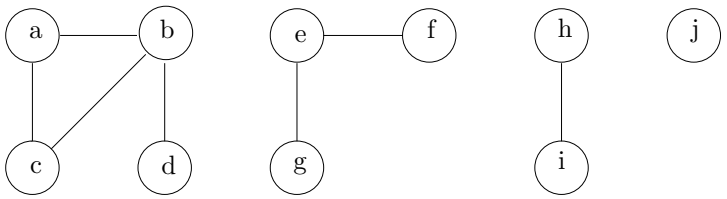


Σχήμα 6.16: Ένα παράδειγμα χρήσης των UNION και FIND.

Ένα παράδειγμα των πράξεων UNION και FIND φαίνεται στο σχήμα 6.16. Για την ένωση των ομάδων που περιέχουν τα στοιχεία 5 (αριστερή) και 7 (δεξιά) χρησιμοποιούμε τις UNION και FIND ως εξής: FIND$(5) = 6$, FIND$(7) = 9$ και UNION$(6, 9) = 9$ οπότε προκύπτει τελικά η ομάδα του σχήματος.

Μία από τις πολλές εφαρμογές των ασύνδετων συνόλων είναι η εύρεση των συνεκτικών συνιστωσών ενός γράφου. Ας θεωρήσουμε το γράφο του σχήματος 6.17.

Χρησιμοποιούμε τον αλγόριθμο του σχήματος 6.18 για να δημιουργήσουμε τη δομή ασύνδετων συνόλων η οποία θα μας βοηθήσει στην εύρεση των συνεκτικών συνιστωσών του γράφου.



Σχήμα 6.17: Ένας γράφος με τέσσερις συνεκτικές συνιστώσες: $\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}$.

CONNECTED-COMPONENTS(G)
1. **for each** $v \in V$
2. MAKE-SET(v)
3. **for each** $(u, v) \in E$
4. $a =$ FIND(u)
5. $b =$ FIND(v)
6. **if** $a \neq b$ **then** UNION(a, b)
7. **end for**

Σχήμα 6.18: Αλγόριθμος Εύρεσης Συνεκτικών Συνιστωσών με χρήση της δομής ασύνδετων συνόλων

Στον πίνακα του σχήματος 6.19 φαίνεται η κατασκευή των συνόλων καθώς εκτελείται η CONNECTED-COMPONENTS.

Πλευρά	Συλλογή από ασύνδετα σύνολα									
αρχικά σύνολα	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

Σχήμα 6.19: Κατασκευή των συνόλων για τον γράφο του σχήματος 6.17.

Έχοντας λοιπόν κατασκευάσει τα σύνολα, μπορούμε να βρούμε εύκολα αν δύο κόμβοι ανήκουν στην ίδια συνεκτική συνιστώσα, ελέγχοντας απλά εάν οι αντιπρόσωποι των ομάδων τους (το αποτέλεσμα της FIND σε κάθε έναν) ταυτίζονται.

Ο χρόνος εκτέλεσης της πράξης UNION είναι σταθερός αφού η πρόσβαση σε κάθε στοιχείο γίνεται σε σταθερό χρόνο και η πράξη UNION κάνει μια απλή ανάθεση δείκτη. Όσο για την πράξη FIND, μπορεί να εκτελεσθεί σε γραμμικό χρόνο στην περίπτωση που η αναπαράσταση των στοιχείων ενός συνόλου έχει εκφυλιστεί σε γραμμική λίστα. Εάν όμως κατά την ένωση δύο συνόλων επιλέγουμε να ενώνουμε το σύνολο με τα λιγότερα στοιχεία σε αυτό με τα περισσότερα στοιχεία τότε ο χρόνος εκτέλεσης της FIND γίνεται $O(\log n)$ (άσκηση).

6.5 Ελάχιστα Μονοπάτια

Το πρόβλημα εύρεσης του συντομότερου μονοπατιού σε έναν κατευθυνόμενο γράφο με βάρη, αναφέρεται στην εύρεση του μονοπατιού από έναν κόμβο αφετηρία, σε έναν κόμβο τερματισμού με το ελάχιστο συνολικό βάρος.

Το πρόβλημα αυτό είναι ένα από τα παλαιότερα διατυπωμένα και πιο σημαντικά της Θεωρίας Γράφων. Παρουσιάζει πληθώρα εφαρμογών στην Επιχειρησιακή Έρευνα, στα Δίκτυα, στο Διαδίκτυο, κ.α. ενώ εμφανίζεται συχνότατα σαν υποπρόβλημα σε άλλα ενδιαφέροντα προβλήματα.

Έστω $G = (V, E, W)$ ο γράφος, με V το σύνολο των κόμβων, E το σύνολο των ακμών του και W η συνάρτηση βαρών $w(i, j)$ που απεικονίζει την ακμή $[i, j]$ σε έναν πραγματικό αριθμό.

Θα συμβολίζουμε ένα μονοπάτι με $p = \langle u_0, u_1, \ldots, u_k \rangle$, με $u_i \in V$ και το κόστος του μονοπατιού είναι: $w(p) = \sum_{i=1}^{k} w(u_{i-1}, u_i)$. Αν s είναι ο κόμβος αφετηρία και t ο κόμβος τερματισμού, ο στόχος μας είναι να βρούμε το μονοπάτι $p^*_{s,t} = \langle s, \ldots, t \rangle$ με ελάχιστο βάρος.

Υπάρχουν οι εξής παραλλαγές του προβλήματος αυτού:

- **Εύρεση όλων των Ελάχιστων Μονοπάτιων από μία Αφετηρία (Single Source Shortest Paths)**: Εδώ μας δίνεται ο κόμβος αφετηρία s και ζητούνται τα ελάχιστα μονοπάτια προς όλους του άλλους κόμβους του γράφου.

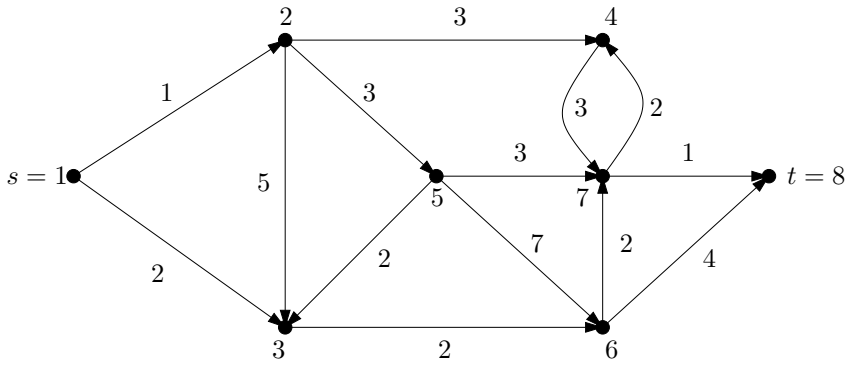
  Για το πρόβλημα αυτό, θα περιγράψουμε τους αλγορίθμους των Dijkstra και Bellman.

- **Εύρεση Ελαχίστων Μονοπατιών για όλα τα Ζευγάρια Κόμβων (All Pairs Shortest Paths)**: Για κάθε $s \in V$ και για κάθε $t \in V$ βρες το ελάχιστο μονοπάτι $p^*_{s,t}$.

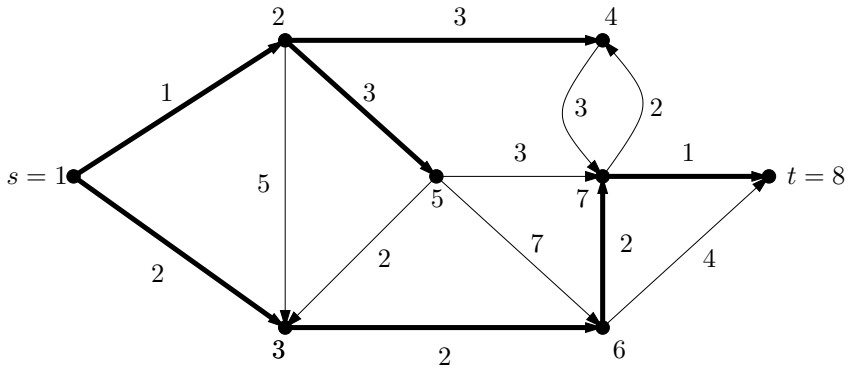
  Γνωστότεροι αλγόριθμοι που λύνουν το πρόβλημα αυτό, είναι ο αλγόριθμος των Floyd-Warshall πολυπλοκότητας $O(n^3)$ και ο αλγόριθμος του Johnson για αραιούς γράφους πολυπλοκότητας $O(n^2 \log n)$.

Το πρόβλημα εύρεσης του μονοπατιού για μία δεδομένη αφετηρία και τερματισμό, αντιμετωπίζεται με το να βρεθούν τα συντομότερα μονοπάτια από την αφετηρία προς οποιονδήποτε άλλο κόμβο.

Έστω για παράδειγμα ο γράφος του σχήματος 6.20. Τα συντομότερα μονοπάτια, από τον κόμβο αφετηρία s προς κάθε άλλο κόμβο φαίνονται στο σχήμα 6.21.



Σχήμα 6.20: Κατευθυνόμενος γράφος $G = (V, E)$ όπου s ο κόμβος αφετηρία και t ο κόμβος τερματισμού



Σχήμα 6.21: Το δένδρο συντομότερων μονοπατιών με έντονες πλευρές στον γράφο $G = (V, E)$

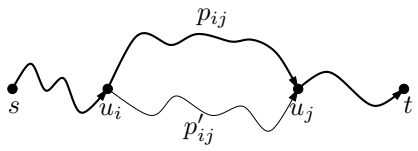
Στο σχήμα φαίνεται ότι οι λύσεις δημιουργούν ένα δένδρο συντομότερων μονοπατιών. Για παράδειγμα το βέλτιστο κόστος από το s στο t είναι $w_{13} + w_{36} + w_{67} + w_{78} = 7$

Αν παρατηρήσουμε το δένδρο συντομότερων μονοπατιών θα δούμε ότι κάθε υπο-μονοπάτι που συναντάται είναι το ίδιο βέλτιστο. Για παράδειγμα το μονοπάτι

από τον κόμβο 3 στον κόμβο 7 είναι το καλύτερο δυνατό μονοπάτι με αφετηρία τον κόμβο 3 και τερματισμό τον κόμβο 7. Το γεγονός αυτό δεν είναι τυχαίο. Οφείλεται στο ότι το δένδρο συντομότερων μονοπατιών μπορεί να διασπαστεί με βέλτιστο τρόπο. Θα αποδείξουμε αυτόν τον ισχυρισμό:

Βέλτιστα διασπώμενη δομή του δένδρου συντομότερων μονοπατιών

Έστω ότι το ελάχιστο μονοπάτι από τον κόμβο s στον κόμβο t είναι το $p^*_{st} = \langle s, u_1, \ldots, u_i, \ldots, u_j, \ldots, u_k, t\rangle$. Τότε κάθε υπο-μονοπάτι αυτού $p_{ij} = \langle u_i, \ldots, u_j\rangle$ είναι το ελάχιστο μονοπάτι από τον κόμβο u_i στο u_j:
Πράγματι, το κόστος του (ελάχιστου) μονοπατιού από τον s στον t είναι: $w(p_{s,i}) + w(p_{i,j}) + w(p_{j,t})$.



Σχήμα 6.22: Το μονοπάτι p'_{ij} είναι μικρότερου βάρους από το μονοπάτι p_{ij}

Αν υπήρχε μονοπάτι $p'_{i,j}$ με $w(p'_{i,j}) < w(p_{i,j})$, τότε και το μονοπάτι $p_{s,t}$ (βλ. σχήμα 6.22) δεν θα ήταν βέλτιστο, γιατί αντικαθιστώντας το $p_{i,j}$ με το $p'_{i,j}$ θα κερδίζαμε κάτι, πράγμα που έρχεται σε αντίφαση με τη δήλωση μας ότι το $p_{s,t}$ έχει ελάχιστο βάρος.

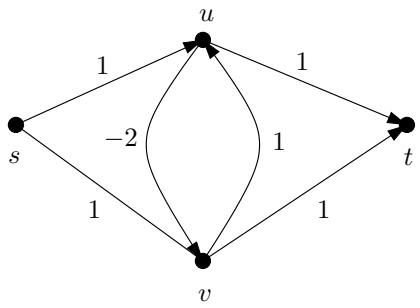
Θα πρέπει ωστόσο να κάνουμε δύο βασικές παρατηρήσεις σχετικά με τα βάρη και την δομή των γράφων εισόδου:

Ακμές αρνητικού βάρους: Επειδή είναι δυνατόν να έχουμε ακμές με αρνητικό βάρος, πρέπει να εξετάσουμε αν υπάρχουν κύκλοι στον γράφο με αρνητικό βάρος.

Αν δεν υπάρχουν κύκλοι με αρνητικό βάρος, το πρόβλημα παραμένει καλά διατυπωμένο ακόμη κι αν επιστρέφει λύση με αρνητικό συνολικό βάρος. Αν ωστόσο υπάρχει κύκλος με αρνητικό συνολικό βάρος, όπως στο σχήμα 6.23, τότε δεν μπορεί να βρεθεί συντομότερο μονοπάτι από τον κόμβο s στον κόμβο t, γιατί για οποιοδήποτε μονοπάτι μπορούμε να βρούμε ένα με μικρότερο βάρος κάνοντας ένα ακόμη πέρασμα από τον κύκλο αρνητικού βάρους.

Άρα το συντομότερο μονοπάτι από το s στο t για αυτήν την ειδική περίπτωση έχει βάρος $-\infty$. Αν ο γράφος δεν περιέχει κύκλους αρνητικού βάρους τότε οι αρνητικές ακμές που υπάρχουν δεν μας ενοχλούν.

Κύκλοι: Όπως είδαμε ένα συντομότερο μονοπάτι δεν μπορεί να περιέχει κύκλο αρνητικού βάρους. Επίσης όμως δεν μπορεί να περιέχει κύκλο θετικού βάρους γιατί αφαιρώντας τον, θα κερδίζαμε κάτι στο βάρος του μονοπατιού. Έτσι απομένουν μονάχα οι κύκλοι με μηδενικό βάρος. Άν το μονοπάτι περιέχει κύκλο μηδενικού βάρους, τότε αφαιρώντας τον πέρνουμε ένα μονοπάτι με το ίδιο βάρος



Σχήμα 6.23: Το μονοπάτι $s \to u \to v \to t$ περιέχει έναν κύκλο αρνητικού βάρους

και λιγότερους κόμβους. Επαναλαμβάνοντας μπορούμε να καταλήξουμε σε ένα μονοπάτι που δεν έχει κύκλους. Με αυτήν την συλλογιστική καταλήγουμε ότι ένα συντομότερο μονοπάτι δεν περιέχει κύκλους. Έτσι αφού κάθε ακυκλικό μονοπάτι σε έναν γράφο μπορεί να έχει το πολύ $|V|-1$ ακμές, μπορούμε να επικεντρώσουμε την προσοχή μας, μονάχα σε μονοπάτια που έχουν το πολύ $|V|-1$ ακμές.

6.5.1 Ο αλγόριθμος του Dijkstra

Ο πρώτος αλγόριθμος που θα δούμε λύνει το πρόβλημα εύρεσης όλων των μονοπατιών από μία δεδομένη αφετηρία, με την προϋπόθεση ότι δεν υπάρχουν ακμές αρνητικού βάρους. Ο αλγόριθμος αυτός, που προτάθηκε από τον E.W. Dijkstra το 1954, στηρίζεται στην παρατήρηση της βέλτιστα διασπώμενης δομής του δένδρου συντομότερων μονοπατιών. Συγκεκριμένα, ξεκινώντας από τον κόμβο αφετηρία $s \in V$, σε κάθε βήμα θα οριστικοποιείται το βέλτιστο μονοπάτι προς έναν κόμβο. Επειδή σε κάθε βήμα οριστικοποιείται και ένας κόμβος (μία ετικέτα του κόμβου) ο αλγόριθμος ανήκει στην γενικότερη κατηγορία οριστικοποίησης ετικετών.

Τις ετικέτες διατηρεί ένας πίνακας $FIXE[x]$, $x = 1 \ldots n$ που δείχνει για ποιους κόμβους έχουμε βρει το συντομότερο μονοπάτι. Δηλαδή, αν $FIXE[i] = TRUE$ τότε ο κόμβος i έχει οριστικοποιηθεί. Για έναν μη οριστικοποιημένο κόμβο i του γράφου ($FIXE[i] = FALSE$) διατηρούμε δύο τιμές, την $V[i]$ (κόστος μονοπατιού) που δείχνει ποια είναι η τιμή του βέλτιστου μονοπατιου προς τον κόμβο i, στο τρέχον βήμα του αλγορίθμου, και την $P[i]$ (πατέρας) που δείχνει από ποιον κόμβο φθάνουμε στον κόμβο i σε αυτό το βέλτιστο μονοπάτι. Για έναν κόμβο i που δεν είναι προσπελάσιμος από τους φιξαρισμένους κόμβους θέτουμε $P[i] = 0$ και $V[i] = +\infty$.

Ο αλγόριθμος σε κάθε βήμα επιλέγει να φιξάρει εκείνο τον κόμβο που έχει

το ελάχιστο μονοπάτι από τον κόμβο αφετηρία, ανάμεσα σε όλους τους μη οριστικοποιημένους κόμβους του γράφου. Στη συνέχεια, ενημερώνει τους πίνακες P, V και $FIXE$ με τις αναθεωρημένες τιμές τους.

ΑΛΓΟΡΙΘΜΟΣ DIJKSTRA($G = (V, E, W), s \in V$)

1. Αρχικοποίησε $FIXE$ σε $FALSE$
2. Αρχικοποίησε P σε 0
3. Αρχικοποίησε V σε ∞
4. Θέσε $V[s] = 0$, $P[s] = s$
5. **repeat**
6. Επέλεξε έναν κόμβο x ΜΗ φιξαρισμένο και με ελάχιστο $V[x]$
7. **if** $V[x] < \infty$ **then**
8. Θέσε $FIXE[x] = TRUE$
9. **for each** y επόμενος του x και ΜΗ φιξαρισμένος
10. **if** $V[x] + W(x, y) < V[y]$ **then**
11. $V[y] = V[x] + w[x, y]$
12. $P[y] = x$
13. **end if**
14. **end for**
15. **end if**
16. **until** $\{x|V[x] < \infty\} = \emptyset$

Σχήμα 6.24: Αλγόριθμος εύρεσης συντομότερων μονοπατιών με αφετηρία τον κόμβο s. Αν το σύνολο $\{x|V[x] = \infty\}$ είναι μη κενό, τότε ο γράφος είναι μη συνεκτικός

Θα κάνουμε μία εφαρμογή του αλγορίθμου στον γράφο του σχήματος 6.20 παρακολουθώντας την εξέλιξη των πινάκων που δημιουργούνται στον πίνακα 6.1. Επίσης στο σχήμα 6.25 φαίνεται ότι το δένδρο συντομότερων μονοπατιών εξαπλώνεται σαν μια κηλίδα, που σε κάθε βήμα προστίθεται και ένας κόμβος.

Ανάλυση Πολυπλοκότητας

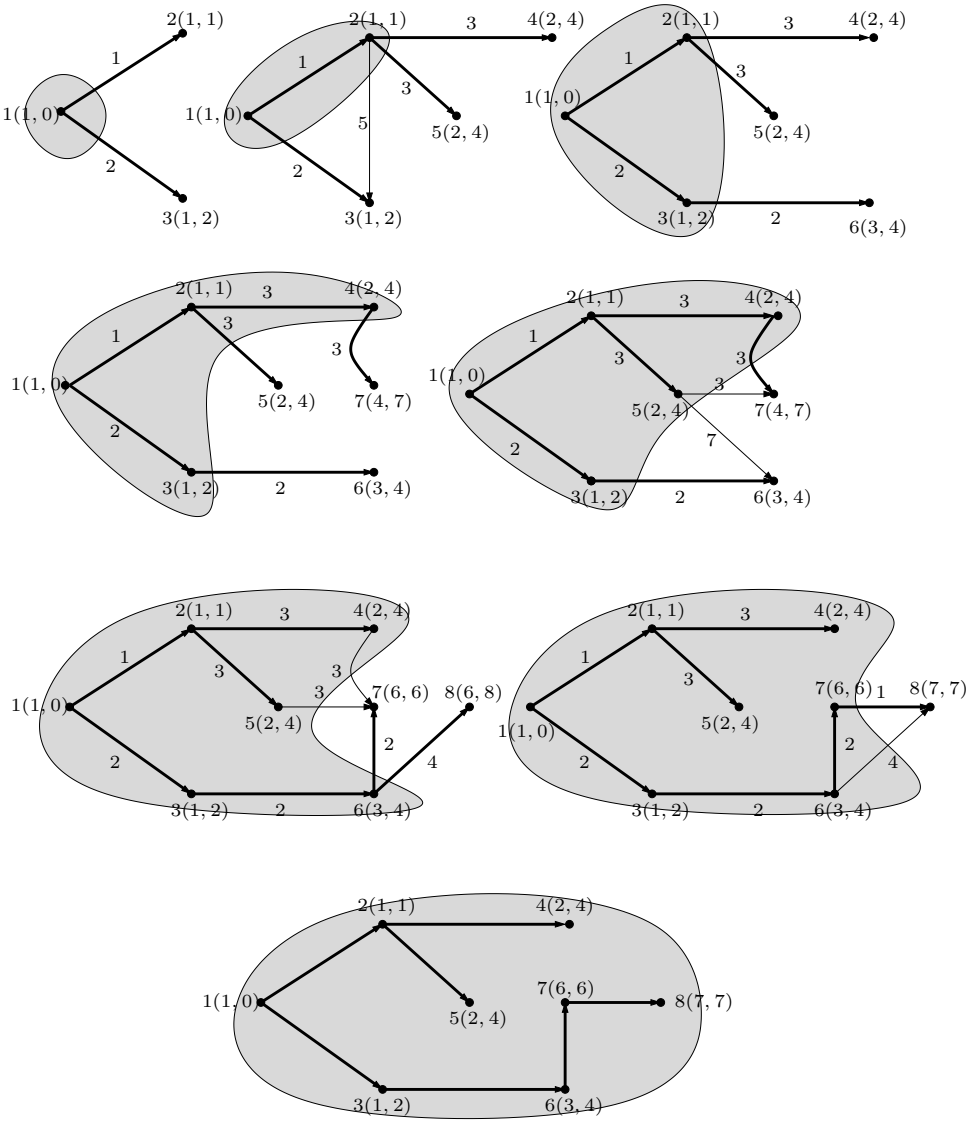
Σε κάθε βήμα οριστικοποιείται ένας κόμβος του γράφου. Άρα ο εξωτερικός βρόγχος (Repeat...Until) θα τρέξει n φορές. Στο εσωτερικό του βρόγχου επιλέγεται πρώτα ο κόμβος που είναι πιο κοντά σε σχέση με τους κόμβους που έχουν οριστικοποιηθεί. Η επιλογή του κόμβου γίνεται σε $O(n)$ χρόνο. Έπειτα γίνεται ενημέρωση των πινάκων παρακολούθησης P και V για κάθε επόμενο του κόμβου που επιλέχθηκε σε χρόνο $\mathcal{O}(d^+(v)) = \mathcal{O}(n)$. Άρα στον εσωτερικό βρόγχο η επιλογή και η ενημέρωση γίνεται σε $\mathcal{O}(n)$. Άρα τελικά η πολυπλοκότητα του αλγορίθμου είναι $O(n^2)$.

Ωστόσο μπορούμε να επιτύχουμε καλύτερο χρόνο, αν οργανώσουμε τον πίνακα V σε δομή σωρού. Ο πίνακας V χρησιμοποιείται για την επιλογή σε κάθε βήμα εκείνου του κόμβου που απέχει λιγότερο από την αφετηρία και καθότι χρειαζόμαστε σε κάθε βήμα τον μικρότερο, η οργάνωση σε δομή σωρού, θα μας επιτρέψει να κάνουμε την επιλογή σε $O(logn)$ (επιλογή και αναδιοργάνωση σωρού) και την εισαγωγή και αναδιοργάνωση σε $O(1)$. Με την εκλέπτυνση αυτή, ο χρόνος του

		1	2	3	4	5	6	7	8
Βήμα 1	FIXE	**1**	0	0	0	0	0	0	0
	P	1	**1**	**1**	0	0	0	0	0
	V	0	**1**	**2**	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Βήμα 2	FIXE	1	**1**	0	0	0	0	0	0
	P	1	1	1	**2**	**2**	0	0	0
	V	0	1	2	**4**	**4**	$+\infty$	$+\infty$	$+\infty$
Βήμα 3	FIXE	1	1	**1**	0	0	0	0	0
	P	1	1	1	2	2	**3**	0	0
	V	0	1	2	4	4	**4**	$+\infty$	$+\infty$
Βήμα 4	FIXE	1	1	1	**1**	0	0	0	0
	P	1	1	1	2	2	3	**4**	0
	V	0	1	2	4	4	4	**7**	$+\infty$
Βήμα 5	FIXE	1	1	1	1	**1**	0	0	0
	P	1	1	1	2	2	3	4	0
	V	0	1	2	4	4	4	7	$+\infty$
Βήμα 6	FIXE	1	1	1	1	1	**1**	0	0
	P	1	1	1	2	2	3	**6**	**6**
	V	0	1	2	4	4	4	**6**	**8**
Βήμα 7	FIXE	1	1	1	1	1	1	**1**	0
	P	1	1	1	2	2	3	6	**7**
	V	0	1	2	4	4	4	6	**7**
Βήμα 8	FIXE	1	1	1	1	1	1	1	**1**
	P	1	1	1	2	2	3	6	7
	V	0	1	2	4	4	4	6	7

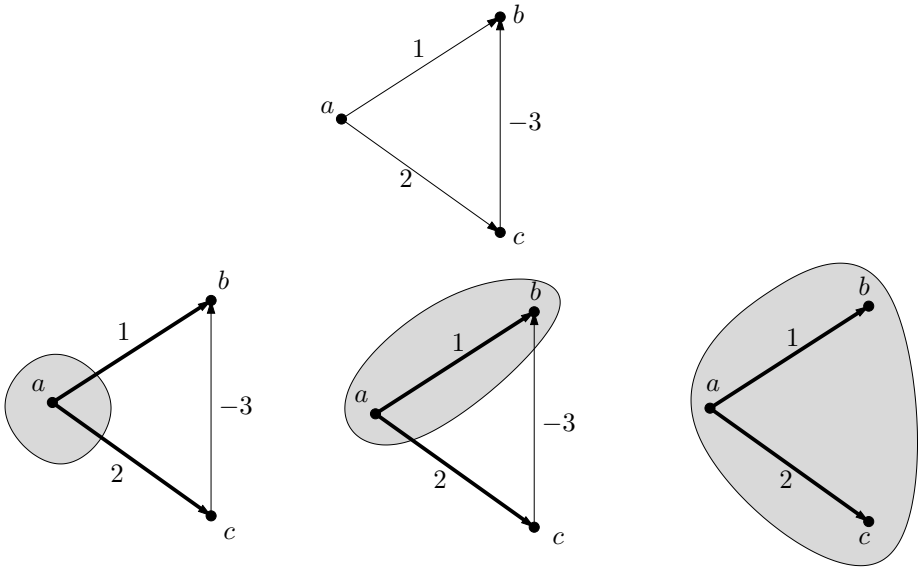
Πίνακας 6.1: Η εκτέλεση του αλγορίθμου Dijkstra στο παράδειγμα του σχήματος 6.20

αλγορίθμου πέφτει σε $O(nlogn + m)$ που για αραιούς γράφους ($m = O(n)$) είναι ανάλογο του $O(nlogn)$, ενώ για πυκνούς γράφους παραμένει $O(n^2)$ (θεωρώντας $m = O(n^2)$).



Σχήμα 6.25: Παράδειγμα Εκτέλεσης αλγορίθμου Dijkstra. Σε κάθε κόμβο πρώτος αριθμός είναι ο κόμβος και στο ζευγάρι που ακολουθεί, πρώτος αριθμός είναι ο πατέρας και δεύτερος η τιμή του συντομότερου μονοπατιού από την αφετηρία $s = 1$.

Ακμές αρνητικού βάρους Ο αλγόριθμος δεν δουλεύει αν υπάρχουν ακμές αρνητικού βάρους στον γράφο. Αυτό συμβαίνει γιατί όταν οριστικοποιείται ένας κόμβος στον γράφο, δεν γίνεται ποτέ να επιστρέψουμε για να διορθώσουμε το μονοπάτι που βρέθηκε αρχικά. Η οριστικοποίηση αυτή, καθιστά αδύνατη την διόρθωση των μονοπατιών που μπορούν να δημιουργηθούν με χρησιμοποιήση αρνητικών ακμών. Τα συμπεράσματα αυτά φαίνονται στον γράφο που ακολουθεί με αφετηρία τον κόμβο a (βλ. σχήμα 6.26)



Σχήμα 6.26: Ο αλγόριθμος Dijkstra δεν δουλεύει με ακμές αρνητικού βάρους. Το συντομότερο μονοπάτι είναι $a \to c \to b$ από τον κόμβο a στον κόμβο b με συνολικό βάρος -1 και όχι $a \to b$ βάρους 1 που δίνει ο αλγόριθμος.

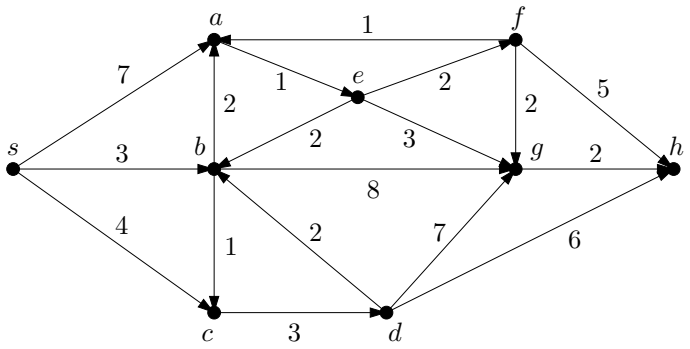
6.5.2 Ο αλγόριθμος του Bellman

Ο αλγόριθμος του Bellman είναι ένας αποδοτικός αλγόριθμος για την εύρεση των συντομότερων μονοπατιών από την αφετηρία σε κάθε άλλο κόμβο του γράφου. Ο αλγόριθμος αυτός ανήκει στην γενική κατηγορία των αλγορίθμων τροποποίησης ετικετών, όπου σε κάθε κόμβο αντιστοιχίζεται μία ετικέτα που τροποποιείται κατά τη διάρκεια εκτέλεσης του αλγορίθμου.

Όπως δείξαμε σε προηγούμενη ενότητα κάθε βέλτιστο μονοπάτι, έχει το πολύ $n-1$ ακμές. Ο αλγόριθμος του Bellman κατασκευάζει διαδοχικά όλα τα μονοπάτια με $1, 2, \ldots, n-1$ ακμές που ξεκινούν από τον κόμβο - αφετηρία s. Κατά την

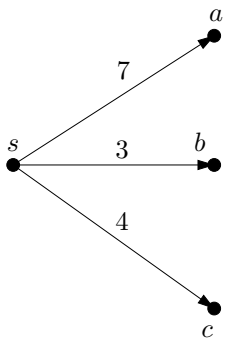
κατασκευή αυτή, αν βρει ένα καλύτερο μονοπάτι προς κάποιο κόμβο στόχο, τότε διορθώνει την ετικέτα του κόμβου - στόχου, που διατηρεί το κόστος του καλύτερου μονοπατιού που είχε βρεθεί μέχρι τότε.

Θα δούμε την εκτέλεση της ιδέας αυτής στον γράφο - παράδειγμα του σχήματος 6.27.



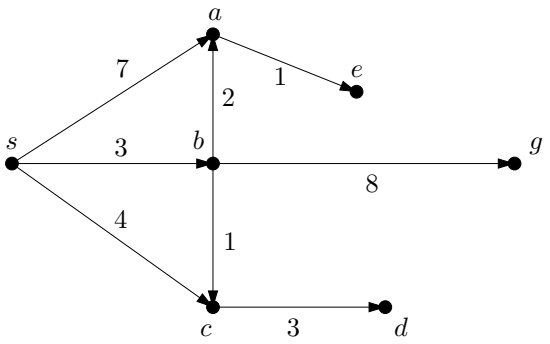
Σχήμα 6.27: Κατευθυνόμενος γράφος με 9 κόμβους και s η αφετηρία

Στο πρώτο βήματα υπολογίζονται τα συντομότερα μονοπάτια μίας ακμής ξεκινώντας από τον κόμβο αφετηρία (σχήμα 6.28).



Σχήμα 6.28: Μονοπάτια μήκους 1.

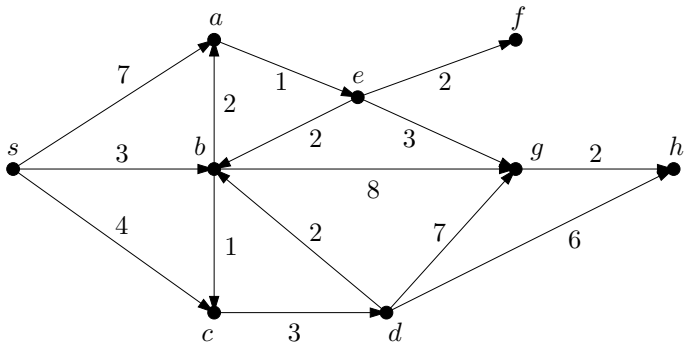
Για τους κόμβους a, b, c έχει βρεθεί ένα μονοπάτι από τον κόμβο αφετηρία. Οι αντίστοιχες ετικέτες τους είναι $7, 3, 4$ που υποδηλώνουν το μήκος του μονοπατιού που έχει βρεθεί. Στο δεύτερο βήμα του αλγορίθμου εξετάζονται τα μονοπάτια μήκους 2 (σχήμα 6.29).



Σχήμα 6.29: Μονοπάτια μήκους 2.

Η εξερεύνηση αυτή είτε τροποποιεί τις ετικέτες, είτε τις αφήνει αμετάβλητες. Για τον κόμβο a έχει βρεθεί καλύτερο μονοπάτι, το $\langle s, b, a\rangle$ με κόστος 5, για τους b και c οι ετικέτες μένουν 3 και 4 αντίστοιχα, ενώ για πρώτη φορά βρίσκονται μονοπάτια για τους d, e, g με αντίστοιχες ετικέτες $8, 11, 7$[2]. Στο επόμενο βήμα εξετάζονται τα μονοπάτια μήκους 3(σχήμα 6.30).

[2]Παρατηρούμε ότι η ετικέτα του e είναι 8 που είναι το συντομότερο μονοπάτι με 2 ακμές και όχι 6 που είναι το μονοπάτι με 3 ακμές



Σχήμα 6.30: Μονοπάτια μήκους 3.

Για τους κόμβους a, b, c δεν παρατηρείται βελτίωση. Για τον κόμβο e βρίσκουμε καλύτερο μονοπάτι, το $\langle s, b, a, e\rangle$ με κόστος 6. Για τους κόμβους d, g δεν βρίσκουμε κάτι καλύτερο, ενώ προστίθενται οι κόμβοι f, h με ετικέτες 10 και 13. Ο αλγόριθμος θα συνεχίσει για μονοπάτια μεγέθους $4, \ldots, 8$.

Όπως γνωρίζουμε, ένας γράφος έχει εκθετικό πλήθος μονοπατιών, οπότε η διαδοχική κατασκευή των μονοπατιών μήκους $1, 2, \ldots, n-1$ θα έδινε έναν κακό (εκθετικό) αλγόριθμο. Ωστόσο δεν είναι ανάγκη να κατασκευάζονται εξαντλητικά όλα τα μονοπάτια. Αρκεί να παρατηρήσουμε ότι στο παραπάνω αλγοριθμικό σχήμα υπεισέρχονται τρεις περιπτώσεις:

- Βρίσκονται κόμβοι στους οποίους δεν είχαμε ξαναφτάσει. Όπως για παράδειγμα στην εξερεύνηση μονοπατιών μήκους 2 που φτάσαμε για πρώτη φορά στους κόμβους d, e, g. Σε αυτήν την περίπτωση πρέπει να εξετάζουμε κάθε μη εισαχθέντα κόμβο για το αν ένας προηγούμενος του έχει μπεί στο δένδρο μέχρι εκείνη τη στιγμή.

- Βρίσκονται καλύτερα μονοπάτια προς κόμβους που έχουμε ξαναφτάσει. Όπως για παράδειγμα στον κόμβο e κατά την εξερεύνηση μονοπατιών μήκους 3. Σε αυτήν την περίπτωση πρέπει να ελέγχεται ο προηγούμενος για το αν έχει βελτιωθεί η τιμή του στο προηγούμενο βήμα.

- Βρίσκονται νέα μονοπάτια προς κάποιον κόμβο, αλλά αυτά τα μονοπάτια είναι χειρότερα από αυτά που έχουν βρεθεί.

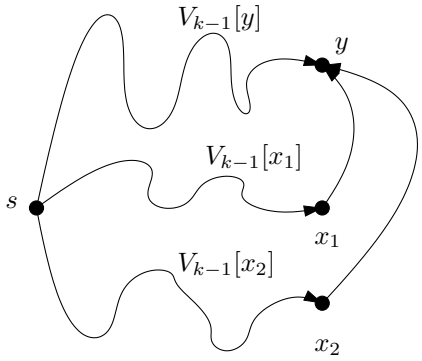
Αυτά μας οδηγούν στο συμπέρασμα, ότι όταν ελέγχονται μονοπάτια μήκους k για κάθε κόμβο πρέπει να ελέγχονται οι προηγούμενοί του. Η παρατήρηση αυτή μας οδηγεί στην κάτωθι αναδρομική σχέση για τις ετικέτες που συντηρούνται:

$$V_k(y) = \min_x \{V_{k-1}(y), V_{k-1}(x) + W(x, y)\}$$

με x κάθε προηγούμενος του y. Η σχέση αυτή δηλώνει ότι το συντομότερο μονοπάτι με το πολύ k ακμές από τον s προς τον y είναι το μικρότερο από τα εξής:

- Το βάρος του συντομότερου μονοπατιού με το πολύ $k-1$ ακμές
- Το κόστος του συντομότερου μονοπατιου με $k-1$ ακμές που καταλήγει σε έναν x προηγούμενο του y συν το βάρος της ακμής $[x, y]$.

Οι δύο αυτές περιπτώσεις συνοψίζονται στο σχήμα 6.31:



Σχήμα 6.31: Το συντομότερο μονοπάτι από τον s στον κόμβο y είναι αυτό με την μικρότερη τιμή ανάμεσα στα $V_{k-1}[y]$, $V_{k-1}[x_1]+W(x_1, y)$ και $V_{k-1}[x_2]+W(x_2, y)$

Για την υλοποίηση του αλγορίθμου χρησιμοποιείται ένας διδιαστατος πίνακας $V[k][j]$ που δείχνει ποιο είναι το κόστος του συντομότερου μονοπατιού ξεκινώντας από τον κόμβο αφετηρία προς τον κόμβο $j(j = 1 \ldots n)$ χρησιμοποιώντας το πολύ $k(k = 1 \ldots n-1)$ ακμές. Η αρχικοποίηση των κόμβων πλην της αφετηρίας, γίνεται με $+\infty$. Επιπλέον διατηρείται ένας βοηθητικός πίνακας P (''πατέρας'') όπου το $P[i]$ δείχνει από ποιον κόμβο φτάσαμε στον κόμβο i. Με βάση τα παραπάνω έχουμε την υλοποίηση σε ψευδογλώσσα, όπως φαίνεται στο σχήμα 6.32.

Παρατηρήστε ότι στο βήμα που εξετάζουμε τα μονοπάτια μεγέθους k απαιτούμε να έχει βελτιωθεί το μονοπάτι προς τουλάχιστον έναν κόμβο (μεταβλητή *stable*). Αν αυτό δεν επιτευχθεί σημαίνει ότι ο αλγόριθμος έχει παγώσει (δηλαδή κανένας κόμβος δεν πρόκειται να αλλάξει ετικέτα) και δεν υπάρχει λόγος να εξετάσουμε τα μονοπάτια με πλήθος ακμών $k+1, \ldots, n-1$. Άρα και ότι η τιμή του k στο τέλος του αλγόριθμου υποδεικνύει και το μεγιστο μήκος των μονοπατιών που βρέθηκαν.

Αν για έναν κόμβο u, ισχύει ότι $V[n-1][u] = +\infty$, τότε προφανώς ο κόμβος αυτός δεν είναι προσπελάσιμος από τον κόμβο αφετηρία.

Ας δούμε την εκτέλεση του αλγορίθμου στον γράφο - παράδειγμα του σχήματος 6.27. Το ζεύγος (α, λ) σε κάθε κελί του πίνακα υποδεικνύει την κατάσταση των

```
Αλγοριθμος Bellman(G = (V, E, W), s ∈ V)
 1.  Αρχικοποίησε P[i], i = 1 ... n σε 0
 2.  Αρχικοποίησε V[0][i], i = 1 ... n σε +∞
 3.  Θέσε V[0][s] = 0, P[s] = s, k = 0
 4.  repeat
 5.      k = k + 1
 6.      for i = 1 to n
 7.          Θέσε V[k][i] = V[k − 1][i]
 8.      stable = TRUE
 9.      for y = 1 το n
10.          for each x προηγούμενο του y
11.              if V[k − 1][x] ≠ +∞ and V[k][y] > V[k − 1][x] + W[x, y]
12.                  V[k][y] = V[k − 1][x] + W[x, y]
13.                  P[y] = x
14.                  stable = FALSE
15.              end if
16.          end for
17.      end for
18.  until stable or k = n − 1
```

Σχήμα 6.32: Αλγόριθμος του Bellman στο γράφο G ξεκινώντας από τον κόμβο s

πινάκων στο τρέχον βήμα του αλγορίθμου. Το λ είναι το κόστος του καλύτερου μονοπατιού μέ το πολύ k ακμές ($V[k][y]$) και το $\alpha = P(y)$ είναι ο κόμβος από τον οποίο φτάσαμε στον κόμβο στόχο y. Με έντονα γράμματα έχουν σημειωθεί τα σημεία που παρουσιάστηκε βελτίωση στο κόστος.

Η εκτέλεση του αλγορίθμου στο παράδειγμα									
	s	a	b	c	d	e	f	g	h
$k=0$	$s,0$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$
$k=1$	$s,0$	$\mathbf{s,7}$	$\mathbf{s,3}$	$\mathbf{s,4}$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$	$s,+\infty$
$k=2$	$s,0$	$\mathbf{b,5}$	$s,3$	$s,4$	$\mathbf{c,7}$	$\mathbf{a,8}$	$s,+\infty$	$\mathbf{b,11}$	$s,+\infty$
$k=3$	$s,0$	$b,5$	$s,3$	$s,4$	$c,7$	$\mathbf{a,6}$	$\mathbf{e,10}$	$\mathbf{e,9}$	$\mathbf{d,13}$
$k=4$	$s,0$	$b,5$	$s,3$	$s,4$	$c,7$	$a,6$	$\mathbf{e,8}$	$e,9$	$\mathbf{g,11}$
$k=5$	$s,0$	$b,5$	$s,3$	$s,4$	$c,7$	$a,6$	$e,8$	$e,9$	$g,11$

Ανάλυση Πολυπλοκότητας

Στον εξωτερικό βρόγχο, ο αλγόριθμος κάνει μία επανάληψη από $1,\ldots,n-1$ εξετάζοντας τα αντίστοιχου μήκους μονοπάτια. Σε κάθε επανάληψη, ελέγχονται οι προηγούμενοι κάθε κόμβου δηλαδή συνολικά όλες οι ακμές του γράφου. Πράγματι, αν απεικονίσουμε τον γράφο με λίστα γειτνίασης προηγουμένων κόμβων, η ανάκτηση των προηγουμένων ενός κόμβου απαιτεί χρόνο $O(d^-(y))$ και η συνολική πολυπλοκότητα γίνεται $O(nm)$. Η μνήμη που απαιτεί είναι $\Theta(n^2)$, αλλά μπορεί να

βελτιωθεί αν παρατηρήσουμε ότι σε κάθε βήμα χρειάζονται μόνο οι δύο τελευταίες γραμμές του πίνακα. Με τον τρόπο αυτό η συνολική μνήμη πέφτει σε $\Theta(n)$.

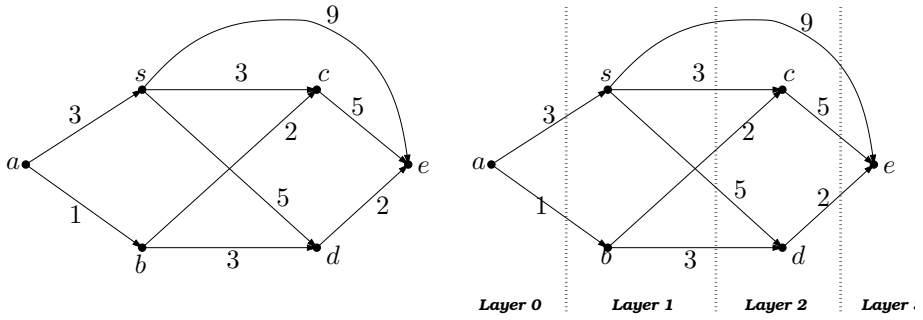
Τελικά, η εύρεση του μονοπατιού χρησιμοποιεί τον ''πατέρα'' κάθε κόμβου μέχρι τον κόμβο αφετηρία. Π.χ. για τον κόμβο h (βλ. σχήμα 6.27 και πίνακα εκτέλεσης του αλγορίθμου) έχουμε $h \leftarrow g \leftarrow e \leftarrow a \leftarrow b \leftarrow s$. Για τον κόμβο d έχουμε αντίστοιχα $d \leftarrow c \leftarrow s$. Η πολυπλοκότητα εύρεσης της λύσης είναι $O(n)$.

Κύκλοι αρνητικού βάρους

Ο αλγόριθμος δουλεύει με ακμές αρνητικού βάρους σε αντίθεση με τον αλγόριθμο του Dijkstra (**Άσκηση:** να εξεταστεί στο παράδειγμα του σχήματος 6.25). Επιπρόσθετα όμως μπορεί με μια μικρή τροποποίηση να εντοπίσει αν ο γράφος έχει κύκλο αρνητικού βάρους και να εκτυπώσει ένα κατάλληλο μήνυμα. Λαμβάνοντας υπόψιν μας ότι κάθε βέλτιστο μονοπάτι έχει το πολύ $n-1$ ακμές, ο αλγόριθμος δεν θα έχει παγώσει αν υπάρχει κύκλος αρνητικού βάρους, αφού θα συνεχίσει να ανακαλύπτει μονοπάτια με καλύτερο κόστος. Άρα αν τρέξουμε τον αλγόριθμο για n βήματα και παρατηρηθεί βελτίωση στο μονοπάτι προς κάποιον κόμβο στο τελευταίο βήμα αυτό σημαίνει ότι υπάρχει κύκλος αρνητικού βάρους.

6.5.3 Ο αλγόριθμος του Bellman για γράφους χωρίς κύκλο

Όταν ο γράφος εισόδου είναι ακυκλικός, μπορούμε να εκμεταλλευτούμε την διαίρεση του γράφου σε επίπεδα (τοπολογική ταξινόμηση) για να κάνουμε πιο αποδοτικό τον αλγόριθμο του Bellman. Για παράδειγμα έστω ο ακόλουθος γράφος και η τοπολογική του ταξινόμηση (σχήμα 6.33):



Σχήμα 6.33: Ένας ακυκλικός γράφος αριστερα και η διαίρεση του σε επίπεδα δεξιά (τοπολογικά ταξινόμημένος)

Μπορούμε να παρατηρήσουμε ότι η εύρεση των συντομότερων μονοπατιών μπορεί να γίνει κατά επίπεδα, αφού όταν εξετάζουμε έναν κόμβο του γράφου, θα έχουν ήδη βρεθεί τα συντομότερα μονοπάτια για τους προηγούμενούς του. Άρα

η ενημέρωση των συντομότερων μονοπατιών μπορεί να γίνει κατά επίπεδα παραλάσσοντας τον αλγόριθμο του Bellman ως εξής: Ο πίνακας $Sorted$ περιέχει τους κόμβους του γράφου G τοπολογικά ταξινομημένους, δηλαδή για κάθε κατευθυνόμενη πλευρά $(Sorted[i], Sorted[j])$ έχουμε $i < j$. Επομένως θα έχουμε για $i < j$ και την κατευθυνόμενη πλευρά $(Sorted[i], Sorted[j])$ ότι $Layer[Sorted[i]] \leq Layer[Sorted[j]]$.

ΑΛΓΟΡΙΘΜΟΣ BELLMAN-ACYCLIC($G = (V, E, W), s \in V$)

1. Αρχικοποίησε $P[i], i = 1 \ldots n$ σε 0
2. Αρχικοποίησε $V[i], i = 1 \ldots n$ σε $+\infty$
3. Θέσε $V[s] = 0$, $P[s] = s$
4. **for each** x επόμενος του s
5. $\quad V[x] = W[s, x]$
6. **for** $i = 1$ **το** n
7. $\quad x = Sorted[i]$
8. $\quad$ **if** $Layer[x] > Layer[s]$ **then**
9. $\quad\quad$ **for each** y επόμενος του x με $V[x] + W[x, y] < V[y]$
10. $\quad\quad\quad V[y] = V[x] + W[x, y]$
11. $\quad\quad\quad P[y] = x$
12. $\quad\quad$ **end for**
13. $\quad$ **end for**
14. **end for**

Σχήμα 6.34: Ο αλγόριθμος του Bellman τροποποιημένος ώστε να εκμεταλλεύεται την ακυκλικότητα του γράφου G. Ο πίνακας sorted είναι η διάταξη των κόμβων του γράφου σύμφωνα με την τοπολογική του ταξινόμηση.

Ο αλγόριθμος ελέγχει μόνο τους κόμβους που είναι σε επόμενο επίπεδο από τον κόμβο αφετηρία, αφού δεν υπάρχει σίγουρα επόμενος προς κόμβους προηγουμένων επιπέδων. Στη συνέχεια για κάθε κόμβο σύμφωνα με την τοπολογική ταξινόμηση (πίνακας sorted), διορθώνονται οι ετικέτες των επομένων του. Ο αλγόριθμος ελέγχει για κάθε κόμβο ακριβώς τους επόμενούς του, άρα η συνολική πολυπλοκότητά του, είναι $O(m)$.

6.6 Δέντρα Επικάλυψης

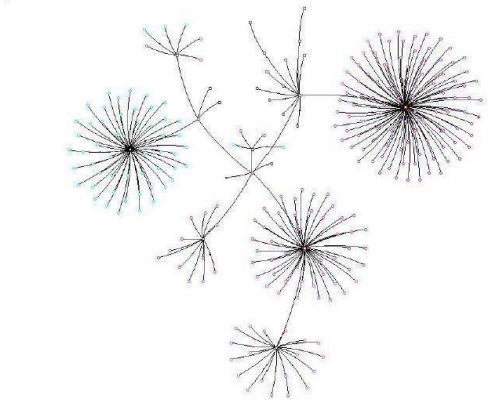
Σε πολλές εφαρμογές όπως παραδείγματος χάριν στο σχεδιασμό ηλεκτρονικών κυκλωμάτων ή στα τηλεπικοινωνιακά δίκτυα έχουμε ένα γράφο με βάρη $G = (V, E, W)$, στον οποίο αναζητούμε ένα ακυκλικό υποσύνολο των πλευρών E και το οποίο εμπεριέχει όλους τους κόμβους V. Ας δούμε ένα παράδειγμα. Στο σχεδιασμό ηλεκτρονικών κυκλωμάτων είναι συχνά απαραίτητο να καλωδιώσουμε τους υποδοχείς (pins) των διαφόρων συνιστωσών (πυκνωτές, ...) μεταξύ τους για να τους κάνουμε ηλεκτρονικά ισοδύναμους. Για να συνδέσουμε ένα σύνολο n υπ-

οδοχέων μπορούμε να χρησιμοποιήσουμε μια διάταξη $n-1$ καλωδίων. Κάθε τέτοια διάταξη συνδέει δυο υποδοχείς. Απο όλες τις διατάξεις που υπάρχουν η ζητούμενη είναι αυτή που χρησιμοποιεί τη μικρότερη ποσότητα καλωδίων.

Το πρόβλημα αυτό μπορούμε να το μοντελοποιήσουμε με ένα συνεκτικό, μη κατευθυνόμενο γράφο με βάρη $G = (V, E, W)$, όπου V είναι το σύνολο των υποδοχέων, E είναι το σύνολο των πιθανών συνδέσεων μεταξύ ζευγαριών υποδοχέων και για κάθε ακμή $(u, v) \in E$ το βάρος $w(u, v)$ αντιστοιχεί στο κόστος σύνδεσης των u και v. Αυτό που θέλουμε να βρούμε είναι ένα ακυκλικό υποσύνολο $E' \subseteq E$ που συνδέει όλες τις κορυφές του γράφου και του οποίου το συνολικό βάρος $w(E') = \sum_{(u,v)\in E'} w(u, v)$ ελαχιστοποιείται. Πριν προχωρήσουμε στον επακριβή ορισμό του προβλήματος παραθέτουμε τον ορισμό του δέντρου.

Ορισμός 22. Ισοδύναμοι ορισμοί για το δέντρο:
Το $G = (V, E)$ είναι ένας μη κατευθυνόμενος

- συνεκτικός γράφος ο οποίος δεν περιέχει κύκλους.
- γράφος τάξης n ο οποίος δεν περιέχει κύκλους και ο οποίος έχει $n-1$ ακμές.
- γράφος ο οποίος δεν περιέχει κύκλους και στον οποίο δημιουργείται κύκλος με την προσθήκη μιας μόνο ακμής.



Σχήμα 6.35: Ένα δέντρο επικάλυψης ενός υπογράφου του γράφου του Web

Παρατηρούμε απο τον ορισμό οτι τα δέντρα λόγω του οτι είναι ακυκλικά είναι οι μικρότεροι συνεκτικοί υπογράφοι τάξης n ενός γράφου G τάξης n.

Συνεχίζοντας με τον ορισμό του προβλήματος παρατηρούμε οτι το E' είναι ακυκλικό και άρα είναι δέντρο. Επιπλέον επειδή καλύπτει όλες τις κορυφές του γράφου το ονομάζουμε δέντρο επικάλυψης (spanning tree).

Ορισμός 23. Ένα δέντρο επικάλυψης T ενός γράφου $G = (V, E)$ είναι ένα υποσύνολο E' των ακμών του γράφου τέτοιο ώστε το $T = (V, E')$ να είναι δέντρο. Δηλαδή, ένα δέντρο επικάλυψης του G είναι ένας υπογράφος του που συνδέει όλους τους κόμβους του και δεν περιέχει κύκλους.

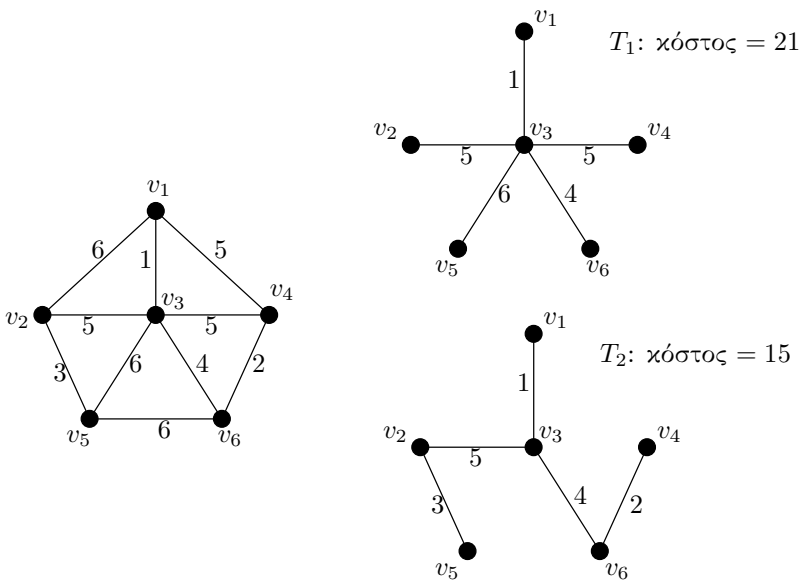
Σχήμα 6.36: Ένας γράφος $G = (V, E)$ και δεξιά ένα δέντρο επικάλυψης

6.6.1 Πρόβλημα εύρεσης ενός δέντρου επικάλυψης ελάχιστου κόστους

Έστω $G = (V, E, W)$ ένας απλός συνεκτικός γράφος με βάρη στις ακμές. Έστω ότι κάθε ακμή $e \in E$ έχει βάρος $w(e)$.

Ορισμός 24. Το Δέντρο Επικάλυψης Ελάχιστου Κόστους (minimum cost spanning tree) είναι ένα δέντρο $T = (V, E')$ με το μικρότερο συνολικό βάρος $W(T) = \sum_{e \in E'} w(e)$.

Στο σχήμα 6.37 βλέπουμε δύο διαφορετικά δέντρα επικάλυψης με συνολικά βάρη 21 το πρώτο και 15 το δεύτερο. Παρατηρούμε ότι εαν το βάρος κάθε ακμής είναι 1, το πρόβλημα συνίσταται στο να βρούμε ένα οποιοδήποτε δέντρο επικάλυψης (κόστους $n - 1$).



Σχήμα 6.37: Ένας γράφος $G = (V, E, W)$ αριστερά και δύο δέντρα επικάλυψης T_1 και T_2 με $W(T_1) = 21$ και $W(T_2) = 15$

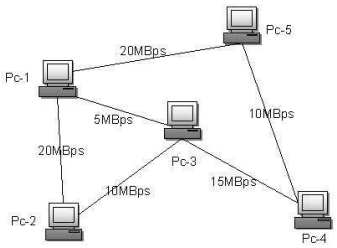
Σε πολλές εφαρμογές αντί να αναζητούμε ένα δέντρο επικάλυψης ελαχίστου κόστους (ΔΕΕΚ) αναζητούμε ένα δέντρο επικάλυψης μεγίστου κόστους (ΔΕΜΚ). Οι αλγόριθμοι που θα μελετήσουμε παρακάτω εφαρμόζονται και στις δύο περιπτώσεις.

Ας δούμε ένα παράδειγμα όπου απαιτείται ένα ΔΕΜΚ. Έστω ένα μικρό δίκτυο υπολογιστών όπως φαίνεται στο σχήμα 6.38. Οι υπολογιστές είναι οι κόμβοι του αντίστοιχου γράφου και οι συνδέσεις είναι οι πλευρές του γράφου. Πάνω στις συνδέσεις έχουμε τη μέγιστη χωρητικότητα. Επιθυμούμε να κρατήσουμε τον ελάχιστο αριθμό συνδέσεων από τις διαθέσιμες ώστε να είναι δυνατή η επικοινωνία κάθε ζεύγους υπολογιστών, επιτυγχάνοντας τον μέγιστο όγκο δεδομένων. Άρα στο δίκτυό μας, αναζητούμε ένα δέντρο επικάλυψης μεγίστου βάρους.

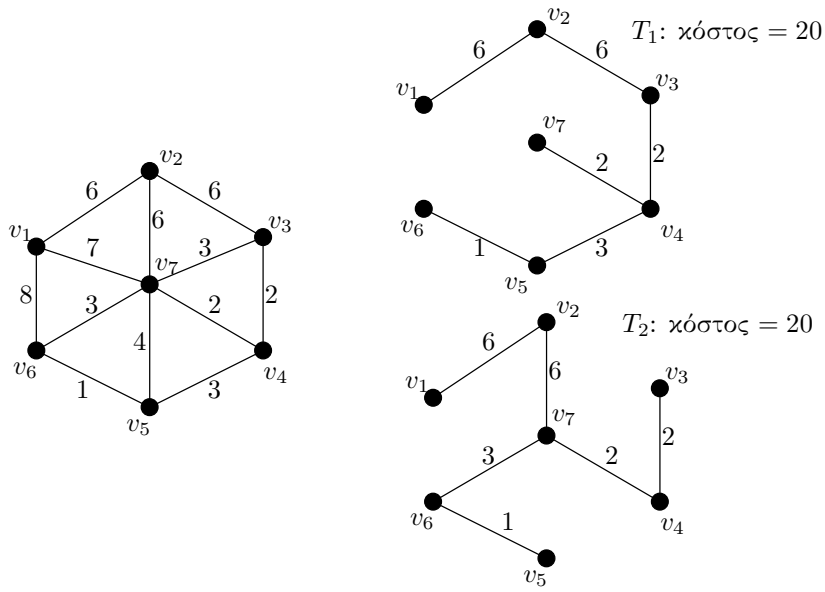
Ας παρατηρήσουμε ότι το ΔΕΕΚ ή ΔΕΜΚ δεν είναι μοναδικά. Ενδέχεται να υπάρχουν πολλά με το ίδιο κόστος (βλ. σχήμα 6.39). Στη συνέχεια θα περιγράψουμε αλγόριθμους για την ανεύρεση ενός εξ΄ αυτών.

Άσκηση: Δείξτε ότι αν τα βάρη στο γράφο είναι διαφορετικά ανα δυο τότε το Δέντρο Επικάλυψης Ελάχιστου Κόστους (minimum spanning tree) είναι μοναδικό.

Ας παρατηρήσουμε ότι αν για να λύσουμε το πρόβλημα βρίσκαμε όλα τα δέντρα επικάλυψης του γράφου και έπειτα επιλέγαμε, ανάμεσα σε αυτά, αυτό με το



Σχήμα 6.38: Ένα δίκτυο επικοινωνιών. Πάνω στις συνδέσεις έχουμε την μέγιστη χωρητικότητα.



Σχήμα 6.39: Ένας γράφος $G = (V, E, W)$ και δύο δέντρα επικάλυψης με το ίδιο ελάχιστο κόστος

ελάχιστο κόστος τότε ο αριθμός των δέντρων επικάλυψης θα ήταν εκθετικός σε σχέση με τον αριθμό των κορυφών του γράφου. Από το 1875 έχουμε το παρακάτω θεώρημα.

Θεώρημα 8. (Arthur Cayley, 1875) Σε ένα πλήρη γράφο, ο αριθμός των δέντρων επικάλυψης είναι ακριβώς n^{n-2}, όπου n είναι η τάξη του γράφου.

n	1	2	3	4	5	6	7	8	10
ΔΕΕΚ	1	1	3	16	125	1296	16807	262144	10^8

□

Είδαμε λοιπόν οτι η ένα προς ένα απαρίθμηση των δέντρων επικάλυψης με σκοπό να βρούμε το ελάχιστο οδηγεί σε εκθετικό χρόνο. Ωστόσο το πρόβλημα εύρεσης του ΔΕΕΚ ή ΔΕΜΚ μπορεί να λυθεί πολυωνυμικά. Οι αλγόριθμοι που θα δούμε παρακάτω (ο αλγόριθμος του Prim και ο αλγόριθμος του Kruskal) στηρίζουν την ορθότητά τους στο παρακάτω θεώρημα για να επιλύσουν βέλτιστα το πρόβλημα σε πολυωνυμικό χρόνο.

Θεώρημα 9. Έστω $F = (F_1, F_2, \ldots, F_k)$ με $F_i = (V_i, E_i)$ ένα δάσος ενός συνεκτικού γράφου $G = (V, E, W)$ και (u, v) η πλευρά με το ελάχιστο κόστος έχουσα ένα άκρο στο V_1. Τότε, ανάμεσα σε όλα τα δέντρα επικάλυψης που περιέχουν αυτό το δάσος, υπάρχει ένα, το καλύτερο το οποίο περιέχει την πλευρά (u, v).

Απόδειξη: Έστω $T = \bigcup_{i=1}^{k} E_i$ οι πλευρές του δάσους F.

Έστω A_i ένα οποιοδήποτε δέντρο επικάλυψης περιέχον το σύνολο των πλευρών T. Ας υποθέσουμε ότι υπάρχει ένα δέντρο επικάλυψης $A = (V, E')$ περιέχον T αλλά όχι την πλευρά (u, v) και του οποίου το κόστος είναι ελάχιστο, δηλαδή

$$\text{Κόστος}(A) \leq \text{Κόστος}(A_i) \tag{6.1}$$

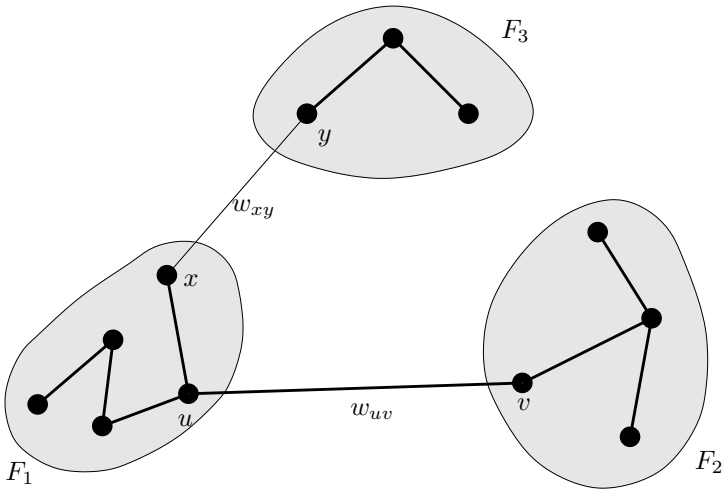
Αν προσθέσουμε την πλευρά (u, v) μέσα στο E' δημιουργούμε έναν κύκλο c. Υπάρχει λοιπόν πάνω στον κύκλο c μια πλευρά (x, y) τέτοια ώστε $x \in V_1$ και $y \in V \backslash V_1$. Έχουμε λοιπόν $W_{xy} > W_{uv}$ και $(x, y) \notin T$. Αντικαθιστώντας (x, y) με (u, v) παίρνουμε ένα νέο δέντρο επικάλυψης $A' = (V, E' \bigcup (u, v) \backslash (x, y))$ τέτοιο ώστε:

$$\text{Κόστος}(A') < \text{Κόστος}(A)$$

το οποίο είναι σε αντίφαση με την 6.1.

6.6.2 Αλγόριθμος Prim

Ο αλγόριθμος του Prim ξεκινάει απο ένα δέντρο A αποτελούμενο απο έναν μόνο κόμβο και σε κάθε επανάληψη αυξάνει το δέντρο A μέχρι να καλυφθούν όλες οι κορυφές του γράφου. Ο κόμβος αυτός επιλέγεται αυθαίρετα. Σε κάθε βήμα προστίθεται μια ακμή στο δέντρο A που συνδέει μια κορυφή του A με κάποια



Σχήμα 6.40: Ένα δάσος F_1, F_2, F_3 του γράφου G. Η πλευρά (u, v) με το ελάχιστο κόστος ανάμεσα στις πλευρές που έχουν το ένα άκρο τους στο V_1 θα ανήκει στο δέντρο επικάλυψης ελαχίστου κόστους στον γράφο G.

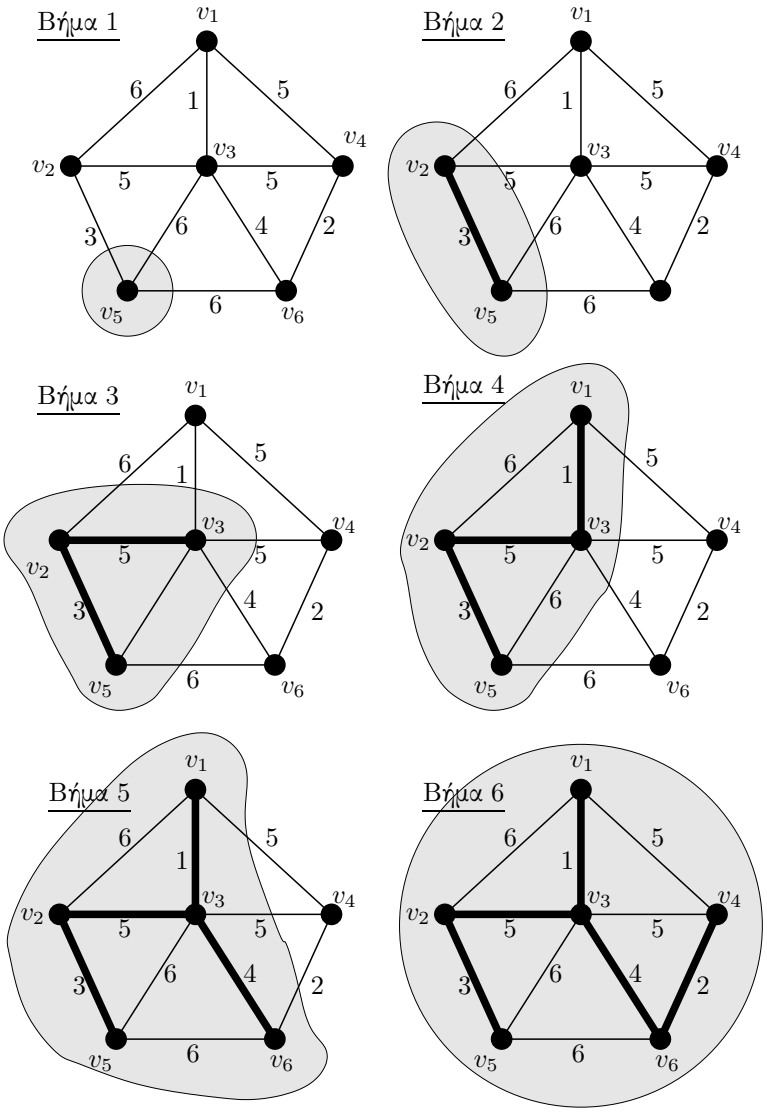
κορυφή εκτός του A. Η ακμή η οποία επιλέγεται κάθε φορά είναι αυτή που έχει το ελάχιστο κόστος ανάμεσα στις υποψήφιες για εισαγωγή ακμές. Η στρατηγική που ακολουθεί ο αλγόριθμος είναι άπληστη (greedy) αφού σε κάθε βήμα το δέντρο αυξάνει κατα μια ακμή, δηλαδή κάνει μια επιλογή την οποία δεν αναθεωρεί στην συνέχεια.

PRIM $(G = (V, E, W))$
1. $T = \{u\}$ /* Οποιοσδήποτε κόμβος $u \in V$ */
2. **for** $i = 2$ **to** $|V|$ **do**
3. $\quad T \leftarrow T \cup \{v\}, v$ πλησιέστερος ελεύθερος κόμβος στο T
4. **end for**

Σχήμα 6.41: Αλγόριθμος Prim για την εύρεση του ΔΕΕΚ στο γράφο G.

Πλησιέστερος κόμβος στο T εννοούμε τον κόμβο v, εκτός του T, και με ελάχιστο βάρος $w(u, v)$ από κάθε κόμβο u μέσα στο T. Ελεύθερος είναι κάθε κόμβος εκτός του δέντρου T. Στο σχήμα 6.42 βλέπουμε την εκτέλεση του αλγορίθμου ξεκινώντας από τον κόμβο v_5. Το συνολικό βάρος του ΔΕΕΚ είναι

15.



Σχήμα 6.42: Εκτέλεση του αλγορίθμου του Prim ξεκινώντας από τον κόμβο v_5.

Για τον υπολογισμό της πολυπλοκότητας του αλγόριθμου του Prim θεωρούμε

για τον γράφο $G = (V, E)$ με $|V| = n$ και $|E| = m$ ότι d_i είναι ο βαθμός του εισερχόμενου κόμβου στο δέντρο στη i επανάληψη. Τότε για την i-επανάληψη, $1 \leq i \leq n-1$, έχουμε ότι σε κάθε βήμα το πλήθος των ακμών που θα πρέπει να εξεταστούν φράσσεται ως εξής:

d_1
$d_1 + d_2$
$d_1 + d_2 + d_3$
$\ldots$
$d_1 + d_2 + d_3 + \ldots + d_i$
$\ldots$
$d_1 + d_2 + d_3 + \ldots + d_i + \ldots + d_{n-1}$

$(n-1)d_1 + (n-2)d_2 + \ldots + (n-i)d_i + \ldots + 2d_{n-2} + d_{n-1} \leq n \sum_{i=1}^{n-1} d_i = O(mn)$

6.6.3 Αλγόριθμος Prim (nearest neighbour)

Όπως είδαμε η πολυπλοκότητα του αλγορίθμου του Prim είναι $O(mn)$. Μπορούμε να τροποποιήσουμε τον αλγόριθμο έτσι ώστε η πολυπλοκότητα του να βελτιωθεί σε $O(n^2)$. Για το σκοπό αυτό ορίζουμε τους πίνακες $near[x]$, $dist[x]$, $x \in V$ έτσι ώστε:

$$near[x] = \begin{cases} x, & \text{αν } x \in T \\ 0, & \text{αν } x \notin T, \nexists y \in T : (x, y) \in E \\ y, & \text{αν } y \text{ ο πλησιέστερος κόμβος στο } T \end{cases}$$

$$dist[x] = \begin{cases} W_{x,y}, & \text{αν } near[x] = y \\ +\infty, & \text{αν } near[x] = 0 \\ 0, & \text{αν } near[x] = x \end{cases}$$

Ο αλγόριθμος στην αρχή επιλέγει αυθαίρετα μια κορυφή και σε κάθε βήμα προσθέτει μια κορυφή v που δεν ανήκει στο τρέχον δέντρο, και απέχει ελάχιστα από αυτό. Για το βήμα του αλγορίθμου που επιλέγει την κορυφή με το μικρότερο κόστος $dist[v]$ χρησιμοποιείται η δομή του σωρού.

```
PRIM (G = (V, E, W))
 1.  V_T = {s} /* Οποιοσδήποτε κόμβος s ∈ V */
 2.  near[s] = s,  E_T = Ø
 3.  for every node i ∈ Γ(s) do
 4.        near[i] = s;  dist[i] = w_{s,i}
 5.  end for
 6.  for every node j ∉ {s} ∪ Γ(s) do
 7.        near[j] = 0;  dist[j] = +∞
 8.  end for
 9.  while |V_T| < n do
10.        Διάλεξε u ∈ V − V_T με ελάχιστο dist[u]
11.        if dist[u] = ∞ then "graph disconnected", exit
12.        V_T = V_T ∪ {u}  E_T = E_T ∪ {(u, near[u])}
13.        for x ∈ (V − V_T) do
14.              if w_ux < dist[x] then
15.                    dist[x] ← w_ux; near[x] ← u
16.              end if
17.        end for
18.  end while
```

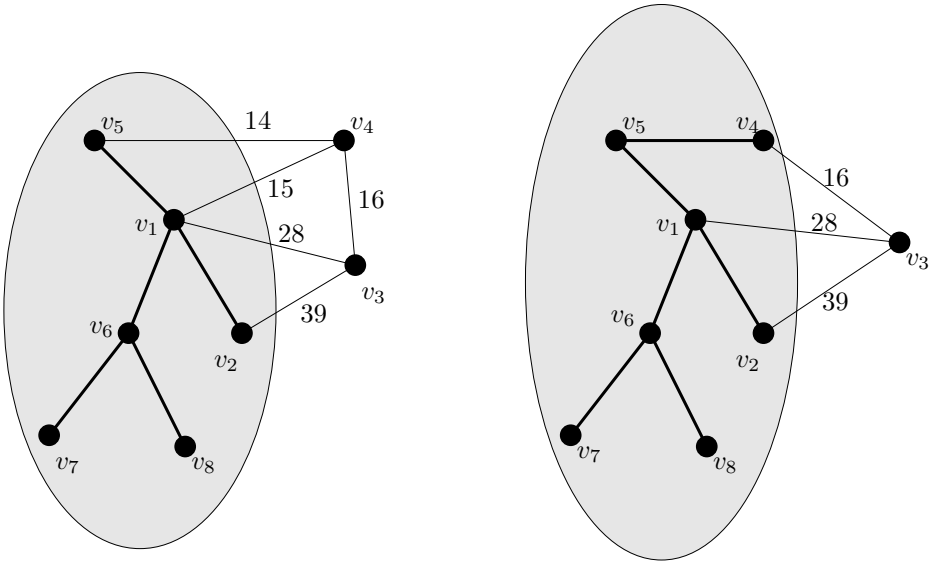
Σχήμα 6.43: Αλγόριθμος Prim για την εύρεση του ΔΕΕΚ στο γράφο G.

Ας δούμε ένα παράδειγμα εκτέλεσης του αλγορίθμου. Στο σχήμα 6.44 αριστερά, δύο κόμβοι είναι εκτός του τρέχοντος δέντρου. Ο πλησιέστερος κόμβος στο δέντρο είναι ο v_4 με $dist[v_4] = 14$. Όταν εισαχθεί και αυτός ο κόμβος στο δέντρο τότε μόνο η πλευρά μεταξύ των κόμβων v_4 και v_3 θα επηρεάσει την απόσταση του κόμβου v_3 από το τρέχον δέντρο.

Η πολυπλοκότητα του τροποποιημένου αλγόριθμου του Prim (nearest neighbour) είναι $O(n^2)$. Πράγματι ο αλγόριθμος απαιτεί $n-1$ επαναλήψεις και σε κάθε επανάληψη χρειάζεται $O(n+d_i)$ πράξεις, όπου d_i ο βαθμός του εισερχόμενου στο δέντρο κόμβου στη i επανάληψη (βλ. σχήμα 6.45).

Παρατηρήσεις:

- Αν ο γράφος G είναι συνεκτικός τότε η εύρεση του δέντρου επικάλυψης ελάχιστου κόστους γίνεται σε $n-1$ επαναλήψεις,
- Αν στην $i < n-1$ επανάληψη δε βρίσκουμε πλευρά (x, y) με x στο δέντρο και y έξω απο το δέντρο και $w(x, y) < \infty$, τότε ο γράφος δεν είναι συνδεδεμένος, (αλλά ήδη γνωρίζουμε έλεγχο συνεκτικότητας σε χρόνο $O(m)$ με την BFS). Σε αυτήν την περίπτωση, μπορούμε να βρούμε ένα δάσος επικάλυψης ελαχίστου κόστους, εκτελώντας και πάλι τον αλγόριθμο, ξεκινώντας από ένα κόμβο εκτός του τρέχοντος δάσους επικάλυψης.
- Ο αλγόριθμος δίνει τη βέλτιστη λύση, (απο το Θεώρημα 8 παρατηρώντας ότι το δάσος εκκίνησης συνίσταται απο n μεμονωμένους κόμβους).



$near[v_4] = v_5, dist[v_4] = 14$
$near[v_3] = v_1, dist[v_3] = 28$

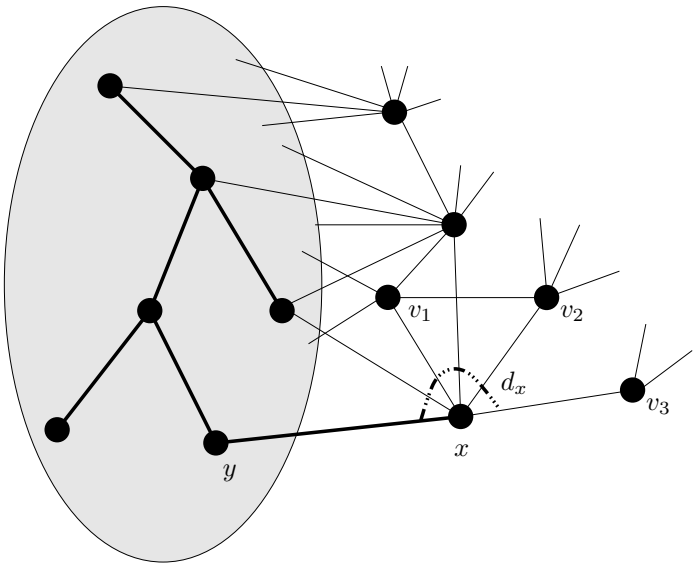
$near[v_4] = v_4, dist[v_4] = 0$
$near[v_3] = v_4, dist[v_3] = 16$

Σχήμα 6.44: Οι αποστάσεις των κόμβων v_4 και v_3 αριστερά. Δεξιά η απόσταση του κόμβου v_3 μετά την προσάρτηση στο δέντρο του κόμβου v_4.

Στη συνέχεια θα δούμε έναν άλλο αλγόριθμο για την εύρεση του ΔΕΕΚ, γνωστός ως αλγόριθμος του Kruskal.

6.6.4 Αλγόριθμος Kruskal

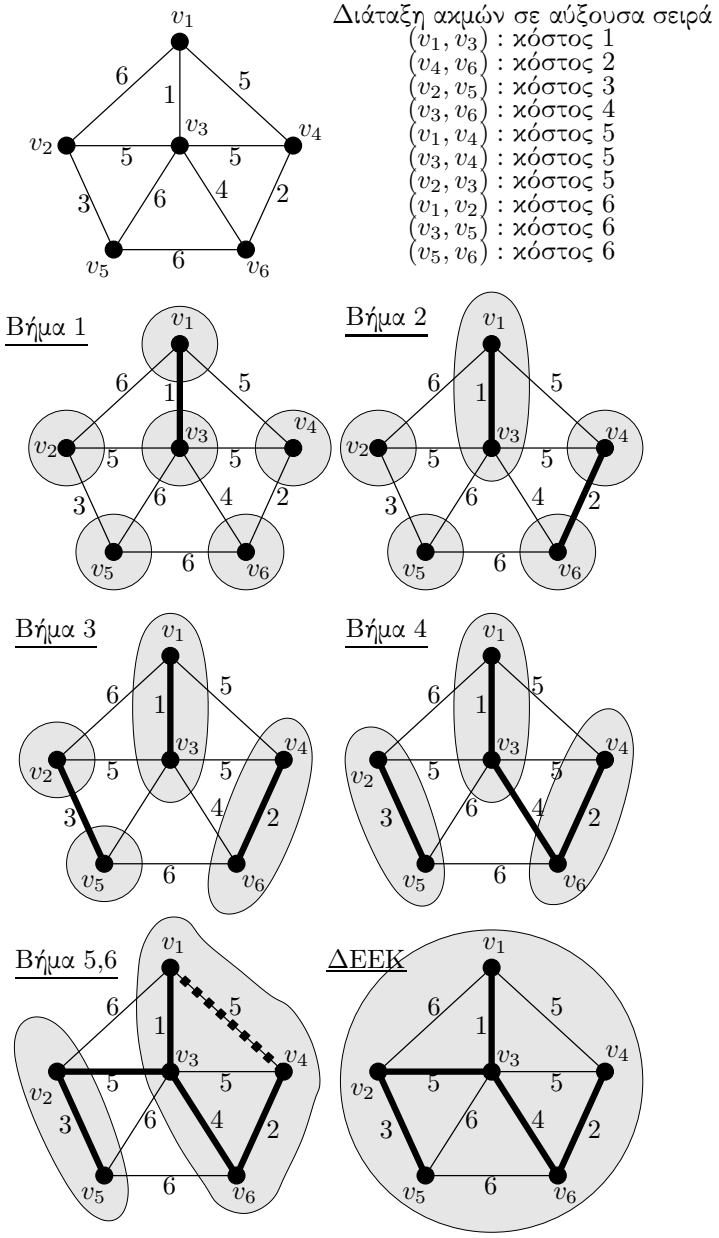
Ο αλγόριθμος ξεκινάει απο ένα δάσος απο n δέντρα, όπου κάθε ένα δέντρο είναι εκφυλισμένο σε ένα μεμονωμένο κόμβο. Εξετάζει τις πλευρές μια-μια και σε κάθε επανάληψη, προσθέτει την ακμή με το μικρότερο κόστος η οποία δεν δημιουργεί κύκλο με τις ήδη επιλεγμένες πλευρές. Σταματάει όταν έχει ένα δέντρο επικάλυψης (συνεκτικός γράφος) ή όταν δεν βρίσκουμε πλέον πλευρά να προσθέσουμε (ο γράφος δεν είναι συνεκτικός).



Σχήμα 6.45: Όταν εισάγεται ο κόμβος x στο δέντρο τότε μόνο οι γειτονές του αρκεί να ελεγθούν για την ενημέρωση των πινάκων *near* και *dist*

```
KRUSKAL (G = (V, E, W))
 1.  T ← ∅
 2.  while (|T| < n − 1) and (E ≠ ∅) do
 3.        e =smallest edge in E
 4.        E = E − {e}
 5.        if T ∪ {e} has no cycle then
 6.              T ← T ∪ {e}
 7.        end if
 8.  end while
 9.  if (|T| < n − 1) then
10.        write "graph disconnected"
```

Ας θεωρήσουμε ένα παράδειγμα. Οι πλευρές (βλ. σχήμα 6.46) είναι ταξινομημένες σε μη φθίνουσα σειρά ως προς τα βάρη. Μετά από πέντε επαναλήψεις ο αλγόριθμος επιστρέφει ένα ΔΕΕΚ βάρους 15.



Σχήμα 6.46: Εκτέλεση του αλγορίθμου Kruskal. Στο βήμα 5 η πλευρά (v_1, v_4) σχηματίζει κύκλο και απορρίπτεται

Η πολυπλοκότητα του αλγόριθμου του Kruskal είναι $O(m \lg n)$, όπου m ο αριθμός των πλευρών και n ο αριθμός των κορυφών, αν θεωρήσουμε ότι οι πλευρές είναι ταξινομημένες και ο έλεγχος ακυκλικότητας γίνεται με τη δομή Union και Find. Καταρχάς η ταξινόμηση των πλευρών κατά αύξουσα τάξη απαιτεί χρόνο $O(m \lg m)$. Σε κάθε επανάληψη η εξακρίβωση σχηματισμού κύκλου γίνεται σε $O(\lg n)$ και αφού ο αλγόριθμος χρειάζεται m επαναλήψεις, η συνολική πολυπλοκότητα του θα είναι $O(m \lg m) + O(m \lg n)$ το οποίο, δεδομένου ότι $m < n^2$, δίνει τελικά $O(m \lg n)$.

Παρατηρήσεις:

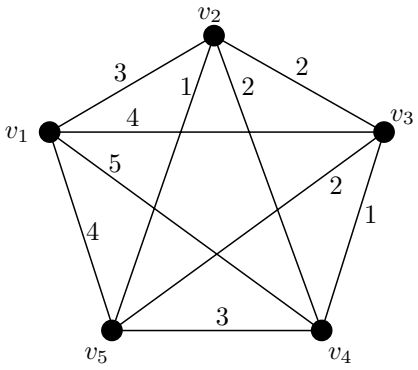
- Ο αλγόριθμος δίνει τη βέλτιστη λύση, (προκύπτει απ΄ ευθείας απο το θεώρημα βελτιστότητας, θεώρημα 8).
- Αν ο γράφος είναι μη συνεκτικός τότε με τον αλγόριθμο του Kruskal βρίσκουμε απ΄ ευθείας ένα δάσος επικάλυψης ελάχιστου κόστους ενώ με τον αλγόριθμο του Prim πρέπει να επαναλάβουμε τον αλγόριθμο σε κάθε συνεκτική συνιστώσα.

Πυκνοί και αραιοί γράφοι:
Καταρχάς όπως είδαμε η πολυπλοκότητα του αλγορίθμου του Prim είναι $O(n^2)$ ενώ η πολυπλοκότητα του αλγορίθμου του Kruskal είναι $O(m \lg n)$, όπου m ο αριθμός των ακμών του γράφου. Ας θεωρήσουμε έναν πυκνό γράφο, δηλαδή $m = O(n^2)$ και έναν αραιό γράφο, δηλαδή $m = O(n)$. Τότε οι πολυπλοκότητες των δύο αλγορίθμων στον πυκνό γράφο είναι $\mathcal{O}(n^2)$ για τον Prim και $\mathcal{O}(n^2 \log n)$ για τον Kruskal. Δηλαδή ο αλγόριθμος Prim στους πυκνούς γράφους είναι μικρότερης πολυπλοκότητας. Αντίθετα στους αραιούς γράφους ο αλγόριθμος Prim εξακολουθεί να είναι πολυπλοκότητας $\mathcal{O}(n^2)$ ενώ ο Kruskal γίνεται $\mathcal{O}(n \log n)$.

6.7 ΔΕΕΚ και το πρόβλημα του Πλανόδιου Πωλητή (TSP)

Δίνεται ένας πλήρης γράφος $G = (V, E, W)$ με βάρη στις ακμές, τάξης n. Στο πρόβλημα του πλανόδιου πωλητή (Traveling Salesman Problem) ζητείται να βρεθεί ένας κύκλος ο οποίος να περνά απο όλες τις κορυφές μία και μόνο μία φορά (κύκλος Hamilton) και ο οποίος να έχει ελάχιστο κόστος.

Στο παράδειγμα του σχήματος 6.47 ο κύκλος $v_1, v_2, v_3, v_5, v_4, v_1$ έχει κόστος 15. Αντίθετα ο κύκλος $v_1, v_5, v_2, v_3, v_4, v_1$ έχει βάρος 13.



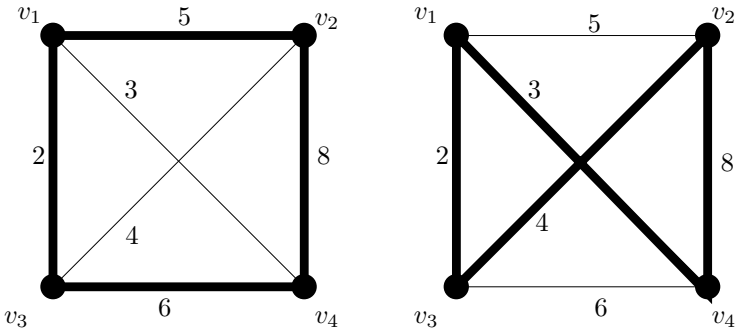
	v_1	v_2	v_3	v_4	v_5
v_1	∞	3	4	5	4
v_2	3	∞	2	2	1
v_3	4	2	∞	1	2
v_4	5	2	1	∞	3
v_5	4	1	2	3	∞

Σχήμα 6.47: Ένας πλήρης γράφος με βάρη στις πλευρές. Δεξιά η αναπαράσταση του γράφου με πίνακα γειτνίασης.

Στο σχήμα 6.48 βλέπουμε δύο λύσεις για το πρόβλημα του πλανώδιου πωλητή στο γράφο K_4. Η δεύτερη λύση είναι η βέλτιστη λύση του προβλήματος του πλανόδιου πωλητή για το συγκεκριμένο γράφο. Γενικά για το TSP δεν γνωρίζουμε και είναι εξαιρετικά αμφίβολο ότι θα βρούμε στο μέλλον πολυωνυμικό αλγόριθμο που να το επιλύει βέλτιστα.

Στη συνέχεια θα δούμε μια ενδιαφέρουσα χρησιμότητα των ΔΕΕΚ και των πολυωνυμικών αλγορίθμων που τα επιλύουν. Θα δείξουμε οτι το δέντρο επικάλυψης ελάχιστου κόστους μπορεί να δώσει ένα **κάτω φράγμα της βέλτιστης λύσης του προβλήματος του πλανόδιου πωλητή**.
Έστω $B(I)$ το κάτω φράγμα, όπου I ένα στιγμιότυπο (instance) του TSP. Υπολογίζουμε το $B(I)$ ως εξής:

- Αφαίρεσε ένα οποιοδήποτε κόμβο v_i απο το γράφο G.
- Βρές ένα δέντρο επικάλυψης ελάχιστου κόστους T στον υπογράφο $G' = (V \backslash \{v_i\}, E')$, όπου $E' = E - \{(v_i, v)\}$ με $v \in \Gamma(v_i)$. Έστω $K(T)$ το



Κόστος = 21 Κόστος = 17

Σχήμα 6.48: Δυο λύσεις κόστους 21 και 17 για το πρόβλημα του πλανώδιου πωλητή στο γράφο K_4

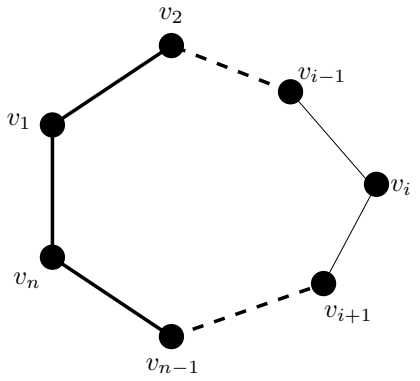
κόστος του δέντρου.

- Θεώρησε τις δύο πλευρές (v_i, v_k) και (v_i, v_l), $k, l \neq i$, με το μικρότερο κόστος w_{il} και w_{ik}.
- Θέσε $B(I) = K(T) + w_{il} + w_{ik}$

Απόδειξη:
Έστω $\{v_1, v_2, \ldots, v_i, \ldots, v_n, v_1\}$ ένας βέλτιστος κύκλος με κόστος C_{opt}. Όταν αφαιρούμε τον κόμβο v_i, έχουμε ένα δέντρο T' (μια αλυσίδα $v_{i+1}, \ldots, v_1, v_2, \ldots, v_{i-1}$). Προφανώς έχουμε ότι:

$$\text{Κόστος}(T) \leq \text{Κόστος}(T').$$

Στο σχήμα 6.50 όταν αφαιρούμε τον κόμβο A από την κλίκα K_5 το ΔΕΕΚ στον γράφο K_4 είναι κόστους 16. Οι δύο μικρότερες πλευρές ανάμεσα στις AE, AD, AC, AB είναι οι AE και AC με $w_{AE} = 2$ και $w_{AC} = 4$. Το κάτω φράγμα $B(I)$ για το TSP είναι ίσο με 22.

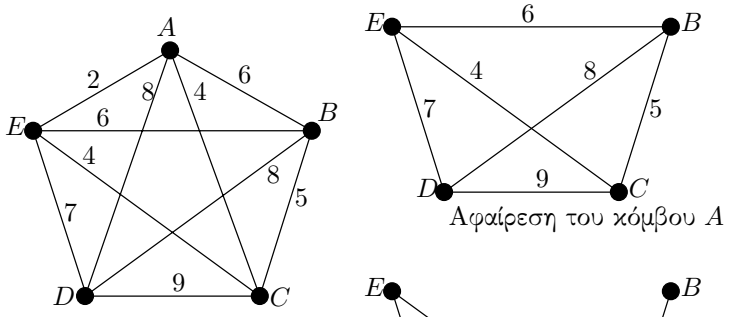


Άρα

$$B(I) = \text{Κόστος}(T) + W_{ik} + W_{il}$$
$$\leq \text{Κόστος}(T') + W_{i,i-1} + W_{i,i+1}$$
$$= \text{Κόστος (Βέλτιστου κύκλου)}$$
$$= C_{opt}$$

.

Σχήμα 6.49: Ένας βέλτιστος κύκλος για το TSP. Όταν αφαιρείται ο κόμβος v_i έχουμε ένα δέντρο επικάλυψης στον πλήρη γράφο K_{n-1}



$$B(I) = K(T) + w_{AE} + w_{AC} = 16 + 2 + 4 = 22$$

Σχήμα 6.50: Αριστερά ένας γράφος με βάρη K_5. Πάνω δεξιά ο εναπομένων γράφος $K_4 = K_5 - \{A\}$. Κάτω δεξιά το ΔΕΕΚ στο γράφο K_4 κόστους 16.

Κεφάλαιο 7

Άπληστοι Αλγόριθμοι

Οι αλγόριθμοι προβλημάτων βελτιστοποίησης (προβλημάτων δηλαδή που στόχος είναι να μεγιστοποιήσουμε την ωφέλεια μας, ή να ελαχιστοποιήσουμε την απώλεια κάνοντας τις καλύτερες επιλογές), συνήθως ακολουθούν μιά σειρά βημάτων, κάνοντας σε κάθε βήμα ένα σύνολο επιλογών.

Ένας άπληστος αλγόριθμος (greedy algorithm) κάνει σε κάθε βήμα την επιλογή που φαίνεται βέλτιστη την δεδομένη στιγμή. Ωστόσο η επιλογή που είναι βέλτιστη μία δεδομένη στιγμή, δεν είναι απαραίτητο ότι θα οδηγήσει στην βέλτιστη λύση του προβλήματος. Αντίθετα όπως θα δούμε, σε κάποια προβλήματα αυτή η ''άπληστη'' επιλογή είναι δυνατόν να οδηγήσει σε μία κακή λύση, οσοδήποτε μακριά από την βέλτιστη.

Ωστόσο έχουμε ήδη μελετήσει άπληστους αλγορίθμους που οδηγούν στην βέλτιστη λύση του προβλήματος (για παράδειγμα οι αλγόριθμοι Prim και Kruskal για την εύρεση ενός δέντρου επικάλυψης ελαχίστου κόστους και ο αλγόριθμος Dijkstra για την εύρεση του ελαχίστου μονοπατιού σε έναν γράφο).

Έτσι, η αντιμετώπιση ενός προβλήματος με μία άπληστη μέθοδο αν και δεν κρύβει κάποια εκλεπτυσμένη μέθοδο είναι συχνά πολύ ισχυρή και αποδοτική. Στην ενότητα αυτή, θα μελετήσουμε την γενική μορφή ενός άπληστου αλγορίθμου και θα δούμε την εφαρμογή του σε προβλήματα που είτε εφαρμόζεται αποδοτικά (βρίσκεται η βέλτιστη λύση) είτε όχι.

7.1 Γενική μορφή ενός Άπληστου Αλγόριθμου

Ένα πρόβλημα βελτιστοποίησης μπορεί να εκφρασθεί μέσω ενός πεπερασμένου συνόλου E που κάθε $e \in E$ δίδει κάποιο όφελος σύμφωνα με την αντικειμένικη συνάρτηση $v(e)$ (που θα θεωρήσουμε ότι παίρνει ακέραιες τιμές ($v(e) \in \mathbf{N}$).

Μία λύση στο πρόβλημα θα είναι ένα υποσύνολο $F \subseteq E$, τέτοιο ώστε να τηρείται το σύνολο περιορισμών του προβλήματος C, δηλαδή $C(F) = true$. Θα λέμε ότι το F είναι μία εφικτή λύση (feasible solution) για το πρόβλημα μας. Ο

στόχος θα είναι να βρεθεί εκείνο το F που είναι βέλτιστο μεταξύ των υποσυνόλων του E, δηλαδή το

$$\sum_{e \in F} v(e)$$

να είναι μέγιστο (αν μιλάμε για πρόβλημα μεγιστοποίησης) ή ελάχιστο (αν μιλάμε για πρόβλημα ελαχιστοποίησης).

Η κατασκευή του F θα γίνεται κατά βήματα. Αρχικά θα είναι κενό και σε κάθε βήμα θα επιλέγεται εκείνο το $e \in E$ που είναι το καλύτερο την δεδομένη στιγμή (άπληστη επιλογή), ικανοποιώντας τους περιορισμούς. Επαναλαμβάνουμε μέχρι να εξαντλήσουμε τα στοιχεία του E. Η αλγοριθμική αυτή προσέγγιση φαίνεται στο σχήμα 7.1.

GREEDY (E : σύνολο στοιχείων)
1. $F = \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. Άπληστη Επιλογή $x \in E$
4. $E = E - \{x\}$
5. **if** $F \cup \{x\}$ feasible **then**
6. $F = F \cup \{x\}$
7. **end if**
8. **end while**
9. return F

Σχήμα 7.1: Γενική Μορφή Άπληστου Αλγόριθμου

Στην γραμμή 5 του αλγορίθμου, είναι ο έλεγχος αν η προσθήκη του x στην λύση δεν παραβιάζει του περιορισμούς, δηλαδή η λύση παραμένει εφικτή.

Ας δούμε πως εφαρμόζεται το σχήμα αυτό στον αλγόριθμο του Kruskal (βλ. σχήμα 7.2).

Το σύνολο E των στοιχείων είναι το σύνολο E των ακμών και σε κάθε βήμα ο αλγόριθμος επιλέγει την ακμή με το ελάχιστο βάρος (άπληστη επιλογή). Η ακμή αφαιρείται από το σύνολο ακμών και ελέγχεται αν μπορεί να μπεί στην λύση, κοιτάζοντας αν δημιουργείται κύκλος. Αν η προσθήκη της ακμής είναι εφικτή, τότε προστίθεται στην λύση και η διαδικασία συνεχίζεται μέχρι να δημιουργηθεί ένα δέντρο επικάλυψης.

Όπως αποδείξαμε στην ενότητα 6.6 ο αλγόριθμος αυτός είναι βέλτιστος δηλαδή το δέντρο επικάλυψης που επιστρέφει είναι ελαχίστου κόστους. Άρα σε αυτό το πρόβλημα η άπληστη επιλογή οδηγεί στην βέλτιστη λύση.

7.2 Ανάθεση ενός πόρου

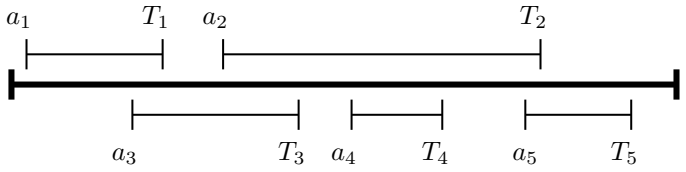
Δίδεται ένας πόρος (για παράδειγμα ένα αυτοκίνητο προς ενοικίαση ή ένα υπολογιστικό σύστημα προς ανάθεση), και ένα σύνολο αιτήσεων $P = \{P_1, \ldots, P_n\}$

```
KRUSKAL (G = (V, E, W))
 1.  T = Ø
 2.  while (|T| < n − 1) and (E ≠ Ø) do
 3.        e =smallest edge in E
 4.        E = E − {e}
 5.        if T ∪ {e} has no cycle then
 6.              T ← T ∪ {e}
 7.        end if
 8.  end while
 9.  if (|T| < n − 1) then
10.        write "graph disconnected"
```

Σχήμα 7.2: Ο αλγόριθμος του Kruskal για την εύρεση ενός δέντρου επικάλυψης ελαχίστου κόστους σε έναν μη κατευθυνόμενο γράφο με βάρη $G = (V, E)$.

χρησιμοποιήσης του πόρου. Κάθε αίτηση P_i, αναφέρεται σε ένα χρονικό διάστημα (a_i, T_i) για τη χρησιμοποίηση του πόρου, με a_i να είναι ο χρόνος εκκίνησης χρησιμοποίησης του πόρου και T_i ο χρόνος απελευθέρωσης του πόρου. Ο στόχος μας είναι να μεγιστοποιήσουμε το πλήθος των πελατών που θα χρησιμοποιήσουν τον πόρο.

Ας θεωρήσουμε το παράδειγμα του σχήματος 7.3. Η βέλτιστη λύση είναι η επιλογή του υποσυνόλου $\{P_3, P_4, P_5\}$ που ικανοποιεί τρεις πελάτες. Αντίθετα η λύση $\{P_1, P_2\}$ δεν είναι βέλτιστη αλλά είναι μέγιστη ως προς την έγκλιση (maximal), δηλαδή δεν μπορεί να προστεθεί κάποιο άλλο διάστημα.



Σχήμα 7.3: Ένα παράδειγμα του προβλήματος ανάθεσης πόρου.

Ένας άπληστος αλγόριθμος (βλ. σχήμα 7.4) για το πρόβλημα αυτό ταξινομεί τις αιτήσεις σύμφωνα με κάποιο άπληστο κριτήριο και έπειτα ελέγχει διαδοχικά τις αιτήσεις, ώστε η υπό κατασκευή λύση να παραμένει εφικτή. Η εφικτότητα εδώ σημαίνει ότι όταν εισάγουμε το αίτημα P_i δεν θα πρέπει να επικαλύπτεται με

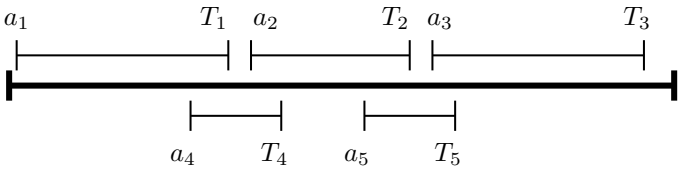
οποιοδήποτε από τα $P_j,\ 1 \leq j < i$.

ΑΝΑΘΕΣΗ-ΠΟΡΟΥ (P : σύνολο αιτήσεων)
1. ΤΑΞΙΝΟΜΗΣΗ(P)
2. $F = \emptyset$
3. **for** $i = 1$ **to** n
4. Επιλογή P_i
5. **if** $F \cup \{P_i\}$ feasible **then**
6. $F = F \cup \{P_i\}$
7. **end if**
8. **end while**
9. return F

Σχήμα 7.4: Άπληστος Αλγόριθμος Ανάθεσης Πόρου

Άρα θα πρέπει να ευρεθεί η ταξινόμηση που θα γίνει στα διαστήματα. Μία διαισθητικά λογική ταξινόμηση είναι η αύξουσα ταξινόμηση μεγέθους του διαστήματος, δηλαδή να δίνουμε πλεονέκτημα σε όσο το δυνατόν μικρότερα διαστήματα, με βάση το γεγονός ότι αφήνουν περισσότερο ελεύθερο χώρο για να επιλέξουμε και άλλα διαστήματα. Μάλιστα στο παράδειγμα του σχήματος 7.3 η προσέγγιση αυτή θα μας επέστρεφε μία βέλτιστη λύση του προβλήματος.

Ωστόσο αυτή η προσέγγιση δεν δουλεύει γενικά. Ας θεωρήσουμε το παράδειγμα του σχήματος 7.5. Εδώ η βέλτιστη λύση είναι το σύνολο $\{P_1, P_2, P_3\}$, ενώ η λύση που επιστρέφει η άπληστη μέθοδος με αύξουσα ταξινόμηση του μεγέθους του διαστήματος είναι η $\{P_4, P_5\}$.



Σχήμα 7.5: Ένα παράδειγμα του προβλήματος ανάθεσης πόρου που η επιλογή του μικρότερου δεν επιστρέφει την βέλτιστη λύση.

Ας θεωρήσουμε τώρα την αύξουσα ταξινόμηση τέλους διαστήματος, δηλαδή στο παράδειγμα του σχήματος 7.3 έχουμε τη διάταξη $[P_1, P_3, P_4, P_2, P_5]$ και στο

παράδειγμα του σχήματος 7.5 $[P_1, P_4, P_2, P_5, P_3]$. Βλέπουμε εδώ, ότι η εφαρμογή του άπληστου αλγορίθμου δίνει τις αντίστοιχες βέλτιστες λύσεις.

Θα αποδείξουμε γιατί αυτή η ταξινόμηση δίνει την βέλτιστη λύση του προβλήματος. Έστω

$$F = \{x_1, x_2, \ldots, x_k, \ldots, x_p\}$$

η λύση που επιστρέφει ο άπληστος αλγόριθμος και έστω

$$Opt = \{y_1, y_2, \ldots, y_k, \ldots, y_q\}$$

η βέλτιστη λύση ($q \geq p$). Θεωρούμε ότι οι δύο αυτές λύσεις είναι ίδιες μέχρι το $(k-1)$-οστό διάστημα και το k-οστό διάστημα είναι διαφορετικό, δηλαδή:

$$x_1 = y_1, x_2 = y_2, \ldots, x_{k-1} = y_{k-1}, x_k \neq y_k$$

Επειδή το x_k διάστημα είναι αυτό που επιλέγει ο άπληστος αλγόριθμος έπεται ότι το τέλος του διαστήματος x_k είναι μικρότερο από αυτό του διαστήματος y_k. Άρα μπορούμε να αντικαταστήσουμε στην βέλτιστη λύση το διάστημα y_k με το διάστημα x_k κερδίζοντας κάτι, αφού το x_k τελειώνει νωρίτερα. Κατασκευάζουμε έτσι την ισοδύναμη βέλτιστη λύση Opt' ως:

$$Opt' = \{x_1, x_2, \ldots, x_{k-1}, x_k, y_{k+1}, \ldots, y_q\}$$

Ωστόσο το ίδιο επιχείρημα μπορεί να επαναληφθεί και για τα υπόλοιπα στοιχεία $y_{k+1}, \ldots, y_q$ που σημαίναι ότι $p = q$ και ότι η λύση που επιστρέφει ο άπληστος αλγόριθμος είναι βέλτιστη.

7.3 Αποθήκευση αρχείων σε δίσκους

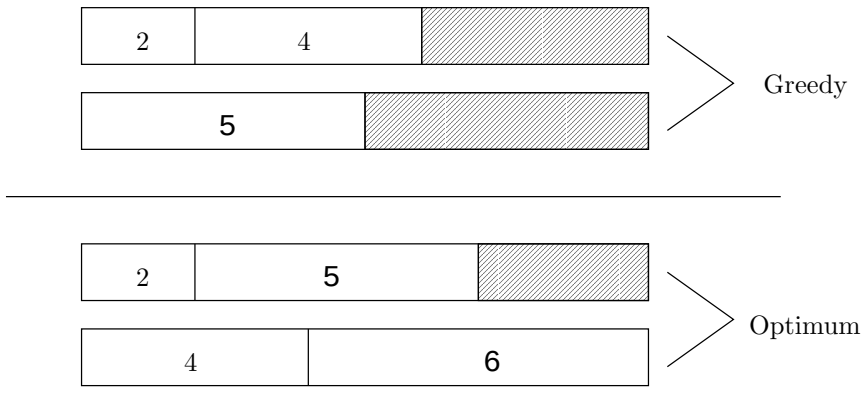
Δίδονται n αρχεία με χωρητικότητες $l_1, l_2, \ldots, l_n$ και δύο δίσκοι ίσης χωρητικότητας L. Θέλουμε να αποθηκεύσουμε όσο το δυνατο περισσότερα αρχεία στους δίσκους, θεωρώντας ότι κάθε αρχείο πρέπει να αποθηκευτεί σε έναν μόνο δίσκο και ότι $\sum_{i=1}^{n} l_i \leq 2L$.

Μία άπληστη μέθοδος για το πρόβλημα αυτό είναι η ταξινόμηση των αρχείων σε αύξουσα σειρά μεγεθών και η διαδοχική τοποθέτηση τους στον πρώτο δίσκο μέχρι να μη μπορούμε να τοποθετήσουμε κάποιο αρχείο. Αυτό το βάζουμε στον δεύτερο δίσκο και συνεχίζουμε. Στο σχήμα 7.6 φαίνεται αυτή η υλοποίηση.

```
ΑΠΟΘΗΚΕΤΣΗ-ΑΡΧΕΙΩΝ (l : σύνολο αρχείων, L : χωρητικότητα δίσκων )
 1.  ΑΥΞΟΤΣΑ-ΤΑΞΙΝΟΜΗΣΗ(l)
 2.  i = 1
 3.  for j = 1 to 2
 4.        sum = 0
 5.        while sum + l_i ≤ L
 6.              τύπωσε i → j
 7.              sum = sum + l_i
 8.              i = i + 1
 9.        end while
10.  end for
```

Σχήμα 7.6: Άπληστος αλγόριθμος αποθήκευσης αρχείων σε δύο ίσης χωρητικότητας δίσκους

Η μέθοδος αυτή δεν δίνει την βέλτιστη λύση, όπως φαίνεται στο παράδειγμα του σχήματος 7.7, όπου οι δίσκοι έχουν χωρητικότητα 10 και υπάρχουν 4 αρχεία με αντίστοιχα μεγέθη $2, 4, 5, 6$



Σχήμα 7.7: Παράδειγμα που ο άπληστος αλγόριθμος δεν επιστρέφει την βέλτιστη λύση. Πάνω είναι η λύση του άπληστου αλγορίθμου με κέρδος 3 και κάτω η βέλτιστη λύση με κέρδος 4.

Η εύρεση κάποιας ταξινόμησης των αρχείων, ώστε να βρεθεί η βέλτιστη λύση με την άπληστη μέθοδο δεν είναι εφικτή. Αυτό οφείλεται στο γεγονός ότι το πρόβλημα είναι NP-δύσκολο, πράγμα που σημαίνει ότι κατά πάσα πιθανότητα δεν

μπορούμε να βρούμε αποδοτικό (πολυωνυμικό) αλγόριθμο και ότι ο μόνος τρόπος για να βρούμε την βέλτιστη λύση είναι να κάνουμε εξαντλητική αναζήτηση σε όλες τις δυνατές επιλογές.

7.4 Το Διακριτό Πρόβλημα Σακιδίου (Discrete Knapsack)

Ένας αλπινιστής προετοιμάζει το σακίδιό του για την ανάβαση στο βουνό. Στη διάθεσή του έχει ένα σύνολο από n αντικείμενα $X = \{x_1, x_2, \ldots, x_n\}$ με βάρη $a_1, a_2, \ldots, a_n$. Κάθε αντικείμενο δίνει ένα κέρδος στον αλπινιστή c_i αν το επιλέξει για να το πάρει μαζί του στην ανάβαση. Ο αλπινιστής πρέπει να επιλέξει το υποσύνολο των αντικειμένων που μεγιστοποιούν το όφελός του, προσέχοντας να μην υπερβαίνεται το βάρος του σακιδίου b. Με άλλα λόγια, αν με $z_i \in \{0, 1\}$ συμβολίζουμε το αν ο αλπινιστής έχει επιλέξει το αντικείμενο i, ο στόχος είναι η μεγιστοποίηση του αθροίσματος:

$$\max \sum_{i=1}^{n} c_i z_i$$

με τον περιορισμό:

$$\sum_{i=1}^{n} a_i z_i \leq b$$

Μία λογική προσέγγιση για την επίλυση του προβλήματος με άπληστο αλγόριθμο, είναι η ταξινόμηση των αντικειμένων σύμφωνα με τον όρο c_i/a_i που εκφράζει το κέρδος του αλπινιστή ανά μονάδα βάρους του αντικειμένου. Τότε οδηγούμαστε στον άπληστο αλγόριθμο του σχήματος 7.8.

DISCRETE-KNAPSACK (X : σύνολο αντικειμένων, a, c : αντίστοιχα βάρη και αξίες, b : βάρος σακιδίου)

1. ΦΘΙΝΟΥΣΑ-ΤΑΞΙΝΟΜΗΣΗ(c_i/a_i)
2. $S = \emptyset$
3. **for** $i = 1$ **to** n
4. **if** $b \geq a_i$
5. $S = S \cup \{x_i\}$
6. $b = b - a_i$
7. **end if**
8. **end for**

Σχήμα 7.8: Άπληστος αλγόριθμος για το διακριτό πρόβλημα σακιδίου

Ωστόσο η προσέγγιση αυτή δεν δουλεύει πάντα. Ας θεωρήσουμε την εξής εφαρμογή: Δίδονται 3 αντικείμενα $\{x_1, x_2, x_3\}$ με βάρη $a = (k + 1, k, k)$ και αξίες

$c = (k+2, k, k)$. Το βάρος του σακιδίου είναι $2k$. Η ταξινόμηση των αντικειμένων σε φθίνουσα σειρά του λόγου c_i/a_i είναι x_1, x_2, x_3 αφού

$$\frac{k+2}{k+1} > \frac{k}{k} = \frac{k}{k}$$

Έτσι ο αλγόριθμος θα επιλέξει το αντικείμενο x_1 με κέρδος $k+2$ αντί να επιλέξει τα αντικείμενα x_2, x_3 με κέρδος $2k$. Βλέπουμε δηλαδή, ότι για μεγάλο k το λάθος που κάνει ο άπληστος αλγόριθμος προσεγγίζει το 100%.

Το πρόβλημα του σακιδίου είναι και αυτό $NP-$δύσκολο, οπότε η προσπάθεια εύρεσης μίας αποδοτικής σειράς εξερεύνησης των στοιχείων θα είναι αποτυχημένη. Ωστόσο στην επόμενη ενότητα, θα μελετήσουμε έναν αλγόριθμο της κατηγορίας του δυναμικού προγραμματισμού που επιλύει βέλτιστα το πρόβλημα σε ψευδοπολυωνυμικό χρόνο.

7.5 Το Συνεχές Πρόβλημα Σακιδίου (Continuous Knapsack)

Το συνεχές πρόβλημα σακιδίου, είναι η παραλλαγή του προβλήματος σακιδίου που επιτρέπεται στον αλπινιστή να πάρει μέρη των αντικειμένων. Με άλλα λόγια οι μεταβλητές απόφασης z_i του ακέραιου προβλήματος παίρνουν συνεχείς τιμές στο $[0, 1]$.

Εδώ η φθίνουσα ταξινόμηση με βάση τον λόγο c_i/a_i δουλεύει αποδοτικά και βρίσκει την βέλτιστη λύση των αντικειμένων που πρέπει να επιλέξει ο αλπινιστής για να έχει μεγιστοποίηση του κέρδους του. Ο άπληστος αλγόριθμος φαίνεται στο σχήμα 7.9

CONTINUOUS-KNAPSACK (X : σύνολο αντικειμένων, a, c : αντίστοιχα βάρη και αξίες, b : βάρος σακιδίου)

1. ΦΘΙΝΟΤΣΑ-ΤΑΞΙΝΟΜΗΣΗ(c_i/a_i)
2. **for** $i = 1$ **to** n Θέσε $z_i = 0$
3. $i = 1$
3. **while** $b \geq a_i$ **do**
4. $\quad z_i = 1$
5. $\quad b = b - a_i$
6. $\quad i = i + 1$
7. **end while**
8. $z_i = b * (c_i/a_i)$

Σχήμα 7.9: Άπληστος αλγόριθμος για το συνεχές πρόβλημα σακιδίου

Κεφάλαιο 8

Δυναμικός Προγραμματισμός

8.1 Εισαγωγικά

Ο δυναμικός προγραμματισμός[1] (dynamic programming), αποτελεί ένα ισχυρό εργαλείο στην ανάπτυξη αποτελεσματικών αλγορίθμων, για την επίλυση προβλημάτων, κύρια συνδυαστικών προβλημάτων βελτιστοποίησης.

Έχουμε ήδη γνωρίσει μια κατηγορία αλγορίθμων, που μοιάζει με τον δυναμικό προγραμματισμό, χωρίς όμως να ταυτίζεται. Πρόκειται για τους *διαίρει - και - βασίλευε* αλγορίθμους. Στην κατηγορία αυτή, η διαδικασία επίλυσης ενός προβλήματος, έγκειται στην υποδιαίρεσή του, σε μια σειρά από μικρότερα ανεξάρτητα υποπροβλήματα τα οποία επιλύουμε αναδρομικά και συνδυάζουμε τις λύσεις τους για να εξάγουμε την λύση στο αρχικό πρόβλημα.

Στον δυναμικό προγραμματισμό, αντίθετα, η επίλυση ενός προβλήματος, έρχεται από την λύση μικρότερων υποπροβλημάτων τα οποία **αλληλοεπικαλύπτονται (overlapping subproblems)**. Κάθε υποπρόβλημα λύνεται μια μόνο φορά και η βέλτιστη λύση αποθηκεύεται σ΄ ένα πίνακα, ώστε να ανακαλείται από εκεί, κάθε φορά που το συγκεκριμένο υποπρόβλημα συναντάται.

Όπως θα δούμε και στη συνέχεια πιο αναλυτικά, τέσσερα είναι τα βασικά βήματα που ακολουθούμε για την ανάπτυξη ενός αλγορίθμου δυναμικού προγραμματισμού.

1. Χαρακτηρισμός της δομής της βέλτιστης λύσης.
2. Αναδρομικός ορισμός της τιμής της βέλτιστης λύσης.
3. Υπολογισμός της τιμής της βέλτισης λύσης από *κάτω προς τα πάνω*.
4. Κατασκευή της βέλτισης λύσης με χρήση πίνακα.

[1] Ο όρος *προγραμματισμός* προέρχεται από μία συστηματική μέθοδο που χρησιμοποιεί πίνακες (tabloid method) και δεν σχετίζεται με την ανάπτυξη κώδικα σε κάποια γλώσσα προγραμματισμού

Στις επόμενες ενότητες θα αναπτύξουμε πιο συστηματικά, τα βασικά συστατικά ενός αλγορίθμου δυναμικού προγραμματισμού και θα προχωρήσουμε στην επίλυση μερικών προβλημάτων με τη χρήση της τεχνικής αυτής.

8.2 Βασικά στοιχεία του δυναμικού προγραμματισμού

Όπως προαναφέρθηκε, ο δυναμικός προγραμματισμός εφαρμόζεται σε προβλήματα βελτιστοποίησης. Επομένως ένα εύλογο ερώτημα που προκύπτει είναι το αν μπορούμε πάντα να αναπτύσσουμε έναν τέτοιο αλγόριθμο. Στα επόμενα θα δούμε τα σημαντικότερα χαρακτηριστικά του δυναμικού προγραμματισμού και πότε θα μπορούμε να εφαρμόζουμε έναν τέτοιο αλγόριθμο.

8.2.1 Βέλτιστα διασπώμενη δομή (optimal substructure)

Το πρώτο βήμα στην ανάπτυξη ενός αλγορίθμου δυναμικού προγραμματισμού είναι, όπως αναφέρθηκε και στην αρχή του κεφαλαίου, ο χαρακτηρισμός της δομής της βέλτιστης λύσης. Θα πρέπει να έχουμε πάντα κατά νου την βασική αρχή που λέει ότι:

Κάθε υποστρατηγική μιας βέλτιστης στρατηγικής είναι και η ίδια βέλτιστη.

Αυτό σημαίνει πρακτικά ότι η βέλτιστη λύση ενός προβλήματος, εμπεριέχει βέλτιστες λύσεις υποπροβλημάτων του και κατά συνέπεια θα πρέπει να διασφαλίζουμε ότι τα υποπροβλήματα που επιλύουμε, είναι αυτά που εμπεριέχει η βέλτιστη λύση του αρχικού προβλήματος.

Όπως θα δούμε και στα παραδείγματα, υπάρχει μια μεθοδολογία για να ανακαλύψουμε την βέλτιστα διασπώμενη δομή ενός προβλήματος, η οποία συνίσταται στα εξής:

1. Αποδεικνύουμε ότι μια οποιαδήποτε λύση στο πρόβλημα, συνίσταται στο να κάνουμε μια επιλογή. Η επιλογή αυτή οδηγεί σε ένα ή περισσότερα υποπροβλήματα τα οποια πρέπει να επιλυθούν. Για παράδειγμα στο πρόβλημα των ελαχίστων μονοπατιών ενός γράφου, η επιλογή αυτή είναι η παρεμβολή ενός κόμβου, έστω p ανάμεσα στον κόμβο αφετηρία u και στον κόμβο προορισμό v. Τότε για να βρούμε ένα μονοπάτι από τον u στον v, αρκεί να βρούμε ένα μονοπάτι από τον u στον p και ένα από τον p στον v.

2. Θεωρούμε τώρα ότι έχουμε δεδομένη την επιλογή η οποία οδηγεί στην βέλτιστη λύση, χωρίς να μας απασχολεί προς το παρόν, πως ορίζεται αυτή.

3. Δεδομένης λοιπόν της επιλογής αυτής, ορίζουμε τα νέα υποπροβλήματα. Για παράδειγμα αν θεωρήσουμε και πάλι το πρόβλημα των ελαχίστων μονοπατιών σε γράφο, τότε αν υποθέσουμε ότι γνωρίζουμε τον κόμβο p που θα οδηγήσει στο ελάχιστο μονοπάτι μεταξύ των u και v, τότε παράγουμε δύο νέα

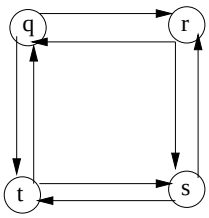
υποπροβλήματα. Το πρώτο είναι το ελάχιστο μονοπάτι από τον u στον p και το δεύτερο αντίστοιχα από τον p στον v.

4. Τέλος αποδεικνύουμε όντως ότι οι βέλτιστες λύσεις των υποπροβλημάτων αυτών οδηγούν στη βέλτιστη λύση του αρχικού προβλήματος. Η απόδειξη αυτή γίνεται συνήθως με *εις άτοπον απαγωγή*, θεωρώντας δηλαδή ότι η λύση καθενός από τα υποπροβλήματα δεν είναι βέλτιστη και έτσι οδηγούμαστε σε άτοπο.

Η τεχνική του δυναμικού προγραμματισμού χρησιμοποιεί την βέλτιστα διασπώμενη δομή, από **κάτω προς τα πάνω** (bottom-up). Δηλαδή, αρχικά επιλύουμε βέλτιστα μικρά υποπροβλήματα και συνδυάζουμε τις λύσεις αυτές ώστε να υπολογίσουμε την βέλτιστη λύση σε όλο και μεγαλύτερα, μεχρι να καταλήξουμε στο αρχικό πρόβλημα. Ο υπολογισμός αυτός απαιτεί να κάνουμε τις επιλογές εκείνων των υποπροβλημάτων που οδηγούν στην βέλτιστη λύση και εδώ ακριβώς χρησιμοποιείται η έννοια του αναδρομικού ορισμού της βέλτιστης λύσης, που αναφέραμε και αρχικά. Επί της ουσίας χρησιμοποιούμε μια αναδρομική σχέση ώστε να υπολογίσουμε την βέλτιστη λύση του αρχικού προβλήματος.

Παρατηρήσεις Αξίζει να σημειωθεί εδώ ότι πρέπει να είναι κανείς ιδιαίτερα προσεκτικός όταν προσπαθεί να δείξει ότι ένα πρόβλημα έχει την ιδιότητα της βέλτιστα διασπώμενης δομής. Ένα αυστηρό κριτήριο που διασφαλίζει την βέλτιστα διασπώμενη δομή είναι αυτό της **ανεξαρτησίας** (independency) των υποπροβλημάτων, εννοώντας ότι η λύση ενός προβλήματος δεν επηρεάζει τα υπόλοιπα. Για να γίνει πιο κατανοητό αυτό, ας θεωρήσουμε δυο γνωστά προβλήματα της θεωρίας γράφων, το πρόβλημα του ελάχιστου μονοπατιού σε ένα γράφο χωρίς βάρη και αυτό του μέγιστου απλού μονοπατιού επίσης σε ένα γράφο χωρίς βάρη.

Στο πρόβλημα του ελάχιστου μονοπατιού σκιαγραφήθηκε η βέλτιστα διασπώμενη δομή στα παραπάνω. Χρησιμοποιώντας τα ίδια επιχειρήματα, θα μπορούσε κάποιος να υποθέσει ότι και για την περίπτωση του μέγιστου απλού μονοπατιού μπορούμε να θεωρήσουμε ότι η βέλτιστα διασπώμενη δομή έγκειται και πάλι στο να βρούμε έναν κόμβο w που να παρεμβάλλεται ανάμεσα στους u και v, ώστε το μέγιστο μονοπάτι $u \rightsquigarrow v$ βρίσκεται υπολογίζοντας τα μέγιστα απλά μονοπάτια $u \rightsquigarrow w$ και $w \rightsquigarrow v$. Αυτό όμως δεν είναι σωστό και γίνεται εύκολα αντιληπτό από το σχήμα 8.1.



Σχήμα 8.1: Ο γράφος του σχήματος δείχνει ότι δεν υπάρχει βέλτιστα διασπώμενη δομή για το πρόβλημα του μέγιστου απλού μονοπατιού

Αν για παράδειγμα ζητάμε να βρούμε το μέγιστο απλό μονοπάτι $q \rightsquigarrow t$, τότε αυτό μπορεί να λυθεί βρίσκοντας δυο μέγιστα απλά μονοπάτια $p_1 : q \rightsquigarrow r$ και $p_2 : r \rightsquigarrow t$. Όμως το p_1 είναι το $q \to t \to s \to r$, που χρησιμοποιεί τις κορυφές t και s. Επομένως το δεύτερο πρόβλημα δεν μπορεί να χρησιμοποιήσει αυτές τις κορυφές, διότι ο συνδυασμός των λύσεων δίνει μέγιστο μονοπάτι που δεν είναι απλό. Επομένως δεν υπάρχει λύση για το δεύτερο υποπρόβλημα πράγμα που συνεπάγεται ότι και το αρχικό μας πρόβλημα δεν έχει λύση.

8.2.2 Επικαλυπτόμενα υποπροβλήματα (Overlapping subproblems)

Το δεύτερο χαρακτηριστικό στοιχείο ενός αλγορίθμου δυναμικού προγραμματισμού είναι η ιδιότητα των **επικαλυπτόμενων υποπροβλημάτων**.

Κάθε αλγόριθμος δυναμικού προγραμματισμού, παράγει πολυωνυμικό, ως προς το μέγεθος της εισόδου, πλήθος διαφορετικών υποπροβλημάτων, τα οποία επιλύει και επιπλέον κάθε ένα από αυτά τα υποπροβλήματα μπορεί να εμφανιστεί αρκετές φορές κατά τη διάρκεια της εκτέλεσης. Η έννοια λοιπόν της επικάλυψης σχετίζεται ακριβώς με το γεγονός ότι κάθε ένα από τα υποπροβλήματα εμφανίζεται, πιθανόν, αρκετές φορές κατά την εκτέλεση. Ο πλεονασμός αυτό αποφεύγεται με την ακόλουθη διαδικασία:

Μόλις επιλυθεί ένα υποπρόβλημα, η τιμή της βέλτιστης λύσης του, αποθηκεύεται σ' ένα πίνακα. Όταν απαιτηθεί να επιλυθεί ξανά, τότε απλά ανακαλείται η τιμή της βέλτισης λύσης του υποπροβλήματος από τον πίνακα (σε σταθερό χρόνο).

Παρατηρήσεις Δεν θα πρέπει να δημιουργείται σύγχυση ανάμεσα στην ιδιότητα της επικάλυψης με αυτή της ανεξαρτησίας για την οποία μιλήσαμε προηγουμένως. Στην πρώτη περίπτωση μιλάμε για το ίδιο υποπρόβλημα που εμφανίζεται περισσότερες φορές κατά την διάρκεια της εκτέλεσης του αλγορίθμου, ενώ στην δεύτερη πρόκειται για ανεξαρτησία των δεδομένων του υποπροβλήματος. Κάθε ένα από τα υποπροβλήματα έχει τα δικά του αποκλειστικά δεδομένα.

Η ιδιότητα των επικαλυπτόμενων υποπροβλημάτων, αποτελεί και βασικό χαρακτηριστικό που καθορίζει την αποδοτικότητα (πολυπλοκότητα) ενός αλγορίθμου δυναμικού προγραμματισμού. Αυτό θα γίνει καλύτερα αντιληπτό, όταν εξετάσουμε τα συγκεκριμένα παραδείγματα.

8.3 Εφαρμογές

Η βασική δυσκολία του δυναμικού προγραμματισμού εγκειται στο γεγονός ότι δεν υπάρχει καμία *μαγική συνταγή* για να βρίσκουμε την βέλτιστα διασπώμενη δομή (αν υπάρχει) ενός προβλήματος. Ο δρόμος έρχεται μέσα από την διαρκή εξάσκηση και μελέτη, των ήδη υπάρχοντων αλγορίθμων, ώστε να αναπτυχθεί η σχετική οικειότητα με το αντικείμενο.

Στην ενότητα αυτή θα παρουσιάσουμε με όσο το δυνατόν πιο απλό και κατανοητό τρόπο μερικές εφαρμογές του δυναμικού προγραμματισμού, σε αντίστοιχα προβλήματα, έτσι ώστε να υπάρξει μια άμεση και πρακτική επαφή με την τεχνική.

Αξίζει εδώ να σημειώσουμε ότι δεν είναι η πρώτη φορα που μελετάμε αλγορίθμους δυναμικού προγραμματισμού, αν παρατηρήσουμε μόνο ότι ο γνωστός αλγόριθμος του Bellman για την εύρεση ελάχιστου μονοπατιού σε γράφο, είναι επίσης ένας αλγόριθμος αυτής της κατηγορίας.

8.3.1 Χρονοδρομολόγηση γραμμής παραγωγής

Το παράδειγμα που θα παρουσιάσουμε παρακάτω είναι μια ειδική περίπτωση του γνωστού προβλήματος εύρεσης του συντομότερου μονοπατιού μεταξύ δυο κόμβων σε ένα γράφο.

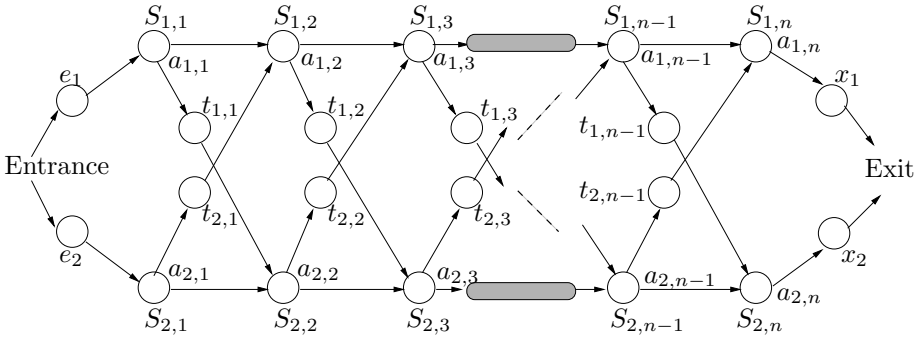
Ένα εργοστάσιο κατασκευής αυτοκινήτων διαθέτει δύο γραμμές παραγωγής, καθεμιά από τις οποίες διαθέτει n σταθμούς συναρμολόγησης. Καθε σταθμός συμβολίζεται με S_{ij}. Οι σταθμοί S_{1j} και S_{2j} εκτελούν ακριβώς την ίδια εργασία αλλά όχι απαραίτητα στον ίδιο χρόνο. Συμβολίζουμε με a_{ij}, το χρόνο που απαιτείται από τον σταθμό S_{ij} για την εκτέλεση μιας συγκεκριμένης λειτουργίας. Συμβολίζουμε επίσης με e_i, το χρόνο που απαιτείται για την εισαγωγή του, προς συναρμολόγηση, αυτοκινήτου στην γραμμή i, όπου $i \in \{1, 2\}$ και με x_i τον χρόνο που απαιτείται για την έξοδο από την i γραμμή.

Αντικειμενικός σκοπός είναι να ελαχιστοποιήσουμε το χρόνο συναρμολόγησης ενός αυτοκινήτου. Για να επιτευχθεί αυτό, το υπο κατασκευή, αυτοκίνητο μπορεί, να αλλάζει γραμμή παραγωγής, κατά τη διαδικασία της συναρμολόγησής του. Για την αλλαγή γραμμής απαιτείται ορισμένος χρόνος. Συμβολίζουμε λοιπόν με t_{ij} το χρόνο που χρειάζεται το αυτοκίνητο για να μεταφερθεί στην άλλη γραμμή έχοντας βγεί από το σταθμό S_{ij}, $j = 1, 2, \ldots, n-1$.

Το πρόβλημα λοιπόν διατυπώνεται ως εξής:

Ζητάμε να αποφασίσουμε από ποιούς σταθμούς θα περάσει το αυτοκίνητο έτσι ώστε να ελαχιστοποιήσουμε το χρόνο κατασκευής του.

Το σχήμα 8.2 δείχνει ένα εργοστάσιο με δυο γραμμές παραγωγής. Κάθε αυτοκίνητο μπορεί να εισαχθεί είτε στη γραμμή 1 ή στη γραμμή 2.



Σχήμα 8.2: Εργοστάσιο κατασκευής αυτοκινήτων με δύο γραμμές παραγωγής

Βήμα 1: Βέλτιστα διασπώμενη δομή

Ας θεωρήσουμε το συντομότερο μονοπάτι μέσω του οποίου μπορεί να φτάσει ένα αυτοκίνητο στο σταθμό S_{1j} ξεκινώντας από την αφετηρία.

Αν $j = 1$ (δηλαδή είμαστε στον πρώτο σταθμό), τότε υπάρχει μόνο ένα μονοπάτι. Διαφορετικά ($j = 2, 3, \ldots, n$) υπάρχουν δύο επιλογές όπως φαίνεται και από το σχήμα 8.2. Είτε μέσω του σταθμού S_{1j-1} ή μέσω του σταθμού S_{2j-1}, απαιτώντας επιπλέον χρόνο t_{2j-1}.

Για να δείξουμε ότι αυτές οι δύο επιλογές οδηγούν σε συντομότερο μονοπάτι σκεφτόμαστε ως εξής:

Αν το συντομότερο μονοπάτι για να φτάσει το αυτοκίνητο στον S_{1j} είναι μέσω του S_{1j-1}, τότε το μόνοπάτι που καταλήγει στον S_{1j-1} είναι και αυτό με τη σειρά του το συντομότερο. Αν δεν ήταν συντομότερο, θα υπήρχε άλλο μονοπάτι, συντομότερο, που καταλήγει στον S_{1j-1} και επομένως μπορούμε να χρησιμοποιήσουμε αυτό για να καταλήξουμε στον S_{1j} και επομένως το μονοπάτι που έχουμε μέχρι τον S_{1j} δεν είναι το συντομότερο, που είναι άτοπο.

Εργαζόμενοι με ακριβώς όμοιο τρόπο καταλήγουμε στο συμπέρασμα ότι αν το συντομότερο μονοπάτι προς τον S_{1j} διέρχεται από τον σταθμό S_{2j-1}, τότε και το μονοπάτι προς τον S_{2j-1} είναι και αυτό το συντομότερο.

Επομένως η **βέλτιστα διασπώμενη δομή** του προβλήματος διατυπώνεται ως εξής:

Η βέλτιστη λύση του προβλήματος της εύρεσης του συντομότερου μονοπατίου μέχρι τον S_{1j} εμπεριέχει τις βέλτιστες λύσεις στα υποπροβλήματα των συντομότερων μονοπατιών προς τους σταθμούς S_{1j-1} και S_{2j-1}.

Δηλαδή, το συντομότερο μονοπάτι έως τον σταθμό S_{1j} είναι:

- Το συντομότερο μονοπάτι έως τον S_{1j-1} και στην συνέχεια κατευθείαν στον S_{1j}
- Το συντομότερο μονοπάτι έως τον S_{2j-1} και στη συνέχεια μεταφορά στον S_{1j} σε χρόνο t_{2j-1}

Με συμμετρικό τρόπο, ορίζουμε το συντομότερο μονοπάτι έως τον S_{2j} σαν

- Το συντομότερο μονοπάτι έως τον S_{2j-1} και στην συνέχεια κατευθείαν στον S_{2j}
- Το συντομότερο μονοπάτι έως τον S_{1j-1} και στη συνέχεια μεταφορά στον S_{2j} σε χρόνο t_{1j-1}

Βήμα 2: Αναδρομικός ορισμός της τιμής της βέλτιστης λύσης

Αφού έχουμε δείξει την βέλτιστα διασπώμενη δομή του προβλήματος, ερχόμαστε τώρα να ορίσουμε με αναδρομικό τρόπο την τιμή της βέλτιστης λύσης, δηλαδή τον χρόνο που απαιτείται για να συναρμολογηθεί ένα αυτοκίνητο ακολουθώντας το συντομότερο μονοπάτι.

Ορίζουμε τον πίνακα f διάστασης $2 \times n$, όπου στη θέση f_{ij} ($i = 1, 2$ και $j = 1, 2, \ldots, n$) αποθηκεύουμε τον ελάχιστο χρόνο μέχρι τον σταθμό S_{ij}.

Η βέλτιστη λύση του προβλήματος θα είναι:

$$f^* = \min\{f_{1n} + x_1, f_{2n} + x_2\} \tag{8.1}$$

διότι έχοντας ακολουθήσει κάποια διαδρομή, το αυτοκίνητο καταλήγει είτε στο σταθμό S_{1n}, ή στον S_{2n}.

Θα διατυπώσουμε τώρα τις δύο αναδρομικές σχέσεις που υπολογίζουν τις δύο γραμμές του πίνακα f.

Έχουμε:

$$f_{1j} = \begin{cases} e_1 + a_{11} & j = 1 \\ \min\{f_{1j-1} + a_{1j}, f_{2j-1} + t_{2j-1} + a_{1j}\} & j \geq 2 \end{cases} \tag{8.2}$$

και όμοια:

$$f_{2j} = \begin{cases} e_2 + a_{21} & j = 1 \\ \min\{f_{2j-1} + a_{2j}, f_{1j-1} + t_{1j-1} + a_{2j}\} & j \geq 2 \end{cases} \tag{8.3}$$

Το πρώτο σκέλος κάθε μιας εκ των δύο εξισώσεων είναι προφανής, αφού για να φτάσει κάθε αυτοκίνητο στον πρώτο σταθμό της i γραμμής παραγωγής, υπάρχει μόνο ένα μονοπάτι με κόστος $e_i + a_{i1}$.

Το δεύτερο σκέλος προκύπτει εύκολα από την βέλτιστα διασπώμενη δομή που δείξαμε προηγουμένως.

Παρατηρήσεις Στις σχέσεις (8.2) και (8.3) εμφανίζεται ο όρος a_{ij}, που ορίσαμε αρχικά. Κατά την ανάλυση της βέλτιστα διασπώμενης δομής δεν αναφερθήκαμε ρητά στον όρο αυτό, αλλά εννοείται ότι εμπεριέχεται στο υποπρόβλημα.

Βήμα 3: Υπολογισμός του ελάχιστου χρόνου

Στο βήμα αυτό, θα κατασκευάσουμε έναν αποδοτικό αλγόριθμο δυναμικού προγραμματισμού, για τον υπολογισμό του ελαχίστου χρόνου που απαιτείται για την κατασκευή ενός αυτοκινήτου. Ο προφανής αναδρομικός αλγόριθμος που προκύπτει από τις σχέσεις (8.2) και (8.3) δεν αποτελεί την καλύτερη επιλογή. Ας δούμε το γιατί.

Έστω $r_i(j)$, $i = 1, 2$ και $j = 1, 2, \ldots, n$ ο αριθμός των αναφορών (πόσες φορές υπολογίζουμε) στο υποπρόβλημα που υπολογίζει το f_{ij}. Τότε:

$$r_1(n) = r_2(n) = 1 \tag{8.4}$$

Αλλά

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1) \tag{8.5}$$

Η (8.5) έχει λύση:

$$r_i(j) = 2^{n-j} \tag{8.6}$$

Δηλαδή ο αναδρομικός αλγόριθμος υπολογίζει το καθένα από τα υποπροβλήματα f_{ij} εκθετικό πλήθος φορών (μόνο το f_{11} υπολογίζεται 2^{n-1} φορές!). Επομένως η πολυπλοκότητα του αναδρομικού αλγορίμου είναι $\Theta(2^n)$.

Στο ερώτημα, αν μπορούμε να κάνουμε καλύτερα από εκθετικά, η απάντηση είναι ναι και ας δούμε πως.

Αρκεί να παρατηρήσουμε ότι για $j \geq 2$, η τιμή f_{ij} εξαρτάται μόνο από τις τιμές των $f_{1,j-1}$ και $f_{2,j-1}$. Επομένως μπορούμε να αποθηκεύουμε κάθε τιμή f_{ij} σε ένα πίνακα (διαστάσεων $2 \times n$) και να ανακαλούμε κάθε φορά την τιμή από κάποιο προηγούμενο βήμα. Με τον τρόπο αυτό μπορούμε να υπολογίσουμε την βέλτιστη διαδρομή που θα ακολουθήσει το αυτοκίνητο, σε χρόνο $\Theta(n)$.

Στον αλγόριθμο που παρουσιάζουμε παρακάτω, χρησιμοποιούμε έναν επιπλέον βοηθητικό πίνακα l, (διάστασης επίσης $2 \times n$). Ο πίνακας αυτός διατηρεί το μονοπάτι το οποίο ακολουθεί, το υπό κατασκευή αυτοκίνητο, από την είσοδο μέχρι την έξοδο. Στο τέλος του αλγορίθμου έχουμε τον ελάχιστο χρόνο f^*, όπως επίσης και την γραμμή παραγωγής l^*, από την οποία εξήλθε το αυτοκίνητο. Έτσι ξεκινώντας αντίστροφα (δηλαδή από την έξοδο προς την αρχή) μπορούμε να ανακατασκευάσουμε το συνολικό μονοπάτι που ακολούθησε το αυτοκίνητο κατά τη διάρκεια της κατασκευής του.

Το τελευταίο βήμα που είναι η **κατασκευή της βέλτισης λύσης**, είναι εύκολη διαδικασία, χρησιμοποιώντας μόνο τον πίνακα l που ορίσαμε προηγουμένως, και αφήνεται ως άσκηση στον αναγνώστη. Επίσης μια χρήσιμη εφαρμογή θα ήταν η ανάπτυξη μιας μεθόδου που να υπολογίζει την ακριβή διαδρομή, αυτή τη φορά όμως από την είσοδο έως την έξοδο (μιας και ο αλγόριθμος FASTEST-WAY υπολογίζει το μονοπάτι σε αντίστροφη σειρά).

8.3.2 Αλυσιδωτός πολλαπλασιασμός πινάκων

Στο πρόβλημα αυτό μας δίνεται μια ακολουθία από n πίνακες, $\langle A_1, A_2, \ldots, A_n \rangle$, Ζητάμε να υπολογίσουμε το γινόμενο

$$A_1 \cdot A_2 \cdot \ldots A_n \tag{8.7}$$

όσο το δυνατόν πιο αποδοτικά (με το λιγότερο δυνατό κόστος). Γνωρίζουμε γενικά ότι το κόστος πολλαπλασιασμού δύο πινάκων A, B με διαστάσεις $p \times q$ και $q \times r$ αντίστοιχα, είναι $p \times q \times r$. Από τη διαπίστωση αυτή προκύπτει εύκολα ότι το κόστος του πολλαπλασιασμού (8.7) μεταβαλλεται ανάλογα με τη σειρά που θα εκτελέσουμε τους πολλαπλασιασμούς. Για να το επιτύχουμε αυτό, θα χρησιμοποιήσουμε την έννοια της πλήρους παρενθετοποίησης (**full parenthesization**), η οποία ορίζει την σειρά προτεραιότητας των πολλαπλασιασμών (Το περισσότερο εμφωλιασμένο γινόμενο έχει μεγαλύτερη προτεραιότητα). Δίνουμε τον ακόλουθο αναδρομικό ορισμό

Ορισμός 1 (Πλήρης παρενθετοποίηση)**.** Μια πλήρης παρενθετοποίηση είναι είτε ένας πίνακας είτε το γινόμενο δύο πλήρως παρενθετοποιημένων πινάκων, μέσα σε παρενθέσεις.

```
FASTEST-WAY(a, t, e, x, n)
 1.  f_11 ← e_1 + a_{1,1}
 2.  f_21 ← e_2 + a_{2,1}
 3.  for j = 2 to n do
 4.      if f_{1,j-1} + a_{1,j} ≤ f_{2,j-1} + t_{2,j-1} + a_{1,j} then
 5.          f_{1j} = f_{1,j-1} + a_{1,j}
 6.          l_{1j} = 1
 7.      else
 8.          f_{1j} = f_{2,j-1} + t_{2,j-1} + a_{1,j}
 9.          l_{1j} = 2
10.      end if
11.      if f_{2,j-1} + a_{2,j} ≤ f_{1,j-1} + t_{1,j-1} + a_{2,j} then
12.          f_{2j} = f_{2,j-1} + a_{2,j}
13.          l_{2j} = 2
14.      else
15.          f_{2j} = f_{1,j-1} + t_{1,j-1} + a_{2,j}
16.          l_{2j} = 1
17.      end if
18.  if f_{1n} + x_1 ≤ f_{2n} + x_2 then
19.      f* = f_{1n} + x_1
20.      l* = 1
21.  else
22.      f* = f_{2n} + x_2
23.      l* = 2
24.  end if
```

Παράδειγμα Έστω το γινόμενο $A_1 \cdot A_2 \cdot A_3 \cdot A_4$. Μια πλήρης παρενθετοποίηση του γινομένου αυτού είναι η:

$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$

ή ακόμα και η

$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$

Παρατήρηση Αν οι διαστάσεις των πινάκων είναι για παράδειγμα $10 \times 20, 20 \times 30, 30 \times 40$ και 40×50 τότε η πρώτη παρενθετοποίηση δίνει: 74.000 πολλαπλασιασμούς, ενώ η δεύτερη παρενθετοποίηση 81.000. Γενικά, εάν μας δίνονται n πίνακες υπάρχουν $\Omega(2^n)$ διαφορετικές παρενθετοποιήσεις του γινομένου τους. Επομένως η προφανής τεχνική της εξερεύνησης όλων των δυνατών παρενθετοποιήσεων δεν έχει πρακτική αξία. Στα επόμενα θα συζητάμε για την παρενθετοποίηση δύο πινάκων, και θα εννοούμε ισοδύναμα τον πολλαπλασιασμό τους.

Άσκηση: Να δειχθεί ότι ο υπολογισμός του αριθμού των παρενθετοποιήσεων δίδεται από την αναδρομή:

$$T(n) = \begin{cases} 1 & \text{αν } n = 1 \\ \sum_{k=1}^{n-1} T(k)T(n-k) & \text{αν } n \geq 2 \end{cases} \tag{8.8}$$

Επίσης να δειχθέι ότι η λύση είναι $\Omega(2^n)$.

Βήμα 1: Βέλτιστα διασπώμενη δομή

Όπως και πριν το πρώτο βήμα είναι να ορίσουμε την βέλτιστα διασπώμενη δομή, ώστε στη συνέχεια να χτίσουμε την βέλτιστη λύση στο πρόβλημα. Στην ανάλυση που ακολουθεί θα χρησιμοποιήσουμε τον συμβολισμό A_{ij} εννοώντας το γινόμενο $A_i \cdot A_{i+1} \dots A_j$ όπου $i \leq j$. Επομένως μια πλήρης παρενθετοποίηση του γινομένου A_{ij} συνίσταται στην πλήρη παρενθετοποίηση των γινομένων A_{ik} και $A_{k+1,j}$ για κάποιο $i \leq k < j$. Το κόστος λοιπόν της παρενθετοποίησης του γινομένου A_{ij} είναι το άθροισμα του κόστους του γινομένου A_{ik} και του $A_{k+1,j}$ συν το κόστος του πολλαπλασιασμού των δύο αυτών γινομένων.

Συμβολικά έχουμε:

$$\text{cost}(A_{ij}) = \text{cost}(A_{ik}) + \text{cost}(A_{k+1,j}) + \text{cost}(A_{ik} \cdot A_{k+1,j}) \tag{8.9}$$

Επομένως η βέλτιστη παρενθετοποίηση του γινομένου A_{ij} είναι αυτή που προκύπτει από την βέλτιστη παρενθετοποίηση των γινομένων A_{ik} και $A_{k+1,j}$.

Η βέλτιστη λύση του προβλήματος μας, έγκειται στον υπολογισμό του ελαχίστου κόστους της παρενθετοποίησης του γινομένου A_{1n}.

Βήμα 2: Αναδρομικός ορισμός της τιμής της βέλτιστης λύσης

Στην εξίσωση (8.9) επεισέρχεται ο όρος $\text{cost}(A_{ik} \cdot A_{k+1,j})$, που είναι το κόστος υπολογισμού του γινομένου των πινάκων A_{ik}, $A_{k+1,j}$. Ο πίνακας A_{ik} έχει διαστάσεις $p_{i-1} \times p_k$ και αντίστοιχα ο $A_{k+1,j}$, $p_k \times p_j$. Επομένως

$$\text{cost}(A_{ik} \cdot A_{k+1,j}) = p_{i-1} \times p_k \times p_j. \qquad 1 \leq i \leq k < j \leq n$$

Ορίζουμε τον πίνακα m διάστασης $n \times n$, όπου στη θέση $m[i,j]$ αποθηκεύουμε το ελάχιστο κόστος που απαιτείται για τον υπολογισμό του γινομένου A_{ij}. Με βάση αυτό προκύπτει ότι στη θέση $m[1,n]$ αποθηκεύεται το ελάχιστο κόστος υπολογισμού του γινομένου A_{1n}. Ας δώσουμε τώρα τον πλήρη ορισμό του πίνακα m. Αν $i = j$ τότε το γινόμενο $A_{ii} = A_i$ δηλαδή εκφυλίζεται στην τετριμένη περίπτωση του ενός πίνακα. Επομένως $m[i,i] = 0$ για $i = 1, 2 \dots, n$.

Ας δούμε τώρα την περίπτωση όπου $i < j$. Η βέλτιστη παρενθετοποίηση του γινομένου A_{ij}, το διαχωρίζει στα γινόμενα A_{ik} και $A_{k+1,j}$. Επομένως το κόστος μιας παρενθετοποίησης του γινομένου A_{ij} θα είναι ίσο με το άθροισμα $m[i,k] + m[k,j] + p_{i-1} \times p_k \times p_j$ και αντίστοιχα το ελάχιστο κόστος

$$m[i,j] = \min_{i \leq k < j} \Big\{ m[i,k] + m[k,j] + p_{i-1} \cdot p_k \cdot p_j \Big\}.$$

Συνοπτικά λοιπόν έχουμε:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \Big\{ m[i,k] + m[k,j] + p_{i-1} \cdot p_k \cdot p_j \Big\} & i < j \end{cases} \tag{8.10}$$

Βήμα 3: Υπολογισμός του ελαχίστου κόστους

Πριν προχωρήσουμε στην ανάπτυξη του αλγορίθμου δυναμικού προγραμματισμού, αξίζει να σημειώσουμε ότι στο πρόβλημά μας, εμφανίζεται ένας σχετικά μικρός αριθμός από διακριτά (διαφορετικά) υποπροβλήματα. Το πλήθος αυτό είναι της τάξης $\Theta(n^2)$. Εντούτοις μια αναδρομική υλοποίηση της εξίσωσης (8.10) θα υπολόγιζε πολλές φορές την τιμή για το ίδιο υποπρόβλημα. Αυτό είναι το δεύτερο χαρακτηριστικό των προβλημάτων που επιδέχονται αλγορίθμους δυναμικού προγραμματισμού και το οποίο ορίσαμε ως **ιδιότητα των επικαλυπτόμενων υποπροβλημάτων**.

Για την ανάπτυξη του αλγορίθμου δυναμικού προγραμματισμού, θα χρησιμοποιήσουμε έναν βοηθητικό πίνακα s, διάστασης $n \times n$ ο οποίος σε κάθε θέση του αποθηκεύει την τιμή του k για την οποία επιτεύχθηκε το ελάχιστο της εξίσωσης (8.10). Επίσης θεωρούμε ότι ο πίνακας A_i, έχει διάσταση $p_{i-1} \times p_i$. Ο αλγόριθμος δέχεται σαν είσοδο την ακολουθία διαστάσεων $p = \langle p_0, p_1, \ldots, p_n \rangle$ με μήκος $n+1$ και επιστρέφει τους πίνακες m και s.

Από την εξίσωση (8.10) διαπιστώνουμε ότι το κόστος $m[i,j]$ υπολογισμού $j-i+1$ πινάκων[2], εξαρτάται από το κόστος υπολογισμού λιγότερων από $j-i+1$ πινάκων. Πιο συγκεκριμένα για $k = i, i+1, \ldots, j-1$ ο πίνακας A_{ik} είναι γινόμενο $k-i+1 < j-i+1$ πινάκων και αντίστοιχα ο πίνακας $A_{k+1,j}$ είναι γινόμενο $j-k < j-i+1$ πινάκων. Δηλαδή ο αλγόριθμος δυναμικού προγραμματισμού συμπληρώνει τον πίνακα m επιλύοντας υποπροβλήματα μεγέθους διαρκώς αυξανόμενου.

MATRIX-CHAIN-ORDER(p)

```
1.  n = length[p] − 1
2.  for i = 1 to n do
3.      m[i, i] = 0
4.  end for
5.  for l = 2 to n do
6.      for i = 1 to n − l − 1 do
7.          j = i + l − 1
8.          m[i, j] = ∞
9.          for k = i to j − 1 do
10.             q = m[i, k] + m[k + 1, j] + p_{i−1} · p_k · p_j
11.             if q < m[i, j] then
12.                 m[i, j] = q
13.                 s[i, j] = k
14.             end if
15.         end for
16.     end for
17. end for
18. return m, s
```

---

[2] Το γινόμενο A_{ij} περιέχει $j-i+1$ πίνακες

Παρατηρήσεις Ο αλγόριθμος MATRIX-CHAIN-ORDER, υπολογίζει το ελάχιστο πλήθος βαθμωτών γινομένων που απαιτούνται για τον υπολογισμό του γινομένου των n πινάκων. Επίσης προκύπτει πολύ εύκολα ότι η πολυπλοκότητα του αλγορίθμου MATRIX-CHAIN-ORDER είναι της τάξης $O(n^3)$ (άσκηση).

Βήμα 4: Κατασκευή της βέλτιστης λύσης

Έχοντας συμπληρώσει τον πίνακα m και τον s μένει τώρα να δώσουμε έναν αλγόριθμο που να κατασκευάζει την βέλτιστη παρενθετοποίηση του γινομένου των πινάκων. Ο αλγόριθμος αυτός εκμεταλεύεται τον πίνακα s, ο οποίος αποθηκεύει στη θέση $s[i,j]$ την τιμή του k για την οποία επιτυγχάνεται το ελάχιστο κόστος $m[i,j]$. Έχουμε λοιπόν:

PRINT-OPTIMAL-PARENTS(s,i,j)
1. **if** $i=j$ **then**
2. τυπωσε ''A''$_i$
3. **else**
4. τύπωσε '('
5. PRINT-OPTIMAL-PARENTS$(s,i,s[i,j])$
6. PRINT-OPTIMAL-PARENTS$(s,s[i,j]+1,j)$
7. τύπωσε ')'
8. **end if**

Στην περίπτωση όπου $i=j$, ο αλγόρθμος απλά εκτυπώνει τον πίνακα A_i (αφού όπως είπαμε και στην αρχή το γινόμενο A_{ii} εκφυλίζεται στον πίνακα A_i). Στις γραμμές $4-5$ εκτυπώνονται αναδρομικά η αριστερή και δεξιά παρενθετοποίηση σε σχέση με τον πίνακα A_i.

8.3.3 Το πρόβλημα του σακιδίου

Ολοκληρώνουμε την μελέτη μας στον δυναμικό προγραμματισμό, με ένα πολύ γνωστό πρόβλημα της συνδυαστικής βελτίστοποίησης, το **πρόβλημα του σακιδίου** (**Knapsack**). Στο πρόβλημα αυτό μας δίνονται, ένα σύνολο $X=\{x_1,x_2,\ldots,x_n\}$ από n αντικείμενα που το καθένα έχει βάρος a_i και ένα κέρδος c_i, $i=1,2,\ldots,n$. Επίσης μας δίνεται ένας ακέραιος W. Ζητάμε να επιλέξουμε ένα υποσύνολο $Y\subseteq X$ τέτοιο ώστε $\sum_{x_i\in Y} c_i = \max$ και επιπλέον $\sum_{x_i\in Y} a_i \leq W$. Δηλαδή θέλουμε να επιλέξουμε τόσα αντικείμενα ώστε να μεγιστοποιήσουμε το κέρδος μας, χωρίς όμως να ξεπεράσουμε το συνολικό βαρος W. Για να υποδηλώσουμε ότι ένα αντικείμενο x_i έχει επιλεγεί, θέτουμε $x_i=1$ ενώ σε αντίθετη περίπτωση θέτουμε $x_i=0$.

Είναι γνωστό ότι το πρόβλημα αυτό είναι $\mathcal{NP}-\text{hard}$. Παρ΄ όλ΄ αυτά θα δώσουμε, συνοπτικά, έναν αλγόριθμο δυναμικού προγραμματισμού, που επιλύει το πρόβλημα αυτό σε **ψευδοπολυωνικό** χρόνο.

Βήμα 1: Βέλτιστα διασπώμενη δομή

Θεωρούμε την οικογένεια προβλημάτων:

$$P_k(y) = \Big\{ \max \sum_{j=1}^{k} c_j x_j, \quad \sum_{j=1}^{k} a_j x_j \leq y,\ x_j = 0 \text{ or } 1 \Big\} \tag{8.11}$$

όπου το y παίρνει τιμές από 0 έως το W και αντίστοιχα το k μεταβάλλεται από 1 έως n.

Η σχέση (8.11) υποδηλώνει μια οικογένεια πλήθους $n(W+1)$ υποπροβλημάτων, αυξανόμενου μεγέθους, που το κάθένα προκύπτει από το προηγούμενο, αυξάνοντας τον χώρο των δεδομένων εισόδου κατά ένα στοιχείο και αυξάνοντας διαδοχικά το συνολικό επιτρεπτό βάρος μέχρι το W. Για παράδειγμα, το πρόβλημα $P_2(y)$ ορίζει το πρόβλημα του σακιδίου όπου το σύνολο $Q = \{x_1, x_2\}$ για κάποιο $0 \leq y \leq W$. Το υποπρόβλημα $P_3(y)$ προκύπτει αν θέσουμε όπου $X = X \cup \{x_3\}$ για $0 \leq y \leq W$.

Βήμα 2: Αναδρομικός ορισμός της τιμής της βέλτιστης λύσης

Έστω $f_k(y)$ η τιμή της βέλτιστης λύσης του υποπροβλήματος $P_k(y)$. Τότε είναι αρκετά εύκολο να εξάγουμε από την σχέση (8.11), ότι

$$f_{k+1}(y) = \begin{cases} f_k(y) & \text{αν } a_{k+1} > y \\ \max\Big\{ f_k(y),\ f_k(y - a_{k+1}) + c_{k+1} \Big\}, & \text{διαφορετικά} \end{cases} \tag{8.12}$$

για $1 \leq k \leq n$ και $0 \leq y \leq W$.

Η αναδρομική εξίσωση (8.12), δηλώνει ότι η βέλτιστη λύση του υποπροβλήματος $P_{k+1}(y)$ είναι είτε η ίδια με αυτήν του $P_k(y)$ ή αυτή που προκύπτει αν μπορούμε να εισάγουμε το στοιχείο x_{k+1} αυξάνοντας το κέρδος κατά c_{k+1} και ελαττώνοντας το διαθέσιμο βάρος y κατά a_{k+1}. Η βέλτιστη λύση του προβλήματος $P_n(W)$ έχει την τιμή $f_n(W)$.

Για $k = 1$ έχουμε $f_1(y) = 0$ αν $a_1 > y$ και $f_1(y) = c_1$ αν $a_1 \leq y$, για όλα τα y, μικρότερα ή ίσα του W. Επίσης έχουμε $f_i(0) = 0, \quad \forall \quad i = 1, 2, \ldots, n$.

Βήμα 3: Υπολογισμός της βέλτιστης λύσης

Παρατηρούμε ότι στην αναδρομική εξίσωση (8.12), η τιμή του $f_{k+1}(y)$ εξαρτάται μόνο από τις τιμές $f_k(y)$ για $0 \leq y \leq W$. Επίσης για να υπολογίσουμε το $f_{k+1}(y)$ είναι απαραίτητο να γνωρίζουμε την τιμή του $f_k(y)$ για όλα τα y με $0 \leq y \leq W$. Τέλος για κάθε y με $0 \leq y \leq W$, ορίζουμε και το βοηθητικό διάνυσμα x_k^y ως εξής:

$$x_{k+1}^y = \begin{cases} 0 & f_{k+1}(y) = f_k(y) \\ 1 & \text{διαφορετικά} \end{cases} \tag{8.13}$$

Κάθε ένα από τα διανύσματα αυτά μας δίνει, τα αντικείμενα που συμμετέχουν στην λύση του υποπροβλήματος $P_k(y)$. Με βάση αυτές τις παρατηρήσεις είμαστε

```
DYNAMIC-KNAPSACK(A, C, W)
 1.  n = length(C)
 2.  for y = 0 to W do
 3.      if a_1 > y then
 4.          f_1(y) = 0
 5.      else
 6.          f_1(y) = c_1
 7.      end if
 8.  end for
 9.  for k = 1 to n − 1 do
10.      for y = 0 to W do
11.          if a_{k+1} > y then
12.              f_{k+1}(y) = f_k(y)
13.              x^y_{k+1} = 0
14.          else if f_k(y) > f_k(y − a_{k+1}) + c_{k+1}
15.              f_{k+1}(y) = f_k(y)
16.              x^y_{k+1} = 0
17.          else
18.              f_{k+1}(y) = f_k(y − a_{k+1}) + c_{k+1}
19.              x^y_{k+1} = 1
20.          end if
21.      end for
22.  end for
23.  return f, x
```

σε θέση τώρα να δώσουμε τον αλγόριθμο για την επίλυση του προβλήματος του σακιδίου.

Μπορούμε εύκολα να παρατηρήσουμε ότι η πολυπλοκότητα του αλγορίθμου είναι της τάξης $O(nW)$, η οποία εξαρτάται πολυωνυμικά από το n αλλά εξαρτάται και από το μέγιστο δυνατό βάρος W το οποίο μπορεί να είναι εκθετική συνάρτηση του n. Από αυτό προκύπτει ότι η πολυπλοκότητα του αλγορίθμου δεν είναι αυστηρά πολυωνυμική αλλά **ψευδοπολυωνυμική**.

Στον πίνακα 8.1 φαίνεται η εκτέλεση του αλγορίθμου όταν μας δίδονται 6 αντικείμενα με βάρη $a_1 = 9,\ a_2 = 8,\ a_3 = 6,\ a_4 = 5,\ a_5 = 4,\ a_6 = 1$ και αξίες $c_1 = 20,\ c_2 = 16,\ c_3 = 11,\ c_4 = 9,\ c_5 = 7,\ c_6 = 1$ και το βάρος του σακιδίου είναι $W = 12$.

Βήμα 4: Κατασκευή της βέλτιστης λύσης

Ολοκληρώνουμε την μελέτη μας πάνω στο πρόβλημα του σακιδίου με την κατασκευή της βέλτιστης λύσης. Όπως είπαμε και στην προηγούμενη παράγραφο, καθένα από τα διανύσματα x_k^y έχει τιμές 1 ή 0 ανάλογα με το αν το στοιχείο x_k συμμετέχει ή όχι, στη λύση του υποπροβλήματος $P_k(y)$.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	20(1)	20(1)	20(1)	20(1)
2	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	16(1)	20(0)	20(0)	20(0)	20(0)
3	0(0)	0(0)	0(0)	0(0)	0(0)	11(1)	11(1)	16(0)	20(0)	20(0)	20(0)	20(0)
4	0(0)	0(0)	0(0)	0(0)	9(1)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	20(0)
5	0(0)	0(0)	0(0)	7(1)	9(0)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	23(1)
6	1(1)	1(1)	1(1)	7(0)	9(0)	11(0)	12(1)	16(0)	20(0)	21(1)	21(1)	23(0)

Πίνακας 8.1: Παράδειγμα εκτέλεσης προβλήματος σακιδίου. Στο κελί (k, y) έχει σημειωθεί το βέλτιστο κέρδος από τον συνδυασμό των αντικειμένων $1, \ldots, k$ με βάρος το πολύ y ($f_k(y)$) και σε παρένθεση αν χρησιμοποιείται το αντικείμενο k (x_k^y).

Μια γρήγορη μέθοδος για την εξαγωγή της λύσης είναι αυτή του σχήματος 8.3.

CONSTRUCT-KNAPSACK(x)
1. τύπωσε: αξία βέλτιστης λύσης: $f_n(W)$
2. τύπωσε 'Αντικείμενα στην λύση: '
3. **while** $W > 0$ **do**
4. **if** $x_k^W = 1$ **then**
5. τύπωσε: k
6. $W = W - a_k$
7. **end if**
8. $k = k - 1$
9. **end while**

Σχήμα 8.3: Αλγόριθμος κατασκευής βέλτιστής λύσης από τον πίνακα x_k^y.

Ο χρόνος εκτέλεσης του αλγορίθμου είναι $O(n)$ και η εκτέλεσή του στο προηγούμενο παράδειγμα φαίνεται στον πίνακα 8.2.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	20(1)	20(1)	20(1)	20(1)
2	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	**16(1)**	20(0)	20(0)	20(0)	20(0)
3	0(0)	0(0)	0(0)	0(0)	0(0)	11(1)	11(1)	**16(0)**	20(0)	20(0)	20(0)	20(0)
4	0(0)	0(0)	0(0)	0(0)	9(1)	11(0)	11(0)	**16(0)**	20(0)	20(0)	20(0)	20(0)
5	0(0)	0(0)	0(0)	7(1)	9(0)	11(0)	11(0)	16(0)	20(0)	20(0)	20(0)	**23(1)**
6	1(1)	1(1)	1(1)	7(0)	9(0)	11(0)	12(1)	16(0)	20(0)	21(1)	21(1)	**23(0)**

Πίνακας 8.2: Παράδειγμα υπολογισμού της βέλτιστης λύσης. Με έντονα γράμματα έχουν σημειωθεί οι θέσεις του πίνακα που περνάει ο αλγόριθμος ώστε να τυπώσει την βέλτιστη λύση, που είναι η επιλογή των αντικειμένων $2, 5$ με συνολικό κέρδος 23.

Κεφάλαιο 9

Εξαντλητική Αναζήτηση

Στις προηγούμενες ενότητες συναντήσαμε προβλήματα βελτιστοποίησης που δεν κατέστη δυνατό να βρούμε έναν αποδοτικό (πολυωνυμικό) αλγόριθμο που βρίσκει την βέλτιστη λύση οποιουδήποτε στιγμιοτύπου του προβλήματος.

Μάλιστα, όπως θα δούμε σε επόμενο κεφάλαιο μια ολόκληρη οικογένεια προβλημάτων, τα $NP-$ δύσκολα προβλήματα, παρά την επίμονη προσπάθεια των επιστημόνων τις τελευταίες δεκαετίες, δεν έχει κατορθωθεί να βρεθεί αποδοτικός αλγόριθμος για κανένα από αυτά.

Στην περίπτωση αυτή, η μόνη αντιμετώπιση που είναι εφικτή για την εύρεση της βέλτιστης λύσης, είναι να απαριθμήσουμε όλες τις δυνατές λύσεις έτσι ώστε να βρούμε την καλύτερη. Οι αλγόριθμοι αυτοί είναι κατά κανόνα εκθετικοί και δεν μπορούμε να τους εμπιστευθούμε για μεγάλες διαστάσεις των προβλήματων.

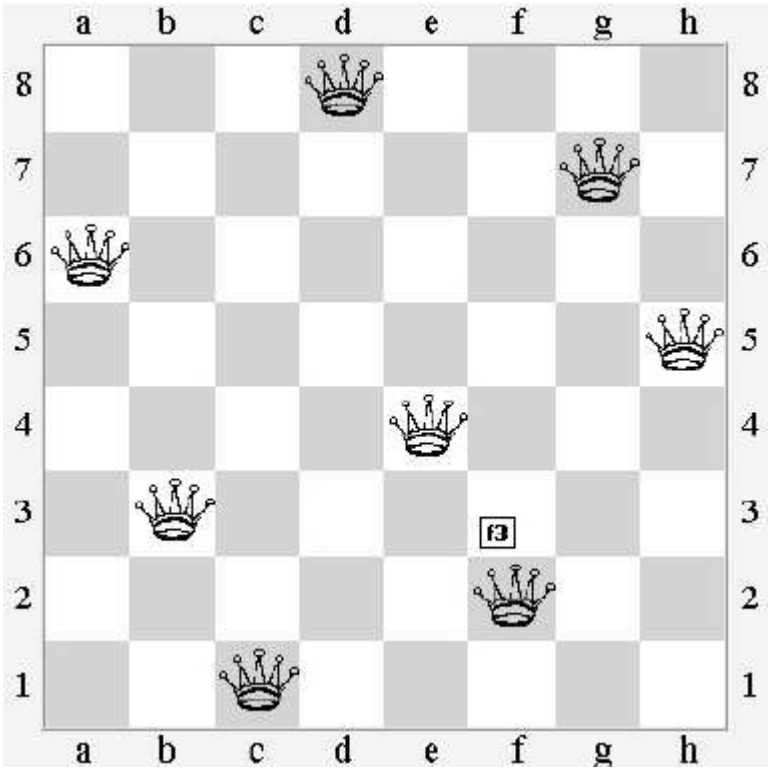
9.1 Το πρόβλημα των βασιλισσών

Το πρόβλημα των βασιλισσών είναι ένα κλασσικό συνδυαστικό πρόβλημα. Θέλουμε να τοποθετήσουμε στην σκακιέρα 8 βασίλισσες, προσέχοντας να μην έχουμε συγκρούσεις, δηλαδή δύο βασίλισσες να μην είναι στην ίδια γραμμή, ή στην ίδια στήλη, ή στην ίδια διαγώνιο. Μία λύση του προβλήματος φαίνεται στο σχήμα 9.1.

Θεωρώντας την γενίκευση του προβλήματος, που έχουμε N βασίλισσες μπορούμε να το μοντελοποιήσουμε ως εξής: Θεωρούμε ότι η βασίλισσα k ($1 \leq k \leq N$), μπαίνει στο τετράγωνο (i_k, j_k), άρα για κάθε συνδυασμό 2 βασιλισσών k_1, k_2 θα πρέπει να ισχύουν:

- $i_{k_1} \neq i_{k_2}$: Οι βασίλισσες δεν βρίσκονται στην ίδια γραμμή.
- $j_{k_1} \neq j_{k_2}$: Οι βασίλισσές δεν βρίσκονται στην ίδια στήλη.
- $|i_{k_1} - i_{k_2}| \neq |j_{k_1} - j_{k_2}|$: Οι βασίλισσες δεν βρίσκονται στην ίδια διαγώνιο.

Για να κατασκευάσουμε μία λύση του προβλήματος, θεωρούμε ότι η $k-$οστή βασίλισσα θα μπει στην $k-$η γραμμή, άρα ισχύει για κάθε k ($1 \leq k \leq N$) ότι



Σχήμα 9.1: Μία λύση του προβλήματος των 8 βασίλισσων. Καμία βασίλισσα δεν βρίσκεται στην ίδια γραμμή, στήλη ή διαγώνιο με κάποια άλλη.

$i_k = k$ και προσπαθούμε να βρούμε την στήλη της γραμμής στην οποία θα μπει η βασίλισσα. Η υλοποίηση αυτής της ιδέας φαίνεται στον αλγόριθμο του σχήματος 9.2.

Η συνάρτηση QUEENS είναι αναδρομική και όταν κληθεί με όρισμα 1 μας επιστρέφει την λύση στο πρόβλημα. Σε κάθε αναδρομική κλήση προσπαθεί να τοποθετήσει την βασίλισσα i σε μία στήλη της γραμμής i που δεν δημιουργεί σύγκρουση με τις ήδη τοποθετημένες βασίλισσες $1, \ldots, i-1$. Η στήλη στην οποία τοποθετείται αποθηκεύεται στην i-η θέση του πίνακα pos. Αυτό γίνεται ελέγχοντας διαδοχικά όλες τις θέσεις της γραμμής αν είναι συμβατές (δεν συγκρούονται) με τις βασίλισσες που έχουν τοποθετηθεί προηγούμενα, μέχρι να βρεθεί μία θέση που είναι συμβατή. Ο έλεγχος της μη σύγκρουσης γίνεται με την συνάρτηση COMPATIBLE που φαίνεται στο σχήμα 9.3.

```
QUEENS (i : βασίλισσα)
 1.  if i > N then /* N : αριθμός βασιλισσών */
 2.       τύπωσε την λύση
 3.  else
 4.       for j = 1 to N do
 5.            if COMPATIBLE (i, j) then
 6.            pos[i] = j
 7.            QUEENS (i + 1)
 8.            end if
 9.       end for
10.  end if
```

Σχήμα 9.2: Αναδρομικός αλγόριθμος εύρεσης λύσης του προβλήματος των βασιλισσών

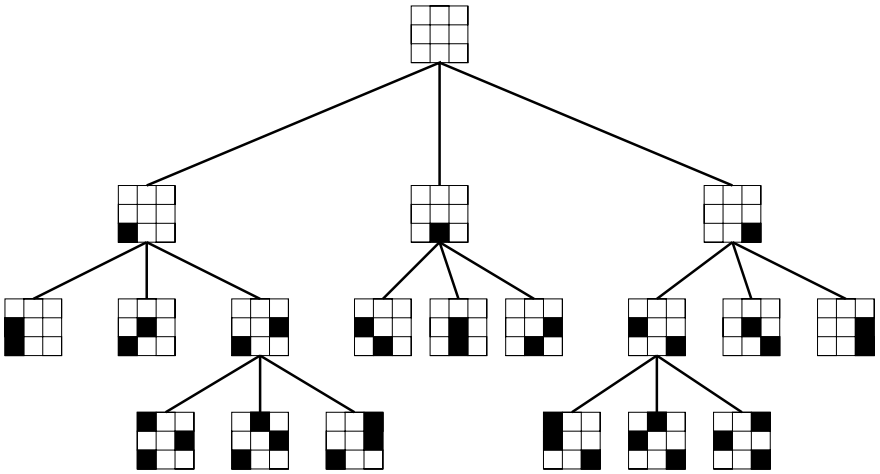
```
COMPATIBLE (i, j)
1.  c = TRUE, k = 1
2.  while c and k < i do
3.       c = not CONFLICT (i, j, k, pos[k])
4.       k = k + 1
5.  end while
6.  return c

CONFLICT (i_1, j_1, i_2, j_2)
1.  if i_1 = i_2 or j_1 = j_2 or abs(i_1 − i_2) = abs(j_1 − j_2) then
2.       return TRUE
3.  else return FALSE
```

Σχήμα 9.3: Συνάρτηση COMPATIBLE για τον έλεγχο της σύγκρουσης με τις ήδη τοποθετημένες βασίλισσες, όταν η $i-$η βασίλισσα τοποθετηθεί στο (i, j) τετράγωνο.

Ενδιαφέρον παρουσιάζει η εκτέλεση του παραπάνω αναδρομικού αλγορίθμου. Ας θεωρήσουμε το δέντρο των αναδρομικών κλήσεων του σχήματος 9.4 στο πρόβλημα των 3 βασιλισσών.

Η αναζήτηση της λύσης γίνεται σε μία πρώτα κατά βάθος λογική. Όταν δεν μπορεί να ευρεθεί θέση για την i-οστή βασίλισσα(ολοκληρώνονται οι αναδρομικές κλήσεις COMPATIBLE(i, j) με $1 \leq j \leq N$), ο έλεγχος επιστρέφεται στην $(i-1)$-οστή βασίλισσα και γίνεται τοποθέτηση της στην επόμενη θέση. Αυτή η διαδικασία ονομάζεται οπισθοδρόμηση (backtracking).

Η διαδικασία αυτή είναι εκθετική (άσκηση). Θα δούμε τώρα την μοντελοποίηση



Σχήμα 9.4: Το δέντρο των αναδρομικών κλήσεων στο πρόβλημα των 3 βασιλισσών. Η αναζήτηση γίνεται σε μία πρώτα κατά βάθος λογική και σταματά όταν η τοποθέτηση της βασίλισσας δημιουργεί σύγκρουση με τις ήδη τοποθετημένες. Στις 3 βασίλισσες το πρόβλημα δεν έχει λύση.

του προβλήματος με γράφο.

Θεωρούμε ότι V είναι το σύνολο των θέσεων της σκακιέρας ($|V| = N^2$) και μια ακμή $((i_1, j_1), (i_2, j_2))$ ανήκει στο E αν τα αντίστοιχα τετράγωνα συγκρούονται. Ο στόχος μας είναι να βρούμε ένα υποσύνολο των κόμβων X πληθικού αριθμού N που δεν συγκρούονται μεταξύ τους. Όντως αν θεωρήσουμε ότι υπάρχουν N τέτοιοι κόμβοι στον γράφο, αυτό σημαίνει ότι υπάρχουν N θέσεις της σκακιέρας που δεν συγκρούονται μεταξύ τους, άρα σε αυτές τις θέσεις μπορούμε να τοποθετήσουμε τις βασίλισσες.

Το πρόβλημα εύρεσης του μέγιστου πλήθους κόμβων ενός γράφου που δεν συνδέονται μεταξύ τους, είναι ένα πολύ γνωστό πρόβλημα της συνδυαστικής βελτιστοποίησης, γνωστό και ως πρόβλημα του Ανεξαρτήτου Υποσυνόλου. Το πρόβλημα αυτό στην γενική του περίπτωση είναι $NP-$δύσκολο, δηλαδή οι γρηγορότεροι αλγόριθμοι που το επιλύουν μέχρι σήμερα είναι ''κακοί'' (εκθετικοί) αλγόριθμοι. Αν όμως ο γράφος είναι χορδικός (chordal), ή διαστημάτων (interval), τότε μπορούμε να σχεδιάσουμε αποδοτικό (πολυωνυμικό) αλγόριθμο.

Διάφορες προσπάθειες έχουν γίνει για την βελτίωση της (εκθετικής) πολυπλοκότητας των αλγορίθμων που αντιμετωπίζουν $NP-$δύσκολα προβλήματα. Ενδεικτικά αναφέρουμε την μέθοδο του Διαχωρισμού και Εκτίμησης (Branch and Bound) με την οποία γίνεται αποκοπή κομματιών του χώρου αναζήτησης με χρήση πάνω και κάτω φραγμάτων. Ωστόσο καμία μέθοδος δεν κατορθώνει να ρίξει τον

χρόνο από εκθετικό σε πολυωνυμικό.

Κεφάλαιο 10

NP-πληρότητα

10.1 Εισαγωγή

Η θεωρία πολυπλοκότητας ασχολείται με την ιεράρχηση των υπολογιστικών προβλημάτων με βάση τη δυσκολία επίλυσής τους. Σε πολύ υψηλό επίπεδο μας ενδιαφέρει να μπορούμε να ξεχωρίσουμε τα «δύσκολα» από τα «εύκολα» προβλήματα. Φυσικά υπάρχουν τουλάχιστον δύο έννοιες που θα πρέπει να οριστούν στο αμέσως υποκείμενο επίπεδο, και αυτές είναι το «πρόβλημα» και η «δυσκολία». Το πρόβλημα στη θεωρητική πληροφορική ορίζεται ανεξάρτητα από τα δεδομένα εισόδου: θέλουμε να έχουμε μια γενικευμένη περιγραφή του προβλήματος (μέσω παραμέτρων), ώστε να μπορούμε να αναπτύξουμε γενικές διαδικασίες επίλυσης (αλγόριθμους) που θα δουλεύουν για οποιαδήποτε είσοδο (τιμές των παραμέτρων). Όταν έχουμε τιμές για τις παράμετρους ενός προβλήματος (δεδομένα εισόδου) αναφερόμαστε στο σύνολο της περιγραφής και των δεδομένων εισόδου σαν «στιγμιότυπο» του προβλήματος. Για παράδειγμα, ας θεωρήσουμε το πρόβλημα του σακιδίου (Knapsack):

Δεδομένων n αντικειμένων με βάρη w_j, $j = 1 \ldots n$ και αξίες p_j, $j = 1 \ldots n$ ζητάμε ένα υποσύνολο αντικειμένων συνολικού βάρους το πολύ W που έχουν τη μέγιστη συνολική αξία.

Ένα στιγμιότυπο του προβλήματος είναι το ακόλουθο:

Δεδομένων 5 αντικειμένων με βάρη $1, 4, 5, 2, 3$ και αξίες $3, 4, 7, 2, 5$ αντίστοιχα, ζητάμε ένα υποσύνολο αντικειμένων συνολικού βάρους το πολύ 7 που έχουν τη μέγιστη συνολική αξία.

Η δυσκολία επίλυσης ενός προβλήματος ορίζεται σε σχέση με τους υπολογιστικούς πόρους που καταναλώνονται από την καλύτερη γνωστή μέθοδο επίλυσης. Οι υπολογιστικοί πόροι κατά κύριο λόγο συνίστανται από το μέγεθος απαιτούμενης μνήμης, καθώς και από τον απαιτούμενο ασυμπτωτικό χρόνο επίλυσης. Τόσο η κατανάλωση μνήμης, όσο και ο χρόνος επίλυσης μετρώνται ασυμπτωτικά σαν συναρτήσεις του μεγέθους των δεδομένων εισόδου. Το μέγεθος των δεδομένων

εισόδου διαφέρει σε κάθε στιγμιότυπο ενός προβλήματος. Για το λόγο αυτό μας ενδιαφέρει η οριακή (ασυμπτωτική) συμπεριφορά της κατανάλωσης μνήμης και του χρόνου επίλυσης. Πιο απλά, μας ενδιαφέρει η συμπεριφορά ενός αλγόριθμου σε ότι αφορά την επίλυση των «μεγάλων» στιγμιοτύπων ενός προβλήματος.

Ανάλογα με τη φύση του προβλήματος και το είδος του υπολογισμού, μπορούν να οριστούν διαφορετικά μέτρα δυσκολίας. Για παράδειγμα ένα πρόβλημα που αφορά στην επεξεργασία δεδομένων που είναι αποθηκευμένα σε εξωτερικό δίσκο, επιλύεται χοντρικά σε τόσο χρόνο όσες είναι οι απαιτούμενες πράξεις εισόδου/εξόδου (I/O) από και προς το δίσκο. Αυτό συμβαίνει διότι το I/O είναι γενικά περισσότερο χρονοβόρο από οποιαδήποτε επεξεργασία και προσπέλαση μνήμης. Στην περίπτωση αυτή το μέτρο δυσκολίας του προβλήματος ορίζεται από την απαιτούμενη μνήμη καθώς και από το απαιτούμενο ασυμπτωτικό πλήθος I/O που λαμβάνουν χώρα κατά τον υπολογισμό.

Στο κεφάλαιο αυτό μελετάμε τη δυσκολία των προβλημάτων από την άποψη του χρόνου εκτέλεσης του ταχύτερου γνωστού αλγόριθμου. Σαν «δύσκολο» θα χαρακτηρίζεται ένα πρόβλημα για το οποίο ο καλύτερος γνωστός αλγόριθμος έχει χρόνο εκτέλεσης εκθετικό ως προς το μέγεθος της εισόδου (π.χ. $O(2^n)$). Σαν «εύκολο» χαρακτηρίζουμε ένα πρόβλημα το οποίο έχει αλγόριθμο πολυωνυμικού χρόνου. Έτσι το πρόβλημα του πλανώδιου πωλητή είναι δύσκολο, ενώ το πρόβλημα του μονοπατιού ελάχιστου κόστους είναι εύκολο ($O(m+n\log n)$ είναι η πολυπλοκότητα της καλύτερης γνωστής υλοποίησης του αλγορίθμου του Dijkstra, όπου m είναι το πλήθος των ακμών του γράφου και n το πλήθος των κορυφών).

Αξίζει να αναφέρουμε ότι αυτή η διάκριση μεταξύ εύκολων και δύσκολων προβλημάτων προτάθηκε για πρώτη φορά από τον Jack Edmonds το 1965, και παγιώθηκε τόσο λόγω της πρακτικής της αξίας (οι εκθετικού χρόνου αλγόριθμοι είναι πολύ αργοί σε οποιοδήποτε υπολογιστή, για αρκετά μεγάλο στιγμιότυπο προβλήματος) όσο και λόγω της θεωρητικής ανάπτυξης που προσέφερε στην πληροφορική σε συνδυασμό με την δουλειά των θεμελιωτών της θεωρίας της NP-Πληρότητας, Richard Karp και Stephen Cook.

10.2 Οι κλάσεις πολυπλοκότητας P και NP

Η θεωρία πολυπλοκότητας ταξινομεί τα προβλήματα σε κλάσεις (σύνολα) ισοδυναμίας που ορίζουν ότι τα προβλήματα στην ίδια κλάση έχουν την ίδια δυσκολία. Ιδιαίτερου ενδιαφέροντος στο πλαίσιο αυτό είναι οι κλάσεις P (Deterministic Polynomial Time) και NP (Non deterministic Polynomial Time). Γενικά θα μπορούσαμε να πούμε ότι η κλάση NP περιλαμβάνει τα περισσότερα από τα ενδιαφέροντα προβλήματα. Η κλάση P περιλαμβάνει τα εύκολα προβλήματα της NP. Θα προχωρήσουμε σε πιο τυπικούς ορισμούς οι οποίοι επαλυθεύουν τη διαίσθηση ότι $P \subseteq NP$. Θα πρέπει να τονίσουμε ότι οι κλάσεις P και NP ορίζονται ως προς **προβλήματα απόφασης**, δηλαδή προβλήματα στα οποία καλούμαστε να απαντήσουμε μια συγκεκριμένη ερώτηση με ΝΑΙ ή ΟΧΙ. Σαν παράδειγμα τέτοιου προβλήματος αναφέρουμε την ερώτηση «Έχει ένας δεδομένος γράφος $G(V,E)$ Hamilton κύκλο;» (δηλαδή κύκλο που διέρχεται μία φορά από κάθε κορυφή του γράφου). Στη συνέχεια δίνουμε τους ορισμούς για τις κλάσεις P και NP και

επεξηγούμε:

Ορισμός 25. Η κλάση P περιλαμβάνει όλα εκείνα τα προβλήματα απόφασης, τα οποία επιλύονται από ένα ντετερμινιστικό αυτόματο σε πολυωνυμικό χρόνο.

Ορισμός 26. Η κλάση NP περιλαμβάνει όλα τα προβλήματα απόφασης που επιλύονται από ένα μη ντετερμινιστικό αυτόματο σε πολυωνυμικό χρόνο.

Ένας ισοδύναμος ορισμός της κλάσης NP έχει ως εξής:

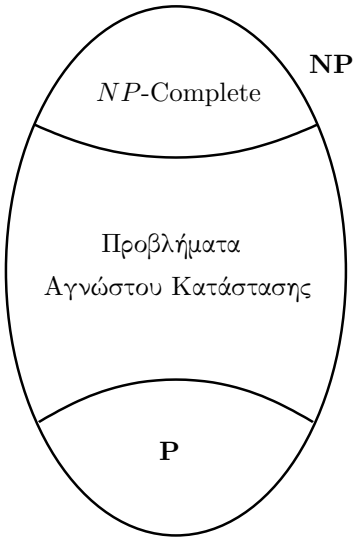
Ορισμός 27. Η κλάση NP περιλαμβάνει όλα τα προβλήματα απόφασης για τα οποία αν μας δοθεί ένα πιστοποιητικό της απάντησης ΝΑΙ, μπορούμε να επαληθεύσουμε σε πολυωνυμικό χρόνο ότι είναι σωστή.

Δεδομένου του όγκου της θεωρίας των αυτομάτων δε θα επεκταθούμε εδώ, αλλά θα εξηγήσουμε διαισθητικά την έννοια των ορισμών. Για τους σκοπούς της συζήτησης μπορούμε να σκεφτόμαστε ένα αυτόματο σαν ένα αλγόριθμο, ή έναν σύγχρονο υπολογιστή. Τα ντετερμινιστικά αυτόματα αντιστοιχούν στις μηχανές και στους αλγορίθμους που μπορούμε να περιγράψουμε και να υλοποιήσουμε κατά τους γνωστούς μας τρόπους. Έχουν την ιδιαιτερότητα ότι *κάθε βήμα υπολογισμού καθορίζεται (αποφασίζεται) πλήρως από το προηγούμενο:* διαισθητικά αυτός είναι και ο μόνος τρόπος με τον οποίο γνωρίζουμε πως να υπολογίζουμε.

Τα μη ντετερμινιστικά αυτόματα έχουν τη δυνατότητα να επιλέγουν το επόμενο βήμα τους μέσα από ένα σύνολο δυνατών υπολογιστικών βημάτων, και ανεξάρτητα από το πιο ήταν το προηγούμενο βήμα. Αυτό σημαίνει ότι δύο διαφορετικές εκτελέσεις ενός μη ντετερμινιστικού αλγορίθμου δεν αντιστοιχούν αναγκαστικά στην ίδια ακολουθία υπολογιστικών βημάτων (ακόμα και αν το πρώτο τους βήμα είναι ίδιο). Θεωρούμε ότι δεν υπάρχει καμία σαφής αιτία η οποία δικαιολογεί μια συγκεκριμένη διαδοχή βημάτων ενός μη ντετερμινιστικού αλγόριθμου: ο αλγόριθμος «μαντεύει» το σωστό επόμενο βήμα, αλλά δεν το αποφασίζει. Προφανώς τα μη ντετερμινιστικά αυτόματα είναι ένα θεωρητικό κατασκεύασμα, που γενικεύει τα ντετερμινιστικά, υπό την έννοια ότι μια διαδοχή βημάτων ενός ντετερμινιστικού αυτομάτου μπορεί να ανήκει και σε ένα μη ντετερμινιστικό, αλλά το αντίθετο δεν ισχύει.

Με ένα παράδειγμα επεξηγούμε τον εναλλακτικό ορισμό 27 της κλάσης NP: θεωρούμε το πρόβλημα απόφασης του αν ένας δεδομένος γράφος $G(V, E)$ έχει Hamilton κύκλο ή όχι. Σαν πιστοποιητικό αληθούς απάντησης σε αυτή την ερώτηση μας δίνεται μια διαδοχή κόμβων i_j, $j = 0, 1, \ldots, |V| - 1$ του γράφου. Τότε μπορούμε σε $O(|V|)$ χρόνο να ελέγξουμε κατά πόσο η διαδοχή που δώθηκε είναι κύκλος, απλώς ελέγχοντας αν $(i_j, i_{(j+1) mod |V|}) \in E$ για $j = 0, 1 \ldots, |V| - 1$. Προφανώς ένα πιστοποιητικό μπορεί να βρεθεί σε πολυωνυμικό χρόνο για όλα τα προβλήματα της P αφού αυτά επιλύονται σε πολυωνυμικό χρόνο.

Από τα προηγούμενα προκύπτει και τυπικά ότι $P \subseteq NP$. Για κάθε κλάση πολυπλοκότητας υπάρχουν κάποια προβλήματα, τα οποία είναι *πλήρη* για την κλάση αυτή, δηλαδή την περιγράφουν πλήρως υπό την ακόλουθη έννοια: η αποδοτική επίλυση ενός πλήρους προβλήματος για μια κλάση πολυπλοκότητας θα επιφέρει την επίλυση όλων των ισοδύναμων προβλημάτων που ανήκουν στην κλάση αυτή. Εδώ μας ενδιαφέρουν τα NP-πλήρη προβλήματα. Τα NP-πλήρη προβλήματα είναι



Σχήμα 10.1: Η κλάση NP: περιλαμβάνει εύκολα προβλήματα ($P \subseteq NP$), προβλήματα πλήρη για την κλάση (NP-complete), καθώς και προβλήματα για τα οποία δε γνωρίζουμε πολυωνυμικό αλγόριθμο, αλλά δεν έχουμε αποδείξει ότι είναι πλήρη για την NP.

«δύσκολα» σύμφωνα με όσα γνωρίζουμε. Είναι προβλήματα για τα οποία γνωρίζουμε ότι ανήκουν στην NP, αλλά δεν γνωρίζουμε αν ανήκουν στην P: ουσιαστικά ο καλύτερος αλγόριθμος που έχουμε για τα προβλήματα αυτά είναι εκθετικού χρόνου, και επιπλέον, η αποδοτική επίλυση ενός από αυτά θα επιφέρει την αποδοτική επίλυση όλων.

Στις επόμενες δύο ενότητες ορίζουμε πιο τυπικά τα προβλήματα απόφασης και τα προβλήματα βελτιστοποίησης, σχολιάζουμε το πρώτο πρόβλημα που αποδείχθηκε NP-πλήρες, και εξηγούμε πώς τα δεύτερα μπορούν να αναχθούν στα πρώτα, και επομένως να κατηγοριοποιηθούν στις κλάσεις P και NP.

10.3 Προβλήματα Απόφασης και $NP-$πληρότητα

Ένα πρόβλημα *απόφασης* ορίζεται από ένα πεπερασμένο σύνολο S, μέσα στο οποίο αναζητείται ένα στοιχείο του που ικανοποιεί μια ιδιότητα P. Το ερώτημα στο οποίο καλούμαστε να απαντήσουμε είναι αν υπάρχει ένα τέτοιο στοιχείο $s \in S$. Ο πιο προφανής τρόπος για να απαντηθεί το ερώτημα είναι η εξαντλητική εξέταση του S, και η εύρεση ή όχι του στοιχείου $s^\star \in S$ ώστε να ισχύει η $P(s^\star)$. Δίνουμε τον ορισμό της NP-πληρότητας:

Ορισμός 28. (**NP-πληρότητα**) Ένα πρόβλημα απόφασης X είναι NP-πλήρες αν:

1. $X \in NP$.
2. Η επίλυση κάθε άλλου προβλήματος απόφασης $\Pi \in NP$ ανάγεται σε πολυωνυμικό χρόνο στην επίλυση του X. Αυτό γράφεται σαν: $\Pi \leq_P X$ για κάθε $\Pi \in NP$.

Επεξηγούμε το δεύτερο σκέλος του ορισμού. Η έννοια της *αναγωγής* έχει μια οικεία σημασία που συνοψίζεται στο ότι αν μπορούσαμε να επιλύσουμε το X σε πολυωνυμικό χρόνο, θα μπορούσαμε να επιλύσουμε και το Π σε πολυωνυμικό χρόνο, με την προϋπόθεση ότι η αναγωγή είναι επίσης πολυωνυμικού χρόνου. Η αναγωγή ενός προβλήματος Π σε ένα πρόβλημα X σημαίνει ότι κάθε στιγμιότυπο του Π μπορεί να διατυπωθεί σαν στιγμιότυπο του X με συγκεκριμένους μετασχηματισμούς (αντιστοιχίες). Θα πρέπει όμως οι αντιστοιχίες αυτές να μπορούν να υπολογιστούν «εύκολα», σε χρόνο πολυωνυμικό. Τότε το X είναι τόσο δύσκολο όσο και το Π με μια αθροιστική πολυωνυμική χρονική επιβάρυνση (λόγω υπολογισμού της αναγωγής), η οποία όμως δε χειροτερεύει (ασυμπτωτικά) τη διαδικασία επίλυσης του Π. Θα εξηγήσουμε πιο αναλυτικά τη σημασία των αναγωγών σε επόμενη ενότητα.

10.3.1 Το πρόβλημα της ικανοποιησιμότητας

Ίσως το πιο διάσημο πρόβλημα απόφασης στην θεωρία πολυπλοκότητας είναι το πρόβλημα της *ικανοποιησιμότητας* (Satisfiability Problem - SAT) ένος τύπου λογικής πρώτης τάξης σε *κανονική συζευκτική μορφή* (Conjunctive Normal Form - CNF). Ένας τύπος σε CNF ορίζεται πάνω από ένα σύνολο λογικών μεταβλητών x_i, $i = 1 \ldots n$, με $x_i \in \{true, false\}$. Αποτελείται από τη σύζευξη m προτάσεων (clauses) c_j, και κάθε πρόταση αποτελείται από τη διάζευξη *λεκτικών* (literals). Ένα λεκτικό, τέλος, είναι μια λογική μεταβλητή ή η άρνησή της. Ακολουθεί ένα παράδειγμα CNF λογικού τύπου:

$$F = (x_1 \vee \bar{x}_2 \vee x_5) \wedge (x_2 \vee \bar{x}_4 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_5)$$

Το SAT για τον τύπο F ορίζεται με $S = \{true, false\}^5$, δηλαδή τα στοιχεία του S είναι όλα τα διανύσματα ανάθεσης τιμών στο $\{true, false\}$ των λογικών μεταβλητών. Μας ενδιαφέρει να αποφασίσουμε αν ο τύπος είναι ικανοποιήσιμος ή όχι (ουσιαστικά αν ο τύπος αποτελεί ένα «θεώρημα» για το σύμπαν $\{true, false\}^5$). Η ιδιότητα P για κάθε ανάθεση $s \in \{true, false\}^5 = S$ ορίζεται σαν η αλήθεια της πρότασης $F(s) = true$. Αν βρούμε $s^\star \in S$ τέτοιο ώστε $F(s^\star) = true$, τότε αποαφσίζουμε ότι η F είναι ικανοποιήσιμη. Αν $F(s) = false$ για κάθε $s \in S$ τότε αποφασίζουμε ότι η F είναι μη ικανοποιήσιμη.

Το SAT έιναι πρώτο πρόβλημα που αποδείχθηκε πλήρες για την κλάση NP. Αυτό σημαίνει ότι η εύρεση ενός αλγόριθμου πολυωνυμικού χρόνου για το SAT θα σημάνει την επίλυση σε πολυωνυμικό χρόνο όλων των προβλημάτων της κλάσης NP, και τότε θα γνωρίζουμε ότι $P = NP$. Αυτή τη στιγμή δε γνωρίζουμε αν υπάρχει πολυωνυμικού χρόνου αλγόριθμος για το SAT, πράγμα που μεταφράζεται στο πιο διάσημο ανοιχτό πρόβλημα της θεωρητικής πληροφορικής:

$$P = NP?$$

10.4 Προβλήματα Συνδυαστικής Βελτιστοποίησης

Ένα *στιγμιότυπο* προβλήματος βελτιστοποίησης ορίζεται από ένα πεπερασμένο χώρο *εφικτών* λύσεων S και από μια συνάρτηση *αποτίμησης* των λύσεων $f : S \to \Re$. Το πρόβλημα συμβολίζεται γενικευμένα με $\langle S, f \rangle$. Μια εφικτή λύση $s \in S$ του προβλήματος ικανοποιεί τους *περιορισμούς* του προβλήματος. Μερικά παραδείγματα ακολουθούν:

- Στο πρόβλημα του *Δέντρου Επικάλυψης Ελάχιστου Κόστους* ενός γράφου $G(V, E, W)$ με κόστη στις ακμές, το σύνολο S περιέχει όλα τα υποσύνολα ακμών του γράφου που επάγουν δέντρο στο σύνολο των κόμβων: $S = \{E' \subseteq E | G[E'] \text{ είναι δέντρο}\}$.

- Στο πρόβλημα του σακιδίου (Knapsack) το σύνολο S αποτελείται από τα υποσύνολα αντικειμένων που «χωράνε» μέσα στο σακίδιο.

- Στο πρόβλημα του πλανώδιου πωλητή (Travelling Salesman Problem - TSP) το σύνολο S ορίζεται από όλους τους δυνατούς κύκλους του γράφου, οι οποίοι περνούν μία φορά από όλους τους κόμβους του γράφου. Στον πλήρη γράφο αυτό ουσιαστικά συνεπάγεται όλες τις δυνατές μεταθέσεις των κόμβων.

Η συνάρτηση f ονομάζεται *αντικειμενική συνάρτηση* (objective function), και προσφέρει μια αποτίμηση της κάθε εφικτής λύσης $s \in S$. Στα προβλήματα που αναφέρθηκαν η f δίνει αντίστοιχα το κόστος του δέντρου επικάλυψης, την αξία του υποσυνόλου αντικειμένων, και το κόστος του κάθε κύκλου. Τυπικά μας ενδιαφέρει να βελτιστοποιήσουμε την f πάνω από το σύνολο των εφικτών λύσεων (που είναι το πεδίο ορισμού της). Ζητάμε δηλαδή εφικτή λύση $s^\star \in S$ τέτοια ώστε:

$$f(s^\star) = opt_{s \in S} f(s)$$

Όπου το *opt* αντικαθίσταται από min ή max ανάλογα με τον ορισμό του προβλήματος.

10.5 Προβλήματα $NP-$δύσκολα

Οι κλάσεις P και NP ορίστηκαν σε σχέση με προβλήματα απόφασης (ΠΑ). Αυτό μας δίνει τη δυνατότητα να κατηγοριοποιήσουμε και προβλήματα συνδυαστικής βελτιστοποίησης (ΠΣΒ) ως προς τη δυσκολία τους, καθώς σε κάθε ΠΣΒ μπορεί να αντιστοιχηθεί ένα ΠΑ. Αντίστροφα, συμβαίνει ένας αλγόριθμος που επιλύει ένα ΠΑ να μπορεί να χρησιμοποιηθεί για την επίλυση του αντίστοιχου ΠΣΒ.

Με την εισαγωγή ενός αριθμού k στα δεδομένα ενός ΠΣΒ εύκολα το μετατρέπουμε σε ΠΑ ζητώντας $s \in S$ τέτοιο ώστε $f(s) \leq k$. Τότε μπορούμε να μελετήσουμε τη δυσκολία του ΠΣΒ μελετώντας τη δυσκολία του αντίστοιχου ΠΑ:

ένας αλγόριθμος για την επίλυση του ΠΣΒ θα σήμαινε άμεσα την επίλυση του αντίστοιχου ΠΑ. Επομένως το ΠΣΒ είναι τουλάχιστον τόσο δύσκολο όσο το αντίστοιχό του ΠΑ. Αντίστροφα, αν έχουμε έναν αλγόριθμο για την επίλυση του ΠΑ, τότε μπορούμε να τον χρησιμοποιήσουμε για την επίλυση του αντίστοιχου ΠΣΒ, χρησιμοποιώντας διχοτομική αναζήτηση πάνω στο πεδίο τιμών του k.

Αν το ΠΑ είναι NP-πλήρες, τότε το ΠΣΒ είναι **NP-δύσκολο (NP-hard)**. Εδώ εισάγουμε τον ορισμό της NP-δυσκολίας:

Ορισμός 29. (**NP-hardness**) Ένα πρόβλημα X είναι NP-δύσκολο αν είναι τουλάχιστον τόσο δύσκολο όσο ένα NP-πλήρες πρόβλημα.

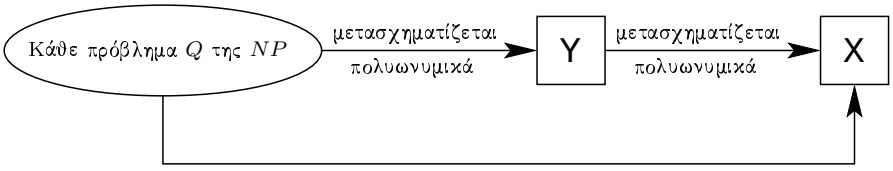
Πρακτικά ο ορισμός αυτός διαφέρει από την NP-πληρότητα στο ότι *αφορά προβλήματα τα οποία δεν ανήκουν στην κλάση* NP. Ας σημειωθεί ότι τα προβλήματα βελτιστοποίησης δεν ανήκουν στην NP διότι *δεν είναι προβλήματα απόφασης*. Αν το αντίστοιχο ΠΑ ενός ΠΣΒ είναι NP-πλήρες, τότε όλα τα προβλήματα της κλάσης NP ανάγονται πολυωνυμικά στο ΠΑ αυτό εξόρισμού. Όμως αν επιλύαμε το ΠΣΒ θα επιλύαμε εύκολα και το ΠΑ και, επομένως, όλα τα προβλήματα της NP. Αυτό σημαίνει ότι όλα τα προβλήματα της NP ανάγονται πολυωνυμικά στο συγκεκριμένο ΠΣΒ. Επομένως μπορούμε να πούμε ότι:

Ορισμός 30. Ένα πρόβλημα X είναι NP-δύσκολο αν για κάθε $\Pi \in NP$ ισχύει $\Pi \leq_P X$ και $X \notin NP$. Αν επιπλέον $X \in NP$ τότε το X είναι NP-πλήρες.

10.6 Αναγωγές

Η έννοια της NP-πληρότητας κατέκτησε κεντρική θέση στη θεωρητική πληροφορική για δύο λόγους:

- Τα NP-πλήρη και NP-δύσκολα προβλήματα συνεχίζουν να αντιστέκονται σε οποιαδήποτε προσπάθεια επίλυσης σε πολυωνυμικό χρόνο.
- Για τα περισσότερα προβλήματα, για τα οποία δε γνωρίζουμε πολυωνυμικό αλγόριθμο, συμβαίνει να έχει αποδειχθεί η NP-πληρότητά τους.



Πολυωνυμικός μετασχηματισμός

Φυσικά το δεύτερο δεν ισχύει για όλα τα δυσεπίλυτα προβλήματα, καθώς για κάποια η κατάστασή τους είναι τελείως άγνωστη. Πάντως είναι καλή πρακτική να προσπαθούμε να αποδείξουμε την NP-πληρότητα ενός προβλήματος όταν έχουμε αποτύχει να βρούμε πολυωνυμικού χρόνου αλγόριθμο μετά από εκτεταμένη προσπάθεια (και το αντίστροφο).

Για να αποδείξουμε ότι ένα πρόβλημα $X \in NP$ είναι NP-πλήρες προσπαθούμε να το *ανάγουμε* σε πολυωνυμικό χρόνο σε ένα άλλο πρόβλημα Y, για το οποίο γνωρίζουμε ήδη ότι είναι NP-πλήρες. Σχεδιάζουμε δηλαδή έναν αλγοριθμικό μετασχηματισμό πολυωνυμικού χρόνου των στιγμιοτύπων του Y σε στιγμιότυπα του X[1]. Το νόημα της πρακτικής αυτής εικονίζεται στο παραπάνω σχήμα. Οι αναγωγές πολυωνυμικού χρόνου έχουν μεταβατική ιδιότητα: αν όλα τα προβλήματα της NP μετασχηματίζονται στο Y (πράγμα το οποίο ισχύει αν το Y είναι γνωστό ότι είναι NP-πλήρες) και το Y μετασχηματίζεται σε πολυωνυμικό χρόνο στο X, τότε όλα τα προβλήματα της NP ανάγονται στο X και, επομένως, το X είναι NP-πλήρες.

10.7 Αγνώστου Κατάστασης Προβλήματα

Υπάρχουν προβλήματα της κλάσης NP για τα οποία δε γνωρίζουμε πολυωνυμικού χρόνου αλγόριθμο που τα επιλύει, ενώ δεν έχουμε επίσης απόδειξη της NP-πληρότητάς τους. Ίσως το πιο διάσημο παράδειγμα από αυτά είναι ο *ισομορφισμός* δύο γράφων. Το πρόβλημα αυτό ορίζεται ως εξής: δεδομένων δύο γράφων $G(V(G), E(G))$ και $H(V(H), E(H))$ με $|V(G)| = |V(H)|$ και $|E(G)| = |E(H)|$ υπάρχει μία απεικόνιση $f : V(G) \to V(H)$ $1-1$ και επί, ώστε αν $(f(u), f(v)) \in E(H)$, τότε $(u, v) \in E(G)$;

Είναι εύκολο να δει κανείς ότι το πρόβλημα αυτό ανήκει στο NP: στην περίπτωση θετικής απάντησης στο ερώτημα αυτό με μια απεικόνιση f σαν πιστοποιητικό μπορούμε να ελέγξουμε την ορθότητα της απάντησης σε πολυωνυμικό $O(|V(G)|^2)$ χρόνο.

Ένα δεύτερο διάσημο πρόβλημα του οποίου η κατάσταση ήταν άγνωστη μέχρι το 2002 είναι η απόφαση του αν ένας δεδομένος αριθμός n είναι πρώτος η όχι. Η εργασία των Agrawal, Kayal, Saxena αποδεικνύει ότι το πρόβλημα αυτό ανήκει στο P, καθώς δίνεται ένας πολυωνυμικού χρόνου αλγόριθμος μετά από δεκαετίες αναζήτησης.

[1] Παρατηρήστε ότι ενώ η αναγωγή γίνεται από το άγνωστο πρόβλημα στο γνωστό, ο πολυωνυμικός μετασχηματισμός μετατρέπει τα στιγμιότυπα του γνωστού προβλήματος σε στιγμιότυπα του άγνωστου προβλήματος.

Βιβλιογραφία

[1] A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN, Data Structures and Algorithms, Addison-Wesley, 1983.

[2] TH.H. CORMEN, CH.E. LEISERSON, R.L. RIVEST, C. STEIN, Introduction to algorithms, 1st edition: MIT Press, 1990 and 2nd edition: MIT Press, 2001.

[3] M.R. GAREY, D.S. JOHNSON, Computers and Intractability: A guide to the theory of NP-completeness, W.H. Freeman and company, 1979.

[4] J. KLEINBERG, E. TARDOS, Algorithm Design, Addison-Wesley, 2005.

[5] H.R. LEWIS, C.H. PAPADIMITRIOU, Elements of the Theory of Computation, 2nd Edition, Prentice-Hall, 1998.

[6] C.H. PAPADIMITRIOU, Computational Complexity, Addison-Wesley, 1994.

[7] C.H. PAPADIMITRIOU, K. STEIGLITZ, Combinatorial Optimization, Prentice-Hall, 1982.

[8] R. SEDGEWICK, Algorithms, Second Edition, Addison-Wesley, 1988.

[9] S.S. SKIENNA, The Algorithm Design Manual, Springer-Verlag, 1998.