Στάθης Ζάχος Άρης Παγουρτζής

Θεμελιώσεις της Επιστήμης των Υπολογιστών

Σημειώσεις

Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα 2015

Θεμελιώσεις της Επιστήμης των Υπολογιστών

Στάθης Ζάχος Άρης Παγουρτζής

Πρόλογος

Σκοπός των σημειώσεων αυτών είναι η εισαγωγή στις θεμελιώδεις έννοιες της Επιστήμης των Υπολογιστών, με έμφαση σε αυτές που σχετίζονται με τη Θεωρητική Πληροφορική και τη Θεωρία Υπολογισμού.

Περιλαμβάνονται: εισαγωγή στην έννοια του υπολογισμού και του αλγορίθμου, Θεωρία Αυτομάτων, Τυπικών Γλωσσών και Γραμματικών, Σχεδίαση και Ανάλυση Αλγορίθμων, Αλγόριθμοι Γράφων, Λογική για την Επιστήμη των Υπολογιστών, Μοντέλα Υπολογισμού, Υπολογισιμότητα και Υπολογιστική Πολυπλοκότητα.

Η ύλη των σημειώσεων αυτών διδάσκεται στους φοιτητές των μαθημάτων "Εισαγωγή στην Επιστήμη Υπολογιστών" της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών ΕΜΠ και "Θεμελιώδη Θέματα Επιστήμης Υπολογιστών" της Σχολής Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών ΕΜΠ.

Προσοχή: οι σημειώσεις δεν υποκαθιστούν την παρακολούθηση του μαθήματος.

Θα θέλαμε να ευχαριστήσουμε τους παρακάτω σπουδαστές, οι περισσότεροι εκ των οποίων είναι πλέον διδάκτορες, που βοήθησαν στην προετοιμασία αυτών των σημειώσεων: Γιάννη Κασσιό, Κατερίνα Ποτίκα, Παναγιώτη Χείλαρη, Βαγγέλη Μπαμπά, Αντώνη Αχιλλέως, Γεωργία Καούρη, και Αντώνη Αντωνόπουλο.

Στάθης Ζάχος – Άρης Παγουρτζής, 2015

Περιεχόμενα

Πρόλογος **i**

Περιεχόμενα **iii**

1 Εισαγωγή **1**

1.1 Κλάδοι Επιστήμης των Υπολογιστών 2

1.2 Επανάληψη, επαγωγή, αναδρομή 3

1.2.1 Επανάληψη (Iteration) 3

1.2.2 Αναδρομή (Recursion) 3

1.2.3 Επαγωγή (Induction) 4

1.2.4 Μερική και ολική ορθότητα 5

1.3 Δομημένος προγραμματισμός και modularity 5

1.4 Παράλληλες, ταυτόχρονες, κατανεμημένες διεργασίες 7

1.4.1 Δίκτυα ταξινόμησης 7

1.5 Ταξινόμηση treesort με δένδρο δυαδικής αναζήτησης 8

1.6 Το θεώρημα τεσσάρων χρωμάτων (four color theorem) 9

2 Αλγόριθμοι **13**

2.1 Αλγόριθμοι και Πολυπλοκότητα 13

2.1.1 Τι είναι αλγόριθμος 13

2.1.2 Πολυπλοκότητα αλγορίθμων και προβλημάτων 14

2.1.3 Ντετερμινιστικοί και μη ντετερμινιστικοί αλγόριθμοι . . . 15

2.2 Μαθηματικοί Συμβολισμοί 17

2.3 Εύρεση μέγιστου κοινού διαιρέτη 19

2.3.1 Ένας απλός αλγόριθμος για το gcd 19

2.3.2 Αλγόριθμος με αφαιρέσεις για το gcd 20

2.3.3 Αλγόριθμος του Ευκλείδη 20

2.4 Ύψωση σε δύναμη με επαναλαμβανόμενο τετραγωνισμό (repeated squaring) . . . 21
2.5 Αριθμοί Fibonacci (Υπολογισμός) . . . 21
2.5.1 Αναδρομικός αλγόριθμος . . . 22
2.5.2 Επαναληπτικός αλγόριθμος . . . 22
2.5.3 Αλγόριθμος με πίνακα . . . 22
2.5.4 Σύγκριση αλγορίθμων . . . 22
2.6 Δυαδική αναζήτηση (Binary Search) . . . 23
2.7 Εύρεση του μεγαλύτερου και του μικρότερου στοιχείου . . . 24
2.8 Πολλαπλασιασμός ακεραίων (integer multiplication) . . . 25
2.9 Πολλαπλασιασμός πινάκων (matrix multiplication) . . . 26
2.10 Τρίγωνο Pascal . . . 27
2.11 Πρώτοι αριθμοί – Παραγοντοποίηση – Κρυπτοσύστημα RSA . . . 29

3 Αλγόριθμοι γράφων **37**
3.1 Γραφήματα . . . 37
3.1.1 Εισαγωγικές έννοιες – Ορισμός . . . 37
3.1.2 Υπογράφος . . . 39
3.1.3 Βαθμός κορυφής . . . 40
3.1.4 Δρόμος - Μονοπάτι - Κύκλος . . . 40
3.1.5 Παράσταση Γράφου . . . 41
3.1.6 Προσανατολισμένος Γράφος . . . 43
3.1.7 Συνεκτικός Γράφος . . . 44
3.2 Δέντρα . . . 45
3.2.1 Γενικά . . . 45
3.3 Ελάχιστο συνδετικό δέντρο (MST) . . . 47
3.4 Το πρόβλημα των συντομότερων μονοπατιών . . . 53
3.5 Διάσχιση γράφων . . . 54
3.5.1 Γενικά . . . 54
3.5.2 Αναζήτηση κατά πλάτος (Breadth First Search) . . . 55
3.5.3 Αναζήτηση κατά βάθος (Depth First Search) . . . 57
3.6 Μέγιστη ροή – Ελάχιστη τομή (Max Flow – Min Cut) . . . 58
3.7 Πολυπλοκότητα γραφοθεωρητικών προβλημάτων . . . 61

4 Πεπερασμένα αυτόματα και κανονικές παραστάσεις **65**
4.1 Εισαγωγή . . . 65
4.2 Ντετερμινιστικά πεπερασμένα αυτόματα . . . 66

4.3 Μη ντετερμινιστικά πεπερασμένα αυτόματα 68
4.4 Κανονικές παραστάσεις 73
4.5 Ελαχιστοποίηση DFA 76
4.6 Pumping Lemma για κανονικά σύνολα 78

5 Μη κανονικές γλώσσες και γραμματικές **85**
5.1 Εισαγωγή 85
5.2 Κανονικές Γραμματικές 85
5.3 Γραμματικές χωρίς συμφραζόμενα και αυτόματα στοίβας 87
5.3.1 Συντακτικά δένδρα 88
5.3.2 Απλοποίηση και κανονικές μορφές 91
5.3.3 Αυτόματα στοίβας 92
5.4 Γενικές γραμματικές 94
5.5 Γραμματικές με συμφραζόμενα 95

6 Λογική στην Επιστήμη των Υπολογιστών **99**
6.1 Προτασιακή Λογική 99
6.2 Κατηγορηματικός Λογισμός 100
6.3 Πρωτοβάθμια Λογική 101

7 Υπολογιστικά μοντέλα **105**
7.1 Μηχανές Turing 105
7.1.1 Μηχανές με εναδική (unary) αναπαράσταση αριθμών . . . 106
7.1.2 Μηχανές με δυαδική (binary) αναπαράσταση αριθμών . . 107

8 Υπολογισιμότητα και Υπολογιστική Πολυπλοκότητα **109**
8.1 Ιστορία - Εισαγωγή 109
8.2 Υπολογιστικά μοντέλα 112
8.3 Υπολογιστική Πολυπλοκότητα 114
8.4 Αναγωγές μεταξύ προβλημάτων 115
8.4.1 Αναγωγές πολυωνυμικού χρόνου 116
8.4.2 Αναγωγές: δύο τρόποι χρήσης 117

Κεφάλαιο 1

Εισαγωγή

Ο όρος *Επιστήμη των Υπολογιστών* (Computer Science) χρησιμοποιείται περισσότερο στην Αμερική. Στην Ευρώπη χρησιμοποιούμε τον όρο *Πληροφορική* (Informatics), ενώ στην Μεγάλη Βρετανία λένε συχνά *Επιστήμη των Υπολογισμών* (Computing Science).

Όπως είπε και ο διαπρεπής E. Dijkstra, η Επιστήμη των Υπολογιστών έχει την ίδια σχέση με τους υπολογιστές που έχει η Αστρονομία με τα τηλεσκόπια.

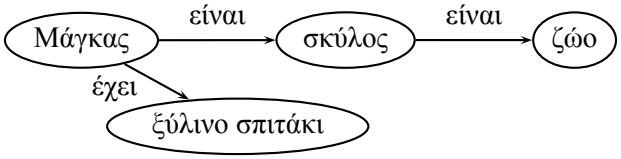
Σήμερα η Επιστήμη των Υπολογιστών άπτεται σχεδόν οποιασδήποτε ανθρώπινης δραστηριότητας. Η Πληροφορική είναι η συστηματική μελέτη αλγοριθμικών διαδικασιών που περιγράφουν και επεξεργάζονται πληροφορίες. Πιο συγκεκριμένα, η Επιστήμη των Υπολογισμών περιλαμβάνει: θεωρία, ανάλυση, σχεδίαση, αποδοτικότητα, υλοποίηση και εφαρμογή αλγοριθμικών διαδικασιών. Το βασικό ερώτημα με το οποίο ασχολείται η Επιστήμη των Υπολογιστών είναι τι μπορεί να *μηχανοποιηθεί* και μάλιστα με αποδοτικό τρόπο. Είναι κατά βάση *αφαιρετική* επιστήμη: Κατασκευάζουμε το σωστό μοντέλο για ένα πρόβλημα και επινοούμε τις κατάλληλες μηχανιστικές τεχνικές για την επίλυσή του.

Συχνά η εύρεση μίας καλής αφαίρεσης για ένα πρόβλημα μπορεί να είναι δύσκολη, λόγω των θεμελιωδών περιορισμών ως προς τις διαδικασίες (tasks) που μπορούν να επιτελέσουν οι υπολογιστές και την ταχύτητα με την οποία εκτελούν οι υπολογιστές αυτές τις διαδικασίες. Ένα δύσκολο τέτοιο πρόβλημα είναι η *αναπαράσταση γνώσης* (knowledge representation). Παρ' όλα αυτά έχουμε κάνει πρόοδο επινοώντας διάφορες αφαιρέσεις οι οποίες μας βοηθούν να κατασκευάσουμε προγράμματα που κάνουν ορισμένα είδη συλλογισμών. Μία τέτοια αφαίρεση είναι ο κατευθυνόμενος γράφος (directed graph· βλέπε σχήμα 1.1).

'Αλλη μία χρήσιμη αφαίρεση είναι η *τυπική λογική*, η οποία μας επιτρέπει να χειριζόμαστε *γεγονότα* (facts) μέσω της εφαρμογής *συμπερασματικών κανόνων* (rules of inference).

Για την επίλυση προβλημάτων χρησιμοποιούνται τα παρακάτω εργαλεία:

1. *μοντέλα δεδομένων (data models)*, που είναι οι αφαιρέσεις που περιγράφουν



Σχήμα 1.1: Γράφος για την αναπαράσταση γνώσης.

το πρόβλημα, για παράδειγμα, όπως αναφέραμε προηγουμένως, η λογική και οι γράφοι·

2. *δομές δεδομένων (data structures)*, που είναι οι δομές των γλωσσών προγραμματισμού που παριστάνουν τα μοντέλα δεδομένων. Για παράδειγμα στην Pascal, οι πίνακες, οι εγγραφές, οι δείκτες μας επιτρέπουν να κατασκευάσουμε δομές δεδομένων που παριστάνουν πιο πολύπλοκες αφαιρέσεις, όπως οι γράφοι·

3. *αλγόριθμοι*, που είναι οι τεχνικές με τις οποίες παίρνουμε λύσεις για τα προβλήματα μέσω της επεξεργασίας των αντίστοιχων δομών δεδομένων.

1.1 Κλάδοι Επιστήμης των Υπολογιστών

Ακολουθεί μία πρόχειρη απαρίθμηση διαφόρων κλάδων της Επιστήμης των Υπολογιστών. Κοινό χαρακτηριστικό όλων τους είναι το περιεχόμενό τους: το θεωρητικό κομμάτι, η δημιουργία μοντέλων και το σχεδιαστικό – κατασκευαστικό τους κομμάτι:

1. Υπολογισιμότητα και Πολυπλοκότητα – Μοντέλα Υπολογισμού
2. Θεωρία Αυτομάτων και Τυπικών Γλωσσών
3. Αλγόριθμοι και Δομές Δεδομένων
4. Γλώσσες Προγραμματισμού και Μεταγλωττιστές
5. Αρχιτεκτονική Υπολογιστών και Δικτύων (hardware)
6. Αριθμητικοί και Συμβολικοί Υπολογισμοί
7. Λειτουργικά - Παράλληλα - Κατανεμημένα Συστήματα
8. Μεθοδολογία - Τεχνολογία Λογισμικού (software)
9. Βάσεις Δεδομένων και Διαχείριση Πληροφοριών
10. Τεχνητή Νοημοσύνη και Ρομποτική
11. Επικοινωνία ανθρώπου - υπολογιστή. Πολυμέσα

12. Κρυπτογραφία και ασφάλεια
13. Δίκτυα Επικοινωνιών - Ευφυή Δίκτυα - Διαδίκτυο
14. Υπολογιστική βιολογία

1.2 Επανάληψη, επαγωγή, αναδρομή

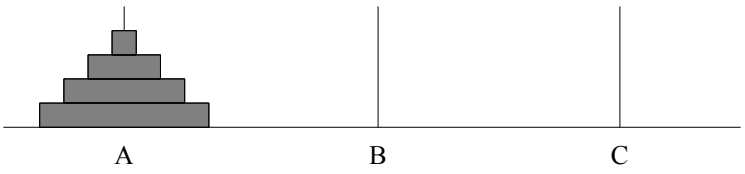
Θα συζητήσουμε διάφορες αρχές πάνω στις οποίες βασίζονται πολλές κατασκευές: π.χ. αλγεβρικές δομές όπως 'Αλγεβρες Boole και η Αναδρομική Μέθοδος.

1.2.1 Επανάληψη (Iteration)

Η ισχύς των υπολογιστών έγκειται στην δυνατότητά τους να εκτελούν με μεγάλη ταχύτητα *επαναληπτικά* την ίδια διαδικασία (πιθανώς παραμετροποιημένη). Ο απλούστερος τρόπος για να εκτελέσουμε μία διαδικασία επαναληπτικά είναι με μία δομή ελέγχου όπως η **for** της Pascal.

1.2.2 Αναδρομή (Recursion)

Αρκετά προβλήματα λύνονται εύκολα με την βοήθεια *αναδρομικών* διαδικασιών ή συναρτήσεων. Αναδρομικές είναι οι συναρτήσεις ή διαδικασίες που καλούν τον εαυτό τους μία ή περισσότερες φορές προκειμένου να λύσουν σχετικά υποπροβλήματα. Στον ορισμό της αναδρομικής διαδικασίας διαπιστώνουμε ότι το αρχικό πρόβλημα διασπάται σε μικρότερα προβλήματα του ίδιου τύπου με το αρχικό.



Σχήμα 1.2: Πύργοι του Ανόι ($n = 4$).

Ένα πρόβλημα για την λύση του οποίου είναι πιο εύκολο να κατασκευάσουμε έναν αναδρομικό από ότι έναν επαναληπτικό αλγόριθμο είναι οι "Πύργοι του Ανόι": Έχουμε τρεις πασάλους, έστω A, B, C, και n δίσκους, που είναι όλοι διαφορετικοί σε μέγεθος μεταξύ τους. Αρχικά όλοι οι δίσκοι, οι οποίοι έχουν μία τρύπα στην μέση, ούτως ώστε να περνάνε στους πασάλους, βρίσκονται περασμένοι στον πάσαλο A, έτσι που να μην υπάρχει μικρότερος δίσκος κάτω από κάποιον μεγαλύτερό του (βλέπε σχήμα 1.2). Σκοπός είναι να μετακινήσουμε όλους τους δίσκους από τον πάσαλο A στον πάσαλο B, έναν κάθε φορά έτσι ώστε ποτέ να μην βρεθεί (στον ίδιο πάσαλο) μικρότερος δίσκος κάτω από μεγαλύτερο, χρησιμοποιώντας τον (βοηθητικό) πάσαλο C.

Ένας αναδρομικός αλγόριθμος:

```
procedure move_anoi(n from X to Y using Z)
begin
   if n = 1 then move top disk from X to Y
   else begin
            move_anoi(n−1 from X to Z using Y);
            move top disk from X to Y;
            move_anoi(n−1 from Z to Y using X)
        end
end
```

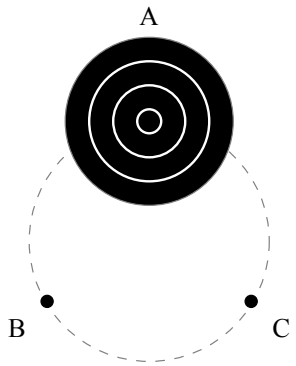
Για να λύσουμε το πρόβλημα του σχήματος 1.2 αρκεί να εκτελέσουμε την διαδικασία ως εξής: move_anoi(4 from A to B using C)

'Ασκηση: Δείξτε ότι ο ακόλουθος επαναληπτικός αλγόριθμος λύνει το πρόβλημα των πύργων του Ανόι:

Θεωρούμε ότι οι πάσαλοι είναι τοποθετημένοι επί κύκλου όπως στο σχήμα 1.3. Ο επαναληπτικός αλγόριθμος λειτουργεί ως εξής:

Επαναλάμβανε συνέχεια τα παρακάτω βήματα μέχρι να μην μπορείς να εκτελέσεις το δεύτερο βήμα:

1. *μετακίνησε κατά την θετική φορά τον μικρότερο δίσκο·*
2. *κάνε την μοναδική επιτρεπτή κίνηση που δεν αφορά τον μικρότερο δίσκο.*



Σχήμα 1.3: Πύργοι του Ανόι ($n = 4$): για τον επαναληπτικό αλγόριθμο.

1.2.3 Επαγωγή (Induction)

Η *επαγωγή* είναι μία σημαντική μέθοδος για την απόδειξη διαφόρων προτάσεων. Είναι πολύ χρήσιμη κατά την απόδειξη της ορθότητας των προγραμμάτων. Επίσης, η επαγωγή χρησιμοποιείται στον λεγόμενο επαγωγικό τρόπο ορισμού, π.χ.: "Μία λίστα είναι είτε η κενή λίστα είτε ένα στοιχείο ακολουθούμενο από μία λίστα".

1.2.4 Μερική και ολική ορθότητα

Στα προγράμματα συνήθως διακρίνουμε τρία επίπεδα ορθότητας: **συντακτική** ορθότητα (αντιστοιχεί στο context free κομμάτι της γλώσσας), **νοηματική** ορθότητα (αντιστοιχεί στο context sensitive κομμάτι της γλώσσας) και **σημασιολογική** ορθότητα. Στα προγράμματα ο compiler ελέγχει τα 2 πρώτα επίπεδα ορθότητας. Η σημασιολογική ορθότητα δεν μπορεί φυσικά να ελεγχθεί από τον compiler και άρα ελέγχεται με δοκιμές (testing) ή με μαθηματική επαλήθευση (verification) σε σχέση με κάποιες προδιαγραφές (specifications). Γενικά, επιδιώκουμε ένα πρόγραμμα να έχει απλή δομή, δηλαδή να αποτελείται από δομικά στοιχεία (building blocks), επαναχρησιμοποιούμενες ενότητες (modules) για τις οποίες έχει ήδη γίνει αυστηρή επαλήθευση ορθότητας. Υπάρχουν πολλών ειδών σημασιολογίες προγραμμάτων (program semantics):

- **Λειτουργική σημασιολογία** (operational semantics). Περιγράφει την υπολογιστική ακολουθία που εκτελείται.
- **Δηλωτική σημασιολογία** (denotational semantics). Ορίζει μόνο τη συνάρτηση εισόδου-εξόδου.
- **Αξιωματική σημασιολογία** (axiomatic semantics). Περιγράφει τις σχετικές ιδιότητες που πρέπει απαραίτητα να ικανοποιούνται από την είσοδο και την έξοδο.

Θα περιοριστούμε σε μια ελάχιστη περιγραφή της τρίτης. Διάφορες ιδιότητες βεβαιώνονται (are asserted) σε ορισμένα σημεία της ροής του προγράμματος. Αυτές οι συνθήκες που βεβαιώνονται, και πρέπει να αποδειχθεί ότι ικανοποιούνται, λέγονται **βεβαιώσεις** (assertions). Οι βεβαιώσεις βρόχου λέγονται **αναλλοίωτες του βρόχου** (loop invariants).

Δεν αρκεί να αποδείξουμε την ορθότητα σε περίπτωση τερματισμού. Η απόδειξη ορθότητας σε περίπτωση τερματισμού λέγεται **μερική ορθότητα** (partial correctness). Για την **ολική ορθότητα** (total correctness) χρειάζεται και απόδειξη τερματισμού. **Συνθήκη τερματισμού** (termination condition) λέγεται μια γνησίως φθίνουσα θετική συνάρτηση $f(t)$ που εγγυάται τον τερματισμό όταν $f(t) = 0$ (όπου t ο χρόνος, ο αριθμός των βημάτων που έχουν εκτελεστεί).

Στην περίπτωση που, αντί για κατηγορηματικό λογισμό και φυσικούς αριθμούς, χρησιμοποιούμε πράξεις και ιδιότητες (αξιώματα) κάποιας άλλης συγκεκριμένης αλγεβρικής δομής η αξιωματική σημασιολογία ονομάζεται συνήθως **αλγεβρική σημασιολογία** (algebraic semantics).

1.3 Δομημένος προγραμματισμός και modularity

Η ιδεολογία του δομημένου προγραμματισμού υποστηρίζει ότι για κάθε ανεξάρτητη συγκεκριμένη λειτουργία πρέπει να γράφεται μια ανεξάρτητη διαδικασία. Θα

πρέπει λοιπόν να αναλύεται όσο γίνεται το πρόβλημα σε ανεξάρτητα υποπροβλήματα και για το καθένα από αυτά να γράφεται κάποιο υποπρόγραμμα (διαδικασία ή συνάρτηση). Επίσης, πρέπει να αποφεύγεται η χρήση παρενεργειών. Αυτό σημαίνει ότι κάθε υποπρόγραμμα πρέπει να κάνει μια συγκεκριμένη λειτουργία, χωρίς να επηρεάζει το κυρίως πρόγραμμα ή άλλα υποπρογράμματα. Η επικοινωνία των επιμέρους μονάδων (πέρασμα τιμών κ.τ.λ.) γίνεται με χρήση παραμέτρων (βλέπε παρακάτω). Έτσι, το πρόγραμμα γίνεται ευανάγνωστο και είναι εύκολη η μεταφορά και η χρήση των υποπρογραμμάτων σε άλλα προγράμματα .

- Είναι ευκολότερο να γράψουμε ένα δομημένο πρόγραμμα, επειδή πολύπλοκα προβλήματα διασπώνται σε έναν αριθμό μικρότερων, απλούστερων εργασιών.
- Είναι ευκολότερο να ανιχνεύουμε λάθη σε δομημένο πρόγραμμα.
- Ένα σχετικό πλεονέκτημα είναι ο χρόνος που εξοικονομείται με δομημένα προγράμματα. Μπορούν να διορθωθούν ή να τροποποιηθούν πιο εύκολα. Όταν έχουμε κατασκευάσει μια διαδικασία που κάνει κάτι συγκεκριμένο, θα μπορούμε να τη χρησιμοποιούμε σε άλλα προγράμματα δίχως να σπαταλάμε χρόνο σε συγγραφή νέων υποπρογραμμάτων.

Σε ορισμένες γλώσσες, όπως η Modula-2, αλλά και στις περισσότερες σύγχρονες γλώσσες, υποστηρίζεται η διάσπαση του κώδικα σε διαφορετικές αλληλοσυνεργαζόμενες **ενότητες** οι οποίες ονομάζονται **modules**. Κάθε μια από τις ενότητες αυτές περιέχει τα δικά της αντικείμενα (σταθερές, τύπους, μεταβλητές, υποπρογράμματα). Μια τέτοια ενότητα μπορεί να αναφέρεται σε (ή να καλεί) αντικείμενα που είτε είναι εσωτερικά σ' αυτήν είτε έχουν εισαχθεί, με αυστηρό τρόπο, από άλλες ενότητες. Βέβαια για να γίνει δυνατή η εισαγωγή τέτοιων αντικειμένων θα πρέπει να έχει γίνει επιτρεπτή η εξαγωγή τους στην ενότητα που έχουν οριστεί. Τελικά, ένα πρόγραμμα αποτελείται από ένα σύνολο ενοτήτων, εντελώς ανεξαρτήτων ως προς την υλοποίηση, οι οποίες όμως επικοινωνούν μεταξύ τους με αυστηρά καθορισμένο τρόπο. Έτσι είναι δυνατή η δημιουργία μιας "**βιβλιοθήκης**" **προγραμμάτων**, οριζόμενων από το χρήστη ώστε κώδικας που έχει γραφτεί μια φορά να μπορεί να χρησιμοποιείται από πολλά προγράμματα που δυνατόν να χρειάζονται τις ίδιες λειτουργίες. Ο βασικός λόγος για αυτή τη δόμηση *είναι η επιθυμία μας για την υλοποίηση μιας ιεραρχίας αφαίρεσης (abstraction) στην οποία modules που βρίσκονται σε υψηλότερο επίπεδο από κάποια άλλα να μπορούν να χρησιμοποιούν αντικείμενα που έχουν οριστεί σε αυτά χωρίς να χρειάζονται να γνωρίζουν τίποτα για τον τρόπο υλοποίησής τους (implementation hiding).*

Απαγορεύοντας έτσι την πρόσβαση στο εσωτερικό της κάθε ενότητας μπορούμε να φτιάξουμε ενότητες οι οποίες (είμαστε σίγουροι ότι) λειτουργούν σωστά και έτσι, σε μετέπειτα στάδιο κατά την χρησιμοποίησή τους από άλλη ενότητα, θα έχουμε μικρύνει κατά πολύ την περιοχή αναζήτησης λαθών.

Εφόσον όμως κάθε ενότητα μπορεί να εισαγάγει αντικείμενα από διάφορες άλλες, ο μεταφραστής (compiler) θα πρέπει να έχει πρόσβαση στην περιγραφή της δομής

των δεδομένων έτσι ώστε να εξασφαλίζεται η σωστή επικοινωνία. Οδηγούμαστε έτσι στην *περαιτέρω διάσπαση ενότητας σε μέρος ορισμού (definition part) και μέρος υλοποίησης (implementation part) τα οποία μπορούν να δημιουργηθούν ανεξάρτητα.*

1.4 Παράλληλες, ταυτόχρονες, κατανεμημένες διεργασίες

Γενικά, οι αλγόριθμοι στους οποίους αναφερθήκαμε ως τώρα εκτελούνται *σειριακά*, όπως λέμε, σε σε έναν υπολογιστή. Παρ' όλα αυτά υπάρχουν ορισμένα προβλήματα, τα οποία μπορούμε να λύσουμε, κατανέμοντας το υπολογιστικό φορτίο σε περισσότερους από έναν υπολογιστές/επεξεργαστές που λειτουργούν *παράλληλα*. Χάρη στον *ταυτοχρονισμό* (concurrency), μειώνεται σημαντικά ο χρόνος που παίρνουμε απάντηση στο πρόβλημα.

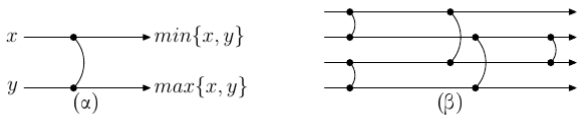
Πολλά προβλήματα σε αυτήν την περιοχή προκύπτουν από τις λεγόμενες κατανεμημένες (distributed) διεργασίες. Αυτές είναι διεργασίες που από την φύση τους εκτελούνται παράλληλα, για παράδειγμα οι συνδιαλέξεις σε ένα τηλεφωνικό δίκτυο.

Το μεγάλο πρόβλημα, στα παράλληλα συστήματα είναι πώς θα *παραλληλοποιήσουμε* έναν σειριακό αλγόριθμο. Για πολλούς αλγορίθμους, αυτό είναι εφικτό με σημαντική επιτυχία.

Παρ'όλα αυτά υπάρχουν και διεργασίες που είναι όπως λέμε *εγγενώς σειριακές* (inherently sequential). Σε αυτές, είναι τόσο έντονη η εξάρτηση ενός βήματος από τα προηγούμενα που δεν είναι δυνατόν τα βήματα να εκτελεστούν παράλληλα και ανεξάρτητα μεταξύ τους.

1.4.1 Δίκτυα ταξινόμησης

Σκοπός των δικτύων ταξινόμησης είναι η ταξινόμηση μίας ακολουθίας αριθμών χρησιμοποιώντας μόνο συγκρίσεις, με όσο το δυνατό μεγαλύτερη παραλληλία. Τα δίκτυα ταξινόμησης αναπαρίστανται με οριζόντιες γραμμές, μία για κάθε είσοδο. Οι συγκρίσεις μεταξύ δύο αριθμών αναπαρίστανται με κατακόρυφες συνδέσεις δύο γραμμών. Κάθε σύγκριση θεωρείται ότι "κοστίζει" μία χρονική μονάδα και η συνολική καθυστέρηση αντιστοιχεί στον αριθμό των κόμβων σε μια οριζόντια γραμμή. Η έξοδος είναι ταξινομημένη από "πάνω" προς τα "κάτω". Στο σχήμα 1.4(α) παρουσιάζεται ένας συγκριτής ενώ στο σχήμα 1.4(β) δίνεται ένα παράδειγμα δικτύου ταξινόμησης τεσσάρων εισόδων.



Σχήμα 1.4: (α) Συγκριτής (β) Δίκτυο ταξινόμησης 4 εισόδων

Η ποιότητα των δικτύων ταξινόμησης χαρακτηρίζεται από το χρόνο που χρειάζονται για να ταξινομήσουν τις εισόδους τους (*βάθος* δικτύου) και το πλήθος των συγκριτών που χρησιμοποιούν (*μέγεθος* δικτύου). Το βάθος του δικτύου ορίζεται ως το μέγιστο βάθος των γραμμών του, ενώ το βάθος μίας γραμμής είναι το πλήθος των συγκρίσεων που πραγματοποιούνται σ' αυτή. Για παράδειγμα, το βάθος και το μέγεθος του δικτύου του σχήματος 1.4(β) είναι 3 και 5 αντίστοιχα. Το μέγεθος αντιστοιχεί στον σειριακό χρόνο ταξινόμησης και το βάθος στον παράλληλο.

1.5 Ταξινόμηση treesort με δένδρο δυαδικής αναζήτησης

Θα δώσουμε έναν αλγόριθμο ταξινόμησης που βασίζεται στα δένδρα δυαδικής αναζήτησης (binary search tree).

Δίνεται μία λίστα αριθμών.

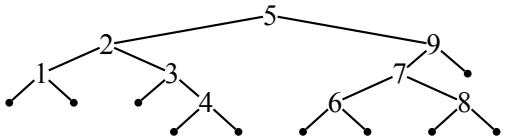
Παίρνουμε ένα ένα τα στοιχεία της λίστας με την σειρά και κατασκευάζουμε ένα δυαδικό δένδρο αναζήτησης. Αυτό κατασκευάζεται ως εξής: Το πρώτο στοιχείο της λίστας τοποθετείται στην ρίζα του δένδρου. Τα υπόλοιπα στοιχεία διασχίζουν το δένδρο μέσω των κόμβων μέχρι να βρουν την θέση τους σε κάποιο φύλλο του δένδρου ως εξής: αν το ήδη τοποθετημένο στοιχείο στον κόμβο είναι μεγαλύτερο από το στοιχείο που επιδιώκουμε να τοποθετήσουμε το προωθούμε στο δεξιό υποδένδρο του κόμβου, αλλιώς στο αριστερό. Επίσης, σε κάθε νέο στοιχείο που τοποθετούμε στο δένδρο φροντίζουμε να δημιουργούμε δύο κόμβους παιδιά οι οποίοι είναι κενοί (empty).

Μετά διασχίζουμε το δένδρο με την λεγόμενη ενδοδιατεταγμένη (inorder) σειρά, οπότε προκύπτουν τα στοιχεία ταξινομημένα. Μία αναδρομική διαδικασία που τυπώνει τα στοιχεία του δένδρου σε ενδοδιατεταγμένη σειρά είναι η εξής:

```
procedure inorder(t: treenode)
begin
    if t is not empty then
    begin
        inorder(left branch of t);
        write(element at t);
        inorder(right branch of t)
    end
end
```

Το δυαδικό δένδρο που προκύπτει από την εκτέλεση του αλγορίθμου για είσοδο: 5, 2, 3, 9, 7, 1, 8, 4, 6 φαίνεται στο σχήμα 1.5. Εκτελέστε ενδοδιατεταγμένη διάσχιση για να δείτε ότι όντως τα στοιχεία τυπώνονται ταξινομημένα.

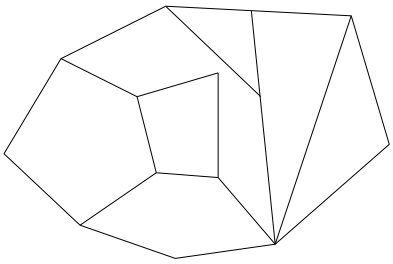
Ο αλγόριθμος δεν είναι βέλτιστος. Η κατασκευή του δυαδικού δένδρου, αν αυτό δεν προκύψει ισορροπημένο, μπορεί να πάρει $O(n^2)$ βήματα.



Σχήμα 1.5: Δένδρο αναζήτησης

1.6 Το θεώρημα τεσσάρων χρωμάτων (four color theorem)

Αναφερόμαστε στους γνωστούς επίπεδους χάρτες, όπως αυτός του σχήματος 1.6, όπου οι φραγμένες περιοχές αντιπροσωπεύουν χώρες.



Σχήμα 1.6: Επίπεδος χάρτης

Το ερώτημα είναι (δεδομένου ενός τέτοιου χάρτη):

Πόσα χρώματα αρκούν για τον χρωματισμό όλων των χωρών, ούτως ώστε χώρες που συνορεύουν (δηλαδή έχουν γραμμή, όχι απλώς σημείο για κοινό σύνορο) να έχουν διαφορετικό χρώμα;

Με την πρώτη ματιά, φαίνεται ότι το πλήθος των απαιτούμενων χρωμάτων μπορεί να γίνει πολύ μεγάλο αν δοθεί κάποιος ιδιαίτερα περίπλοκος χάρτης. Παρ' όλα αυτά, *είναι αρκετά τέσσερα χρώματα, σε κάθε περίπτωση*. Η παραπάνω πρόταση είναι γνωστή ως *θεώρημα τεσσάρων χρωμάτων*. Δοκιμάστε, για παράδειγμα, να χρωματίσετε τον χάρτη του σχήματος 1.6 με τέσσερα χρώματα.

Το πρόβλημα (αν αρκούν τέσσερα χρώματα) τέθηκε για πρώτη φορά το 1852 από τον Guthrie και αργότερα από τον Cayley το 1878. Ένα χρόνο μετά, ο Kempe παρουσίασε μία απόδειξη, η οποία όμως ήταν λανθασμένη. Σημειωτέον ότι η απόδειξη ότι αρκούν πέντε χρώματα είναι σχετικά εύκολη και την έδωσε ο Heawood το 1890, μεταβάλλοντας κάπως την λανθασμένη απόδειξη του Kempe. Από τότε, πολλοί επιδίωξαν να αποδείξουν το θεώρημα των τεσσάρων χρωμάτων.

Η πρώτη γενικά αποδεκτή απόδειξη δημοσιεύτηκε από τους Appel και Haken το 1977. Σε γενικές γραμμές, η απόδειξη λέει ότι αν είναι δυνατόν να χρωματιστεί κάθε ένας χάρτης από ένα πεπερασμένο σύνολο περιπτώσεων, τότε μπορεί να χρωματιστεί οποιοσδήποτε χάρτης. Το πρόβλημα είναι ότι πρέπει να αποδειχθεί για

κάθε χάρτη από το πεπερασμένο σύνολο περιπτώσεων ότι αρκούν τέσσερα χρώματα και στην απόδειξη των Appel και Haken οι περιπτώσεις αυτές είναι περίπου 1700 το πλήθος και ορισμένες από αυτές αρκετά περίπλοκες. Λόγω του μεγάλου πλήθους των περιπτώσεων, ο χρωματισμός των περιπτώσεων γίνεται με την βοήθεια υπολογιστή. Επιπλέον και οι 1700 περίπου περιπτώσεις έχουν προκύψει με την βοήθεια υπολογιστή. Με λίγα λόγια, δεν φαίνεται να είναι δυνατόν να ελέγξει την απόδειξη κάποιος άνθρωπος χωρίς την βοήθεια του υπολογιστή και για αυτόν τον λόγο η απόδειξη των Appel και Haken δέχθηκε αρκετή κριτική. Η απόδειξη βασίζεται στην ορθότητα (βλέπε ενότητα 1.2.4) των χρησιμοποιηθέντων προγραμμάτων. Όμως, λόγω της περιπλοκότητας των προγραμμάτων, η ορθότητα τους δεν έχει πλήρως αποδειχθεί. Επιπλέον, αν θέλουμε να είμαστε απόλυτα αυστηροί, ούτε για τους μεταγλωττιστές με τους οποίους μεταφράστηκαν τα προγράμματα έχουμε απόδειξη ορθότητας.

Πάντως, η απόδειξη τελικά έγινε αποδεκτή, ειδικά επειδή επιπλέον έχουν εμφανιστεί και νεότερες αποδείξεις, όπως για παράδειγμα αυτή των Robertson, Sanders, Seymour, Thomas, οι οποίες μειώνουν κάπως τον αριθμό των εξεταζόμενων περιπτώσεων, αλλά και πάλι ο έλεγχος αν αρκούν τέσσερα χρώματα βασίζεται σε υπολογιστή.

Ασκήσεις

1. Αναδρομή – Επανάληψη – Επαγωγή:

   (α) Εκφράστε τον αριθμό μετακινήσεων δίσκων που κάνει ο αναδρομικός αλγόριθμος για τους πύργους του Ανόι, σαν συνάρτηση του αριθμού των δίσκων n.

   (β) Βρείτε τη σχέση ανάμεσα στον αριθμό των μετακινήσεων του αναδρομικού και τον αριθμό των μετακινήσεων του επαναληπτικού αλγορίθμου.

   (γ) Δείξτε ότι ο αριθμός των μετακινήσεων του αναδρομικού αλγορίθμου είναι ο ελάχιστος μεταξύ όλων των δυνατών αλγορίθμων για το πρόβλημα αυτό.

2. Δυαδική αναζήτηση:

   (α) Περιγράψτε (σε ψευδοκώδικα) διαδικασία / συνάρτηση treeinsert(T, k), που να δέχεται ένα δένδρο δυαδικής αναζήτησης T και έναν αριθμό k, και να εισάγει τον αριθμό k στην σωστή θέση του δένδρου, ώστε αυτό να διατηρεί την ιδιότητα του δένδρου δυαδικής αναζήτησης. Υποθέστε ότι, δεδομένου του T, έχετε πρόσβαση στο 'αριστερό' / 'δεξί' παιδί του T (π.χ. μέσω κάποιας συνάρτησης leftchild(T) / rightchild(T)).

   (β) Ποια είναι η πολυπλοκότητα της συνάρτησής σας, όταν εκτελεστεί διαδοχικά για n αριθμούς; Μελετήστε την πολυπλοκότητα χειρότερης περίπτω-

σης και, προαιρετικά, την πολυπλοκότητα της μέσης περίπτωσης (θεωρώντας κάθε διάταξη των n αριθμών εισόδου ισοπίθανη).

(γ) Υλοποιήστε την συνάρτησή σας σε γλώσσα προγραμματισμού της επιλογής σας, και συνδυάστε την με κατάλληλη διάσχιση ώστε να φτιάξετε μια συνάρτηση treesort που να δέχεται λίστα ακεραίων και να την επιστρέφει ταξινομημένη.

3. Θεώρημα τεσσάρων χρωμάτων:

   (α) Αποδείξτε ότι υπάρχουν περιπτώσεις χαρτών που δεν μπορούν να χρωματιστούν με 3 χρώματα.

   (β) Αποδείξτε ότι 5 χρώματα αρκούν για χρωματισμό οποιουδήποτε χάρτη.

4. Δίκτυα ταξινόμησης:

   (α) Σχεδιάστε ένα δίκτυο ταξινόμησης οκτώ εισόδων. Τι βάθος και τι μέγεθος έχει το δίκτυό σας; Τι σημαίνει αυτό για τον χρόνο σειριακής και τι για τον χρόνο παράλληλης εκτέλεσης του αντίστοιχου αλγορίθμου ή κυκλώματος;

   (β) Μπορείτε να σχεδιάσετε καλύτερη μέθοδο ταξινόμησης οκτώ αριθμών με συγκρίσεις; Ποιος είναι ο αριθμός συγκρίσεων της μεθόδου σας; Είναι ελάχιστος; Αποδείξτε τους ισχυρισμούς σας.

Κεφάλαιο 2

Αλγόριθμοι

2.1 Αλγόριθμοι και Πολυπλοκότητα

Η ονομασία Αλγόριθμος προέρχεται από το όνομα του ’Αραβα Μαθηματικού Al-Khowârizmi (με καταγωγή από το Ουζμπεκιστάν, που έζησε στη Βαγδάτη τον 9ο αιώνα μ.χ). Ήταν ο πρώτος που διατύπωσε τους κανόνες για τις 4 βασικές αριθμητικές πράξεις (από δικό του βιβλίο προέρχεται και η ’Αλγεβρα).

Υπενθυμίζουμε ότι σκοπός ενός αλγορίθμου είναι η επίλυση ενός προβλήματος. Πριν ξεκινήσουμε ας θέσουμε ορισμένες ερωτήσεις για τα προβλήματα: Ποιος ορίζει πρόβλημα; Τι είναι πρόβλημα; Πώς μοντελοποιούμε ένα πρόβλημα; Τι είναι λύση; Πώς αναπαριστούμε μια λύση (γλώσσα); Είναι πράγματι μια προτεινόμενη λύση, λύση του προβλήματος (ορθότητα); Ποια είναι η αποδοτικότητα της μεθόδου εξεύρεσης λύσης (πολυπλοκότητα); Πότε είναι μια λύση καλή; Καλύτερη; Βέλτιστη; Ποία είναι τα όρια εφαρμοσιμότητας μιας οποιασδήποτε λύσης; Είναι ένα πρόβλημα δύσκολο; Πώς χειριζόμαστε τα δύσκολα προβλήματα; Και μια απάντηση: Λύνουμε προβλήματα με συνδυασμό ευφυΐας, διαίσθησης, τύχης, πείρας και σκληρής δουλειάς.

Σε έναν αλγόριθμο μας ενδιαφέρουν: ορθότητα, πολυπλοκότητα, αν εφαρμόζεται γενικά (δηλαδή για όλα τα πιθανά στιγμιότυπα εισόδου), αν είναι βέλτιστος, ακριβής ή προσεγγιστικός, πιθανοτικός κ.τ.λ.

2.1.1 Τι είναι αλγόριθμος

Η έννοια αλγόριθμος είναι πρωταρχική έννοια της θεωρίας αυτής. Γι’ αυτό δεν ορίζεται. Εδώ δίνουμε μία άτυπη εξήγηση.

Αλγόριθμος είναι ένα πεπερασμένο σύνολο κανόνων, οι οποίοι περιγράφουν μία μέθοδο (που αποτελείται από μία σειρά υπολογιστικών διεργασιών) για να λυθεί ένα συγκεκριμένο πρόβλημα. Τα αντικείμενα πάνω στα οποία επενεργούν αυτές οι διεργασίες λέγονται **δεδομένα** (*data*).

Ο αλγόριθμος χαρακτηρίζεται από τα παρακάτω πέντε στοιχεία:

- Κάθε εκτέλεση είναι *πεπερασμένη*, δηλαδή τελειώνει ύστερα από έναν πεπερασμένο αριθμό διεργασιών ή βημάτων (*finiteness*).
- Κάθε κανόνας του ορίζεται επακριβώς και η αντίστοιχη διεργασία είναι συγκεκριμένη (*definiteness*).
- Έχει μηδέν ή περισσότερα μεγέθη *εισόδου* που δίδονται εξαρχής, πριν αρχίσει να εκτελείται ο αλγόριθμος (*input*).
- Δίδει τουλάχιστον ένα μέγεθος σαν αποτέλεσμα (*έξοδο-output*) που εξαρτάται κατά κάποιο τρόπο απ'τις αρχικές εισόδους.
- Είναι *μηχανιστικά* αποτελεσματικός, δηλαδή όλες οι διαδικασίες που περιλαμβάνει μπορούν να πραγματοποιηθούν με ακρίβεια και σε πεπερασμένο χρόνο "με μολύβι και χαρτί" (*effectiveness*).

2.1.2 Πολυπλοκότητα αλγορίθμων και προβλημάτων

Στην πράξη, το ενδιαφέρον δεν σταματά στο να βρεθεί ένας αλγόριθμος που επιλύει ένα πρόβλημα, αλλά προχωρά στη μελέτη των μετρήσιμων ιδιοτήτων που χαρακτηρίζουν την **αποδοτικότητα** μιας υπολογιστικής μεθόδου. Αυτά τα μεγέθη (αγαθά-*resources*) είναι π.χ. ο χρόνος υπολογισμού, ο χώρος σε μνήμη υπολογιστή, ο αριθμός προκαταρκτικών διαδικασιών που προαπαιτούνται και είναι αυτά που ορίζουν την **πολυπλοκότητα** (*complexity*) του αλγορίθμου. Ονομάζουμε **πολυπλοκότητα ενός προβλήματος** την πολυπλοκότητα ενός **βέλτιστου** (*optimal*) αλγορίθμου που λύνει το πρόβλημα.

Η συμπεριφορά του αλγορίθμου μελετάται κυρίως σε δύο περιπτώσεις. Στην **χειρότερη** (*worst case*) και στην **μέση** (*average case*), μιας δεδομένης κατανομής πιθανών **στιγμιοτύπων** (*instances*) του προβλήματος. Μια άλλη ανάλυση ενδιαφέρεται για την **μακροπρόθεσμη απόσβεση** (*amortization*) επαναληπτικής χρήσης ενός αλγορίθμου. Η μελέτη της πολυπλοκότητας ενός αλγορίθμου μας επιτρέπει πολλές φορές να αποφανθούμε αν αυτός είναι **βέλτιστος** (*optimal*) για το συγκεκριμένο πρόβλημα. Αυτό προϋποθέτει ότι έχουμε **τα άνω (με αλγόριθμο)** και **κάτω (με απόδειξη) φράγματα του χρόνου** (ή και **του χώρου)** που επαρκούν και απαιτούνται για την επίλυση ενός προβλήματος και επίσης προϋποθέτει ότι αυτά ταυτίζονται.

Το κόστος ενός αλγορίθμου εξαρτάται φυσικά και από την είσοδο (input). Λογικά το κόστος αυξάνεται με την αύξηση του μεγέθους της εισόδου. Από την άλλη μεριά το κόστος μπορεί να διαφέρει για διαφορετικές εισόδους ίδιου μεγέθους.

Αντιστοιχούμε λοιπόν σ'έναν αλγόριθμο μία συνάρτηση, η οποία μας δίδει την ποσότητα χώρου ή χρόνου που απαιτείται για την επίλυση ενός προβλήματος με δεδομένα διαφόρου μεγέθους.

Το κόστος ενός αλγορίθμου ορίζεται με τη βοήθεια της παρακάτω συνάρτησης:

$$\text{κόστος αλγορίθμου}(n) = \max_{\substack{\text{για όλες τις δυνατές εισόδους } x \text{ μεγέθους } n}} \{\text{κόστος αλγορίθμου για είσοδο } x\}$$

Και το κόστος ενός προβλήματος, με τη βοήθεια της συνάρτησης:

$$\text{κόστος προβλήματος } (n) = \min_{\substack{\text{για όλους τους} \\ \text{αλγόριθμους A που} \\ \text{επιλύουν το πρόβλημα}}} \{\text{κόστος του αλγορίθμου } A(n)\}$$

Αριθμητική Πολυπλοκότητα – Πολυπλοκότητα Ψηφιοπράξεων (Arithmetic *vs.* Bit Complexity)

Η μέτρηση πολυπλοκότητας γίνεται συνήθως μετρώντας το πλήθος των στοιχειωδών αριθμητικών πράξεων ή εντολών (κάθε απλή αριθμητική πράξη, και κάθε απλή εντολή μιας γλώσσας προγραμματισμού θεωρούνται ότι έχουν μοναδιαίο κόστος) και τότε λέγεται *αριθμητική πολυπλοκότητα* (*a*rithmetic complexity). Η πιο ακριβής μέτρηση πολυπλοκότητας που λαμβάνει υπ'όψη της το πλήθος των στοιχειωδών ψηφιοπράξεων (π.χ. η σύγκριση δύο αριθμών b ψηφίων απαιτεί b ψηφιοπράξεις, ο πολλαπλασιασμός τους απαιτεί, με τον σχολικό αλγόριθμο, b^2 ψηφιοπράξεις, κ.ο.κ.) λέγεται *πολυπλοκότητα ψηφιοπράξεων* (*b*it complexity) και είναι απαραίτητη όταν στην είσοδο εμφανίζονται 'μεγάλοι' αριθμοί, των οποίων το μέγεθος επηρεάζει σημαντικά το πλήθος ή/και το κόστος των αριθμητικών πράξεων που θα εκτελέσει ο αλγόριθμος. [1]

2.1.3 Ντετερμινιστικοί και μη ντετερμινιστικοί αλγόριθμοι

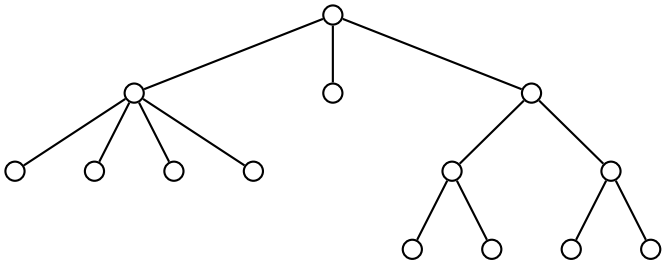
Ένας αλγόριθμος μπορεί να είναι **ντετερμινιστικός** (*deterministic*) ή **μη ντετερμινιστικός** (*nondeterministic*). Ο ντετερμινιστικός αλγόριθμος διακρίνεται από τα παρακάτω στοιχεία:

- Ο υπολογισμός που προτείνει είναι γραμμικός. Σε κάθε υπολογιστικό βήμα υπάρχει μία μοναδική επιτρεπτή επόμενη κατάσταση (configuration, διαμόρφωση).
- Η υπολογιστική διαδικασία προχωρεί βήμα προς βήμα και πάντοτε σταματάει για οποιαδήποτε είσοδο.

Σχηματικά ο ντετερμινιστικός αλγόριθμος φαίνεται στο σχήμα 2.1. Ο μη ντετερμινιστικός αλγόριθμος παριστάνεται στο σχήμα 2.2. Όπως φαίνεται στο σχήμα 2.2

[1] Για τις ανάγκες ανάλυσης των περισσότερων αλγορίθμων που θα δούμε στις σημειώσεις αυτές αρκεί η εκτίμηση της αριθμητικής πολυπλοκότητας. Όπου θα χρειαστούμε την πολυπλοκότητα ψηφιοπράξεων αυτό θα αναφέρεται ρητά.

Σχήμα 2.1: Ντετερμινιστικός αλγόριθμος



Σχήμα 2.2: Μη ντετερμινιστικός αλγόριθμος

ο μη ντετερμινιστικός υπολογισμός είναι δέντρο, μη γραμμικός: για κάθε υπολογιστική διαμόρφωση (κόμβο του δέντρου) μπορεί να υπάρχουν πολλές, μία ή και καμία νόμιμες επόμενες υπολογιστικές διαμορφώσεις. Συνεπώς ο ντετερμινιστικός αλγόριθμος είναι ειδική περίπτωση μη ντετερμινιστικού αλγορίθμου. Σε μια άλλη ισοδύναμη περιγραφή ο μη ντετερμινιστικός αλγόριθμος αποτελείται από δύο διαφορετικές φάσεις. Στην πρώτη φάση ο μη ντετερμινιστικός αλγόριθμος επιλέγει μια επιτρεπτή ακολουθία από υπολογιστικές διαμορφώσεις (δηλαδή ένα μονοπάτι στο δέντρο) και στη δεύτερη φάση, με μια καθαρά ντετερμινιστική διαδικασία ελέγχει αν το αποτέλεσμα που δίνει η πρώτη φάση αποτελεί λύση του προβλήματος. Η έννοια του μη ντετερμινιστικού αλγορίθμου δεν είναι κατ' ανάγκη πρακτικά χρήσιμη για την επίλυση προβλημάτων. Χρησιμεύει όμως στην θεωρητική ταξινόμηση της δυσκολίας των προβλημάτων.

Θέλοντας να τυποποιήσουμε δηλαδή να ορίσουμε αυστηρά την έννοια του αλγορίθμου, είναι απαραίτητο να ορίσουμε ένα συγκεκριμένο υπολογιστικό μοντέλο. Πολλοί επιστήμονες, όπως οι *A. Turing, A. Church, S. Kleene, E. Post, A. Markov* κ.α., ασχολήθηκαν με το θέμα αυτό και όρισαν διάφορα υπολογιστικά μοντέλα.

Το υπολογιστικό μοντέλο που αντιστοιχεί στον πιο "φυσικό" και διαισθητικό ορισμό του αλγορίθμου είναι η **μηχανή Turing**. Σύμφωνα με την αξιωματική **"θέση των Church-Turing"**:

> *"Κάθε αλγόριθμος μπορεί να περιγραφεί με τη βοήθεια μιας μηχανής Turing"*

ή διατυπωμένη αλλιώς:

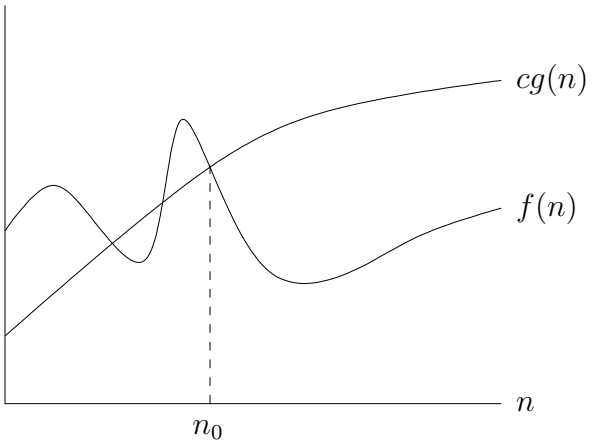
> *"Όλα τα γνωστά και άγνωστα υπολογιστικά μοντέλα είναι μηχανιστικά ισοδύναμα"*

δηλαδή:

> *"Για μια συγκεκριμένη συνάρτηση f, δοθέντος ενός αλγορίθμου σ'ένα υπολογιστικό μοντέλο μπορούμε με τη βοήθεια μηχανής (ή προγράμματος: compiler) να κατασκευάσουμε, για την ίδια συνάρτηση f, αλγόριθμο σ'ένα άλλο υπολογιστικό μοντέλο".*

2.2 Μαθηματικοί Συμβολισμοί

Συνήθως δεν μας ενδιαφέρει να μετρήσουμε ακριβώς το κόστος εκτέλεσης ενός αλγόριθμου αλλά να βρούμε μόνο την τάξη μεγέθους του κόστους. Μας ενδιαφέρει η ασυμπτωτική συμπεριφορά του αλγόριθμου. Με άλλα λόγια αναζητούμε την οριακή αυξητική τάση της συνάρτησης που εκφράζει την πολυπλοκότητα του αλγόριθμου καθώς αυξάνεται το μέγεθος της εισόδου (input). Έτσι λοιπόν ένας αλγόριθμος που έχει καλύτερη ασυμπτωτική συμπεριφορά (μικρότερη τάξη μεγέθους ή, ισοδύναμα, μικρότερο ρυθμό αύξησης) από έναν άλλο για το ίδιο πρόβλημα, θα είναι και η καλύτερη επιλογή για όλα τα μεγέθη της εισόδου (εκτός ίσως από τα πολύ μικρά).



Σχήμα 2.3: $f = O(g)$

Ορίζουμε τώρα κάποια σύμβολα που χρησιμοποιούνται στη μελέτη της **αυξητικής τάσης** (*rate of growth*) μιας συνάρτησης:

Έστω $f : \mathbb{N}\backslash\{0\} \to \mathbb{N}\backslash\{0\}$ και $g : \mathbb{N}\backslash\{0\} \to \mathbb{N}\backslash\{0\}$, όπου $\mathbb{N}$ οι φυσικοί αριθμοί. Να σημειώσουμε εδώ ότι μας ενδιαφέρουν μόνο μονότονα αύξουσες συναρτήσεις (η συνάρτηση $\frac{1000}{n}$ π.χ. δεν εμφανίζεται ως πολυπλοκότητα αλγορίθμου).

$$O(g) = \{f \mid \exists c > 0,\ \exists n_0 : \forall n > n_0 \ \ f(n) \leq cg(n)\}$$

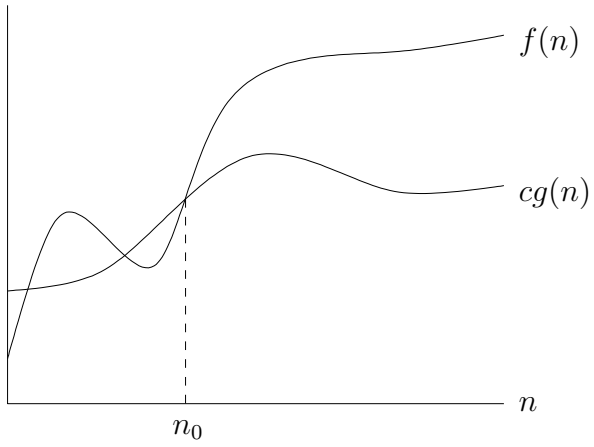
Το $O(g)$ περιλαμβάνει "χοντρικά" τις συναρτήσεις μικρότερης ή ίδιας τάξης μεγέθους με την g.

$$\Omega(g) = \{f \mid \exists c > 0,\ \exists n_0 : \forall n > n_0 \ \ f(n) \geq cg(n)\}$$

Το $\Omega(g)$ περιλαμβάνει "χοντρικά" τις συναρτήσεις μεγαλύτερης ή ίδιας τάξης μεγέθους με την g.

$$\Theta(g) = \{f \mid \exists c_1 > 0,\ \exists c_2 > 0,\ \exists n_0 : \forall n > n_0 \ \ c_1 \leq \frac{f(n)}{g(n)} \leq c_2\}$$

Το $\Theta(g)$ περιλαμβάνει "χοντρικά" τις συναρτήσεις της ίδιας τάξης μεγέθους με την g.



Σχήμα 2.4: $f = \Omega(g)$

Αν $g \in O(f)$ συνήθως γράφουμε $g(n) = O(f(n))$ και λέμε ότι συνάρτηση g είναι τάξης μεγέθους f.

Ισχύουν: $\Theta(f) = O(f) \bigcap \Omega(f)$ και $f \in \Theta(g) \iff (f \in O(g)$ και $\ g \in O(f))$.

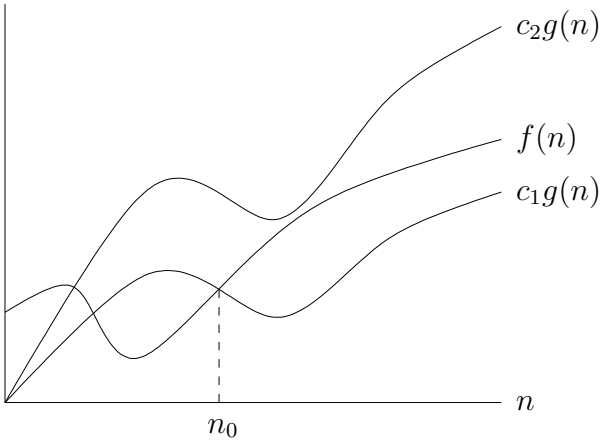
Αν $p = c_k n^k + c_{k-1} n^{k-1} + \cdots + c_0$, δηλαδή πολυώνυμο βαθμού k, τότε $p \in \mathrm{O}(n^k)$ ή $p(n) = \mathrm{O}(n^k)$. Επίσης $p \in \Omega(n^k)$ ή $p(n) = \Omega(n^k)$. Συνεπώς $p(n) = \Theta(n^k)$.

Ορίζουμε $\mathrm{O}(poly) = \bigcup \mathrm{O}(n^k)$.

Γενικά ισχύει ότι: $O(1) < O(\log^* n) < O(\log(n)) < O(\sqrt{n}) < O(n) < O(n\log(n)) < O(n^2) < \ldots < O(poly) < O(2^n) < O(n!) < O(n^n) < O(2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}^{n})$

Σημείωση: γράφουμε "$<$" αντί "$\subset$".

Ορισμός 2.1. $\log^* n$ δίνει πόσες φορές πρέπει να λογαριθμήσουμε το n για να πάρουμε 1.



Σχήμα 2.5: $f = \Theta(g)$

2.3 Εύρεση μέγιστου κοινού διαιρέτη

Δίνονται δύο θετικοί ακέραιοι a και b. Αναζητούμε τον μέγιστο κοινό διαιρέτη τους— ΜΚΔ (greatest common divisor — gcd):

$$z := gcd(a, b)$$

Μία πρώτη ιδέα είναι να παραγοντοποιήσουμε τους αριθμούς σε γινόμενο πρώτων (από το Θεώρημα μοναδικής παραγοντοποίησης σε πρώτους, για κάθε θετικό ακέραιο αυτό το γινόμενο είναι μοναδικό) και να εντοπίσουμε τους κοινούς παράγοντες, δηλαδή να ανάγουμε το πρόβλημα σε αυτό της εύρεσης πρώτων παραγόντων. Δυστυχώς, ο αλγόριθμος δεν είναι ιδιαίτερα αποδοτικός. Μάλιστα, δεν γνωρίζουμε καν αν υπάρχει αλγόριθμος πολυωνυμικού χρόνου για το πρόβλημα εύρεσης πρώτων παραγόντων.

2.3.1 Ένας απλός αλγόριθμος για το gcd

$z := min(a, b)$;
while (a **mod** $z \neq 0$) **or** (b **mod** $z \neq 0$) **do** $z := z - 1$;
return (z)

Η ορθότητα του παραπάνω αλγορίθμου είναι προφανής: Δεν υπάρχει αριθμός $w > z$ τέτοιος ώστε να διαιρεί και τον a και τον b.

Η πολυπλοκότητα του αλγορίθμου είναι $O(\min(a, b))$ στην χειρότερη περίπτωση και αυτό συμβαίνει όταν οι αριθμοί a, b είναι πρώτοι μεταξύ τους. Ακόμη όμως και στην μέση περίπτωση ο αλγόριθμος δεν αποδίδει καλά, δεδομένου ότι αν το

$\min(a, b)$ δεν διαιρεί ακριβώς το $\max(a, b)$, τότε ο βρόχος επαναλαμβάνεται τουλάχιστον $\min(a, b)/2$ φορές.

2.3.2 Αλγόριθμος με αφαιρέσεις για το gcd

$i := a$; $j := b$;
while $i \neq j$ **do if** $i > j$ **then** $i := i - j$ **else** $j := j - i$;
return (i)

Η απόδειξη της ορθότητας βασίζεται στο εξής: Αν w διαιρεί το a και το b (με $a > b$) τότε διαιρεί και το $a - b$.

Όσο για την πολυπλοκότητα, αυτή είναι $O(\max(a, b))$ για την χειρότερη περίπτωση, όταν για παράδειγμα το $b = 1$. Πάντως, στην μέση περίπτωση ο αλγόριθμος με τις αφαιρέσεις είναι καλύτερος από τον προηγούμενο αλγόριθμο.

2.3.3 Αλγόριθμος του Ευκλείδη

$i := a$; $j := b$;
while $(i > 0)$ **and** $(j > 0)$ **do**
  if $i > j$ **then** $i := i$ **mod** j **else** $j := j$ **mod** i;
return $(i + j)$

Η απόδειξη της ορθότητας βασίζεται στο εξής: Αν w διαιρεί το a και το b (με $a > b$) τότε διαιρεί και το a **mod** b.

Ο αλγόριθμος έχει πολυπλοκότητα χειρότερης περίπτωσης $O(\log(a + b))$, αλλά και $O(log(\min(a, b)))$. Συνήθως, όμως, τερματίζει πιο γρήγορα.

Μάλιστα, την χειρότερη απόδοση ο αλγόριθμος του Ευκλείδη την παρουσιάζει αν του δοθούν ως είσοδος δύο διαδοχικοί αριθμοί της ακολουθίας Fibonacci:

$$F_k = \begin{cases} 0 & \text{για } k = 0 \\ 1 & \text{για } k = 1 \\ F_{k-1} + F_{k-2} & \text{για } k \geq 2 \end{cases}$$

Ισχύει $F_k = (\phi^k - \hat{\phi}^k)/\sqrt{5}$, όπου $\phi = (1 + \sqrt{5})/2 \approx 1.618$ (γνωστή και ως η *χρυσή τομή*) και $\hat{\phi} = (1 - \sqrt{5})/2$. Ας σημειώσουμε ότι επειδή $|\hat{\phi}| < 1$, το F_k είναι ίσο με το $\phi^k/\sqrt{5}$ στρογγυλοποιημένο στον πλησιέστερο ακέραιο, οπότε προκύπτει το παρακάτω πόρισμα:

Πόρισμα 2.2. $\log_\phi(F_k) + 1 \leq k \leq \log_\phi(F_k) + 2$

Έχουμε τα παρακάτω αποτελέσματα σχετικά με την πολυπλοκότητα:

Πρόταση 2.3. *Ο αλγόριθμος του Ευκλείδη για $a = F_{k+1}$ και $b = F_k$ έχει χρονική πολυπλοκότητα $\Theta(k)$.*

Πόρισμα 2.4. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι* $\Omega(\log(a+b))$.

Λήμμα 2.5. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι* $O(\log(a+b))$.

Απόδειξη. Σε δύο επαναλήψεις, ο μεγαλύτερος από τους δύο αριθμούς τουλάχιστον υποδιπλασιάζεται. Οι λεπτομέρειες αφήνονται ως άσκηση. □

Θεώρημα 2.6. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι* $\Theta(\log(a+b))$.

Απόδειξη. Προκύπτει από το παραπάνω λήμμα ότι ο αλγόριθμος θα εκτελέσει το πολύ $2\log(a+b)$ επαναλήψεις. Το πλήθος βημάτων κάθε επανάληψης φράσσεται από μία μικρή σταθερά. □

2.4 Ύψωση σε δύναμη με επαναλαμβανόμενο τετραγωνισμό (repeated squaring)

Για να υπολογίσουμε το a^n, ο απλοϊκός αλγόριθμος χρειάζεται n πολλαπλασιασμούς, επομένως έχει (αριθμητική) πολυπλοκότητα $O(n)$. Ο παρακάτω αλγόριθμος επιτυγχάνει σημαντική βελτίωση.

function fastpower(a, n)
  result := 1;
  while n>0 **do**
    if odd(n) **then** result:=result*a;
    $n := n$ **div** 2;
    $a := a$*a
  return result

Παράδειγμα 2.7. $a^{13} = a^{1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0}$

Η (αριθμητική) πολυπλοκότητα του αλγορίθμου είναι $O(\log n)$, δηλαδή πολυωνυμική ως προς το μήκος της εισόδου.

2.5 Αριθμοί Fibonacci (Υπολογισμός)

Παρακάτω δίνονται 3 αλγόριθμοι για τον υπολογισμό του n-οστού αριθμού Fibonacci (βλ. και πιο πάνω για ορισμό), και συγκρίνεται η πολυπλοκότητά τους.

2.5.1 Αναδρομικός αλγόριθμος

function $F(n)$
  if (n<2) **then return** n
  else return $F(n-1)$+$F(n-2)$;

Πολυπλοκότητα: $T(n) = T(n-1) + T(n-2) + c$, δηλαδή η $T(n)$ ορίζεται όπως η $F(n)$ (συν κάτι μικρό), οπότε (μπορούμε να αποδείξουμε ότι):

$$(n) > F(n) = \Omega(1.62^n)$$

2.5.2 Επαναληπτικός αλγόριθμος

function $F(n)$
  a:=0; b:=1;
  for i:=2 **to** n **do**
    c:=b; b:=a+b; a:=c;
  return b;

Η πολυπλοκότητα είναι γραμμική, $O(n)$.

2.5.3 Αλγόριθμος με πίνακα

Μπορούμε να γράψουμε τον υπολογισμό σε μορφή πινάκων:

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$

Από το παραπάνω συμπεραίνουμε ότι:

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Αν χρησιμοποιηθεί ο αλγόριθμος επαναλαμβανόμενου τετραγωνισμού για την ύψωση σε δύναμη του πίνακα, τότε ο αριθμός των αριθμητικών πράξεων μειώνεται στο $O(\log n)$. Έτσι, ο αλγόριθμος αυτός έχει αριθμητική πολυπλοκότητα που είναι πολυωνυμική ως προς το μέγεθος της εισόδου. Δεν συμβαίνει όμως το ίδιο για την πολυπλοκότητα ψηφιοπράξεων του αλγορίθμου και την χωρική πολυπλοκότητα. Η εκτίμηση αυτών των πολυπλοκοτήτων, και η σύγκριση με τις αντίστοιχες του επαναληπτικού αλγορίθμου αφήνεται ως άσκηση.

2.5.4 Σύγκριση αλγορίθμων

Θεωρήστε τέσσερα προγράμματα με αριθμό βημάτων $O(2^n)$, $O(n^2)$, $O(n)$, και $O(\log n)$ που το καθένα χρειάζεται 1 δευτερόλεπτο για να υπολογίσει το $F(100)$ (σε διαφορετικούς υπολογιστές). Στον παρακάτω πίνακα, φαίνεται πόσα δευτερόλεπτα θα χρειαστούν για να υπολογιστεί το $F(n)$.

	$c \cdot 2^n$	$c \cdot n^2$	$c \cdot n$	$c \cdot \log n$
$F(100)$	1	1	1	1
$F(101)$	2	1.02	1.01	1.002
$F(110)$	1024	1.21	1.1	1.02
$F(200)$	2^{100}	4	2	1.15

2.6 Δυαδική αναζήτηση (Binary Search)

Η δυαδική αναζήτηση είναι ίσως το πιο απλό παράδειγμα της τεχνικής **Divide and Conquer**. Στην δυαδική αναζήτηση, για να διαπιστώσουμε αν ένα δοσμένο κλειδί είναι στοιχείο ενός ήδη ταξινομημένου πίνακα (έστω μη φθίνουσα διάταξη), συγκρίνουμε το κλειδί με το στοιχείο που βρίσκεται στη μεσαία θέση του πίνακα. Αν το κλειδί είναι μικρότερο τότε θα βρίσκεται (αν υπάρχει) στο πρώτο μισό του πίνακα, αλλιώς στο δεύτερο μισό. Συνεχίζοντας αναδρομικά την εφαρμογή της παραπάνω διαδικασίας, καταλήγουμε στο να εντοπίσουμε, αν υπάρχει, το δοσμένο κλειδί. Μια υλοποίηση της δυαδικής αναζήτησης είναι αυτή που φαίνεται στον αλγόριθμο 2.1.

Αλγόριθμος 2.1 Δυαδική αναζήτηση (Binary Search)

```
function BinSearch (a:array[p..q] of item; search: item): info;
  var first, last, mid: index; found: boolean;
begin
  first := p; last := q; found := (search=a[first]) or (search=a[last]);
  while not found and (first<last) do
  begin
    mid := (first+last) div 2;
    if search < a[mid] then begin last := mid−1; first := first+1 end
                       else begin first := mid; last := last−1 end
    found := (search = a[first]) or (search = a[last])
  end
  if found then if search=a[first] then return(first) else return(last)
           else return('not found')
end
```

Αν $T(n)$ είναι η συνάρτηση που δίνει την χρονική πολυπλοκότητα του αλγορίθμου 2.1, τότε εύκολα καταλαβαίνουμε ότι ισχύει:

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ T(\frac{n}{2}) + c & , \text{για } n > 1 \end{cases}$$

Στην περίπτωση που $n > 1$ έχουμε:

$$T(n) \quad = \quad T(\frac{n}{2}) + c = T(\frac{n}{4}) + 2c = T(\frac{n}{8}) + 3c = \ldots = T(\frac{n}{2^k}) + kc$$

Αν ο πίνακας έχει 2^k στοιχεία ($n = q-p+1 = 2^k$), τότε έχουμε ότι $T(n) = T(1)+c\log n$. Άρα ο αλγόριθμος 2.1, σε ταξινομημένο πίνακα στοιχείων χρειάζεται χρόνο τάξης $O(\log n)$.

Σημείωση 2.1. Η τεχνική της δυαδικής αναζήτησης εφαρμόζεται άμεσα και σε περιπτώσεις που αναζητούμε ρίζες μονότονων συναρτήσεων σε συγκεκριμένο διάστημα ακέραιων ή και πραγματικών αριθμών (κατά προσέγγιση). Η πολυπλοκότητά της αποδεικνύεται ότι είναι $O(\log n + k)$ όπου n το εύρος του διαστήματος και k η ακρίβεια σε δεκαδικά ψηφία. Η απόδειξη αφήνεται ως άσκηση.

2.7 Εύρεση του μεγαλύτερου και του μικρότερου στοιχείου

Έστω ένας πίνακας S_n που τα στοιχεία του είναι ακέραιοι αριθμοί μη ταξινομημένοι. Ζητάμε να βρούμε το μεγαλύτερο και το μικρότερο στοιχείο του πίνακα. Ο απλός αλγόριθμος δουλεύει ως εξής: με $n-1$ συγκρίσεις (για $n \geq 2$) βρίσκουμε το μεγαλύτερο από τα στοιχεία του πίνακα και έπειτα με $n-2$ βρίσκουμε το μικρότερο από τα $n-1$ στοιχεία. Δηλαδή συνολικά έχουμε $2n-3$ συγκρίσεις. Η τεχνική **Divide and Conquer** χωρίζει την ακολουθία των n ακεραίων σε δύο υποακολουθίες μήκους $\frac{n}{2}$ και υπολογίζει το μεγαλύτερο και το μικρότερο στοιχείο αυτών των υποακολουθιών. Στη συνέχεια με δύο συγκρίσεις, μια ανάμεσα στα μεγαλύτερα και μια ανάμεσα στα μικρότερα στοιχεία, βρίσκει το μεγαλύτερο και το μικρότερο στοιχείο της αρχικής ακολουθίας. Τα μεγαλύτερα και τα μικρότερα στοιχεία των υποακολουθιών υπολογίζονται αναδρομικά. Αν $T(n)$ είναι η συνάρτηση που δίνει τον αριθμό των συγκρίσεων, είναι εύκολο να δούμε ότι ισχύει:

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

Για $n > 2$ έχουμε:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 2 = \\ &= 4T(\frac{n}{4}) + 4 + 2 = \\ &= 8T(\frac{n}{8}) + 8 + 4 + 2 = \\ &= \ldots = \\ &= 2^k T(\frac{n}{2^k}) + 2\sum_{i=0}^{k-1} 2^i = \\ &= 2^k T(\frac{n}{2^k}) + 2\frac{2^k - 1}{2-1} = \\ &= 2^k T(\frac{n}{2^k}) + 2^{k+1} - 2 \end{aligned}$$

Όταν ο πίνακας έχει $n = 2^{k+1}$ στοιχεία ($\frac{n}{2} = 2^k$) τότε έχουμε:

$$T(n) = \frac{n}{2}T(2) + 2^{k+1} - 2 = \frac{3}{2}n - 2 \tag{2.1}$$

2.8 Πολλαπλασιασμός ακεραίων (integer multiplication)

Το πρόβλημα του κλασσικού πολλαπλασιασμού δύο n-bit ακεραίων, έστω των X και Y, περιλαμβάνει τον υπολογισμό n μερικών γινομένων μεγέθους n, είναι δηλαδή μια πράξη με πολυπλοκότητα της τάξης $O(n^2)$. Η τεχνική Divide and Conquer χωρίζει τον καθένα από τους X και Y σε δύο ακεραίους που έχουν μήκος $\frac{n}{2}$ bits ο καθένας. Αν υποθέσουμε ότι το n είναι μια δύναμη του 2 τότε έχουμε:

$$X : \boxed{a}\boxed{b} \quad = a \cdot 2^{\frac{n}{2}} + b$$

$$Y : \boxed{c}\boxed{d} \quad = c \cdot 2^{\frac{n}{2}} + d$$

Τότε το γινόμενο των X και Y εκφράζεται ως εξής:

$$X\ Y = ac \cdot 2^n + (ad + bc) \cdot 2^{\frac{n}{2}} + bd \tag{2.2}$$

Για να υπολογίσουμε αυτό το γινόμενο χρειάζεται να κάνουμε τέσσερις πολλαπλασιασμούς $\frac{n}{2}$-bit ακεραίων, τρεις προσθέσεις, το πολύ $2n$-bit, ακεραίων και δύο ολισθήσεις. Αφού οι προσθέσεις και οι ολισθήσεις είναι πράξεις πολυπλοκότητας $O(n)$ μπορούμε να γράψουμε την παρακάτω αναδρομική σχέση για να περιγράψουμε τη συνάρτηση χρονικής πολυπλοκότητας του πολλαπλασιασμού:

$$T(n) = \begin{cases} a & \text{, για } n = 1 \\ 4T(\frac{n}{2}) + cn & \text{, για } n > 1 \end{cases}$$

Κάνοντας πράξεις όπως και στις προηγούμενες περιπτώσεις καταλήγουμε στο:

$$T(n) = (c+1)n^2 - cn = O(n^2)$$

Δηλαδή η πράξη του πολλαπλασιασμού αν γίνει με τη χρήση της 2.1, έχει την ίδια πολυπλοκότητα με τον κλασικό πολλαπλασιασμό. Μπορούμε να βελτιώσουμε την κατάσταση αν μειώσουμε τον αριθμό των υποπροβλημάτων και αυτό γίνεται με την παρακάτω παρατήρηση:

$$(ad + bc) \ = \ [(a-b)(d-c) + ac + bd] \tag{2.3}$$

Συνδυάζοντας τις 2.1 και 2.2 έχουμε:

$$X\ Y = ac \cdot 2^n + [(a-b)(d-c) + ac + bd]\ 2^{\frac{n}{2}} + bd \tag{2.4}$$

Για τον υπολογισμό της 2.3 χρειάζονται μόνο τρεις πολλαπλασιασμοί $\frac{n}{2}$-bit ακεραίων $(ac,\ (a-b)(d-c), bd)$ και οι υπόλοιπες πράξεις έχουν τάξη πολυπλοκότητας $O(n)$. Η συνάρτηση $T(n)$ αυτή τη φορά έχει ως εξής (a σταθερά):

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 3T(\frac{n}{2}) + cn & , \text{για } n > 1 \end{cases}$$

Τελικά έχουμε ότι:

$$T(n) \ = \ O(n^{\log_2 3}) \ = \ O(n^{1.59})$$

2.9 Πολλαπλασιασμός πινάκων (matrix multiplication)

Παρόμοιο με την προηγούμενη εφαρμογή είναι και το πρόβλημα πολλαπλασιασμού δύο πινάκων. Αν A και B είναι δύο πίνακες $n \times n$, το γινόμενό τους $A \times B$ είναι ένας $n \times n$ πίνακας C, του οποίου τα στοιχεία δίνονται από τον τύπο:

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

Η κλασσική μέθοδος πολλαπλασιασμού, δηλαδή η άμεση εφαρμογή του παραπάνω τύπου, έχει πολυπλοκότητα χρόνου $O(n^3)$. Η μέθοδος DIVIDE AND CONQUER, προτείνει τον διαχωρισμό κάθε πίνακα σε τέσσερις υποπίνακες διαστάσεων $\frac{n}{2} \times \frac{n}{2}$ ο καθένας. Αν θεωρήσουμε ότι $n = 2^k$, τότε το γινόμενο των πινάκων $A \times B$ θα δίνεται από τον τύπο:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

όπου

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

και η πολυπλοκότητα του αλγορίθμου ως προς το χρόνο θα είναι:

$$T(n) \ = \ 8T(\frac{n}{2}) + 4c(\frac{n}{2})^2 \ = \ O(n^3)$$

(c σταθερά) ενώ η πολυπλοκότητα χώρου θα είναι $S(n) = O(1)$. Σημειωτέον ότι για τον πολλαπλασιασμό αυτό θα χρειαστούν οκτώ πολλαπλασιασμοί $\frac{n}{2} \times \frac{n}{2}$ πινάκων και τέσσερις προσθέσεις $\frac{n}{2} \times \frac{n}{2}$ πινάκων. Η πολυπλοκότητα $O(n^3)$ προκύπτει με προσεκτικές αντικαστάσεις ή με τη μέθοδο του Κυρίαρχου Όρου (Master Theorem).

Το 1969 ο *Volker Strassen* απέδειξε ότι για να προσδιορίσουμε τα C_{ij}, αρκεί να χρησιμοποιήσουμε μόνον επτά πολλαπλασιασμούς $\frac{n}{2} \times \frac{n}{2}$ πινάκων και δεκαοκτώ

προσθέσεις $\frac{n}{2} \times \frac{n}{2}$ πινάκων. Σύμφωνα με τη μέθοδο του, πρώτα υπολογίζονται οι επτά $\frac{n}{2} \times \frac{n}{2}$ πίνακες $P,\ Q,\ R,\ S,\ T,\ U,\ V$ και μετά υπολογίζονται τα C_{ij}:

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(-B_{11} + B_{12}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Τα C_{ij} υπολογίζονται ως εξής:

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

Η πολυπλοκότητα χρόνου σε αυτήν την περίπτωση είναι:

$$T(n) \ = \ 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 \ \leq \ 43n^{\log 7}$$

Άρα:

$$T(n) \ = \ O(n^{\log 7}) \ \approx \ O(n^{2.81})$$

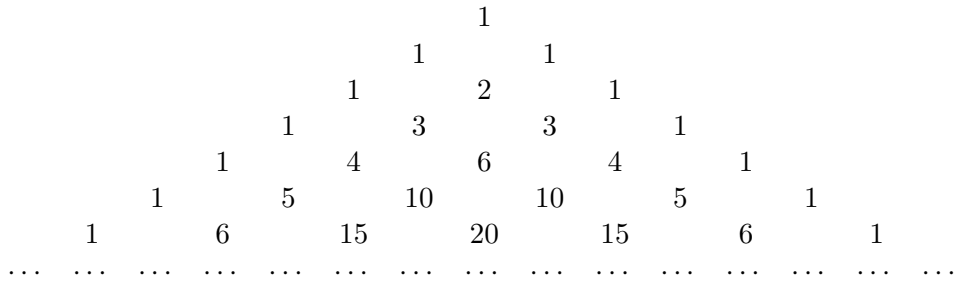
όπως προκύπτει με προσεκτικές αντικαστάσεις ή με τη μέθοδο του Κυρίαρχου Όρου (Master Theorem).

Ο *Strassen* απέδειξε επίσης ότι ο αριθμός 7 (πολλαπλασιασμοί υποπινάκων) δεν μπορεί να βελτιωθεί. Στα τελευταία είκοσι χρόνια έχουν βρεθεί καλύτεροι αλγόριθμοι (*Schonhage, Bini, Pan*) με πολυπλοκότητα $T(n) \approx O(n^{2.38})$. Από την άλλη μεριά το καλύτερο γνωστό κάτω φράγμα είναι $\Omega(n^2)$. Υπάρχει δηλαδή ένα κενό μεταξύ άνω και κάτω φράγματος. Συνεπώς αυτή η βελτίωση πολυπλοκότητας πολλαπλασιασμού πινάκων είναι ένα πεδίο έρευνας με έντονη δραστηριότητα.

Σημείωση 2.2. Πολυπλοκότητα χρόνου ίδια με αυτή του πολλαπλασιασμού δύο πινάκων $(M(n))$, έχει και ο αλγόριθμος εύρεσης αντιστρόφου πίνακα (όπως και ο αλγόριθμος εύρεσης της ορίζουσας του πίνακα, κ.α.), γιατί μπορεί να αναχθεί σε πολλαπλασιασμό πινάκων.

2.10 Τρίγωνο Pascal

Το τρίγωνο του Pascal είναι το (άπειρο) τρίγωνο που φαίνεται στο σχήμα 2.10. Κάθε όρος προκύπτει προσθέτοντας τους ακριβώς από επάνω όρους. Επίσης, ο $k-$οστός όρος της $n-$οστής γραμμής του τριγώνου, $a_{n,k}$ προκύπτει και από τον τύπο:



Σχήμα 2.6: Το τρίγωνο του Pascal

$$a_{n,k} = \left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k! \cdot (n-k)!}$$

Οι όροι του τριγώνου του Pascal λέγονται και διωνυμικοί συντελεστές, γιατί ο $a_{n,k}$ είναι ο $k-$οστός συντελεστής του αναπτύγματος του διωνύμου $(a+b)^{n}$΄:

$$(a+b)^n = \left(\begin{array}{c} n \\ 0 \end{array} \right) a^n + \left(\begin{array}{c} n \\ 1 \end{array} \right) a^{n-1}b + \ldots + \left(\begin{array}{c} n \\ n \end{array} \right) a^0 b^n$$

Ένας αλγόριθμος για την εύρεση της $n-$οστής γραμμής του τριγώνου είναι ο αλγόριθμος 2.2. Χρησημοποιεί έναν πίνακα $1 \times n$.

$$a_{n+1,k+2} = a_{n,k+1} + a_{n,k+2}$$

Αλγόριθμος 2.2 Αλγόριθμος για εύρεση της $n-$οστής γραμμής του τριγώνου του Pascal. Στηρίζεται στην παραπάνω σχέση.

```
program pasctri (n);
 var i, j:integer;
     a: array [0..n] of integer;
begin
  for i := 0 to n do
  begin
    a[i] := 1;
    for k := i downto 1 do a[k] := a[k - 1] + a[k]
  end
  output(a)
end.
```

Ο αλγόριθμος αυτός χρειάζεται $\frac{n^2}{2}$ βήματα για να υπολογίσει τη $n-$οστή γραμμή. Υπάρχει όμως και πιο αποδοτικός αλγόριθμος. Αυτός υπολογίζει κάθε όρο της

$n-$οστής γραμμής του τριγώνου και τον χρησιμοποιεί για να υπολογίσει τον επόμενο. Στηρίζεται στη σχέση

$$\begin{pmatrix} n \\ k+1 \end{pmatrix} = \begin{pmatrix} n \\ k \end{pmatrix} \cdot \frac{n-k}{k+1},$$

που είναι προφανής από τον ορισμό των $\begin{pmatrix} n \\ k \end{pmatrix}$.

Αλγόριθμος 2.3 Αλγόριθμος σε $n+1$ βήματα για την εύρεση της $n-$οστής γραμμής του τριγώνου του Pascal.

```
program pasctri2 (n);
 var i:integer;
     a: array [0..n] of integer;
begin
  a[0] := 1
  for i := 1 to n do
    a[i] := a[i − 1] ∗ (n − i)div(i + 1);
  output(a)
end.
```

2.11 Πρώτοι αριθμοί – Παραγοντοποίηση – Κρυπτοσύστημα RSA

Ίσως τα σπουδαιότερα, και σίγουρα τα πλέον "ιστορικά" προβλήματα της Θεωρίας Αριθμών είναι το Primality (έλεγχος αν ένας αριθμός είναι πρώτος) και το Factoring (παραγοντοποίηση). Στην ενότητα αυτή θα συζητήσουμε την πολυπλοκότητά τους και το ρόλο τους στην κρυπτογραφία δημοσίου κλειδιού.

Primality ('Ελεγχος αν ένας αριθμός είναι πρώτος). Το πρόβλημα Primality ορίζεται ως εξής: "Δίνεται ακέραιος n. Είναι πρώτος;". Το πρόβλημα αυτό είναι υπολογιστικά ευεπίλυτο, αν και αυτό δεν είναι καθόλου προφανές. Συγκεκριμένα, ανήκει στην κλάση **P** όπως έδειξαν πρόσφατα οι Agrawal, Kayal και Saxena, που επινόησαν τον αλγόριθμο AKS το 2002 (δες 8.3 για ορισμούς). Για πολλά χρόνια δεν ήταν γνωστό αν το πρόβλημα ήταν στο **P**, αλλά είχαν επινοηθεί ήδη από την δεκαετία του '70 αποδοτικοί *πιθανοτικοί* αλγόριθμοι από τους Miller-Rabin και Solovay-Strassen. Οι αλγόριθμοι αυτοί χρησιμοποιούνται ακόμη και σήμερα καθώς είναι καλύτερης πολυπλοκότητας από τον αλγόριθμο AKS και επίσης πιο εύκολοι στην υλοποίησή τους.

Παρακάτω παρουσιάζουμε τον έλεγχο Miller-Rabin. Στην αρχή θα περιγράψουμε έναν ακόμη απλούστερο πιθανοτικό αλγόριθμο ελέγχου πρώτων, τον έλεγχο Fermat (που όμως αποτυγχάνει σε ελάχιστες περιπτώσεις όπως θα εξηγήσουμε).

Θεώρημα 2.1 (Μικρό Θεώρημα Fermat). *Αν n πρώτος τότε για κάθε $a, 1 \leq a < n-1$, ισχύει*

$$a^{n-1} \bmod n = 1$$

Το θεώρημα αποδεικνύεται με χρήση στοιχειώδους Θεωρίας Αριθμών και η απόδειξη μπορεί να βρεθεί στα περισσότερα σχετικά εγχειρίδια ή και στο διαδίκτυο (WikiPedia).

Από το μ. Θεώρημα Fermat προκύπτει άμεσα ο παρακάτω απλός αλγόριθμο ελέγχου πρώτων αριθμών, που όμως δεν δουλεύει σωστά για κάποιους (πολύ λίγους) αριθμούς, συγκεκριμένα για τους αριθμούς Carmichael (περισσότερα παρακάτω).

Αλγόριθμος 2.4 Έλεγχος πρώτων αριθμών (primality test) Fermat

Input: n

1. randomly choose $a, 1 < a < n-1$
2. **if** $a^{n-1} \bmod n \neq 1$ **then output** 'Composite' (* με βεβαιότητα *)
3. **else output** 'Prime' (* ορθό με μεγάλη πιθανότητα εκτός αν n αριθμός Carmichael *)

Ο αλγόριθμος συμπεριφέρεται ως εξής: Αν ο n είναι πρώτος, τότε η ισότητα ισχύει και πάντα το τεστ θα δώσει σίγουρα τη σωστή απάντηση. Αν ο n είναι σύνθετος, τότε αποδεικνύεται ότι για τουλάχιστον τα μισά a η ισότητα δεν ισχύει, οπότε θα πάρουμε τη σωστή απάντηση με πιθανότητα $\geq 1/2$. Για να αυξήσουμε σημαντικά την πιθανότητα μπορούμε να επαναλάβουμε μερικές φορές (τυπικά 30-40 φορές) με διαφορετικό a. Αν όλες τις φορές βρεθεί να ισχύει η παραπάνω ισότητα τότε λέμε ότι το n "περνάει το test" και ανακηρύσσουμε το n πρώτο αριθμό°αν έστω και μία φορά αποτύχει ο έλεγχος, τότε ο αριθμός είναι σύνθετος.

Σημείωση: Υπάρχουν (λίγοι) σύνθετοι που έχουν την ιδιότητα να περνούν τον έλεγχο Fermat για κάθε a που είναι σχετικά πρώτο με το n, οπότε για αυτούς το test θα αποτύχει όσες δοκιμές και αν γίνουν (εκτός αν πετύχει κανείς a που δεν είναι σχετικά πρώτο με το n, πράγμα πολύ απίθανο). Αυτοί οι αριθμοί λέγονται *Carmichael*.

Ένας έλεγχος πρώτων αριθμών που δεν καλύπτει και την περίπτωση που ο n είναι αριθμός Carmichael, είναι ο έλεγχος Miller-Rabin που αποτελεί βελτίωση του ελέγχου του Fermat και δίνει σωστή απάντηση με πιθανότητα τουλάχιστον 1/2 για *κάθε* φυσικό αριθμό (οπότε με π.χ. 40 επαναλήψεις έχουμε αμελητέα πιθανότητα λάθους για κάθε αριθμό εισόδου). Ο έλεγχος Miller-Rabin περιγράφεται παρακάτω:

Η ορθότητα του ελέγχου βασίζεται στο ότι αν $a^{n-1} \bmod n \neq 1$ τότε ο αριθμός είναι οπωσδήποτε σύνθετος, λόγω του (μικρού) Θεωρήματος του Fermat. Αν όμως $a^{n-1} \bmod n = 1$, τότε αν γράψουμε $n - 1 = 2^s \cdot t$, με t περιττό, η ακολουθία των αριθμών

Αλγόριθμος 2.5 Έλεγχος πρώτων αριθμών (primality test) Miller-Rabin

Input: n

1. **repeat** k times (* παράμετρος ακριβείας, τυπική τιμή $k = 40$ *)
2. randomly choose $a, 1 < a < n - 1$
3. **if** $a^{n-1} \bmod n \neq 1$ **then**
4. **output** 'Composite'; iscomposite := true (* με βεβαιότητα *)
5. **else** compute $s, t : n - 1 = 2^s \cdot t;\ y := a^t;\ z := y^2 \bmod n$
6. **while** $z \not\equiv \pm 1 \pmod{n}$ **do**
7. $y := z;\ z := y^2 \bmod n$
8. **end while**
9. **if** $y \not\equiv -1 \pmod{n} \wedge z \equiv 1 \pmod{n}$ **then**
10. **output** 'Composite'; iscomposite := true (* με βεβαιότητα *)
11. **end repeat**
12. **if not** iscomposite **then output** 'Prime' (* ορθό με πολύ μεγάλη πιθανότητα *)

$$a^t, a^{2t}, \ldots, a^{2^i t}, \ldots a^{2^s t}$$

που προκύπτει με διαδοχικούς τετραγωνισμούς, εμφανίζει σίγουρα σε κάποια θέση ισοτιμία με $1 \pmod{n}$, δηλαδή $\exists i, 0 \leq i \leq s, a^{2^i t} \bmod n = 1$ (γιατί;). Αν για τον προηγούμενο αριθμό στη σειρά, δηλαδή τον $x = a^{2^{i-1} t}$, ισχύει $x \bmod n \neq \pm 1$,[2] τότε έχουμε βρεί μια μη τετριμμένη τετραγωνική ρίζα της μονάδας (modulo n), κάτι που όπως μπορεί να αποδειχθεί οδηγεί άμεσα σε παραγοντοποίηση του n: $\gcd(n, x-1)$ και $\gcd(n, x+1)$ είναι παράγοντες του αριθμού (άσκηση: αποδείξτε γιατί).

Αποδεικνύεται με χρήση θεωρίας ομάδων ότι, αν ο n είναι σύνθετος, τουλάχιστον οι μισές επιλογές για το a παράγουν τέτοιες *ακολουθίες παραγοντοποίησης*. Επομένως η πιθανότητα λάθος απάντησης είναι το πολύ $\frac{1}{2}$. Επαναλαμβάνοντας k φορές μειώνουμε την πιθανότητα αυτή στο $\frac{1}{2^k}$.

[2] Θυμηθείτε ότι στην αριθμητική modulo n το '-1' ταυτίζεται με το $n - 1$, επομένως όταν γράφουμε $x \bmod n \neq \pm 1$ (ή, ισοδύναμα, $x \not\equiv \pm 1 \pmod{n}$) εννοούμε $x \bmod n \neq 1$ και $x \bmod n \neq n - 1$.

Factoring (παραγοντοποίηση σε πρώτους αριθμούς). Το πρόβλημα Factoring ορίζεται ως εξής: "Δίνεται ακέραιος n. Να βρεθούν οι πρώτοι παράγοντές του." Δεν ξέρουμε αν είναι εύκολο ή δύσκολο. Πιστεύουμε ότι δεν είναι στο **P** (άρα μάλλον δύσκολο), αλλά δεν είναι τόσο δύσκολο όσο τα **NP**-πλήρη προβλήματα. Για κβαντικούς υπολογιστές (που δεν έχουμε καταφέρει ακόμα να κατασκευάσουμε) ανήκει στο **P**. Παρ'όλα αυτά, θεωρείται από όλους τους ειδικούς ότι έχει αρκετή δυσκολία ώστε να μην είναι πιθανός ένας πρακτικός αλγόριθμος παραγοντοποίησης ενός αριθμού που διαθέτει χιλιάδες ψηφία.

Factoring και κρυπτοσύστημα RSA. Το RSA είναι ένα κρυπτογραφικό σχήμα δημοσίου κλειδιού για να στείλει η Α (Alice) στον Β (Bob) ένα μήνυμα m (κωδικοποιημένο κείμενο). Δηλαδή, ο Β έχει δύο κλειδιά, ένα δημόσιο, που μπορεί να το γνωρίζει ο καθένας και ένα ιδιωτικό που το ξέρει μόνο αυτός και το χρησιμοποιεί για την αποκρυπτογράφηση.

- Ο Β βρίσκει 2 μεγάλους πρώτους αριθμούς p και q, υπολογίζει το γινόμενο $n = pq$, και ακέραιο e σχετικά πρώτο με το $\phi(n) = (p-1)(q-1)$[3]. Ο e μαζί με τον n αποτελούν το *δημόσιο κλειδί* του Β.

- Ο Β στέλνει στην Α τα n και e. Ή μπορεί να τα έχει μονίμως δημοσιοποιημένα για να τα χρησιμοποιεί οποιοσδήποτε θέλει να του στείλει κρυπτογραφημένο κείμενο.

- Η Α στέλνει στον Β τον αριθμό $c = m^e \bmod n$. Ο c είναι η κρυπτογράφηση του m.

- Ο Β υπολογίζει το m: $m = c^d \bmod n$,
όπου το $d \equiv e^{-1} \pmod{\phi(n)}$, δηλαδή $d \cdot e \bmod \phi(n) = 1$. Το d είναι το *ιδιωτικό κλειδί* του Β.

Αποδεικνύεται ότι αν $m < n$, τότε η αποκρυπτογράφηση είναι σωστή, δηλαδή στο τελευταίο βήμα ισχύει πράγματι ότι $m = c^d \bmod n$.

Παράδειγμα: $p = 11$, $q = 17$, $n = 187$, $e = 21$, $d = 61$, $m = 42$, $c = 9$.

Η ασφάλεια του RSA στηρίζεται στην (εκτιμώμενη) δυσκολία του Factoring: αν μπορούσαμε να κάνουμε παραγοντοποίηση γρήγορα, τότε θα υπολογίζαμε τα p, q, και στη συνέχεια τα $\phi(n), d$. Από την άλλη, το να υπολογίσει κάποιος τον ιδιωτικό εκθέτη d, έχοντας στη διάθεσή του μόνο τα e, n (χωρίς δηλαδή να ξέρει τους παράγοντες p, q του n) είναι πρακτικά το ίδιο δύσκολο με το να παραγοντοποιήσει το n, δηλαδή να λύσει το Factoring: υπάρχει αλγόριθμος που παραγοντοποιεί το n με πολύ μεγάλη πιθανότητα, αν κάποιος γνωρίζει το d (μαζί βέβαια με το e που είναι δημόσιο).

Σημείωση 2.3. Παρ'όλα αυτά, δεν έχει αποδειχθεί με απόλυτη αυστηρότητα η υπολογιστική ισοδυναμία του "σπασίματος" του RSA με το Factoring. Συγκεκριμένα,

[3] Η $\phi(n)$ είναι γνωστή ως συνάρτηση του Euler και εκφράζει το πλήθος των αριθμών $< n$ που είναι σχετικά πρώτοι με τον n

είναι πολύ ενδιαφέρον ανοιχτό ερώτημα, κατά πόσον ένας πολυωνυμικού χρόνου αλγόριθμος που μπορεί να υπολογίσει το m, με είσοδο n, e, c, θα μπορούσε να δώσει έναν αλγόριθμο για την παραγοντοποίηση του n. Δεν γνωρίζουμε δηλαδή κάποια *αναγωγή* (βλ. και σε επόμενο κεφάλαιο) από το πρόβλημα Factoring στο πρόβλημα του "σπασίματος" του RSA, ενώ γνωρίζουμε αναγωγή για το αντίστροφο.

Υλοποίηση του RSA Για την υλοποίηση του RSA χρησιμοποιούνται, μεταξύ άλλων: ένας έλεγχος πρώτων αριθμών (χρησιμεύει για την εύρεση των p, q), ο αλγόριθμος επαναλαμβανόμενου τετραγωνισμού (για την ύψωση σε δύναμη) και ο επεκτεταμένος Ευκλείδειος αλγόριθμος (που επιπλέον εκφράζει τον $\gcd(a, b)$ σαν γραμμικό συνδυασμό των a και b°χρησιμεύει για την εύρεση του d ως αντιστρόφου του e modulo $\phi(n)$: $1 = ke + l\phi(n) \Rightarrow ke \bmod \phi(n) = 1$, οπότε ο k είναι ο αντίστροφος του e modulo $\phi(n)$, δηλαδή ο d).

Ιδιαίτερη προσοχή χρειάζεται στην επιλογή των παραμέτρων. Διάφορα κριτήρια έχουν προταθεί στη σχετική βιβλιογραφία προς αποφυγή περιπτώσεων που μπορούν να οδηγήσουν σε παραγοντοποίηση του n.

Ασκήσεις

1. Ασυμπτωτικός συμβολισμός (i):

   Βάλτε σε φθίνουσα σειρά (αν $f = O(g)$ η f θα βρίσκεται δεξιά της g) τα ακόλουθα: $O(5\sqrt[15]{n})$, $O(n^2 \log n + n!)$, $O(\frac{\log^3 n}{100!})$, $O(5n^5 - 4n^4 + 1)$, $O(n^3 \log^{37} n)$, και $O(3^n)$.

2. Ασυμπτωτικός συμβολισμός (ii):

   Σχεδιάστε σε διάγραμμα Venn τα εξής σύνολα συναρτήσεων:
   $O(\frac{n^2}{\log(100!)}), O(2^n), O(n^3{+}15n), \Omega(2^{2\log n}), O(\log\log n), \Omega(1), \Theta(n^2), \Theta(n!), O(n^{\frac{n}{2}})$.

3. Πύργοι Hanoi:

   Θεωρήστε την παραλλαγή του προβλήματος με 4 πασάλους. Περιγράψτε σε ψευδοκώδικα έναν αλγόριθμο για την μετακίνηση των δίσκων από τον πάσαλο 1 στον πάσαλο 4, με αριθμό κινήσεων σημαντικά μικρότερο από την λύση που χρησιμοποιεί 3 πασάλους μόνο. Εκφράστε τον αριθμό κινήσεων του αλγορίθμου σας σαν συνάρτηση του n.

   Υπόδειξη: σκεφτείτε την μέθοδο "διαίρει και κυρίευε".

4. Αριθμοί Fibonacci (i):

   Υλοποιήστε σε γλώσσα της επιλογής σας τους τρεις αλγορίθμους για υπολογισμό αριθμών Fibonacci και συγκρίνετέ τους πειραματικά (οι δύο ταχύτεροι αλγόριθμοι θα πρέπει να υπολογίζουν τουλάχιστον τον 10^6-οστό όρο). Εξηγήστε τα αποτελέσματα των δοκιμών σας.

5. Αριθμοί Fibonacci (ii):

   Βρείτε την πολυπλοκότητα σε ψηφιοπράξεις (bit complexity) των δύο ταχύτερων αλγορίθμων για υπολογισμό Fibonacci (του επαναληπτικού και του αλγορίθμου με χρήση πίνακα). Τι παρατηρείτε; Πώς επηρεάζεται η πολυπλοκότητα αν χρησιμοποιήσουμε πιο αποδοτικό αλγόριθμο πολλαπλασιασμού (π.χ. Gauss-Karatsuba);.

   Τι συμβαίνει με την χωρική πολυπλοκότητα (κόστος σε χώρο μνήμης);

6. Υπολογισμός κυβικής ρίζας

   (α) Σχεδιάστε απλό αλγόριθμο που να υπολογίζει το ακέραιο μέρος της κυβικής ρίζας ενός φυσικού αριθμού n με διαδοχικές δοκιμές – επιθυμητή πολυπλοκότητα $O(n^{1/3})$.

   (β) Βελτιώστε τον αλγόριθμό σας ώστε να γίνει σημαντικά ταχύτερος – επιθυμητή πολυπλοκότητα $O(\log n)$.

   (γ) Τροποποιήστε τον αλγόριθμο ώστε να υπολογίζει την κυβική ρίζα με ακρίβεια δεκαδικού ψηφίου που θα δίνεται ως παράμετρος (δηλαδή δίνεται επιπλέον φυσικός αριθμός k και ζητείται η κυβική ρίζα του n με ακρίβεια k-οστού δεκαδικού ψηφίου). Δώστε την πολυπλοκότητα του αλγορίθμου σας εκφρασμένη σαν συνάρτηση των n και k.

   Σημείωση: για την πολυπλοκότητα θεωρήστε ότι οι βασικές αριθμητικές πράξεις (πολ/σμός, διαίρεση) κοστίζουν ένα βήμα υπολογισμού (αριθμητική πολυπλοκότητα).

7. Έλεγχος πρώτων αριθμών Fermat

   (α) Υλοποιήστε σε γλώσσα προγραμματισμού της επιλογής σας (που να υποστηρίζει υπολογισμούς με πολύ μεγάλους αριθμούς, εκατοντάδων ψηφίων) τους αλγορίθμους ύψωσης σε δύναμη που μάθαμε (τον απλό και του επαναλαμβανόμενου τετραγωνισμού). Τροποποιήστε τους ώστε να δέχονται και μία 3η παράμετρο, έστω m, και να βρίσκουν το αποτέλεσμα $a^n \bmod m$. Διαπιστώστε με δοκιμές μέχρι ποια περίπου δύναμη του 17 ($\bmod 2^{100}$) μπορείτε να υπολογίσετε με τον κάθε αλγόριθμο σε χρόνο 1 λεπτού. Σχολιάστε.

   (β) Χρησιμοποιήστε τη συνάρτησή σας για να ελέξγετε αν ένας αριθμός είναι πρώτος με τον έλεγχο (test) του Fermat:

   Αν n πρώτος τότε για κάθε a τ.ώ. $1 < a < n-1$, ισχύει

   $$a^{n-1} \bmod n = 1$$

   Ένας πρώτος αριθμός ικανοποιεί τη συνθήκη για οποιαδήποτε επιλογή του a (περνάει πάντα τον έλεγχο Fermat). Όπως είπαμε ήδη, ισχύει (σχεδόν για κάθε αριθμό n) ότι αν ο n είναι σύνθετος, τότε η παραπάνω συνθήκη δεν ικανοποιείται για τουλάχιστον τα μισά a στο διάστημα $(0, n-1]$. Επομένως ένας σύνθετος αριθμός, με πιθανότητα $\geq \frac{1}{2}$ δεν περνάει τον έλεγχο Fermat.

Πώς μπορούμε να αυξήσουμε την πιθανότητα ορθής απάντησης στην περίπτωση που ο αριθμός είναι σύνθετος; Χρησιμοποιήστε τις παραπάνω ιδέες για να ελέγξετε αν n πρώτος για $n = 2^i - 1, i \in \{100, 200, \ldots, 1000\}$.

8. Έλεγχος πρώτων αριθμών Miller-Rabin

   Επεκτείνετε τη συνάρτηση της προηγούμενης άσκησης ώστε να υλοποιήσετε τον έλεγχο Miller-Rabin. Δοκιμάστε τη συνάρτησή σας με είσοδο αριθμούς Carmichael: http://en.wikipedia.org/wiki/Carmichael_number

Κεφάλαιο 3

Αλγόριθμοι γράφων

3.1 Γραφήματα

3.1.1 Εισαγωγικές έννοιες – Ορισμός

Ορισμός 3.1. Γράφος (ή γράφημα) G, ονομάζεται ένα διατεταγμένο ζεύγος συνόλων (V, E), όπου V είναι μη κενό σύνολο στοιχείων και E ένα σύνολο μη διατεταγμένων ζευγών του V, δηλαδή

$$E \subseteq \binom{V}{2}$$

Παράδειγμα 3.2. Αν $V = \{v_1, v_2, v_3, v_4, v_5\}$ είναι ένα μη κενό σύνολο στοιχείων και $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_4, v_5\}\}$ τότε το διατεταγμένο ζεύγος $G = (V, E)$ είναι ένας γράφος.

Τα στοιχεία του μη κενού συνόλου V λέγονται **κορυφές** ή **κόμβοι** (*vertices, nodes*) του γράφου. Τα στοιχεία του συνόλου E λέγονται **ακμές** (*edges*) και μπορούν να συμβολιστούν και με ένα γράμμα, π.χ. e, όπου $e = \{x, y\}, x, y \in V, x \neq y$. Καμιά φορά, καταχρηστικώς, θεωρούμε και ακμές-βρόχους, δηλαδή $e = \{x, x\}$.

Αν $e = \{v_1, v_2\}$ είναι ακμή ενός γράφου G, αυτή ενώνει ή συνδέει τις κορυφές v_1, v_2 του G και μπορεί να συμβολιστεί επίσης ως v_1v_2 ή v_2v_1. Οι κορυφές v_1, v_2 λέγονται **άκρα** (*endpoints*) της ακμής e. Επειδή δε η ακμή e τις συνδέει, λέγονται **γειτονικές** (*adjacent*) κορυφές στο G.

Αν τώρα v_1, v_2 είναι γειτονικές κορυφές στο G, τότε η ακμή v_1v_2 προσπίπτει (*incident*) στις v_1 και v_2. Δύο ακμές που προσπίπτουν στην ίδια κορυφή είναι **γειτονικές** ακμές στο G.

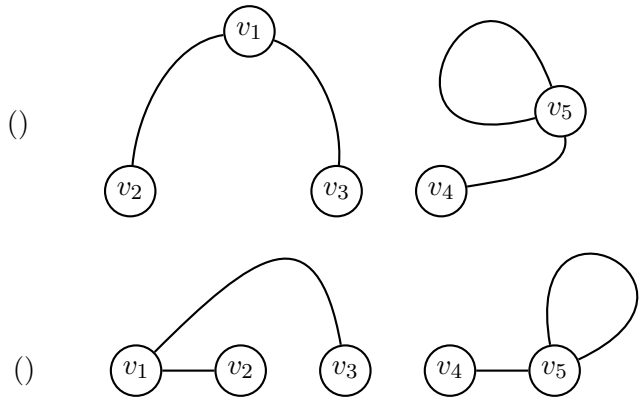
Ο ορισμός του γράφου όπως δόθηκε παραπάνω, δεν διευκολύνει την εποπτική αντίληψη του όρου. Είναι δυνατόν και πολλές φορές επιβάλλεται, για την αναγνώριση και τη μελέτη ιδιοτήτων των γράφων, η απεικόνιση αυτών με τη βοήθεια διαγράμματος.

Για την κατασκευή του διαγράμματος, κάθε κορυφή του γράφου τη σχεδιάζουμε με ένα σημείο, μία κουκίδα και κάθε ακμή με ένα τμήμα καμπύλης γραμμής. Από τον τρόπο κατασκευής του διαγράμματος, είναι φανερό πως δεν υπάρχει μοναδικός τρόπος σχεδίασης ενός γράφου.

Παράδειγμα 3.3. Το διάγραμμα του γραφήματος G με

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_4, v_5\}, \{v_5, v_5\}\}$$

μπορεί να είναι αυτό που φαίνεται στο σχήμα 3.1(α) ή αυτό που φαίνεται στο σχήμα 3.1(β). Οι κορυφές v_1, v_2 είναι γειτονικές στο G ενώ οι v_3, v_4 δεν είναι. Οι ακμές v_1v_2, v_1v_3 είναι γειτονικές στο G ενώ οι v_4v_5, v_1v_2 δεν είναι.



Σχήμα 3.1: Δύο διαφορετικά διαγράμματα για τον γράφο G

Ο αριθμός των κορυφών ενός γράφου $G(V, E)$ ονομάζεται **τάξη** (*order*) του G και συμβολίζεται με $|V|$ και ο αριθμός των ακμών του, **μέγεθος** (*size*) του G και συμβολίζεται με $|E|$. Στην Πληροφορική όμως, συνήθως ονομάζουμε μέγεθος το $n = |V|$.

Παράδειγμα 3.4. Στο γράφο G του Παραδείγματος 2 η τάξη του ισούται με 5 και το μέγεθος με 4. Ο γράφος G μπορεί επίσης να συμβολιστεί και με $G(5, 4)$.

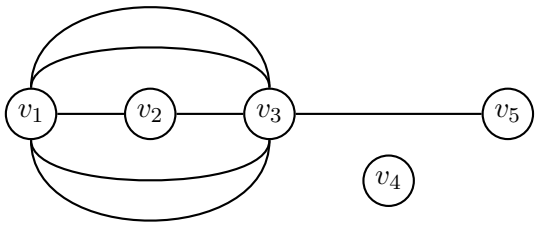
Παρατήρηση 3.5. Από τον ορισμό του γράφου προκύπτει ότι μία ακμή δεν μπορεί να έχει ως άκρα την ίδια κορυφή. Συχνά όμως στην Πληροφορική, όπως αναφέραμε και πιο πάνω, χρειαζόμαστε μια τέτοια ακμή. Η ακμή τότε λέγεται **βρόχος** (*loop*). Επίσης από τον ορισμό του γράφου προκύπτει ότι δεν είναι δυνατή η ύπαρξη περισσοτέρων ακμών με ίδια άκρα, δηλαδή δεν είναι δυνατή η ύπαρξη παράλληλων ακμών. Στο γράφο του Παραδείγματος 2, εφόσον υπάρχει η ακμή v_1v_2 η ύπαρξη μιας παράλληλης της π.χ. v_2v_1, αποκλείεται απ' τον ορισμό.

Ένας γράφος ο οποίος δεν έχει βρόχους, λέγεται στην Πληροφορική **απλός γράφος**. Επειδή σε ότι θα αναφερθεί παρακάτω, δεν επηρεάζει η ύπαρξη ή μη βρόχων

στους γράφους, χωρίς βλάβη της γενικότητας, θα θεωρούμε στο εξής μόνο απλούς γράφους.

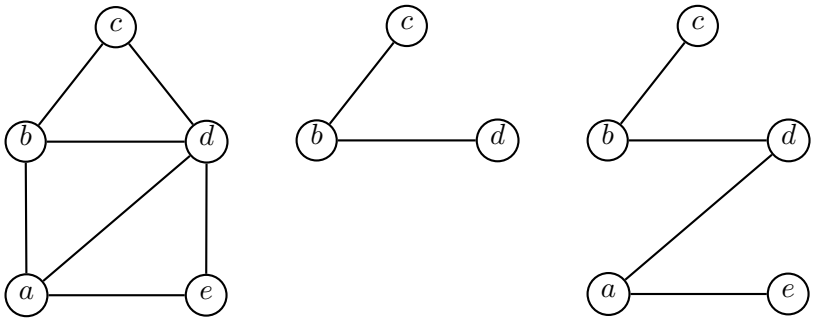
Υπάρχουν όμως και **πολυγραφήματα** (*multigraphs*). Το διάγραμμα ενός πολυγραφήματος μπορεί να περιέχει πολλές ακμές που συνδέουν τις ίδιες κορυφές.

Παράδειγμα 3.6. Στο σχήμα 3.2 φαίνεται ένα πολυγράφημα.



Σχήμα 3.2: Πολυγράφημα

3.1.2 Υπογράφος



Γράφος G Γράφος G' Γράφος G''

Σχήμα 3.3: Ο γράφος G και δύο υπογράφοι αυτού

Ορισμός 3.7. Ένας γράφος $G' = (V', E')$ είναι **υπογράφος** (*subgraph*) ενός άλλου γράφου $G = (V, E)$, αν ισχύει $V' \subseteq V$ και $E' \subseteq E$.

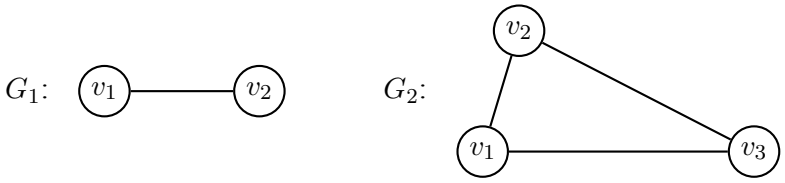
Παράδειγμα 3.8. Στο σχήμα 3.3 ο G' είναι ένας υπογράφος του G.

Ορισμός 3.9. Έστω $G' = (V', E')$ υπογράφος ενός γράφου $G = (V, E)$. Αν ισχύει $V' = V$ τότε ο υπογράφος λέγεται **παράγων υπογράφος** (*spanning subgraph*) του γράφου G.

Παράδειγμα 3.10. Στο σχήμα 3.3 ο G'' είναι παράγων υπογράφος του G.

3.1.3 Βαθμός κορυφής

Ορισμός 3.11. Έστω ένας γράφος $G = (V, E)$. **Βαθμός** (*degree, valence*) μιας κορυφής $v \in V$ ονομάζεται ο αριθμός των ακμών του G που προσπίπτουν στην v και συμβολίζεται με $d_G(v)$ ή $d(v)$. Ένας γράφος $G(V, E)$, για τον οποίο ισχύει $d(v) = k$ για κάθε κορυφή του, λέγεται k-**κανονικός** γράφος.



Σχήμα 3.4: G_1: 1–κανονικός και G_2: 2–κανονικός γράφος

Αποδεικνύεται εύκολα ότι το άθροισμα των βαθμών όλων των κορυφών ενός γράφου, ισούται αριθμητικά με το διπλάσιο του αριθμού των ακμών του. Δηλαδή σε ένα γράφο $G = (V, E)$ έχουμε ότι

$$\sum_{v \in V} d(v) = 2|E|$$

Παράδειγμα 3.12. Στο γράφο G στο σχήμα 3.3 έχουμε $d(b) = 3$, $d(d) = 4$, $d(c) = d(e) = 2$. Στο σχήμα 3.4 ο G_1 είναι $1-$κανονικός γράφος και ο G_2 είναι $2-$κανονικός γράφος.

3.1.4 Δρόμος - Μονοπάτι - Κύκλος

Ορισμός 3.13. Σε ένα γράφο G, μια πεπερασμένη ακολουθία εναλλάξ κορυφών και ακμών του G που αρχίζει και τελειώνει σε κορυφή και που κάθε ακμή που περιέχεται στην ακολουθία προσπίπτει στην κορυφή που προηγείται και σε αυτήν που έπεται, λέγεται **δρόμος** ή **διαδρομή** (*walk*) στο G.

Παράδειγμα 3.14. Στο γράφο G στο σχήμα 3.3 η ακολουθία κορυφών και ακμών του γράφου

$$c\{c, d\}d\{d, b\}b\{b, a\}a\{a, d\}d\{d, b\}b$$

είναι δρόμος στο G.

Ορισμός 3.15. Αν σε έναν δρόμο ενός γράφου κάθε ακμή του δρόμου εμφανίζεται μόνο μία φορά, ο δρόμος λέγεται **δρομίσκος** ή **μονοπάτι** (*trail*).

Παράδειγμα 3.16. Στο γράφο G στο σχήμα 3.3 ο δρόμος

$$d\{d, b\}b\{b, a\}a\{a, d\}d\{d, e\}e$$

είναι δρομίσκος.

Ορισμός 3.17. Ένας δρόμος στον οποίο κάθε κορυφή και κάθε ακμή του εμφανίζονται ακριβώς μία φορά, λέγεται **απλό μονοπάτι** (*path*).

Παράδειγμα 3.18. Στο γράφο G στο σχήμα 3.3 ο δρόμος

$$a\{a,b\}b\{b,c\}c\{c,d\}d\{d,e\}e$$

είναι απλό μονοπάτι.

Ορισμός 3.19. Ένας δρόμος με αρχή και τέλος την ίδια κορυφή, λέγεται **κλειστός δρόμος**, αλλιώς λέγεται **ανοικτός**.

Ορισμός 3.20. Ένας δρόμος που είναι κλειστό μονοπάτι λέγεται **κύκλος** (*cycle*). Ένας δρόμος που είναι απλό κλειστό μονοπάτι λέγεται **απλός κύκλος** (*simple cycle*).

Παράδειγμα 3.21. Στο γράφο G στο σχήμα 3.3

- ο δρόμος $c\{c,b\}b\{b,d\}$ είναι ανοικτός δρόμος
- ο δρόμος $c\{c,b\}b\{b,d\}d\{d,a\}a\{a,b\}b\{b,c\}c$ είναι κλειστός δρόμος
- ο δρόμος $c\{c,b\}b\{b,d\}d\{d,c\}c$ είναι κύκλος

Ορισμός 3.22. Ένας κύκλος που περνά ακριβώς μια φορά από κάθε ακμή ενός γράφου G (χωρίς απαραίτητα να περνά ακριβώς μια φορά και από κάθε κορυφή) ονομάζεται **κύκλος Euler**. Ένας γράφος που έχει κύκλο Euler ονομάζεται **γράφος Euler**. Αποδεικνύεται εύκολα ότι ένας γράφος έχει κύκλο Euler ανν όλες οι κορυφές έχουν άρτιο βαθμό (σχήμα 3.5(α)).

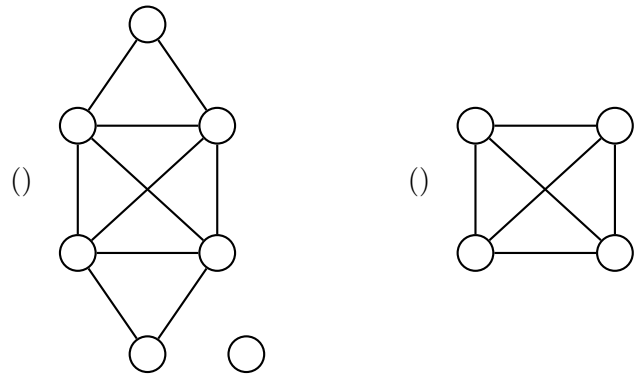
Ορισμός 3.23. Ένας κύκλος που περνά ακριβώς μια φορά από κάθε κορυφή ενός γράφου G (χωρίς απαραίτητα να περνά και από όλες τις ακμές) ονομάζεται **κύκλος Hamilton**. Ένας γράφος που έχει κύκλο Hamilton ονομάζεται **γράφος Hamilton** (σχήμα 3.5(β)).

Σε ένα γράφο G, ένας δρόμος μεταξύ δύο κορυφών u και v του G λέγεται και $(u,v)-$δρόμος ή απλούστερα $uv-$δρόμος. Ο αριθμός των ακμών ενός γράφου που εμφανίζονται σε έναν δρόμο του γράφου, λέγεται **μήκος** του δρόμου.

Παράδειγμα 3.24. Στο παράδειγμα 3.21 τα μήκη των δρόμων με τη σειρά που εμφανίζονται είναι 2, 5 και 3.

3.1.5 Παράσταση Γράφου

Ένας γράφος μπορεί να παρασταθεί με τη βοήθεια του **πίνακα γειτνίασης** (*adjacency matrix*) ή του **πίνακα πρόσπτωσης** (*incidence matrix*) ή των **λιστών γειτνίασης** (*adjacency lists*).



Σχήμα 3.5: (α) Γράφος Euler, (β) Γράφος Hamilton

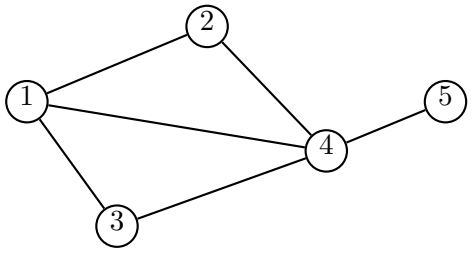
Ορισμός 3.25 (Πίνακας γειτνίασης). Έστω ένας γράφος $G = (V, E)$ με $V = \{v_1, v_2, ..., v_n\}$. Τότε ο γράφος μπορεί να παρασταθεί με τη βοήθεια ενός $n \times n$ πίνακα $A(G)$, όπου

$$A(G) = [a_{ij}], \quad a_{ij} = \begin{cases} 1, & \text{αν } \{v_i, v_j\} \in E \\ 0, & \text{αλλιώς} \end{cases}$$

Ο πίνακας $A(G)$ λέγεται **πίνακας γειτνίασης** (*adjacency matrix*), και είναι συμμετρικός ($a_{i,j} = a_{j,i}$).

Μια άλλη παράσταση είναι με τις **λίστες γειτνίασης** (*adjacency lists*). Η λίστα γειτνίασης μιας κορυφής v περιέχει όλες τις γειτονικές κορυφές της v. Η παράσταση αυτή σε Η/Υ είναι πιο αποδοτική για **αραιούς** γράφους.

Ορισμός 3.26. Οι αραιοί γράφοι έχουν $O(n)$ ακμές ενώ οι πυκνοί γράφοι έχουν $\Omega(n^2)$.



Σχήμα 3.6: Γράφος

Παράδειγμα 3.27. Η αναπαράσταση του γράφου που φαίνεται στο σχήμα 3.6 με τον πίνακα γειτνίασης είναι η παρακάτω:

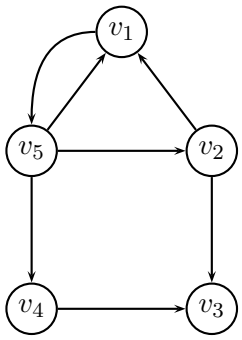
$$A(G) = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Η αναπαράσταση με τις λίστες γειτνίασης είναι η παρακάτω:

$$\begin{array}{lllll} [1]\rightarrow & 2 & 3 & 4 & \\ [2]\rightarrow & 1 & 4 & & \\ [3]\rightarrow & 1 & 4 & & \\ [4]\rightarrow & 1 & 2 & 3 & 5 \\ [5]\rightarrow & 4 & & & \end{array}$$

3.1.6 Προσανατολισμένος Γράφος

Αν στον ορισμό του γράφου αντικαταστήσουμε τα στοιχεία του E με διατεταγμένα ζεύγη στοιχείων του V, παίρνουμε ένα **προσανατολισμένο** ή **κατευθυνόμενο** γράφο (*directed graph, digraph*). Δηλαδή $E \subseteq V \times V$.



Σχήμα 3.7: Κατευθυνόμενος γράφος

Παράδειγμα 3.28. Ο γράφος στο σχήμα 3.7 είναι ένας προσανατολισμένος γράφος. Αν ο γράφος είναι ο $G = (V, E)$ τότε έχουμε:

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E = \{(v_5, v_1), (v_1, v_5), (v_2, v_1), (v_5, v_2), (v_2, v_3), (v_5, v_4), (v_4, v_3)\}$$

Ορισμός 3.29. Έστω x μια κορυφή ενός προσανατολισμένου γράφου. Ο αριθμός των ακμών που προσπίπτουν (εισέρχονται) σε αυτήν την κορυφή ονομάζεται **προσβαθμός** (*in-degree*) της κορυφής x και συμβολίζεται με $deg^-(x)$:

$$deg^-(x) = |\{(y, x) : y \in V \text{ και } (y, x) \in E\}|$$

Ο αριθμός των ακμών που ξεκινούν (εξέρχονται) από την κορυφή x ονομάζεται από-βαθμός (*out-degree*) και συμβολίζεται με $deg^+(x)$:

$$deg^+(x) = |\{(x, y) : y \in V \text{ και } (x, y) \in E\}|$$

Παράδειγμα 3.30. Στον προσανατολισμένο γράφο στο σχήμα 3.7 έχουμε:

$$\begin{aligned} deg^-(v_2) = &\ |\{(v_5, v_2)\}| = 1 \\ deg^+(v_2) = &\ |\{(v_2, v_3), (v_2, v_1)\}| = 2 \end{aligned}$$

3.1.7 Συνεκτικός Γράφος

Ορισμός 3.31. Έστω ένας γράφος $G = (V, E)$. Δύο κορυφές u, v του G είναι **συνδεδεμένες** (*connected*) αν υπάρχει τουλάχιστον ένα uv-μονοπάτι στο G. Η σχέση "σύνδεση δύο κορυφών στο G", είναι μια σχέση ισοδυναμίας στο σύνολο V του G, η οποία δημιουργεί μια διαμέριση (*partition*) σε κλάσεις ισοδυναμίας π.χ. τις $V_1, V_2, \ldots, V_k$. Για τις κλάσεις αυτές ισχύει ότι:

$$\begin{aligned} &V_i \subseteq V && ,1 \leq i \leq k \\ &V_i \bigcap V_j = \emptyset && ,\forall i, j \\ &V_1 \bigcup V_2 \bigcup \ldots \bigcup V_k = V \end{aligned}$$

Προφανώς κάθε ζεύγος κορυφών u, v συνδέονται αν και μόνο αν οι κορυφές u, v ανήκουν στην ίδια κλάση ισοδυναμίας V_i.

Ορισμός 3.32. Έστω ένας γράφος $G = (V, E)$ και $V' \subseteq V$. Ο υπογράφος που έχει σύνολο κορυφών το V' και σύνολο ακμών όλες τις ακμές του G, των οποίων και τα δύο άκρα ανήκουν στο V', λέγεται **παραγόμενος υπογράφος** (*induced subgraph*) από τον V' και συμβολίζεται $G[V']$.

Ορισμός 3.33. Έστω ένα γράφος $G = (V, E)$ και $V_1, V_2, \ldots, V_k$ οι κλάσεις ισοδυναμίας του V που δημιουργούνται απ' τη σχέση "σύνδεση δύο κορυφών". Τα υπογραφήματα $G[V_1], G[V_2], \ldots, G[V_k]$ λέγονται συνεκτικές συνιστώσες (connected components) του γράφου G. Ο αριθμός των συνιστωσών ενός γράφου G συμβολίζεται με (G).

Ορισμός 3.34. Ένας γράφος λέγεται **συνεκτικός** αν αποτελείται από μία μόνο συνιστώσα. Αν ο αριθμός των συνιστωσών ενός γράφου είναι μεγαλύτερος από το 1, ο γράφος λέγεται μη συνεκτικός. Είναι φανερό πως ένας γράφος είναι συνεκτικός, αν για κάθε ζεύγος κορυφών του γράφου υπάρχει ένα μονοπάτι τουλάχιστον, που τις συνδέει.

Παράδειγμα 3.35. Ο γράφος του σχήματος 3.5α είναι μη συνεκτικός με $(G) = 2$. Ο γράφος του σχήματος 3.5β είναι συνεκτικός με $(G) = 1$.

Παρατήρηση 3.36. Στην περίπτωση προσανατολισμένου γράφου οι ακμές σε μονοπάτια (άρα και σε κύκλο) πρέπει να έχουν όλες τον ίδιο προσανατολισμό.

Ορισμός 3.37. Ένας προσανατολισμένος γράφος λέγεται **ισχυρά συνεκτικός** (*strongly connected*) αν για κάθε ζεύγος (u, v) υπάρχει μονοπάτι από το u στο v. Ο γράφος λέγεται **ασθενώς συνεκτικός** (*weakly connected*) αν για κάθε ζεύγος (u, v) υπάρχει μονοπάτι από το u στο v αν αγνοήσουμε τον προσανατολισμό των ακμών.

Ορισμός 3.38. Ένας απλός γράφος $G = (V, E)$ (χωρίς βρόχους και παράλληλες ακμές) ονομάζεται **πλήρης** όταν δύο οποιεσδήποτε κορυφές του είναι γειτονικές. Για ένα πλήρη γράφο προφανώς ισχύει ότι:

$$E = \begin{pmatrix} V \\ 2 \end{pmatrix} \text{ άρα: } |E| = \begin{pmatrix} |V| \\ 2 \end{pmatrix} = \frac{|V|(|V|-1)}{2}$$

Ο πλήρης γράφος με n κορυφές συμβολίζεται με K_n.

Ορισμός 3.39. Ένας γράφος $G(V, E)$ ονομάζεται **διμερής** (*bipartite*) αν το σύνολο των κόμβων V μπορεί να διαμεριστεί σε δύο μη κενά υποσύνολα X και Y έτσι ώστε όλες οι ακμές στο E να ενώνουν έναν κόμβο του X με έναν κόμβο του Y. Ένας πλήρης διμερής γράφος (δηλαδή ο διμερής γράφος στον οποίο κάθε κορυφή του X ενώνεται με κάθε κορυφή του Y) συμβολίζεται με $K_{n,m}$, όπου $n = |X|$ και $m = |Y|$.

Ορισμός 3.40. Ένας γράφος ονομάζεται **επίπεδος** (planar) αν μπορεί να σχεδιαστεί στο επίπεδο έτσι ώστε όλες οι ακμές του να μην διασταυρώνονται, φυσικά εκτός από τις κοινές κορυφές τους.

Έχει αποδειχθεί ότι ικανή και αναγκαία συνθήκη για να είναι ένας γράφος επίπεδος είναι να μην περιέχει υπογράφο ομοιομορφικό με τον K_5 ή τον $K_{3,3}$ (Θεώρημα Kuratowski).

3.2 Δέντρα

3.2.1 Γενικά

Ορισμός 3.41. **Δένδρο** λέγεται ένας συνεκτικός γράφος που δεν περιέχει κύκλους. **Δάσος** λέγεται κάθε γράφος που δεν περιέχει κύκλους. Οι συνεκτικές συνιστώσες ενός δάσους, είναι δέντρα. Ένα δέντρο στο οποίο ξεχωρίζουμε μια κορυφή, την οποία ονομάζουμε **ρίζα**, λέγεται δέντρο με ρίζα (*rooted tree*).

Πρόταση 3.42. *Ένας γράφος είναι δάσος αν και μόνο αν είναι υπογράφος ενός δέντρου.*

Ανάλογοι είναι και οι ορισμοί για προσανατολισμένα δέντρα (δάση). Ο προσανατολισμός θεωρείται από τη ρίζα προς τα φύλλα.

Ένας κατευθυνόμενος γράφος χωρίς κύκλους δεν είναι πάντα δέντρο. Ένας τέτοιος **κατευθυνόμενος ακυκλικός γράφος** (*DAG=Directed Acyclic Graph*) είναι χρήσιμος στην κωδικοποίηση της συντακτικής δομής αριθμητικών εκφράσεων, στην

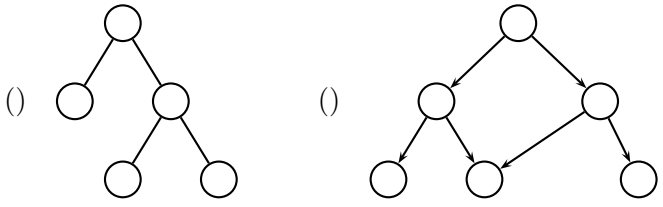
αναπαράσταση μερικών διατάξεων, κ.α. Ένα δέντρο και ένας κατευθυνόμενος ακυκλικός γράφος φαίνονται στο σχήμα 3.8.

Ορισμός 3.43. Έστω ένα δέντρο με ρίζα. Αν x και y είναι κορυφές τέτοιες ώστε η x να βρίσκεται στο μονοπάτι που συνδέει τη ρίζα με την y, τότε η x ονομάζεται **πρόγονος** (*ancestor*) της y και η y **απόγονος** (*descendant*) της x. Αν επιπλέον $x \neq y$ τότε η x είναι **γνήσιος πρόγονος** της y και η y **γνήσιος απόγονος** της x. Αν x είναι γνήσιος πρόγονος της y και $\{x, y\}$ είναι ακμή του δέντρου, τότε η x είναι **γονέας** (*parent*)της y και η y **παιδί** (*child*) της x. Οι κορυφές με τον ίδιο γονέα λέγονται **αδέλφια** (*siblings*). Οι κορυφές που δεν έχουν απογόνους ονομάζονται **τερματικές ή φύλλα** (*terminals, leaves*). Οι κορυφές που δεν είναι τερματικές, λέγονται **εσωτερικές ή μη τερματικές ή κορυφές κλάδων** (*internals, non-terminals, branch nodes*).

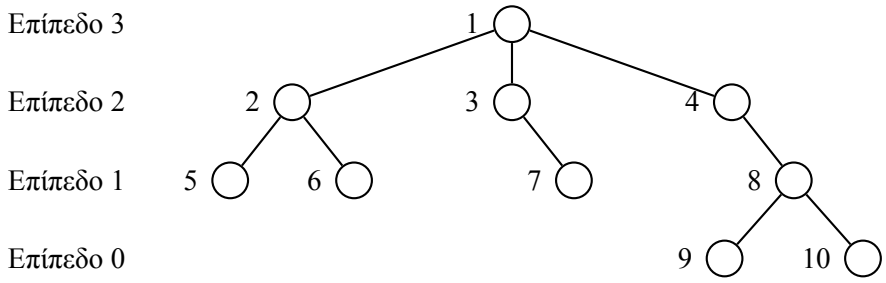
Παράδειγμα 3.44. Στο σχήμα 3.9 έχουμε:

- Η κορυφή 1 είναι η ρίζα του δέντρου.
- Η κορυφή 4 είναι πρόγονος της κορυφής 10.
- Η κορυφή 9 είναι απόγονος της κορυφής 4.
- Η κορυφή 3 είναι γονέας της κορυφής 7.
- Οι κορυφές 5 και 6 είναι αδέλφια με γονέα την κορυφή 2.
- Οι κορυφές 2, 3, 4, 8 είναι εσωτερικές κορυφές.
- Οι κορυφές 5, 6, 7, 9, 10 είναι φύλλα.

Ορισμός 3.45. **Ύψος** (*height*) ενός κόμβου είναι η μέγιστη απόστασή του από απόγονό του. Ύψος δέντρου είναι το ύψος της ρίζας. **Βάθος** (*depth*) ενός κόμβου είναι η απόστασή του από τη ρίζα. **Επίπεδο** (*level*) κόμβου είναι το ύψος του δέντρου πλην το βάθος του κόμβου.



Σχήμα 3.8: (α) Δένδρο, (β) Κατευθυνόμενος ακυκλικός γράφος



Σχήμα 3.9: Δένδρο με αριθμημένες κορυφές

Ορισμός 3.46. **Δυαδικό δέντρο** (*binary tree*) είναι ένα δέντρο με ρίζα στο οποίο κάθε κορυφή έχει το πολύ δύο παιδιά (σχήμα 3.10). Ισοδύναμα, δυαδικό δέντρο είναι ένα πεπερασμένο σύνολο κορυφών που είναι ή κενό, ή αποτελείται από τη ρίζα και δύο ξένα μεταξύ τους δυαδικά δέντρα που ονομάζονται το δεξί και το αριστερό υποδέντρο.

Σε ένα δυαδικό δέντρο στο οποίο κάθε κορυφή του είναι είτε τερματική είτε έχει ακριβώς δύο παιδιά, δεν υπάρχουν **εκφυλισμένοι εσωτερικοί κόμβοι** (*no degenerate branch nodes*) (σχήμα 3.10).

Ένα δυαδικό δέντρο ύψους k το οποίο έχει 2^k φύλλα, όλα στο επίπεδο 0, ονομάζεται εντελώς πλήρες δυαδικό δέντρο (fully complete binary). Πλήρες δ.δ.(ή καμιά φορά σχεδόν πλήρες) ονομάζουμε ένα ε.π.δ.δ. που όμως του λείπουν μερικά ακροδεξιά φύλλα. (σχήμα 3.10).

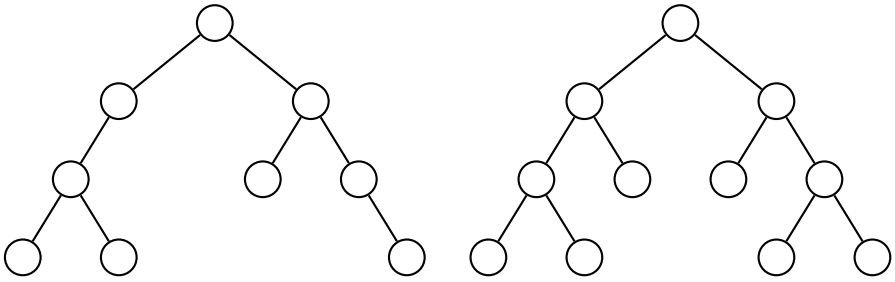
Μια κωδικοποίηση ενός δυαδικού δέντρου στον υπολογιστή μπορεί να γίνει πολύ εύκολα ως εξής: χρησιμοποιούμε ένα μονοδιάστατο πίνακα A και στη θέση $A[1]$ βάζουμε τη ρίζα του δέντρου. Τις θέσεις $A[2], A[3]$ καταλαμβάνουν (αν υπάρχουν) τα δύο παιδιά της ρίζας. Γενικά στις θέσεις $A[2 * k], A[2 * k + 1]$ βρίσκονται τα παιδιά της κορυφής k. Συνεπώς ο γονέας της κορυφής n βρίσκεται στη θέση $A[n \ \ div \ \ 2]$. Σημειώνουμε ότι η αναπαράσταση αυτή χρησιμοποιείται συνήθως όταν το δέντρο είναι (σχεδόν) πλήρες.

3.3 Ελάχιστο συνδετικό δέντρο (MST)

Στο πρόβλημα αυτό έχουμε ένα συνεκτικό γράφο $G(V, E)$ και ένα πίνακα κόστους $c_{|V|\times|V|}$ έτσι ώστε για κάθε ακμή $(u, v) \in E$ να υπάρχει το αντίστοιχο (βάρος) κόστος $c[v, u] > 0$. Ζητάμε ένα δέντρο $T(V, E')$ για το οποίο ισχύει:

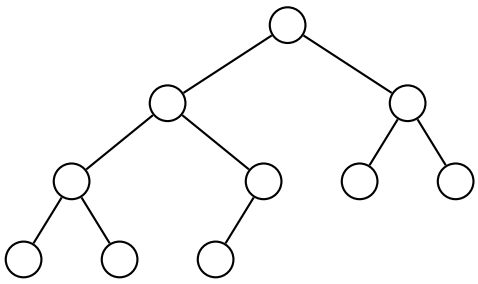
$$\begin{cases} E' & \subseteq E \\ \& & \sum_{(u,v)\in E'} c[u, v] = MINIMUM \end{cases}$$

δηλαδή ένα δέντρο-υπογράφο που συνδέει όλες τις κορυφές του V και το συνολικό κόστος των ακμών του είναι το ελάχιστο δυνατό. Το δέντρο αυτό λέγεται ελαχίστου

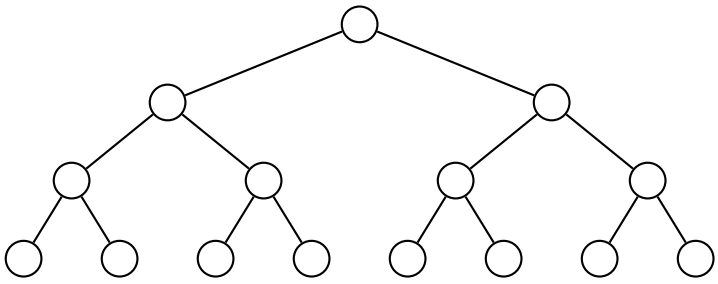


Δυαδικό δένδρο (Binary Tree)

Χωρίς εκφυλισμένους κόμβους



Πλήρες δυαδικό δένδρο
(Complete Binary Tree)

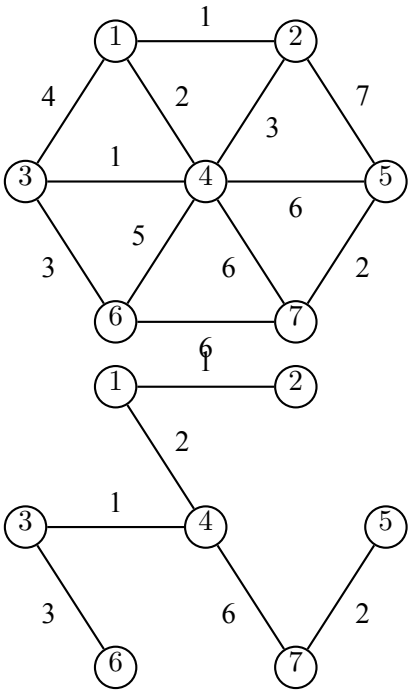


Εντελώς πλήρες δυαδικό δένδρο
(Fully Complete Binary Tree)

Σχήμα 3.10: Δυαδικά δένδρα

κόστους συνδετικό δέντρο του γράφου G (σχήμα 3.11).

Τα ελάχιστα συνδετικά δέντρα βρίσκουν εφαρμογή στο σχεδιασμό τηλεπικοινωνιακών δικτύων. Οι κόμβοι του γράφου αναπαριστούν πόλεις και οι ακμές αναπαριστούν πιθανούς τηλεπικοινωνιακούς συνδέσμους μεταξύ των πόλεων. Το κόστος κάθε ακμής είναι το κόστος κατασκευής του συγκεκριμένου συνδέσμου για το τελικό δίκτυο. Το ελάχιστο συνδετικό δέντρο αναπαριστά ένα τηλεπικοινωνιακό δίκτυο που ενώνει όλες τις πόλεις και έχει το ελάχιστο κόστος κατασκευής. Η άπληστη μέθοδος αρχίζει με την κενή λύση και κατασκευάζει το δέντρο ακμή-



Σχήμα 3.11: Ο γράφος G και ένα ελαχίστου κόστους συνδετικό δένδρο αυτού

ακμή (*edge-by-edge*) ακολουθώντας είτε το κριτήριο του *Prim* είτε το κριτήριο του *Kruskal*, είτε άλλα κριτήρια.

Κριτήριο του Prim: Ξεκινώντας από έναν αρχικό κόμβο (π.χ. τον 1) διαλέγουμε κάθε φορά την ακμή που έχει το ελάχιστο κόστος έτσι ώστε ο νέος υπογράφος να παραμένει δέντρο. Η υλοποίηση που χρησιμοποιεί αυτό το κριτήριο φαίνεται στον αλγόριθμο 3.1 ενώ στο σχήμα 3.12 φαίνεται η ακολουθία των ακμών που προστίθενται στο ελάχιστο συνδετικό δέντρο μέχρι αυτό να πάρει την τελική μορφή του.

Κριτήριο του Kruskal: Διαλέγουμε κάθε φορά την ακμή ελαχίστου κόστους έτσι ώστε ο νέος υπογράφος να μην έχει κύκλους. Σχηματίζουμε έτσι ένα δάσος το οποίο τελικά γίνεται δέντρο. Η υλοποίηση που χρησιμοποιεί αυτό το κριτήριο φαίνεται στον αλγόριθμο 3.2 ενώ στο σχήμα 3.13 φαίνεται η ακολουθία των ακμών που προστίθενται στο ελάχιστο συνδετικό δέντρο μέχρι αυτό να πάρει την τελική μορφή του.

Αν $n = |V|$ και $e = |E|$ τότε ο αλγόριθμος 3.1 (με το κριτήριο του Prim) έχει πολυπλοκότητα $O(n^2)$ ενώ ο αλγόριθμος 3.2 (με το κριτήριο του Kruskal) έχει πολυπλοκότητα $O(e \log e)$. Για συνεκτικούς γράφους ισχύει:

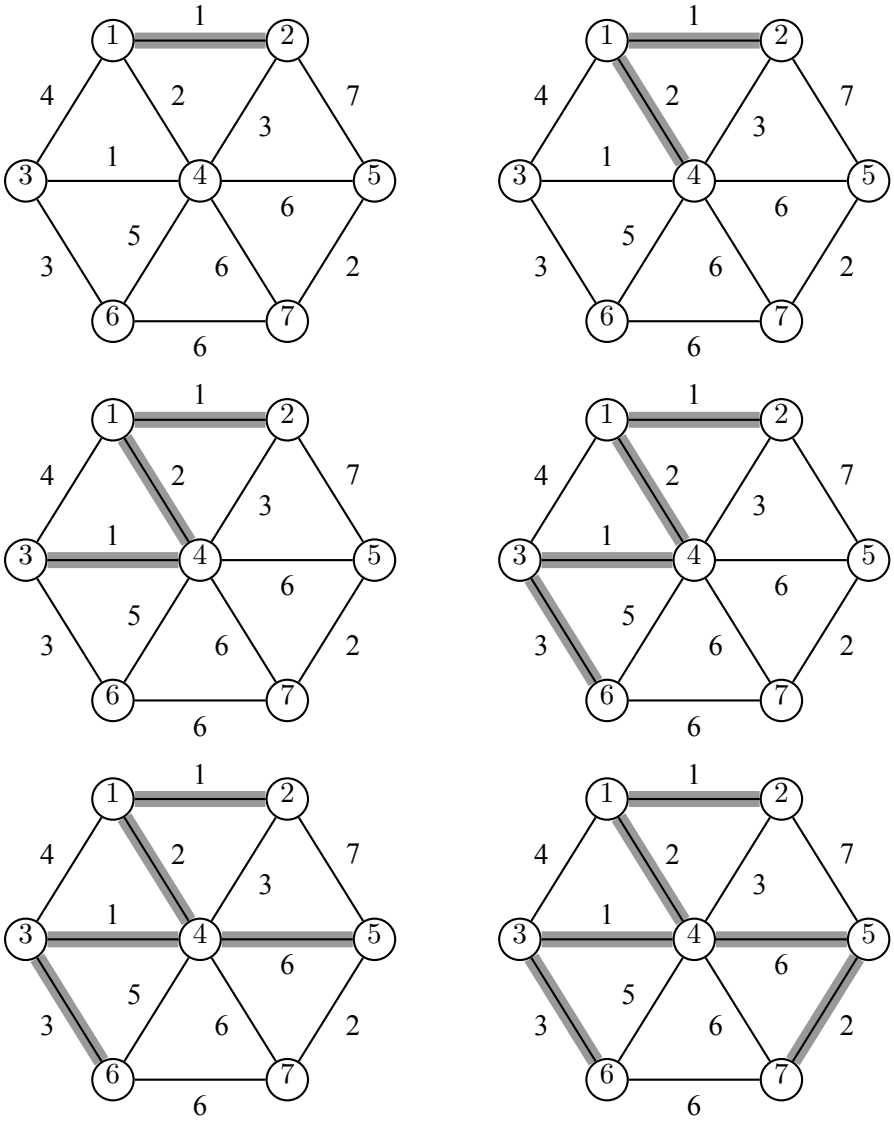
$$n - 1 \leq e \leq \frac{n(n-1)}{2}$$

Αλγόριθμος 3.1 Απλη μέθοδος εύρεσης ελαχίστου κόστους συνδετικού δέντρου (minimum cost spanning tree) με το κριτήριο του Prim

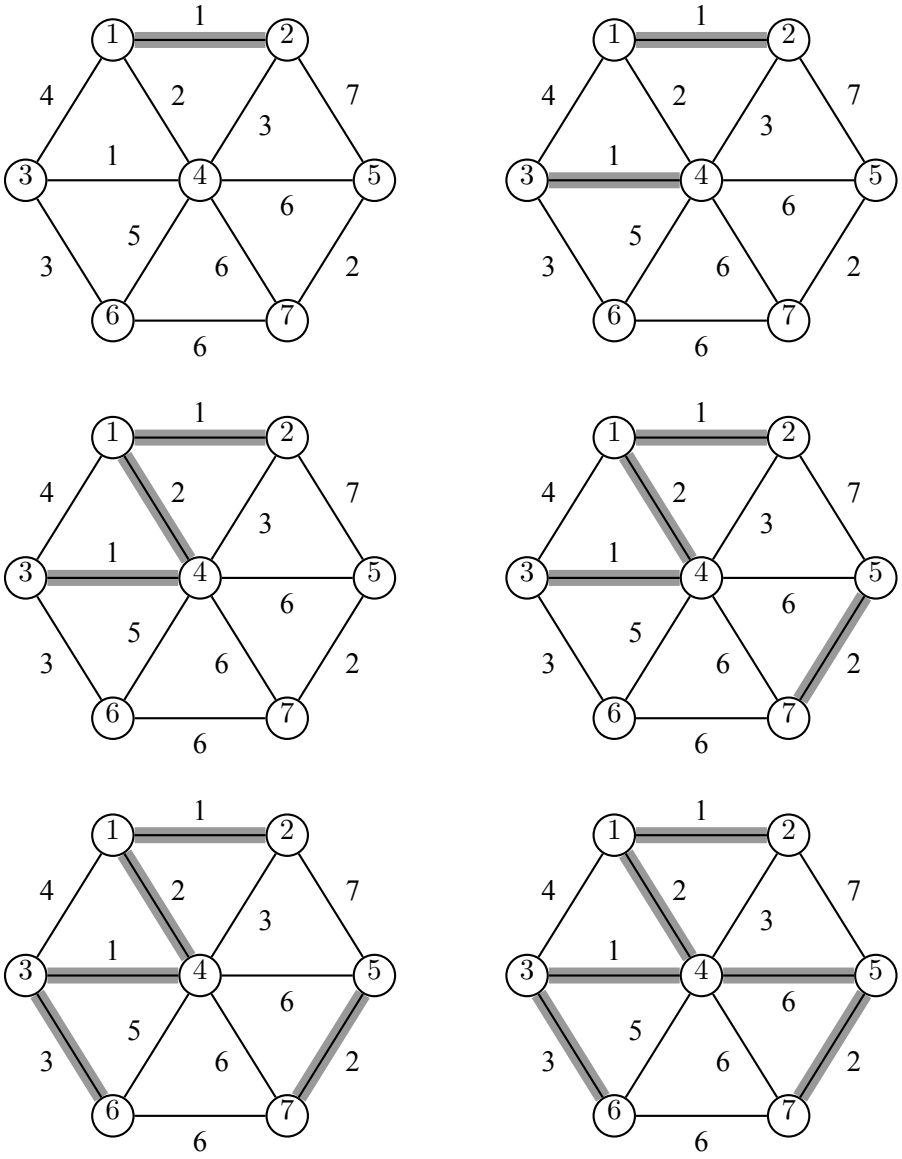
```
procedure Prim (Input: graph G(V,E) with costs on edges;
                       Output: MST with total cost);
  var DistFromTree: array[1..n] of real;
      Edge: array[1..n] of edges; i, j, k, l: integer;
  // Edge[i]: the edge of minimum cost connecting vertex i with current tree
  // DistFromTree[i]: cost of Edge[i]

begin
  DistFromTree[1] := 0
  for k := 2 to n do
      DistFromTree[k] := cost[1,k] ; Edge[k] := (1,k) // for (1,k) ∉ E, cost[1,k] = ∞

  T := ∅
  for i := 2 to n do
  begin
    select j not yet in T, such that DistFromTree[j] is minimum;
    T := T∪ Edge[j]; // edge connecting j is added to the MST
    DistFromTree[j] := 0;
    for k := 2 to n not yet in T do begin
        if cost[j,k] < DistFromTree[k] then
          begin DistFromTree[k] := cost[j,k]; Edge[k] := (j,k) end
    end
  end
end
```

Αλγόριθμος 3.2 Απλη μέθοδος εύρεσης ελαχίστου κόστους συνδετικού δέντρου (minimum cost spanning tree) με το κριτήριο του Kruskal

```
procedure Kruskal (...); // assume edges are ordered by non−decreasing weight
begin
  forest:=empty;
  while |forest| < n − 1 do
  begin
    select Min_Cost edge (u,w) and delete it from E;
    if (u,w) does not create a cycle in forest then
      add it to forest
    else discard it
  end
end
```



Σχήμα 3.12: Εφαρμογή του κριτηρίου του Prim για την εύρεση ελαχίστου κόστους συνδετικού δένδρου



Σχήμα 3.13: Εφαρμογή του κριτηρίου του Kruskal για την εύρεση ελαχίστου κόστους συνδετικού δένδρου

Βλέπουμε λοιπόν ότι ο αλγόριθμος του *Kruskal* έχει καλύτερη απόδοση στους αραιούς (*sparse*) γράφους όπου η πολυπλοκότητά του είναι $O(n \log n)$, σε αντίθεση με τον αλγόριθμο του *Prim* που έχει πολυπλοκότητα $O(n^2)$.

3.4 Το πρόβλημα των συντομότερων μονοπατιών, single source shortest paths problem

Μας δίνεται ένας κατευθυνόμενος γράφος με (θετικά) βάρη και ένας κόμβος του σαν πηγή (*source*). Ζητάμε να βρούμε τα συντομότερα μονοπάτια από την πηγή προς όλες τις άλλες κορυφές του γράφου. Το πρόβλημα της εύρεσης του συντομότερου μονοπατιού από την πηγή προς ένα συγκεκριμένο κόμβο του γράφου είναι εξίσου δύσκολο.

Ο αλγόριθμος που θα περιγράψουμε στηρίζεται στη μέθοδο που αναπτύχθηκε από τον *Dijkstra* (1959). Αριθμούμε τις κορυφές του γράφου και σαν πηγή λαμβάνουμε την κορυφή 1. Έστω V το σύνολο των κορυφών του γράφου και S το σύνολο των κορυφών για τις οποίες το συντομότερο μονοπάτι από την πηγή είναι ήδη γνωστό. Χρησιμοποιούμε τους πίνακες C, D, P. Με $C[i, j]$ συμβολίζουμε το κόστος μετάβασης από την κορυφή i στην κορυφή j, δηλαδή το κόστος μετάβασης της ακμής $i \to j$. Αν η ακμή $i \to j$ δεν υπάρχει, τότε $C[i, j] = \infty$, δηλαδή μια τιμή μεγαλύτερη από οποιοδήποτε πραγματικό κόστος. Το $D[v]$ περιέχει σε κάθε βήμα το κόστος του τρέχοντος συντομότερου από την πηγή στον κόμβο v, ενώ το $P[v]$ περιέχει την αμέσως προηγούμενη κορυφή από την v στο συντομότερο μονοπάτι. Ο αλγόριθμος ξεκινάει με $S = 1$ και προχωράει σε βήματα κατά τα οποία επιλέγει μια κορυφή w έτσι ώστε το $D[w]$ να είναι το ελάχιστο. Στη συνέχεια, αφαιρεί την κορυφή w από το V και την προσθέτει στο S. Έπειτα επαναϋπολογίζει τον πίνακα D για κάθε κορυφή v που ανήκει στο $V - S$ ως εξής:

$$D[v] := min(D[v], D[w] + C[w, v])$$

Αν ισχύει ότι $D[v] > D[w] + C[w, v]$ τότε θέτει $P[v] = w$. Ο αλγόριθμος σταματά όταν $V - S = \emptyset$. Η υλοποίηση της μεθόδου του *Dijkstra* φαίνεται στον αλγόριθμο 3.3.

Από τον αλγόριθμο 3.3 φαίνεται ότι αρχικά το σύνολο $V - S$ έχει $n - 1$ στοιχεία άρα για να βρεθεί το ελάχιστο θα πρέπει να γίνουν $n-2$ συγκρίσεις. Ακολούθως το $V - S$ θα έχει $n - 2$ στοιχεία άρα θα χρειάζονται $n - 3$ συγκρίσεις κ.ο.κ. Συνολικά λοιπόν οι συγκρίσεις είναι:

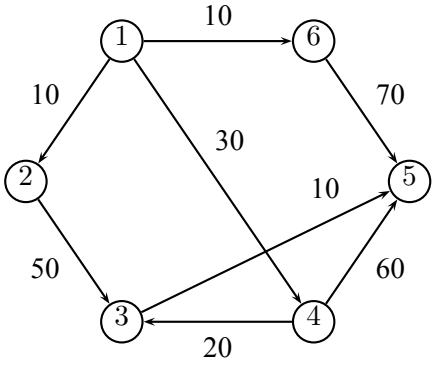
$$\sum_{i=1}^{n-2} i = \frac{(n-1)(n-2)}{2} = O(n^2)$$

Παράδειγμα 3.47. Έστω ότι μας δίνεται ο γράφος που φαίνεται στο σχήμα 3.14. Η εκτέλεση του αλγορίθμου φαίνεται στον πίνακα 3.1. Για να τυπώσουμε το συντομότερο μονοπάτι από ένα κόμβο σε ένα άλλο ανιχνεύουμε τους προκατόχους του

Αλγόριθμος 3.3 Εύρεση συντομότερων μονοπατιών (single source shortest paths) με τη μεθόδο του Dijkstra

```
procedure Dijkstra;
begin (*Αρχικοποίηση *)
  for i:=2 to n do begin D[i]:=cost[1,i]; P[i]:=1 end;
  S := {1};

  for i:=2 to n − 1 do
  begin
    Select w from V − S such that D[w] is minimum;
    S := S + {w};
    for all v in V − S do if D[v] > D[w] + C[w,v] then
        begin P[v] := w; D[v] := D[w] + C[w,v] end
  end
end
```

τελευταίου κόμβου χρησιμοποιώντας τον πίνακα P. Αν για παράδειγμα θέλουμε το συντομότερο μονοπάτι από τον κόμβο 1 στον κόμβο 5 βλέπουμε ότι $P[5] = 3$, δηλαδή ο 3 είναι ο προκάτοχος του 5, $P[3] = 4$ δηλαδή ο 4 είναι προκάτοχος του 3. 'Αρα το συντομότερο μονοπάτι από τον κόμβο 1 στον 5 είναι το $1 \to 4 \to 3 \to 5$.



Σχήμα 3.14: Κατευθυνόμενος γράφος με βάρη

3.5 Διάσχιση γράφων

3.5.1 Γενικά

Οι τεχνικές διάσχισης γράφων μας βοηθούν στο να επισκεπτόμαστε συστηματικά τους κόμβους ενός γράφου G(V,E) έτσι ώστε να δίνουμε γρήγορα απαντήσεις σε προβλήματα όπως τα παρακάτω (*G.A.P., Graph Accessibility Problem*):

- Στο γράφο G υπάρχει μονοπάτι από τον κόμβο v στον κόμβο u;
- Ο γράφος G είναι ακυκλικός;
- Ποιες είναι οι συνεκτικές συνιστώσες (*strong components*) του γράφου G; Δηλαδή ζητάμε να βρεθούν όλα τα υποσύνολα του συνόλου V που είναι τέτοια ώστε όλοι οι κόμβοι που ανήκουν σε αυτά να είναι μεταξύ τους συνδεδεμένοι.
- Ποια είναι τα σημεία σύνδεσης (*articulation points*) του γράφου G; Δηλαδή ζητάμε όλους εκείνους τους κόμβους του γράφου, που αν αφαιρεθούν μαζί με τις προσπίπτουσες ακμές τους, χωρίζουν το γράφο σε δύο ή περισσότερες συνεκτικές συνιστώσες.

3.5.2 Αναζήτηση κατά πλάτος (Breadth First Search)

Στην αναζήτηση κατά πλάτος από κάθε κόμβο v που επισκεπτόμαστε, αναζητούμε κόμβους όσο το δυνατόν κατά πλάτος, δηλαδή αμέσως μετά την επίσκεψη του κόμβου v επισκεπτόμαστε όλους τους γειτονικούς του κόμβους. Έτσι λοιπόν, κατά τη διάρκεια της διαδικασίας αυτής μπορούμε να ξεχωρίσουμε δύο κατηγορίες κόμβων:

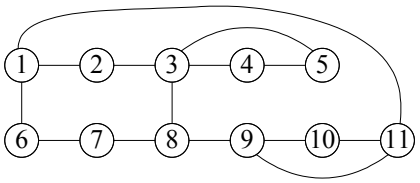
- κόμβους που έχουμε επισκεφθεί (*visited nodes*)
- κόμβους που έχουμε επισκεφθεί αυτούς και όλους τους γειτονικούς τους (*explored nodes*)

Μετά το τέλος της διαδικασίας της αναζήτησης κατά πλάτος στο γράφο G, προκύπτει το συνδετικό δέντρο με προτεραιότητα πλάτους (*breadth first spanning tree*) του γράφου G. Η υλοποίηση της διαδικασίας φαίνεται στον αλγόριθμο 3.4.

Παράδειγμα 3.48. Έστω ο γράφος που φαίνεται στο σχήμα 3.15. Αν καλέσουμε τη διαδικασία **bfs** με **bfs**(1) έχουμε τα βήματα που φαίνονται στο πίνακα 3.2. Το συνδετικό δέντρο με προτεραιότητα πλάτους, είναι αυτό που φαίνεται στο σχήμα 3.16.

			D					P				
Βήμα	S	w	2	3	4	5	6	2	3	4	5	6
-	{1}	-	10	∞	30	∞	10	1	1	1	1	1
2	{1,2}	2		60	30	∞	10		2			
3	{1,2,6}	6		60	30	80					6	
4	{1,2,6,4}	4		50		80			4			
5	{1,2,6,4,3}	3				60					3	
6	{1,2,6,4,3,5}	5										

Πίνακας 3.1: Εφαρμογή της μεθόδου του Dijkstra για το γράφο G

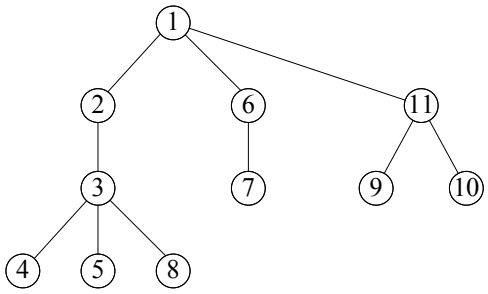


Σχήμα 3.15: Παράδειγμα γράφου στον οποίο θα γίνει αναζήτηση

Αλγόριθμος 3.4 Αναζήτηση κατά πλάτος

```
procedure bfs(v:vertex);
begin
  initialize queue with v; visited[v]:=true;
  repeat dequeue(u);
    for all vertices w adjacent to u do
      if not visited[w] then
        begin visited[w] := true; enqueue(w) end
  until queue is empty
end
```

Οποιαδήποτε υλοποίηση και να χρησιμοποιηθεί, ο αλγόριθμος 3.4 χρειάζεται τον επιπλέον χώρο μιας ουράς μήκους n και ένα πίνακα επίσης μήκους n. Συνεπώς ο χώρος που χρειάζεται είναι της τάξης $O(n)$. Αν ο γράφος υλοποιηθεί με πίνακα γειτονίας, ο χρόνος που χρειάζεται για να διατρέξουμε τον πίνακα είναι $O(n^2)$, ενώ αν έχουμε λίστες γειτονικών κορυφών ο χρόνος είναι $O(n + e)$.



Σχήμα 3.16: Αναζήτηση κατά πλάτος στον γράφο

Για να βρούμε όλες τις συνεκτικές συνιστώσες του γράφου G διασχίζουμε το γράφο με προτεραιότητα πλάτους (*breadth first traversing*). Η υλοποίηση αυτής της διάσχισης φαίνεται στον αλγόριθμο 3.5. Στον αλγόριθμο αυτό είναι φανερό ότι αν ο γράφος είναι συνεκτικός, τότε με την πρώτη κλήση της bfs θα επισκεφθούμε όλους τους κόμβους του.

visited nodes	Queue
1	1
2	2
6	2-6
11	2-6-11
3	6-11-3
7	11-3-7
9	3-7-9
10	3-7-9-10
4	7-9-10-4
5	7-9-10-4-5
8	7-9-10-4-5-8
-	9-10-4-5-8
-	10-4-5-8
-	4-5-8
-	5-8
-	8
-	Ø

Πίνακας 3.2: Εκτέλεση αναζήτησης κατά πλάτος

3.5.3 Αναζήτηση κατά βάθος (Depth First Search)

Στην αναζήτηση κατά βάθος από κάθε κόμβο v που επισκεπτόμαστε αναζητάμε άλλους κόμβους όσο το δυνατό βαθύτερα. Μετά το τέλος της διαδικασίας προκύπτει το συνδετικό δέντρο με προτεραιότητα βάθους (*depth first spanning tree*). Η υλοποίηση της διαδικασίας φαίνεται στον αλγόριθμο 3.6, ενώ το συνδετικό δέντρο με προτεραιότητα βάθους είναι αυτό που φαίνεται στο σχήμα 3.17.

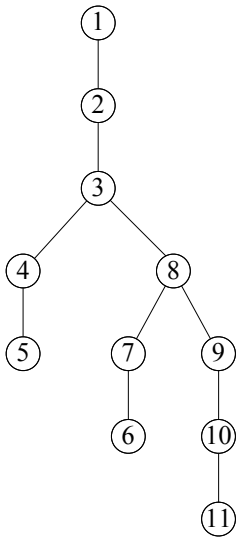
Ο Αλγόριθμος **dft**, που βρίσκει όλες τις συνεκτικές συνιστώσες ενός γράφου, είναι ίδιος με τον **bft**, με την διαφορά ότι αντικαθιστούμε την κλήση της **bfs** με την κλήση της **dfs**.

Αλγόριθμος 3.5 Εύρεση συνεκτικών συνιστωσών

```
procedure bft(G);
begin
  initialize visited with false;
  for i:=1 to n do
    if not visited[i] then bfs(i)
end
```

Αλγόριθμος 3.6 Αναζήτηση κατά βάθος

```
procedure dfs(v:vertex);
begin
  visited[v]:=true;
  for all vertices u adjacent to v do
    if not visited[u] then dfs(u)
end
```



Σχήμα 3.17: Αναζήτηση κατά βάθος στον γράφο

D-Search Ένας ακόμη αλγόριθμος διάσχισης είναι ο **D-search** που περιγράφεται ακριβώς όπως ο **bfs** εκτός από το ότι χρησιμοποιεί μια στοίβα (*stack*) αντί ουράς. Έτσι συμπεριφέρεται σαν ένα κράμα **bfs** και **dfs** (θυμηθείτε σε αυτό το σημείο ότι ο **dfs** μπορεί να υλοποιηθεί και με επαναληπτικό αλγόριθμο, που χρησιμοποιεί στοίβα).

3.6 Μέγιστη ροή – Ελάχιστη τομή (Max Flow – Min Cut)

Ένα πρόβλημα που, όπως θα δούμε, ανάγεται στο πρόβλημα της διάσχισης είναι το *Πρόβλημα Μέγιστης Ροής* (Max Flow Problem): Δοθέντος ενός *δικτύου*, δηλαδή ενός κατευθυνόμενου γράφου με βάρη που αντιπροσωπεύουν *χωρητικότητες* και δύο κόμβων s (source, πηγή), t (target, στόχος), ζητείται να δρομολογηθεί όσο το δυνατόν μεγαλύτερη ροή από τον s στον t έτσι ώστε να ισχύει η αρχή διατήρησης της ροής και να μην παραβιάζονται οι χωρητικότητες των ακμών.

Ας σημειωθεί ότι η *ροή* (όπως και η χωρητικότητα) ορίζεται τυπικά ως μια συνάρ-

τηση πάνω στις ακμές που παίρνει μη αρνητικές πραγματικές τιμές $f : E \to \mathbf{R}_+$ (για την χωρητικότητα συμβολίζουμε με $c : E \to \mathbf{R}_+$).

Η αρχή διατήρησης της ροής απαιτεί σε κάθε κόμβο εκτός των s, t το άθροισμα της f στις εισερχόμενες ακμές να είναι ίδιο με το άθροισμα της f στις εξερχόμενες ακμές (πρβλ. νόμο Kirchhoff). Η *τιμή* της ροής ορίζεται ως η καθαρή ροή που εξέρχεται από τον κόμβο s (ή ισοδύναμα, που εισέρχεται στον κόμβο t), δηλαδή το άθροισμα της f στις εξερχόμενες ακμές του κόμβου s μείον το άθροισμα της f στις εισερχόμενες ακμές του κόμβου s.[1]

Ισχύει το παρακάτω σημαντικό θεώρημα:

Θεώρημα 3.49. *(Max Flow – Min Cut) Η μέγιστη ροή ισούται με την ελάχιστη (ως προς χωρητικότητα) τομή (σύνολο ακμών) που διαχωρίζει τον s από τον t.*

Η απόδειξη του θεωρήματος αυτού, μαζί με τον αλγόριθμο που υπολογίζει τη μέγιστη ροή, δόθηκε από τους Ford και Fulkerson(1962).

Αλγόριθμος 3.7 Αλγόριθμος Ford-Fulkerson

Επαναλαμβάνονται τα παρακάτω βήματα στο δίκτυο για όσο υπάρχει μονοπάτι από τον s στον t (μετά την 1η επανάληψη χρησιμοποιείται το παραμένον δίκτυο):

1. Επιλογή μονοπατιού από τον s στον t. Δρομολόγηση ροής ίσης με την ελάχιστη χωρητικότητα ακμής στο μονοπάτι.
2. Υπολογισμός της *παραμένουσας χωρητικότητας* σε κάθε ακμή του μονοπατιού p (καθώς και στις αντίθετές τους): κατασκευή του *παραμένοντος δικτύου (residual network)*

Ας εξηγήσουμε λίγο περισσότερο το βήμα 2. Έστω f_i η ροή που προστίθεται στην i-οστή επανάληψη (δηλαδή $f_i = \min_{e \in p} c_{i-1}(e)$, όπου c_{i-1} είναι η συνάρτηση χωρητικότητας στο τέλος της $(i-1)$-οστής επανάληψης). Σε κάθε μία από τις ακμές του μονοπατιού p αφαιρούμε χωρητικότητα ίση με f_i και στις αντίθετες ακμές προσθέτουμε χωρητικότητα ίση με f_i (αν δεν υπάρχουν κάποιες από τις αντίθετες ακμές, τις προσθέτουμε). Το δίκτυο που προκύπτει είναι το παραμένον δίκτυο.

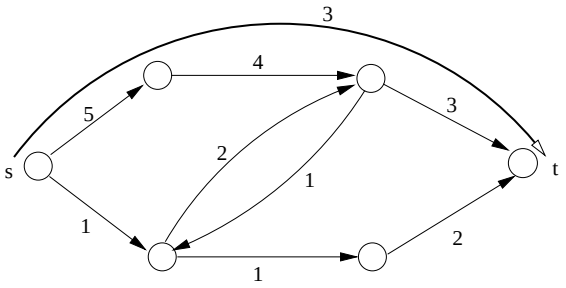
Η τελική ροή είναι το άθροισμα όλων των f_i. Αν σε κάποιο ζεύγος αντίθετων ακμών υπάρχει ροή και στις δύο ακμές, τότε η διαφορά τους αποδίδεται στην ακμή που είχε τη μεγαλύτερη ροή, ενώ η ροή της αντίθετης ακμής μηδενίζεται. Ένα παράδειγμα της εκτέλεσης του αλγορίθμου δίνεται στα Σχήματα 3.18, 3.19, 3.20.

Τα μονοπάτια που χρησιμοποιεί ο αλγόριθμος λέγονται *μονοπάτια επαύξησης (augmenting paths)*.

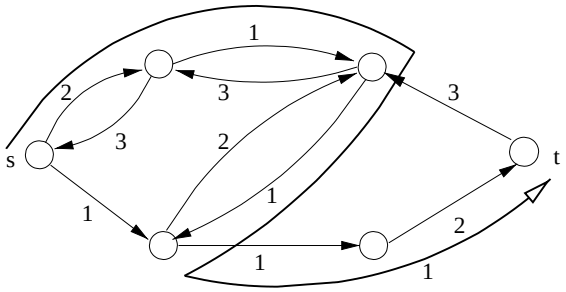
[1] Στη βιβλιογραφία χρησιμοποιούνται και ορισμοί του προβλήματος που επιτρέπουν αρνητική ροή, ή/και απαιτούν διατήρηση της ροής και στους κόμβους s, t (με προσθήκη μιας ακμής (t, s) εάν δεν υπάρχει). Μπορεί να δειχθεί ότι όλοι αυτοί οι ορισμοί είναι ισοδύναμοι με αυτόν που δίνουμε εδώ.

Πολυπλοκότητα του αλγορίθμου: για ακέραιες χωρητικότητες είναι $O(|f^*||E|)$, όπου f^* η τιμή της μέγιστης ροής, καθώς σε κάθε επανάληψη η ροή αυξάνεται τουλάχιστον κατά 1 μονάδα, και η εύρεση μονοπατιού από τον s στον t γίνεται σε χρόνο $(||)$. Επομένως, ο αλγόριθμος αυτός δεν είναι πολυωνυμικού χρόνου στη χειρότερη περίπτωση (γιατί;).

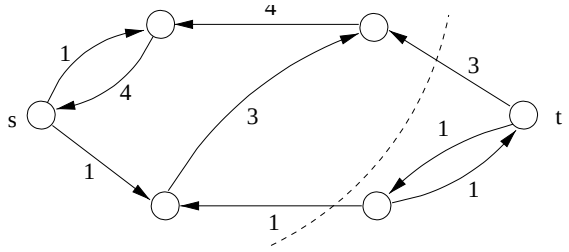
Εάν όμως επιλέγεται κάθε φορά το συντομότερο μονοπάτι, τότε ο αριθμός των επαναλήψεων γίνεται πολυωνυμικός, όπως απέδειξαν οι Edmonds-Karp (1972) - ο αντίστοιχος αλγόριθμος έχει πολυπλοκότητα $O(|V||E|^2)$. Ακόμη ταχύτερος είναι ο αλγόριθμος του Goldberg (1986-87) με πολυπλοκότητα $O(|V|^2|E|)$ και $O(|V|^3)$ (μέθοδος preflow-push).



Σχήμα 3.18: Το αρχικό δίκτυο και η πρώτη ροή, μεγέθους 3, από τον κόμβο s στον t.



Σχήμα 3.19: Το παραμένον δίκτυο και η δεύτερη ροή, μεγέθους 1, από τον κόμβο s στον t.



Σχήμα 3.20: Το παραμένον δίκτυο. Δεν υπάρχει μονοπάτι από τον κόμβο s στον t. Η συνολική ροή (δεν απεικονίζεται) προκύπτει από τη διαφορά της χωρητικότητας σε σχέση με το αρχικό δίκτυο. Η ελάχιστη τομή (με διακεκομμένη γραμμή) έχει συνολική χωρητικότητα 4, ίση με τη μέγιστη ροή.

Πρόβλημα ταιριάσματος (Matching) Αξίζει να σημειωθεί ότι ένα άλλο γνωστό πρόβλημα, το πρόβλημα Maximum Matching (Μέγιστο Ταίριασμα), και επομένως και η ειδική του περίπτωση Perfect Matching (Τέλειο Ταίριασμα), λύνονται, όταν η είσοδος είναι διμερής γράφος, με αναγωγή στο πρόβλημα Maximum Flow.

3.7 Πολυπλοκότητα γραφοθεωρητικών προβλημάτων

Υπάρχουν πολυωνυμικοί (ντετερμινιστικοί) αλγόριθμοι για τα κάτωθι προβλήματα: κύκλος Euler, reachability και διάσχιση γράφων, συνεκτικές συνιστώσες, συντομότερα μονοπάτια, ελάχιστο συνδετικό δένδρο (minimum spanning tree), μέγιστη ροή, ταίριασμα (matching), χρωματισμός ακμών σε διμερείς γράφους (bipartite graph edge coloring).

Αντιθέτως, για τα κάτωθι προβλήματα (που είναι όλα ισοδύναμα από άποψη πολυπλοκότητας, NP-πλήρη) είναι γνωστοί μόνο μη ντετερμινιστικοί αλγόριθμοι πολυωνυμικού χρόνου και η ακριβής πολυπλοκότητά τους είναι ανοιχτό πρόβλημα: κάλυψη με κορυφές (vertex cover), κλίκα (clique), κύκλος Hamilton (Hamilton circuit), πρόβλημα πλανόδιου πωλητή (traveling salesperson problem), 3-χρωματισμός (3-colorability), ισομορφισμός υπογράφων (subgraph isomorphism), 3-διάστατο ταίριασμα (3-dimensional matching, 3DM).

Ασκήσεις

1. Δείξτε ότι ένας ακυκλικός συνεκτικός γράφος με n κόμβους έχει $n-1$ ακμές.
2. Δείξτε ότι ένας συνεκτικός γράφος με n κόμβους και $n-1$ ακμές είναι δέντρο.
3. Σε ένα δένδρο ένας κόμβος λέγεται $1/k$ separator αν μετά την αφαίρεσή του, οι συνεκτικές συνιστώσες που απομένουν έχουν μέγεθος το πολύ n/k, όπου n ο αριθμός των κόμβων του δένδρου.

(α□) Δείξτε ότι σε κάθε δένδρο υπάρχει $1/2$ separator.

(β□) Δείξτε ότι αν σε ένα δένδρο υπάρχει $1/k$ separator (υποθέτοντας $k < n$) τότε υπάρχει κόμβος με βαθμό τουλάχιστον k. Εξετάστε αν ισχύει και το αντίστροφο.

(γ□) Βρείτε αλγόριθμο που αποφαίνεται αν ένα δένδρο έχει $1/(x+3)$ separator, όπου x το τελευταίο ψηφίο του αριθμού μητρώου σας. Αποδείξτε την ορθότητα του αλγορίθμου σας και υπολογίστε την πολυπλοκότητά του.

4. Σε έναν κατευθυνόμενο γράφο, ένας κόμβος με indegree (πλήθος εισερχομένων ακμών) μηδέν λέγεται πηγή. Θεωρήστε ότι ο γράφος δίνεται σε μορφή πίνακα γειτνίασης.

   (α□) Αποδείξτε ότι σε κάθε ακυκλικό κατευθυνόμενο γράφο υπάρχει τουλάχιστον μία πηγή.

   (β□) Δείξτε ότι ένας κατευθυνόμενος γράφος με n κόμβους είναι ακυκλικός αν και μόνο αν μπορούμε να τοποθετήσουμε ετικέτες $1, 2, \ldots, n$ στους κόμβους ώστε όλες οι ακμές να κατευθύνονται από κόμβο με μικρότερη ετικέτα σε κόμβο με μεγαλύτερη ετικέτα.

   (γ□) Περιγράψτε αποδοτικό αλγόριθμο που να αποφαίνεται αν ένας κατευθυνόμενος γράφος είναι ακυκλικός.

5. (α□) Στην όχθη ενός ποταμού βρίσκονται ένας λύκος, ένα πρόβατο κι ένα καφάσι με μαρούλια. Υπάρχει μόνο μια βάρκα, η οποία εκτός από το βαρκάρη μπορεί να μεταφέρει μόνο ένα από τα προηγούμενα κάθε φορά. Όταν ο βαρκάρης είναι παρών τότε επικρατεί ασφάλεια στο σύστημα. Παρόλα αυτά όταν απουσιάζει, κάποια από τα παραπάνω μπορούν το ένα να φάει το άλλο. Συγκεκριμένα ισχύουν τα εξής:

   - Αν ο λύκος και το πρόβατο μείνουν αφύλακτα στην όχθη όσο ο βαρκάρης μεταφέρει το καφάσι με τα μαρούλια, ο λύκος μπορεί να φάει το πρόβατο.
   - Αν το πρόβατο μείνει αφύλακτο μαζί με τα μαρούλια στην όχθη όσο ο βαρκάρης μεταφέρει το λύκο, το πρόβατο θα φάει τα μαρούλια.

   Βρείτε έναν τρόπο ώστε να καταφέρει ο βαρκάρης να τα μεταφέρει και τα τρία άθικτα στην απέναντι όχθη.

   (β□) Γενικεύοντας, έστω ότι υπάρχουν n αντικείμενα $\{x_1, x_2, \ldots x_n\}$, τα οποία ο βαρκάρης επιθυμεί να περάσει στην απέναντι όχθη. Δίνεται γι' αυτά ένας γράφος ασυμβατοτήτων, του οποίου οι κορυφές είναι τα n αντικείμενα και το x_i συνδέεται με το x_j όταν τα x_i και x_j δεν επιτρέπεται να μείνουν αφύλακτα μαζί στην ίδια όχθη (δηλαδή όταν κάποιο από τα δύο μπορεί να φάει το άλλο).

   Βρείτε ποιά πρέπει να είναι τουλάχιστον η χωρητικότητα της βάρκας ώστε το πρόβλημα να λύνεται όταν ο γράφος ασυμβατοτήτων είναι:

- αλυσίδα (chain) P_n,
- δακτύλιος (ring) C_n,
- αστέρι (star) S_n,
- ο πλήρης γράφος K_n,
- πλέγμα (grid) διαστάσεων $\sqrt{n} \times \sqrt{n}$,
  Πλέγμα (grid) διαστάσεων $m \times n$ είναι ο γράφος $G(V, E)$, με $V = \{(i, j) : i = 1 \ldots m, j = 1 \ldots n\}$ και $E = \{((i, j), (i', j')) : |i - i'| + |j - j'| = 1\}$

Πόσες φορές πρέπει ο βαρκάρης να διασχίσει τον ποταμό σε κάθε μια από τις παραπάνω περιπτώσεις; Δώστε μέθοδο επίλυσης.

(γ□) Περιγράψτε έναν αλγόριθμο για τη γενική περίπτωση του προβλήματος: είσοδος η χωρητικότητα της βάρκας και ο γράφος ασυμβατοτήτων.

6. Ένας οδηγός αποφασίζει να κάνει ένα ταξίδι από την πόλη X μέχρι την πόλη Y με το αυτοκίνητό του, το οποίο έχει αυτονομία κίνησης ως προς τη βενζίνη που μπορεί να αποθηκεύσει, k χιλιόμετρα. Αν ο οδηγός διαθέτει χάρτη στον οποίο αναφέρονται όλα τα βενζινάδικα της διαδρομής με τις αποστάσεις τους και επιπλέον επιθυμεί να πραγματοποιήσει όσο το δυνατό λιγότερες στάσεις, πώς πρέπει να προγραμματίσει τον ανεφοδιασμό καυσίμων; Αποδείξτε την ορθότητα του αλγορίθμου που επινοήσατε.

   Σημείωση: δεν υπάρχει ζεύγος διαδοχικών βενζινάδικων που να απέχουν μεταξύ τους πάνω από k χιλιόμετρα.

7. Κάποιος ισχυρίζεται οτι παρόλο που ο αλγόριθμος Dijkstra δε δουλεύει για κατευθυνόμενους γράφους με αρνητικά βάρη, μπορεί να τροποποιηθεί ώστε να δίνει σωστά αποτελέσματα. Πιο συγκεκριμένα αν v είναι η ακμή με το ελάχιστο βάρος (αρνητικό), προτείνει να προσθέσουμε σε όλες τις ακμές βάρος $|w(v)|$ ώστε να αποκτήσουν όλες οι ακμές θετικό βάρος. Στη συνέχεια προτείνει να εφαρμόσουμε τον αλγόριθμο του Dijkstra ως έχει. Εξηγήστε αν η παραπάνω σκέψη λύνει το single-source πρόβλημα σωστά, δίνοντας απόδειξη ή αντιπαράδειγμα.

Κεφάλαιο 4

Πεπερασμένα αυτόματα και κανονικές παραστάσεις

4.1 Εισαγωγή

Στο κεφάλαιο αυτό θα ασχοληθούμε με τυπικές γλώσσες που μπορούν να περιγράψουν υπολογιστικά προβλήματα και επίσης χρησιμεύουν στον προγραμματισμό. Επίσης θα επιδιώξουμε να ταξινομήσουμε γλώσσες ανάλογα με το είδος του **αυτομάτου** (*αφηρημένης υπολογιστικής συσκευής*) που μπορεί να τις αναγνωρίσει.

Ένα αυτόματο έχει μερικές εσωτερικές καταστάσεις: $q_0, q_1, q_{16}, q_{23}, \ldots$ και μία συνάρτηση δ που καθορίζει την επόμενη κατάσταση του αυτομάτου.

Στις πιο απλές συσκευές, που ονομάζονται **μηχανισμοί**, η συνάρτηση μετάβασης (*transition function*) δ έχει ως όρισμα μία κατάσταση q_i και ως τιμή μια άλλη κατάσταση q_j. Δεν υπάρχει δηλαδή είσοδος και έξοδος. Υπολογιστική ακολουθία σε αυτήν την περίπτωση είναι μια ακολουθία από καταστάσεις:

$$q_i \to q_j \to q_k \to q_l \cdots$$

Αν επιπλέον η συσκευή διαβάζει μια συμβολοσειρά εισόδου, σύμβολο προς σύμβολο, από αριστερά προς τα δεξιά και ανάλογα αλλάζει καταστάσεις, τότε ονομάζεται **αναγνωριστής πεπερασμένων καταστάσεων** (*FSA: finite state acceptor*). Η συνάρτηση μετάβασης είναι της μορφής

$$\delta\colon (q_i, a) \to q_j,$$

όπου $\alpha \in \Sigma$ και το Σ το αλφάβητο εισόδου. Στο τέλος το FSA αποδέχεται ή απορρίπτει την είσοδό του.

Αν προσθέσουμε έξοδο σε ένα *FSA*, δηλαδή $\delta\colon (q_i, a) \to (q_j, b)$, όπου $b \in \Delta$ και Δ το αλφάβητο εξόδου, τότε έχουμε τη λεγόμενη μηχανή πεπερασμένων καταστάσεων (*FSM: finite state machine*). Οι FSA και FSM εμφανίζονται συχνά ως χρήσιμα εργαλεία στο λογικό, ψηφιακό σχεδιασμό. Προσθέτοντας στο αυτόματό μας

μνήμη υπό μορφή στοίβας (*stack*) έχουμε πολύ περισσότερες δυνατότητες: το αυτόματο ονομάζεται τότε **αυτόματο στοίβας** (*PDA: push-down automaton*). Αν αντί στοίβας έχουμε απεριόριστη δυνατότητα μνήμης υπό μορφή ταινίας, τότε έχουμε τη γνωστή **μηχανή Turing** (*TM*). **Γραμμικά περιορισμένο αυτόματο** (*LBA: linearly bounded automaton*) είναι μια μηχανή Turing που όμως μπορεί να χρησιμοποιήσει ταινία που το μήκος της είναι μια γραμμική συνάρτηση του μήκους της εισόδου.

Τα διαφορετικά αυτά αυτόματα αναγνωρίζουν κλάσεις γλωσσών που όμως μπορούν να περιγραφούν και με άλλους τρόπους π.χ. με αλγεβρικό τρόπο, ή με **τυπικές γραμματικές**.

Οι γλώσσες προγραμματισμού είναι προφανώς **τυπικές γλώσσες** (*formal languages*) που έχουν αυστηρό συντακτικό ορισμό. Οι τυπικές γλώσσες μπορούν να ταξινομηθούν ανάλογα με τις τυπικές γραμματικές που τις παράγουν. Στη θεωρία τυπικών γλωσσών οι πρωταρχικές έννοιες είναι τα **σύμβολα** (ως αντικείμενα) και η **παράθεση** (ως πράξη).

Ένα **αλφάβητο** Σ είναι ένα πεπερασμένο σύνολο συμβόλων. Μια λέξη ή πρόταση ή συμβολοσειρά ή string είναι μια πεπερασμένου μήκους ακολουθία συμβόλων. Το μήκος του string w συμβολίζεται με $|w|$. Το κενό string συμβολίζεται με ε. Η παράθεση των strings x και y συμβολίζεται με xy. 'Αλλοι χρήσιμοι όροι είναι **πρόθεμα** (*prefix*), **υποσυμβολοσειρά** (*substring*), **υπακολουθία** (*subsequence*), **κατάληξη** (*suffix*), **αντίστροφη** (*reversal*), **παλινδρομική** ή **καρκινική** (*palindrome*).

Ισχύουν $x\varepsilon = \varepsilon x = x$ για όλα τα strings x και $|\varepsilon| = 0$. Το string x^k μπορεί να οριστεί με πρωταρχική αναδρομή:

$$\begin{cases} x^0 = \varepsilon \\ x^{k+1} = x^k x \end{cases}$$

Ορισμός 4.1. Αν Σ είναι ένα αλφάβητο τότε Σ^* είναι το σύνολο όλων των strings από το Σ. Μια γλώσσα L από το Σ δεν είναι παρά κάποιο υποσύνολο του Σ^*.

Παραδείγματα (με αλφάβητο $\Sigma = \{a, b\}$):

- $L_1 = \{w \in \Sigma^* \mid w$ αρχίζει με $a\}$
- $L_2 = \{w \in \Sigma^* \mid w$ περιέχει ζυγό αριθμό από $a\}$
- $L_3 = \{w \in \Sigma^* \mid w$ είναι παλινδρομική $\}$

4.2 Ντετερμινιστικά πεπερασμένα αυτόματα

Θα περιγράψουμε πρώτα τα *ντετερμινιστικά πεπερασμένα αυτόματα* (deterministic finite automata - DFA). Ένα DFA είναι ένα αυτόματο που *αναγνωρίζει* (δηλαδή *αποδέχεται* ή *απορρίπτει*) συμβολοακολουθίες που εμφανίζονται στην ταινία εισόδου του. Η κεφαλή[1] του DFA βρίσκεται αρχικά στο αριστερότερο σημείο της

[1] ώς *κεφαλή* ορίζουμε εκείνο το στοιχείο της ταινίας εισόδου που βρίσκεται την παρούσα στιγμή υπό εξέταση

εισόδου. Μετακινείται δε προς τα δεξιά κατά ένα σύμβολο σε κάθε βήμα μέχρι να σταματήσει στο τέλος της ταινίας εισόδου. Από σύμβαση, η αρχική κατάσταση (initial state) ονομάζεται q_0, κάποιες από τις καταστάσεις αποδέχονται και ονομάζονται τελικές (accepting, final states), και οι υπόλοιπες καταστάσεις απορρίπτουν (rejecting, nonfinal states). Αν το DFA βρίσκεται σε μια κατάσταση που αποδέχεται αφότου η κεφαλή έχει διαβάσει όλη την συμβολοακολουθία είσοδου, λέμε ότι το DFA *αποδέχεται* την είσοδο. Η γλώσσα L, την οποία αναγνωρίζει ένα DFA, είναι το σύνολο των συμβολοακολουθιών που αποδέχεται.

Ορισμός 4.2. Τυπικά ένα DFA είναι μία πεντάδα $M = (Q, \Sigma, \delta, q_0, F)$, όπου:

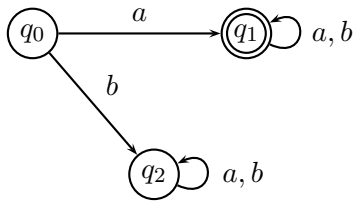
- Q: ένα πεπερασμένο σύνολο από καταστάσεις,
- Σ: ένα αλφάβητο εισόδου ($\Sigma \cap Q = \emptyset$),
- $\delta\colon Q \times \Sigma \to Q$: η συνάρτηση μετάβασης,
- $q_0 \in Q$: η αρχική κατάσταση,
- $F \subseteq Q$: το σύνολο των τελικών καταστάσεων.

Ενα DFA μπορεί επίσης να περιγραφεί (όπως και μια TM) από ένα *πίνακα καταστάσεων* ή από ένα *διάγραμμα καταστάσεων*

Στα παρακάτω παραδείγματα θεωρούμε αλφάβητο $\Sigma = \{a, b\}$.

Παράδειγμα 4.1. $L_1 = \{w \in \Sigma^* \mid w \text{ αρχίζει από } a\}$

Διάγραμμα:

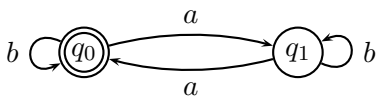


Πίνακας:

	a	b
q_0	q_1	q_2
(q_1)	q_1	q_1
q_2	q_2	q_2

Χρησιμοποιούμε έναν επιπλέον κύκλο για να δείξουμε τις τελικές καταστάσεις.

Παράδειγμα 4.2. $L_2 = \{w \in \Sigma^* \mid w \text{ περιέχει άρτιο πλήθος από } a\}$



	a	b
(q_0)	q_1	q_0
q_1	q_0	q_1

Παράδειγμα 4.3. $L'_3 = \{w \in \Sigma^* \mid w \text{ παλίνδρομη αρτίου μήκους}\}$, δηλαδή $L'_3 = \{ww^R \mid w \in \Sigma^*, w^R = \text{αντίστροφη της } w\}$. Δεν υπάρχει DFA που αποδέχεται την L'_3.

Για να συντομεύσουμε την περιγραφή της συμπεριφοράς του DFA, επεκτείνουμε την συνάρτηση δ ώστε να δέχεται ως όρισμα συμβολοακολουθίες αντί για απλά σύμβολα. Η δ επεκτείνεται λοιπόν ως εξής:

Ορισμός 4.3. $\delta : Q \times \Sigma^* \to Q$ όπου

$$\begin{cases} \delta(q, \varepsilon) = q \\ \delta(q, wa) = \delta(\delta(q, w), a) \end{cases}$$

Ο πιο πάνω ορισμός είναι αναδρομικός, ή, πιο συγκεκριμένα, είναι ορισμός σύμφωνα με το σχήμα της *πρωταρχικής αναδρομής*.

Ορισμός 4.4. Έχουμε:

- Ένα DFA αποδέχεται το string $w \in \Sigma^*$ ανν $\delta(q_0, w) \in F$
- Ένα DFA αποδέχεται τη γλώσσα $L(M) = \{w \mid \delta(q_0, w) \in F\}$
- Η γλώσσα L λέγεται *κανονική* (*regular*) ανν $\exists$ FA $M : L = L(M)$

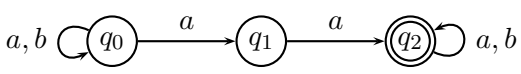
Άσκηση: Δείξτε ότι $\delta(q, uv) = \delta(\delta(q, u), v)$, όπου $u, v \in \Sigma^*$.

Υπάρχουν διάφορες γενικεύσεις και διαφοροποιήσεις ως προς τα DFA:

- NFA: *μη ντετερμινιστικό πεπερασμένο αυτόματο*. Σε κάθε μετάβαση υπάρχει επιλογή της επόμενης κατάστασης από ένα σύνολο πιθανών νομίμων καταστάσεων.
- NFA$_\varepsilon$: *μη ντετερμινιστικό πεπερασμένο αυτόματο με ε-κινήσεις*. Το πεπερασμένο αυτόματο ενδέχεται να αλλάζει την κατάστασή του χωρίς να μετακινείται η κεφαλή στην ταινία εισόδου.

4.3 Μη ντετερμινιστικά πεπερασμένα αυτόματα

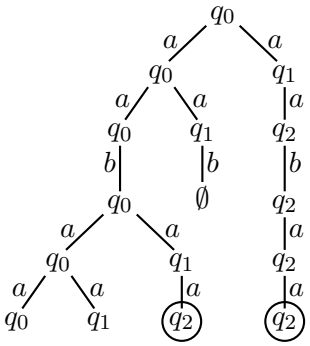
Παράδειγμα 4.4. NFA για $L_4 := \{w \in \Sigma^* \mid w$ περιέχει δύο συνεχόμενα $a\}$



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
(q_2)	$\{q_2\}$	$\{q_2\}$

Ένας υπολογισμός σε ένα NFA δεν είναι απλώς μία γραμμική (νόμιμη) ακολουθία καταστάσεων, αλλά ένα *υπολογιστικό δένδρο* (κάθε κλάδος είναι μία νόμιμη ακολουθία καταστάσεων).

Το δένδρο υπολογισμού για το παραπάνω παράδειγμα για είσοδο $aabaa$:



Η συμβολοσειρά $aabaa$ γίνεται αποδεκτή, επειδή υπάρχει τουλάχιστον ένα νόμιμο μονοπάτι που την αποδέχεται.

Στα μη ντετερμινιστικά αυτόματα, για κάθε είσοδο και κατάσταση, μπορεί να υπάρχει καμμία, μία ή πολλές πιθανές επόμενες καταστάσεις. Αυτό εκφράζεται στον ορισμό ενός NFA από το γεγονός ότι η συνάρτηση μετάβασης δ έχει ως πεδίο τιμών το *δυναμοσύνολο* του Q (Pow(Q)).

Ένα NFA λοιπόν αποτελείται από μια πεντάδα $M = (Q, \Sigma, \delta, q_0, F)$, όπου:

- Q: ένα πεπερασμένο σύνολο από καταστάσεις,
- Σ: ένα πεπερασμένο αλφάβητο εισόδου,
- $\delta : Q \times \Sigma \to \text{Pow}(Q)$: η συνάρτηση μετάβασης
- $q_0 \in Q$: η αρχική κατάσταση και
- $F \subseteq Q$: το σύνολο των τελικών καταστάσεων.

Όπως και πριν, επεκτείνουμε την συνάρτηση μετάβασης δ για να συμπεριλάβουμε στο πεδίου ορισμού της τις συμβολοακολουθίες. Η επεκτεταμένη συνάρτηση μετάβασης $\delta(q, w)$ θα περιέχει το σύνολο όλων των καταστάσεων που μπορούμε να φτάσουμε από το q με είσοδο το w.

Ορισμός 4.5. $\delta : Q \times \Sigma^* \to \text{Pow}(Q)$ όπου

$$\begin{cases} \delta(q, \varepsilon) = \{q\} \\ \delta(q, wa) = \{p \in \delta(r, a) | r \in \delta(q, w)\} = \bigcup\limits_{r \in \delta(q,w)} \delta(r, a) \end{cases}$$

Επεκτείνουμε περαιτέρω την δ, συμπεριλαμβάνοντας στο πεδίο ορισμού το δυναμοσύνολο του Q. Η $\delta(P, w)$ θα περιέχει όλες τις καταστάσεις που μπορούμε να φτάσουμε από οποιαδήποτε κατάσταση στο P με είσοδο w.

Ορισμός 4.6. $\delta : \text{Pow}(Q) \times \Sigma^* \to \text{Pow}(Q)$ όπου

$$\delta(P, w) = \{p \in \delta(q, w) | q \in P\} = \bigcup_{q \in P} \delta(q, w)$$

Παρατήρηση 4.1. Ισχύει $\delta(q, wa) = \delta(\delta(q, w), a)$

Παρατήρηση 4.2. Ισχύει $\delta(P, wa) = \delta(\delta(P, w), a)$

Ορισμός 4.7. Έχουμε:

- Ένα NFA αποδέχεται το string $w \in \Sigma^*$ ανν $\delta(q_0, w) \cap F \neq \emptyset$
- Ένα NFA M αποδέχεται τη γλώσσα $L(M) = \{w | \delta(q_0, w) \cap F \neq \emptyset\}$

Ισοδυναμία DFA και NFA. Όπως φαίνεται από τον ορισμό της δ ενός NFA, ένα DFA είναι μια "υποπερίπτωση" ενός NFA. Παρ' όλα αυτά, τα NFA δεν μας παρέχουν περισσότερες δυνατότητες υπολογισμού από ότι τα DFA. Αυτό αποδεικνύει το παρακάτω θεώρημα:

Θεώρημα 4.5. *(Rabin - Scott)*

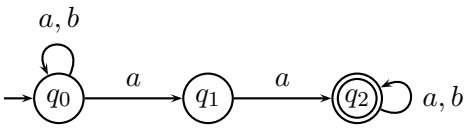
$$\textit{Έστω } M \textit{ ένα NFA. Τότε } \exists \textit{ DFA } M' : L(M) = L(M')$$

Απόδειξη. Ορίζουμε το DFA $M' = (Q', \Sigma', q_0', F', \delta')$ όπου

- $Q' = \text{Pow}(Q)$,
- $\Sigma' = \Sigma$, $q_0' = \{q_0\}$,
- $F' = \{R \in Q' | R \cap F \neq \emptyset\} = \bigcup_{R \cap F \neq \emptyset} (R \in Q')$ και τέλος
- $\delta'(R, a) = \delta(R, a)$, όπου $R \in Q'$.

Μένει να αποδειχθεί ότι $L(M) = L(M')$ με την βοήθεια του ισχυρισμού: $\delta'(q_0', w) = \delta(q_0, w)$. (Αφήνεται ως άσκηση· χρησιμοποιήστε επαγωγή.) □

Παράδειγμα 4.6.

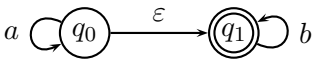


	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\{q_2\}$	$\emptyset$
$(\{q_2\})$	$\{q_2\}$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$(\{q_0, q_2\})$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
$(\{q_1, q_2\})$	$\{q_2\}$	$\{q_2\}$
$(\{q_0, q_1, q_2\})$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

Το DFA φαίνεται στο παράδειγμα 4.9.

Μη ντετερμινιστικά αυτόματα με ε-κινήσεις - NFA$_\varepsilon$

Παράδειγμα 4.7. NFA$_\varepsilon$ για $L_5 := \{a^*b^*\} = \{a^n b^m \mid n, m \in \mathbb{N}\}$



	a	b	ε
q_0	$\{q_0\}$	$\emptyset$	$\{q_1\}$
(q_1)	$\emptyset$	$\{q_1\}$	$\emptyset$

Μπορούμε να γενικεύσουμε το μη ντετερμινιστικό αυτόματο που ορίσαμε ώστε να συμπεριλάβουμε στο πεδίο ορισμού της δ και το ε. Με άλλα λόγια, το αυτόματο μπορεί να αλλάζει κατάσταση χωρίς να διαβάζει κάποια είσοδο, με μια ε-κίνηση. Ένα NFA$_\varepsilon$ λοιπόν είναι μια πεντάδα $M = (Q, \Sigma, \delta, q_0, F)$ όπου

- Q: ένα πεπερασμένο σύνολο από καταστάσεις,
- Σ: ένα πεπερασμένο αλφάβητο εισόδου,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \text{Pow}(Q)$: η συνάρτηση μετάβασης
- $q_0 \in Q$: η αρχική κατάσταση και
- $F \subseteq Q$: το σύνολο των τελικών καταστάσεων.

Για την ανάλυσή μας, είναι χρήσιμο να ορίσουμε το *ε-κλείσιμο* μιας κατάστασης.

Ορισμός 4.8. Ως ε-κλείσιμο: $Q \to \text{Pow}(Q)$ ορίζουμε το

$$\varepsilon\text{-κλείσιμο}(q) = \{p | \text{ τα } p \text{ προσβάσιμα από το } q \text{ μόνο με } \varepsilon\text{-κινήσεις}\}$$

Παρατηρούμε ότι πάντα $q \in \varepsilon$-κλείσιμο(q). Επεκτείνουμε τον ορισμό αυτό:

Ορισμός 4.9. Ως ε-κλείσιμο: $\text{Pow}(Q) \to \text{Pow}(Q)$ ορίζουμε το

$$\varepsilon\text{-κλείσιμο}(P) = \bigcup_{q \in P} \varepsilon\text{-κλείσιμο}(q)$$

Παρατηρούμε επιπλέον ότι ε-κλείσιμο(ε-κλείσιμο(P)) $=$ ε-κλείσιμο(P). Η συνάρτηση μετάβασης δ μπορεί να επεκταθεί. Η επεκτεταμένη συνάρτηση μετάβασης $\delta(q, w)$ θα περιέχει το σύνολο όλων των καταστάσεων που μπορούμε να φτάσουμε από το q με είσοδο το w, καθώς και με οσεσδήποτε αρχικές, ενδιάμεσες και τελικές ε-κινήσεις.

Ορισμός 4.10. $\delta : Q \times \Sigma^* \to \text{Pow}(Q)$ όπου

$$\begin{cases} \delta(q, \varepsilon) = \varepsilon\text{-κλείσιμο}(q) \\ \delta(q, wa) = \varepsilon\text{-κλείσιμο}(\{p \in \delta(r, a) \mid r \in \delta(q, w)\}) = \varepsilon\text{-κλείσιμο}(\bigcup_{r \in \delta(q,w)} \delta(r, a)) \end{cases}$$

Επιπλέον, επεκτείνουμε την δ, συμπεριλαμβάνοντας στο πεδίο ορισμού το δυναμοσύνολο του Q.

Ορισμός 4.11. $\delta : \text{Pow}(Q) \times \Sigma^* \to \text{Pow}(Q)$ όπου

$$\delta(P, w) = \{p \in \delta(q, w) | q \in P\} = \bigcup_{q \in P} \delta(q, w)$$

Ορισμός 4.12. Έχουμε:

- Ένα NFA$_\varepsilon$ αποδέχεται το string $w \in \Sigma^*$ ανν $\delta(q_0, w) \cap F \neq \emptyset$
- Ένα NFA$_\varepsilon$ αποδέχεται τη γλώσσα $L(M) = \{w | \delta(q_0, w) \cap F \neq \emptyset\}$

Ισοδυναμία NFA και NFA$_\varepsilon$. Και πάλι, μπορεί να θεωρήσει κανείς ότι τα NFA είναι μια "υποπερίπτωση" των NFA$_\varepsilon$. Όμως, όπως και πριν, τα NFA$_\varepsilon$ δεν έχουν περισσότερες δυνατότητες υπολογισμού από τα NFA, όπως αποδεικνύει το επόμενο θεώρημα, και άρα και από τα DFA.

Θεώρημα 4.8. *Έστω M ένα NFA$_\varepsilon$ τότε $\exists$ NFA $M' : L(M) = L(M')$*

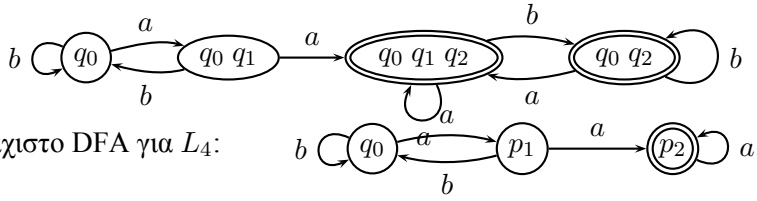
Απόδειξη. Ορίζουμε το NFA $M' = (Q, \Sigma, q_0, F', \delta')$ όπου

$$F' = \begin{cases} F \cup \{q_0\}, & \text{αν } \varepsilon\text{-κλείσιμο}(q_0) \cap F \neq \emptyset \\ F, & \text{ειδάλλως} \end{cases}, \qquad \delta'(q, a) = \delta(q, a)$$

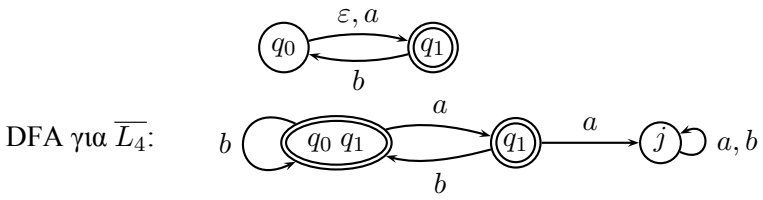
Πλέον, προκειμένου να ισχύει $L(M) = L(M')$, αρκεί να αποδειχθεί ο ισχυρισμός: $\forall w \in \Sigma^* - \{\varepsilon\}$: $\delta'(q_0, w) = \delta(q_0, w)$. ('Ασκηση.) □

Ακολουθούν μερικά παραδείγματα με μη ντετερμινιστικά αυτόματα και ισοδύναμά τους ντετερμινιστικά αυτόματα.

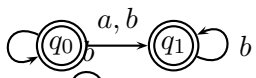
Παράδειγμα 4.9. DFA για L_4 (βλ. παραδ. 4.4):



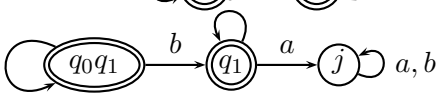
Παράδειγμα 4.10. NFA$_\varepsilon$ για $\overline{L_4}$ (δηλαδή "όχι δύο συνεχόμενα a"):



Παράδειγμα 4.11. NFA για L_5 (βλ. παράδ. 4.7):



Ελάχιστο DFA για L_5:



4.4 Κανονικές παραστάσεις

Παρακάτω, ορίζουμε κάποιες πράξεις επί των γλωσσών.

Ορισμός 4.13. Έστω L, L_1, L_2 γλώσσες επί του ίδιου αλφαβήτου Σ.

- $L_1L_2 := \{uv \mid u \in L_1 \wedge v \in L_2\}$: παράθεση
- $L_1 \cup L_2 := \{w \mid w \in L_1 \vee w \in L_2\}$: ένωση
- $L_1 \cap L_2 := \{w \mid w \in L_1 \wedge w \in L_2\}$: τομή
- $L^0 := \{\varepsilon\}$, $L^{n+1} := LL^n$
- $L^* := \bigcup_{n=0}^{\infty} L^n$: άστρο του Kleene
- $L^+ := \bigcup_{n=1}^{\infty} L^n$

Τώρα πλέον μπορούμε να εισάγουμε επαγωγικά έναν συμβολισμό για τις κανονικές γλώσσες, τις λεγόμενες *κανονικές παραστάσεις*.

Ορισμός 4.14. Κανονική παράσταση είναι:

$\emptyset$: παριστάνει το κενό σύνολο·

ε: παριστάνει το $\{\varepsilon\}$·

a: παριστάνει το $\{a\}$, όπου $a \in \Sigma$·

$(r + s)$: παριστάνει το $R \cup S$, όπου r, s κανονικές παραστάσεις που παριστάνουν τα R, S αντιστοίχως·

(rs): παριστάνει το RS, όπου r, s κανονικές παραστάσεις που παριστάνουν τα R, S αντιστοίχως·

(r^*)**:** παριστάνει το R^*, όπου r κανονική παράσταση που παριστάνει το R.

Σύμβαση: Μπορούμε να περιορίσουμε τις παρενθέσεις αν χρησιμοποιήσουμε την ακόλουθη προτεραιότητα των τελεστών: $*$, παράθεση, ένωση.

Παράδειγμα 4.12.

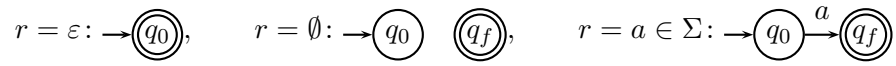
$L_1 = a(a+b)^*$
$L_2 = (b^*ab^*a)^*b^* = (b+ab^*a)^*$
L_3 δεν είναι δυνατόν να παρασταθεί με κανονική παράσταση
$L_4 = (a+b)^*aa(a+b)^*$ (τουλάχιστον δύο συνεχόμενα a)
$\overline{L_4} = (a+\varepsilon)(ba+b)^*$ (όχι συνεχόμενα a)
$L_5 = a^*b^*$

Θεώρημα 4.13. *Μία γλώσσα μπορεί να παρασταθεί με κανονική παράσταση ανν $L = L(M)$ για κάποιο πεπερασμένο αυτόματο M.*

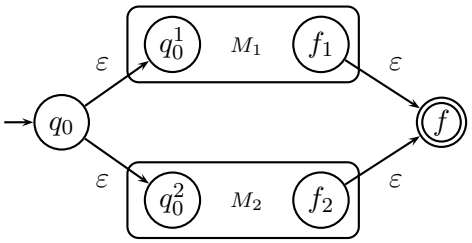
Ιδέα απόδειξης.

$\Rightarrow$ Επαγωγή στην δομή της κανονικής παράστασης. Έστω r η κανονική παράσταση.

1. Επαγωγική Βάση:

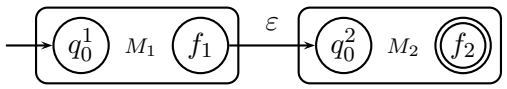


2. Επαγωγικό Βήμα: Υποθέτουμε ότι οι $L(r_1)$ και $L(r_2)$ αναγνωρίζονται από NFA$_\varepsilon$ M_1, M_2 αντιστοίχως με τελική κατάσταση f_1, f_2 αντιστοίχως.
Περίπτωση α: $r = r_1 + r_2$



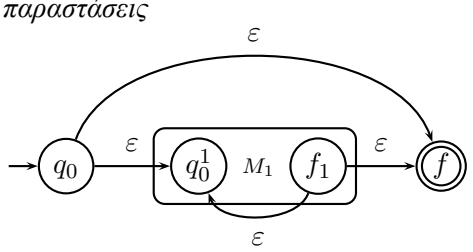
$L(M) = L(M_1) \cup L(M_2)$

Περίπτωση β: $r = r_1 r_2$



$L(M) = L(M_1)L(M_2)$

Περίπτωση γ: $r = r_1^*$



$L(M) = L(M_1)^*$

$\Leftarrow$ Θεωρούμε ότι το DFA είναι της μορφής $= (Q, \Sigma, \delta, q_1, F)$, όπου τα στοιχεία του Q είναι αριθμημένα κατά αύξουσα σειρά και η αρχική κατάσταση είναι η q_1, δηλαδή $Q = \{q_1, \ldots, q_n\}$, όπου $n = |Q|$. Ορίζουμε:

$$R_{ij}^k = \{w \mid \tilde{\delta}(q_i, w) = q_j \text{ και } \forall x \text{ πρόθεμα του } w \text{ με } x \neq w, \varepsilon\colon \tilde{\delta}(q_i, x) = q_l \Rightarrow l \leq k\}$$

Δηλαδή, R_{ij}^k είναι το σύνολο των συμβολοακολουθιών που οδηγούν από την κατάσταση q_i στην q_j χωρίς να περνούν από οποιαδήποτε κατάσταση με δείκτη μεγαλύτερο από k. Προφανώς, αφού δεν υπάρχει κατάσταση με δείκτη μεγαλύτερο από n, το R_{ij}^n περιέχει όλες τις συμβολοακολουθίες από την q_i στην q_j. Μπορούμε να υπολογίσουμε το R_{ij}^k αναδρομικά, σύμφωνα με μια ιδέα των Floyd και Warshall, ως εξής:

Αλγόριθμος Floyd-Warshall

$$R_{ij}^0 = \begin{cases} \{a | \delta(q_i, a) = q_j\}, & \text{αν } i \neq j \\ \{a | \delta(q_i, a) = q_j\} \cup \{\varepsilon\}, & \text{αν } i = j \end{cases}$$

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$$

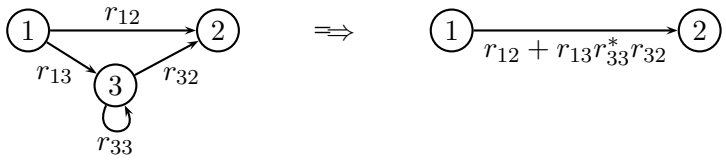
Τέλος, αρκεί να παρατηρήσουμε ότι $L(M) = \bigcup_{q_j \in F} R_{1j}^n$. Η ιδέα των Floyd και Warshall είναι ότι το R_{ij}^k αποτελείται από συμβολοακολουθίες που οδηγούν από την κατάσταση q_i στην q_j είτε χωρίς να περνούν από την κατάσταση q_k (οπότε περιέχονται και στο R_{ij}^{k-1}, είτε περνούν από την κατάσταση q_k μία ή περισσότερες φορές (οι λεπτομέρειες της απόδειξης αφήνονται ως άσκηση.) □

Κατασκευή κανονικής παράστασης από FA Το 2ο σκέλος της πιο πάνω απόδειξης δίνει έναν πλήρη, συστηματικό, αλλά συχνά χρονοβόρο, τρόπο κατασκευής της κανονικής παράστασης που αντιστοιχεί σε ένα FA. Παρακάτω δίνεται ένας πιο γρήγορος τρόπος που βασίζεται στην έννοια του GNFA: γενικευμένο αυτόματο, του οποίου η συνάρτηση μετάβασης δέχεται και κανονικές παραστάσεις (στα βέλη του μπορούμε να γράφουμε κανονικές παραστάσεις αντί για σκέτα σύμβολα).

Κατ' αρχάς, θεωρούμε ότι το FA έχει μια αρχική και μια τελική κατάσταση, διαφορετικές μεταξύ τους. Αν υπάρχουν πάνω από μια τελικές καταστάσεις, προσθέτουμε μια νέα κατάσταση στην οποία βαίνουν όλες με ε-κινήσεις και στη συνέχεια

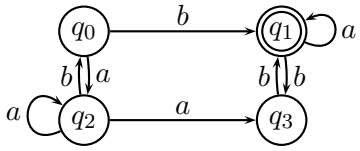
τις κάνουμε μη τελικές. Αντίστοιχα μετατρέπουμε και την αρχική κατάσταση σε μη τελική, αν χρειάζεται.

Φέρνοντας το FA μας σε αυτή τη μορφή, μπορούμε να απαλείψουμε μία μία τις ενδιάμεσες καταστάσεις σύμφωνα με το πιο κάτω σχήμα:

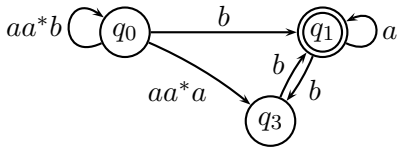


Με διαδοχικές απαλοιφές οδηγούμαστε τελικά στο επιθυμητό GNFA (γενικευμένο αυτόματο, του οποίου η συνάρτηση μετάβασης δέχεται και κανονικές παραστάσεις). Στη συνέχεια θα δώσουμε ένα παράδειγμα με όλα τα ενδιάμεσα βήματα υπολογισμού.

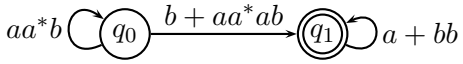
Παράδειγμα 4.14. ’Εστω πως έχουμε το ακόλουθο FA:



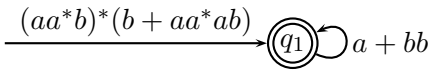
Επιλέγουμε να διαγράψουμε τη κατάσταση q_2, οπότε ενημερώνουμε τις μεταβάσεις των άλλων καταστάσεων:



Επιλέγουμε να διαγράψουμε τη κατάσταση q_3, οπότε ενημερώνουμε τα υπόλοιπα:



Επιλέγουμε να διαγράψουμε τη κατάσταση q_0:



Τελικά, η κανονική έκφραση που προκύπτει είναι:

$$(aa^*b)^*(b + aa^*ab)(a + bb)^*$$

4.5 Ελαχιστοποίηση DFA

Σε πολλές περιπτώσεις μπορούμε να μειώσουμε τον αριθμό καταστάσεων ενός DFA ως εξής: Δύο καταστάσεις μπορούν να συγχωνευτούν σε μία αν είναι και οι

δύο τελικές ή και οι δύο μη τελικές και αν ξεκινώντας από οποιαδήποτε από αυτές το αυτόματο θα έχει το ίδιο αποτέλεσμα (ως προς την αποδοχή ή μη) για οποιαδήποτε συμβολοσειρά διαβαστεί στη συνέχεια. Για παράδειγμα, δύο καταστάσεις που απορρίπτουν και με κάθε σύμβολο πηγαίνουν στον εαυτό τους ('junk' states) μπορούν να συγχωνευτούν σε μία.

Πιο αυστηρά, οι καταστάσεις q_i, q_j μπορούν να συγχωνευτούν αν:

$$\forall w \in \Sigma^* : \delta(q_i, w) \in F \Leftrightarrow \delta(q_j, w) \in F$$

Προσέξτε ότι ο παραπάνω ορισμός περιλαμβάνει και την περίπτωση $w = \varepsilon$: αυτό ισοδυναμεί με την απαίτηση οι δύο καταστάσεις να είναι του ίδιου είδους ως προς την αποδοχή (δηλ. τελικές ή μη τελικές).

Παρατηρήστε ακόμη ότι δεν έχει καμμία σημασία με ποιες συμβολοσειρές μπορεί το αυτόματο να φτάσει στις δύο καταστάσεις. Αυτό που έχει σημασία είναι τι κάνει *από εκεί και μετά*.

Αποδεικνύεται (συνέπεια του Θεωρήματος Myhill-Nerode) ότι σε κάθε κανονικό σύνολο αντιστοιχεί ένα μοναδικό DFA (εκτός ισομορφισμού, δηλαδή εκτός μετονομασίας των καταστάσεων) με ελάχιστο αριθμό καταστάσεων και επιπλέον υπάρχει αλγόριθμος που κατασκευάζει το μοναδικό αυτό DFA. Ο αλγόριθμος αυτός βρίσκει συστηματικά όλα τα ζεύγη διακρίσιμων καταστάσεων όπως περιγράφεται παρακάτω:

Αλγόριθμος 4.1 Αλγόριθμος ελαχιστοποίησης DFA

---

1. Εξαλείφουμε όλες τις απρόσιτες καταστάσεις.

2. Φτιάχνουμε ένα πίνακα για να συγκρίνουμε κάθε ζεύγος καταστάσεων. Αρχικά σημειώνουμε X_0 σε όλα τα ζεύγη όπου η μία κατάσταση είναι τελική ενώ η άλλη δεν είναι.

3. Επαναλαμβάνουμε το παρακάτω για $i := 1, 2, \ldots$ μέχρι που σε κάποια επανάληψη να μην σημειωθεί κανένα νέο ζεύγος στον πίνακα:

   για κάθε ζεύγος q_k, q_j που δεν έχει ήδη σημειωθεί:

   για κάθε σύμβολο $\sigma \in \Sigma$:

   αν το ζεύγος $(\delta(q_k, \sigma), \delta(q_j, \sigma))$ είναι σημειωμένο με X_{i-1}

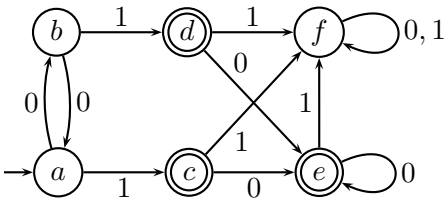
   τότε σημειώνουμε το ζεύγος (q_k, q_j) με X_i.

4. Ζεύγη που δεν έχουν σημειωθεί συγχωνεύονται.

Αποδεικνύεται (η απόδειξη παραλείπεται) ότι ο παραπάνω αλγόριθμος δεν οδηγεί σε αντιφάσεις, δηλαδή αν τα ζεύγη (q_k, q_j) και (q_k, q_m) δεν έχουν σημειωθεί (είναι συγχωνεύσιμα) τότε ούτε το ζεύγος (q_m, q_j) έχει σημειωθεί (επομένως και οι τρεις

καταστάσεις συγχωνεύονται σε μία.

Παράδειγμα 4.15. Έστω το αυτόματο M που φαίνεται στο σχήμα, το οποίο αποδέχεται την γλώσσα $L = 0^*10^*$.

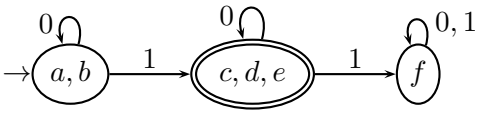


Στον πίνακα οι δείκτες του X δείχνουν σε ποια επανάληψη εγγράφουμε το (οι δείκτες υποδηλώνουν επίσης με πόσα σύμβολα διακρίνονται οι καταστάσεις).

b					
ⓒ	X_0	X_0			
ⓓ	X_0	X_0			
ⓔ	X_0	X_0			
f	X_1	X_1	X_0	X_0	X_0
	a	b	ⓒ	ⓓ	ⓔ

Τελικά οι ισοδύναμες καταστάσεις είναι $a \equiv b$, $c \equiv d \equiv e$.

Το ελάχιστο αυτόματο φαίνεται στο παρακάτω σχήμα.



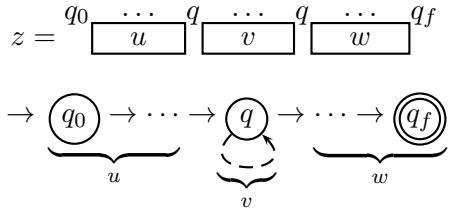
Επεξηγήσεις για τον παραπάνω αλγόριθμος Ο παραπάνω αλγόριθμος βασίζεται στην εξής παρατήρηση: αν δύο καταστάσεις q_i, q_j διακρίνονται με k σύμβολα, δηλαδή υπάρχει συμβολοσειρά w μήκους k που τις οδηγεί σε διαφορετικό αποτέλεσμα, τότε αν δύο καταστάσεις έστω q_m, q_n οδηγούν με κάποιο σύμβολο στις q_i, q_j, τότε οι q_m, q_n διακρίνονται με $k + 1$ σύμβολα. Αυτό ισχύει και αντίστροφα, και επομένως αρκεί ο αλγόριθμος να εξετάζει ένα σύμβολο κάθε φορά: αν υπάρχουν δύο καταστάσεις που διακρίνονται με $k + 1$ σύμβολα, τότε υπάρχει ένα σύμβολο που τις οδηγεί σε k-διακρίσιμες καταστάσεις.

4.6 Pumping Lemma για κανονικά σύνολα

Ένα σημαντικό αποτέλεσμα είναι το Pumping Lemma που χρησιμοποιείται κυρίως για να αποδεικνύουμε ότι συγκεκριμένες γλώσσες δεν είναι κανονικές αλλά και σε αλγορίθμους για να απαντάμε ερωτήσεις σχετικά με πεπερασμένα αυτόματα, όπως αν μια γλώσσα που γίνεται αποδεκτή από ένα πεπερασμένο αυτόματο είναι πεπερασμένη ή άπειρη.

Pumping Lemma. Αν μία γλώσσα είναι κανονική τότε την αποδέχεται ένα DFA $M = \{Q, \Sigma, \delta, q_0, F\}$ με κάποιο συγκεκριμένο αριθμό από καταστάσεις, έστω n, δηλαδή $|Q| = n$. Έστω μία λέξη z που γίνεται αποδεκτή από το αυτόματο M και η οποία έχει μήκος μεγαλύτερο από n.

Καθώς επεξεργαζόμαστε το z, το αυτόματο μας M πρέπει να περάσει ξανά από μία κατάσταση, γιατί δεν υπάρχουν περισσότερες από n καταστάσεις (αρχή του περιστερώνα, pigeonhole principle). Έχουμε ότι ένα μονοπάτι που αποδέχεται το z είναι το ακόλουθο:



Το uvw γίνεται αποδεκτό, όπως επίσης και το uw, ή το $uvvw$ ή γενικά το uv^iw για οποιοδήποτε $i \in \mathbb{N}$. Δηλαδή το v μπορεί να επαναληφθεί όσες φορές θέλουμε.

Λήμμα 4.16 (Pumping Lemma). *Εάν L είναι regular τότε:*
$\exists n \in \mathbb{N}, \forall z \in L$ *με* $|z| \geq n, \exists u, v, w \in \Sigma^*$:
$[z = uvw \ \wedge |uv| \leq n \ \wedge |v| \geq 1 \wedge \forall i \in \mathbb{N} \ (uv^iw \in L)]$

Χρησιμοποιώντας το λήμμα δείχνουμε ότι ένα δοσμένο σύνολο δεν είναι regular. Η μέθοδος είναι η ακόλουθη:

1. Διαλέγεις τη γλώσσα που θέλεις να αποδείξεις πως δεν είναι regular.
2. Ο αντίπαλος (PL) επιλέγει ένα n. Θα πρέπει να μπορείς για οποιοδήποτε πεπερασμένο ακέραιο n διαλέξει, να αποδείξεις ότι η L δεν είναι regular, αλλά από τη στιγμή που ο αντίπαλος έχει διαλέξει ένα n αυτό είναι σταθερό στην απόδειξη.
3. Διαλέγεις ένα string z της L έτσι ώστε $\mid z \mid \geq n$.
4. Ο αντίπαλος (PL) σπάει το z σε u, v και w που ικανοποιούν τους περιορισμούς $|uv| \leq n$ και $|v| \geq 1$.
5. Φτάνεις σε αντίφαση δείχνοντας ότι για κάθε u, v, w που καθορίζονται από τον αντίπαλο, υπάρχει ένα i για το οποίο uv^iw δεν ανήκει στην L. Τότε μπορούμε να συμπεράνουμε ότι η L δεν είναι regular. Η επιλογή του i μπορεί να εξαρτάται από τα n, u, v, w.

Παράδειγμα 4.17. $L = \{a^kb^k \mid k \in \mathbb{N}\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$ δεν είναι regular.

1. Υποθέτουμε ότι L είναι regular και χρησιμοποιούμε το Pumping lemma.
2. PL: $\exists n \in \mathbb{N}$
3. Διαλέγουμε $z = a^n b^n$. Εντάξει επιλογή, διότι $z \in L$, $|z| = 2n \geq n$.
4. PL: z μπορεί να γραφεί: $z = uvw$ με $|uv| \leq n \wedge |v| \geq 1$, οπότε αναγκαστικά $v = a^l$ με $l \geq 1$.
5. Διαλέγουμε $i = 2$: $uvvw = a^{n+l} b^n \in L$.

'Ατοπο

Εναλλακτικά:

1. Υποθέτουμε πάλι ότι η L είναι regular.
2. Συνεπώς, αναγνωρίζεται από ένα DFA M. Αυτό θα έχει κάποιον πεπερασμένο αριθμό καταστάσεων, έστω n.
3. Θέτουμε $z = a^n b^n \in L$.
4. Το DFA M αποδέχεται τη λέξη z. Καθώς τη διατρέχει, μέχρι να φτάσει στο $n-$οστό a, αναγκαστικά τουλάχιστον μία κατάστασή του επαναλαμβάνεται, έστω στο $i-$οστό και στο $j-$οστό a, $i < j$.
5. Έστω $l = j - i$. Τότε, για κάθε $k \geq -1$, το DFA M θα αποδέχεται το string $a^{n+k \cdot l} b^n$. Το substring $a^{k \cdot l}$ θα αντιστοιχεί σε k επαναλήψεις της ίδιας κυκλικής ακολουθίας καταστάσεων του M.

'Ατοπο

Ασκήσεις

1. Αποδείξτε ή ανταποδείξτε (δίνοντας αντιπαράδειγμα) ότι για δύο γλώσσες $L_1, L_2 \subseteq \Sigma^*$ ισχύει $(L_1 \cup L_2)^* = L_1^* \cup L_2^*$
2. Δώστε διάγραμμα καταστάσεων για DFA που αποδέχεται την γλώσσα

$$L = \left\{ w \in \{a,b\}^* \mid \text{κάθε } a \left\langle \begin{array}{l} \text{ακολουθείται αμέσως από ένα } b \\ \text{και έπεται αμέσως ενός } b \end{array} \right. \right\}$$

   Για παράδειγμα, $bababb \in L$, $bbb \in L$, αλλά $ab \notin L$.
   Το DFA σας δεν θα πρέπει να έχει πάνω από τέσσερις καταστάσεις.
3. Κατασκευάστε κανονικές εκφράσεις για κάθε μία από τις γλώσσες:
   (α) Σύνολο συμβολοσειρών του $\{a,b\}^*$ των οποίων το πλήθος των a δεν είναι πολλαπλάσιο του 3.

(β) Σύνολο συμβολοσειρών του $\{a, b\}^*$ που έχουν άρτιο πλήθος a και τελειώνουν σε b.
(γ) Σύνολο συμβολοσειρών του $\{a, b\}^*$ που δεν έχουν δύο συνεχόμενα a.

4. Δύο κανονικές εκφράσεις ονομάζονται ισοδύναμες όταν παράγουν την ίδια γλώσσα. Χρησιμοποιούμε το σύμβολο της ισότητας για να δείξουμε την ισοδυναμία. Έστω r και s κανονικές εκφράσεις. Θεωρήστε την 'εξίσωση' $X = rX + s$. Με την προϋπόθεση ότι η γλώσσα που παράγεται από την r, δηλαδή η $L(r)$, δεν περιέχει την κενή συμβολοσειρά ε, βρείτε την λύση για το X και αποδείξτε ότι είναι μοναδική (εκτός ισοδυναμίας). Ποια είναι η λύση αν η $L(r)$ περιέχει την ε; *Υπόδειξη*: Στην τελευταία περίπτωση η λύση δεν είναι μοναδική. Για παράδειγμα, αμφότερες οι $X = 10^*$ και $X = (1+11)0^*$, που αναπαριστούν διαφορετικές γλώσσες, είναι λύσεις της $X = 1 + X0^*$.

5. Δείξτε ότι $\delta(q, xy) = \delta(\delta(q, x), y)$ για κάθε x, y.

6. Δώστε DFA που αποδέχονται τις ακόλουθες γλώσσες επί του αλφαβήτού $\{0, 1\}$:

   - $\{w \mid$ κάθε υποακολουθία από 5 διαδοχικά σύμβολα περιέχει τουλάχιστον δύο «0»$\}$
   - $\{w \mid w$: δυαδική αναπαράσταση φυσικού αριθμού $\overline{w}$, w αρχίζει με 1, $\overline{w} \equiv 0 \pmod 5 \}$
   - $\{w \mid$ το 3ο σύμβολο πριν το τελευταίο της συμβολοσειράς είναι «1»$\}$
   - $\{w \mid |w|$ διαιρείται από το 2 ή το 3 ή από αμφότερα$\}$
   - $\{w \in \{1, 2, 3\}^* \mid$ άθροισμα ψηφίων της w είναι $\equiv 0 \pmod 4\}$

7. Δώστε NFA για την $\{w \in \{0, 1\}^* \mid$ υπάρχουν δύο «0» που χωρίζονται από μία συμβολοσειρά μήκους $4i$ για κάποιο $i \geq 0 \}$.

8. Κατασκευάστε DFA ισοδύναμα με τα NFA:

   (α) $(\{p, q, r, s\}, \{0, 1\}, \delta_1, p, \{s\})$ (β) $(\{p, q, r, s\}, \{0, 1\}, \delta_2, p, \{q, s\})$

δ_1	0	1
p	p, q	p
q	r	r
r	s	$-$
Ⓢ	s	s

δ_2	0	1
p	q, s	q
Ⓠ	r	q, r
r	s	p
Ⓢ	$-$	p

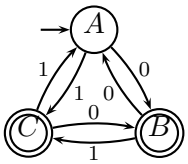
9. Κατασκευάστε FA ισοδύναμα με:

   (α) $10 + (0 + 11)0^*1$

   (β) $01[((10)^* + 111)^* + 0]^*1$

   (γ) $((0 + 1)(0 + 1))^* + ((0 + 1)(0 + 1)(0 + 1))^*$

10. Κατασκευάστε κανονική έκφραση για το:



11. Γράψτε κανονικές εκφράσεις για τις παρακάτω γλώσσες επί του $\{0, 1\}$. Δικαιολογήστε την ορθότητα των κανονικών σας εκφράσεων.

    (α) Το σύνολο των strings που δεν περιέχουν το 101 ως substring.

    (β) Το σύνολο των strings με ίσο πλήθος εμφανίσεων «0» και «1», έτσι ώστε δεν υπάρχει πρόθεμα που να περιέχει δύο περισσότερα «0» από ότι «1», ούτε δύο περισσότερα «1» από ότι «0».

12. Δείξτε τις ακόλουθες ταυτότητες για κανονικές εκφράσεις. Εδώ $r = s$ σημαίνει $L(r) = L(s)$.

    - $r + s = s + r$
    - $(r + s) + t = r + (s + t)$
    - $(rs)t = r(st)$
    - $r(s + t) = rs + rt$
    - $(r + s)t = rt + st$
    - $\emptyset^* = \varepsilon$,
    - $(r^*)^* = r^*$
    - $(\varepsilon + r)^* = r^*$
    - $(r^*s^*)^* = (r + s)^*$

13. Αποδείξτε ή ανταποδείξτε:

    - $(rs + r)^*r = r(sr + r)^*$,
    - $s(rs + s)^*r = rr^*s(rr^*s)^*$,
    - $(r + s)^* = r^* + s^*$.

14. Ποια από τα παρακάτω σύνολα είναι κανονικά; Αποδείξτε!

    - $\{a^{3k} \mid k \in \mathbb{N}\}$
    - σύνολο εξισορροπημένων παρενθέσεων
    - $\{a^i b^j \mid 1 \leq i < j\}$
    - $\{w \mid$ η w περιέχει λιγότερα a από $b\}$
    - $\{a^i b^j a^j \mid i, j \geq 1\}$
    - $\{w \in \{1\}^* \mid |w|$ είναι πρώτος αριθμός$\}$
    - $\{uvw \in \{a, b\}^* \mid$ η uv είναι παλίνδρομη αρτίου μήκους$\}$
    - $\{w \in \{a, b\}^* \mid$ η w δεν έχει 4 συνεχόμενα $b\}$

- $\{uvw \in \{a,b\}^* \mid$ η uw είναι παλίνδρομη αρτίου μήκους$\}$
- $\{w \in \{a,b\}^* \mid$ η w είναι παλίνδρομη$\}$

15. Έστω κανονικό σύνολο L. Ποια από τα παρακάτω σύνολα είναι κανονικά; Δικαιολογήστε τις απαντήσεις σας.

- $\{a_1a_3a_5 \dots a_{2n-1} \mid a_1a_2a_3 \dots a_{2n} \in L\}$
- $\{a_2a_1a_4a_3 \dots a_{2n}a_{2n-1} \mid a_1a_2a_3 \dots a_{2n} \in L\}$
- Cycle$(L) = \{w_1w_2 \mid w_1w_2 \in L,\ w_1, w_2$: strings$\}$
- max$(L) = \{x \in L \mid$ δεν υπάρχει $y \neq \varepsilon$ με $xy \in L\}$
- min$(L) = \{x \in L \mid$ δεν υπάρχει πρόθεμα του x που να είναι στην $L\}$
- init$(L) = \{x \in L \mid$ για κάποιο y, $xy \in L\}$
- $L^R = \{x \mid x^R \in L\}$
- $\{x \mid xx^R \in L\}$

Κεφάλαιο 5

Μη κανονικές γλώσσες και γραμματικές

5.1 Εισαγωγή

Στο κεφάλαιο αυτό θα ασχοληθούμε με μη κανονικές γλώσσες και γραμματικές, καθώς και τα αντίστοιχα αυτόματα που τις αναγνωρίζουν. Πιο συγκεκριμένα για: γλώσσες και γραμματικές *χωρίς συμφραζόμενα (context free)*, γλώσσες και γραμματικές *με συμφραζόμενα (context sensitive)*, και *αναδρομικά αριθμήσιμες (recursively enumerable)* γλώσσες και *γενικές γραμματικές*.

5.2 Κανονικές Γραμματικές

Στις *κανονικές γραμματικές* όλοι οι κανόνες παραγωγής είναι της μορφής:

1. δεξιογραμμικοί (rightlinear): $A \to wB$, $A \to w$, όπου $w \in T^*$, ή
2. αριστερογραμμικοί (leftlinear): $A \to Bw$, $A \to w$, όπου $w \in T^*$.

Θεώρημα 5.1. *Τα κανονικά σύνολα παράγονται από κανονικές (δεξιο- ή αριστερογραμμικές γραμματικές)*

Ιδέα απόδειξης. Περιοριζόμαστε σε δεξιογραμμικές γραμματικές:

- Έστω G_1 δεξιογραμμική. Κατασκευάζουμε ισοδύναμη γραμματική G_2 σε *μεταβατική μορφή*, δηλαδή σε μορφή κανόνων $A \to aB$ ή $A \to a$ (ή ε). Ιδέα για μετατροπή: $A \to a_1 \dots a_n B$ αντικαθίσταται από τους κανόνες: $A \to a_1 A_1$, $A_1 \to a_2 A_2$, $\dots$, $A_{n-1} \to a_n B$, όπου ${}_1, \dots, A_{n-1}$ νέα μη τερματικά σύμβολα.

Μετά κατασκευάζουμε ένα NFA$_\varepsilon$ με $Q = V \cup \{\varepsilon'\}, \Sigma = , q_0 = S, F = \{\varepsilon'\}$ και:

$$B \in \delta(A, a) \text{ ανν } (A \to aB) \in P$$
$$\varepsilon' \in \delta(A, a) \text{ ανν } (A \to a) \in P$$
$$\varepsilon' \in \delta(A, \varepsilon) \text{ ανν } (A \to \varepsilon) \in P$$

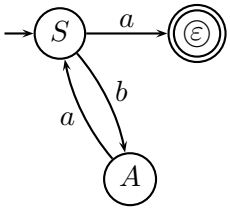
- Έστω DFA. Κατασκευάζουμε γραμματική με $V = Q$, $T = \Sigma$, $S = q_0$ και παραγωγές:

$$(q_i \to aq_j) \in P \text{ ανν } \delta(q_i, a) = q_j, \qquad (q_i \to a) \in P \text{ ανν } \delta(q_i, a) \in F$$

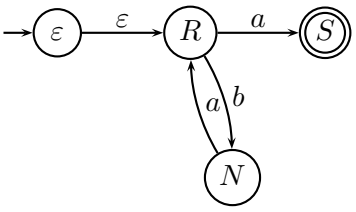
□

Παράδειγμα 5.2. $(ba)^*a$ δεξιογραμμικά $S \to baS \mid a$
αριστερογραμμικά $S \to Ra,\ R \to Rba \mid \varepsilon$

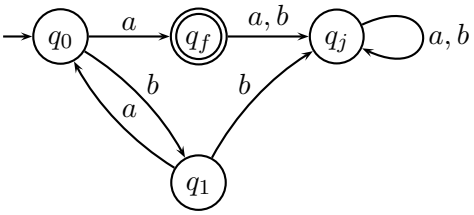
• Για να κατασκευάσουμε F.A. που αποδέχεται την κανονική γλώσσα που παράγεται από τη δεξιογραμμική γραμματική ($S \to baS \mid a$) πρώτα την απλοποιούμε σε *μεταβατική μορφή* ($S \to bA \mid a, A \to aS$) και μετά:



• Για να κατασκεύασουμε F.A. που αποδέχεται την κανονική γλώσσα που παράγεται από την αριστερογραμμική γραμματική ($S \to Ra$, $R \to Rba \mid \varepsilon$) πρώτα απλοποιείται σε μεταβατική μορφή ($S \to Ra, R \to Na \mid \varepsilon, N \to Rb$) και μετά:



• Κατασκευή δεξιογραμμικής γραμματικής από αυτόματο.
Έστω M:



$$\begin{cases} q_0 \to a \mid bq_1 \mid aq_f \\ q_1 \to aq_0 \mid bq_j \\ q_f \to aq_j \mid bq_j \\ q_j \to aq_j \mid bq_j \end{cases}$$

Απαλοιφή σύμβολων που δεν παράγουν τίποτα (non yielding) q_j, q_f και παραγωγών που σχετίζονται με αυτά. Τότε έχουμε:

$$\begin{cases} q_0 \to a \mid bq_1 \\ q_1 \to aq_0 \end{cases}$$

• Κατασκευή αριστερογραμμικής γραμματικής από αυτόματο. ($S := q_f$)

$$\begin{cases} q_f \to q_0a \\ q_0 \to q_1a \mid \varepsilon \\ q_1 \to q_0b \\ q_j \to q_jb \mid q_ja \mid q_fb \mid q_fa \mid q_1b \end{cases}$$

Απαλοιφή συμβόλων στα οποία δεν φθάνουμε ποτέ (unreachable) q_j καθώς και παραγωγών που σχετίζονται με αυτά. Τότε έχουμε:

$$\begin{cases} q_f \to q_0a \\ q_0 \to q_1a \mid \varepsilon \\ q_1 \to q_0b \end{cases}$$

5.3 Γραμματικές χωρίς συμφραζόμενα και αυτόματα στοίβας

Όπως είδαμε στην ενότητα περί ιεραρχίας Chomsky, μια γραμματική $G = (V, T, P, S)$ λέγεται *χωρίς συμφραζόμενα (context free - c.f.)* αν οι κανόνες παραγωγής στο P έχουν τη μορφή $A \to \beta$ όπου $\beta \in (V \cup T)^*$ και $A \in V$. Παρακάτω δίνουμε μερικά παραδείγματα γραμματικών χωρίς συμφραζόμενα:

Παράδειγμα 5.3. Έστω η γραμματική

$$G_1: \quad V = \{S\},\ T = \{a, b\},\ P = \{S \to \varepsilon, S \to aSb\}$$

Μια πιθανή ακολουθία παραγωγών είναι η:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Η γλώσσα που παράγεται από την G_1 είναι η $L(G_1) = \{a^n b^n | n \in \mathbb{N}^*\}$.

Παράδειγμα 5.4. $G_2 = (V, T, P, S)$ με $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *\}$, $V = \{S\}$ και το P περιέχει τους κανόνες:

$$S \to S + S, \quad S \to S * S, \quad S \to 0|1|2|3|4|5|6|7|8|9$$

Τόσο στο πιο πάνω παράδειγμα, όσο και στο παράδειγμα 5.3, είναι εύκολο να καταλάβουμε ποιες είναι οι γλώσσες που παράγονται από τις γραμματικές G_1, G_2. Εν γένει, το να βρούμε την $L(G)$ δεν είναι πάντοτε τόσο προφανές, όπως φαίνεται από το επόμενο παράδειγμα.

Παράδειγμα 5.5. $G_3 = (V, T, P, S), V = \{S, A, B\}, T = \{a, b\}$ και το P περιέχει τους κανόνες:

$$S \to aB \mid bA, \quad A \to a \mid aS \mid bAA, \quad B \to b \mid bS \mid aBB$$

Μια πιθανή ακολουθία παραγωγών της πιο πάνω γραμματική είναι:

$$S \Rightarrow aB \Rightarrow abS \Rightarrow abbA \Rightarrow abba$$

Αν και δεν είναι προφανές, $L(G_3) = \{w \in T^+ \mid w$ έχει ίσο αριθμό a και $b\}$

5.3.1 Συντακτικά δένδρα

Ένας άλλος τρόπος να να περιγράψουμε πως με μια γραμματική προκύπτει μια συμβολοακολουθία, εκτός από τις ακολουθίες παραγωγών μέσω των "⇒", είναι τα *συντακτικά δένδρα* (parse trees).

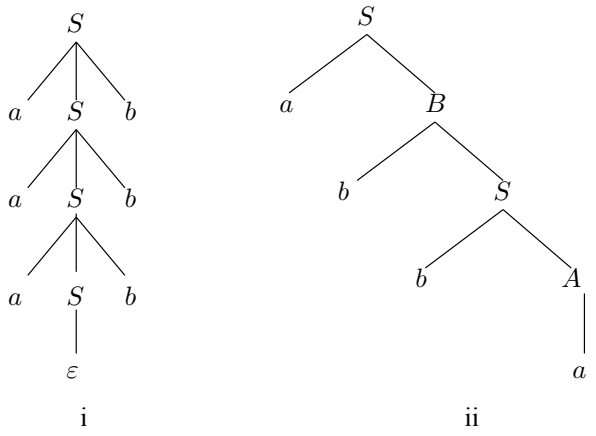
Ορισμός 5.6. Έστω $G = \{V, T, P, S\}$ μια c.f. γραμματική. Ένα δένδρο είναι συντακτικό δένδρο της G αν

1. Κάθε κόμβος του δένδρου έχει μια *επιγραφή*, η οποία είναι ένα σύμβολο στο $V \cup T \cup \{\varepsilon\}$.
2. Η επιγραφή της ρίζας είναι το S.
3. Αν ένας κόμβος είναι εσωτερικός και έχει επιγραφή A, τότε το πρέπει να είναι στοιχείο του V.

4. Αν ο κόμβος n έχει επιγραφή A και οι κόμβοι $n_1, n_2, \ldots, n_k$ είναι παιδιά του n, σε διάταξη από αριστερά προς τα δεξιά, με επιγραφές $X_1, X_2, \ldots, X_k$ αντίστοιχα, τότε ο $A \to X_1 X_2 \ldots X_k$ πρέπει να είναι κανόνας παραγωγής στο P.

5. Αν ένας κόμβος έχει επιγραφή ε, τότε είναι φύλλο και είναι το μοναδικό παιδί του γονέα του.

Για παράδειγμα, τα συντακτικά δένδρα των ακολουθιών παραγωγών των παραδειγμάτων 5.3 και 5.5 φαίνονται στο σχήμα 5.1.



Σχήμα 5.1: Παραδείγματα συντακτικών δένδρων

Ονομάζουμε *φύλλωμα (leaf string)* τη συμβολοακολουθία που προκύπτει από τα φύλλα ενός δένδρου, αν τα διατρέξουμε από το αριστερότερο προς το δεξιότερο. Ορίζουμε ως *A-δένδρο* ενός συντακτικού δέντρου ένα υποδέντρο το οποίο περιλαμβάνει ως ρίζα έναν κόμβο με επιγραφή A καθώς και όλους τους απογόνους αυτού του κόμβου και τις μεταξύ τους ακμές.

Θεώρημα 5.7. *Έστω $G(V, T, P, S)$ μια c.f. γραμματική. Τότε $S \stackrel{*}{\Rightarrow} \alpha$ ανν υπάρχει συντακτικό δένδρο με φύλλωμα το α.*

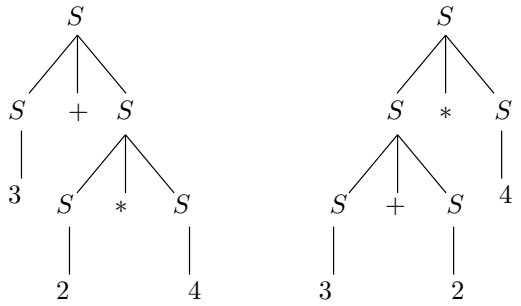
Η απόδειξη της κατεύθυνσης "$\Leftarrow$" γίνεται με επαγωγή ως προς τον αριθμό των εσωτερικών κόμβων, ενώ της "$\Rightarrow$" με επαγωγή ως προς των αριθμό των βημάτων της ακολουθίας παραγωγών (άσκηση).

Διφορούμενες γραμματικές (ambiguous grammars)

Ορισμός 5.8. Μια γραμματική G ονομάζεται *διφορούμενη (ambiguous)* αν υπάρχουν δύο συντακτικά δένδρα που να έχουν ως φύλλωμα ένα $w \in L(G)$.

Για παράδειγμα, η συμβολοακολουθία $3 + 2 * 4$ ανήκει στην $L(G_2)$, του παραδείγματος 5.4, σελίδα 88 και υπάρχουν δύο συντακτικά δένδρα που αντιστοιχούν σε αυτήν, όπως φαίνεται στο σχήμα 5.2. Παρ'όλα αυτά, μπορούμε να κατασκευάσουμε

μια γραμματική που είναι ισοδύναμη με την G_2 και η οποία *δεν* είναι διφορούμενη (άσκηση).



Σχήμα 5.2: Συντακτικά δένδρα διφορούμενης γραμματικής

Υπάρχουν όμως και γλώσσες χωρίς συμφραζόμενα, για τις οποίες όλες οι γραμματικές που τις παράγουν είναι αναγκαστικά διφορούμενες:

Ορισμός 5.9. Μια γλώσσα χωρίς συμφραζόμενα ονομάζεται *εγγενώς διφορούμενη* (inherently ambiguous) αν όλες οι γραμματικές που την παράγουν είναι διφορούμενες.

Παράδειγμα 5.10. Η γλώσσα χωρίς συμφραζόμενα $\{a^i b^j c^k \mid i = j \vee j = k\}$ είναι εγγενώς διφορούμενη.

5.3.2 Απλοποίηση και κανονικές μορφές

Είναι δυνατόν να απλοποιήσουμε μία γραμματική χωρίς συμφραζόμενα ως εξής: Κρατάμε μόνον τα σύμβολα που χρειάζονται (για παράδειγμα εξαλείφουμε όλα τα μη τερματικά που δεν παράγουν συμβολοακολουθίες με τερματικά και όλα τα σύμβολα που δεν μπορούν να εμφανισθούν σε καμία παραγωγή που ξεκινάει από το αρχικό σύμβολο). Επίσης, μπορούμε να εξαλείψουμε κανόνες του τύπου $A \to B$ (unit productions).

Επιπλέον, αν το ε δεν ανήκει στην L, δεν χρειάζονται κανόνες παραγωγής της μορφής $A \to \varepsilon$ (ε-productions). Μάλιστα, αν το ε δεν ανήκει στην L, μπορούμε να κατασκευάσουμε την G με τέτοιο τρόπο ούτως ώστε κάθε κανόνας παραγωγής να έχει είτε τη μορφή $A \to BC$ είτε την $A \to a$, όπου οι A, B και C είναι μεταβλητές και το a είναι τερματικό σύμβολο. Εναλλακτικά, μπορούμε να μετατρέψουμε κάθε κανόνα παραγωγής του G στη μορφή $A \to a$, όπου το α είναι μια συμβολοακολουθία από μεταβλητές (μπορεί και κενή). Αυτές οι δύο ειδικές μορφές ονομάζονται *κανονική μορφή Chomsky* και *κανονική μορφή Greibach*, οπότε έχουμε τα παρακάτω δύο θεωρήματα κανονικών μορφών:

Θεώρημα 5.11 (Κανονικής μορφής Chomsky, CNF). *Κάθε c.f. γλώσσα χωρίς το ε παράγεται από μια γραμματική στην οποία όλες οι παραγωγές είναι της μορφής $A \to BC$ ή $A \to a$. όπου , , C μεταβλητές και a τερματικό.*

Θεώρημα 5.12 (Κανονικής μορφής Greibach, GNF). *Κάθε c.f. γλώσσα χωρίς το ε παράγεται από μια γραμματική στην οποία όλες οι παραγωγές είναι της μορφής $A \to a$, όπου $\alpha \in V^*$, $a \in T$.*

Από γραμματικές στις παραπάνω κανονικές μορφές, προκύπτουν σχετικά απλούστερα συντακτικά δένδρα: για την CNF τα συντακτικά δένδρα έχουν πάντοτε διακλάδωση βαθμού δύο αν προκύπτουν μεταβλητές, αλλιώς από μία μεταβλητή προκύπτει ένα μόνον φύλλο (τερματικό σύμβολο), ενώ για την GNF, τα αριστερά παιδιά κάθε κόμβου είναι πάντοτε τερματικά σύμβολα.

Μπορούμε να εκμεταλλευτούμε αυτές τις ιδιότητες των συντακτικών δένδρων για να επιλύσουμε ταχύτερα ορισμένα προβλήματα όπως αν κάποια συμβολοσειρά x ανήκει στην γλώσσα που παράγει μία γραμματική: Δεδομένης γραμματικής χωρίς συμφραζόμενα G, όχι απαραίτητα σε κανονική μορφή, υπάρχει μηχανιστικός αλγόριθμος ο οποίος για οποιαδήποτε συμβολοσειρά x αποκρίνεται αν $x \in L(G)$ ή όχι. Π.χ. αν συστηματικά κατασκευάσουμε όλες τις παραγόμενες συμβολοσειρές κατά αύξουσα σειρά μήκους, τότε μπορούμε να αποφασίσουμε εάν $x \in L(G)$. Ο αλγόριθμος όμως είναι εκθετικού χρόνου ως προς το μήκος της συμβολοσειράς εισόδου. Αν όμως η γραμματική δίνεται σε κανονική μορφή Chomsky, τότε υπάρχει ταχύτερος αλγόριθμος, πολυπλοκότητας $O(|x|^3)$, ο λεγόμενος αλγόριθμος CYK (από τους Cocke, Younger, Kasami).

function CYK(x: string): **boolean** *(* assumes Chomsky n.f. *)*
begin $n := |x|$
  for $i := 1$ **to** n **do**
    $V_i^1 := \{A \mid (A \to a) \in P \wedge (x)_i = a\}$;
  for $j := 2$ **to** n **do**
    for $i := 1$ **to** $n - j + 1$ **do**
    begin $V_i^j := \emptyset$;
      for $k := 1$ **to** $j - 1$ **do**
      $V_i^j := V_i^j \cup \{A \mid (A \to BC) \in P \wedge B \in V_i^k \wedge C \in V_{i+k}^{j-k}\}$
    end ;
  CYK $:= S \in V_1^n$
end

Ο αλγόριθμος ουσιαστικά ελέγχει όλα τα δυνατά συντακτικά δένδρα, αρχίζοντας από τα τερματικά σύμβολα στα φύλλα και αποδίδοντας σε αυτά πιθανά μη τερματικά σύμβολα που τα παραγάγουν. Ο αλγόριθμος συνεχίζει αποδίδοντας πιθανά μη τερματικά σύμβολα σε κάθε υποσυμβολοσειρά της συμβολοσειράς εισόδου και τέλος ελέγχει αν στην συμβολοσειρά εισόδου έχει αποδοθεί το αξίωμα S.

Παράδειγμα 5.13. Έστω γραμματική σε κανονική μορφή Chomsky:

$$S \to AB \mid BC, \quad A \to BA \mid a, \quad B \to CC \mid b, \quad C \to AB \mid a$$

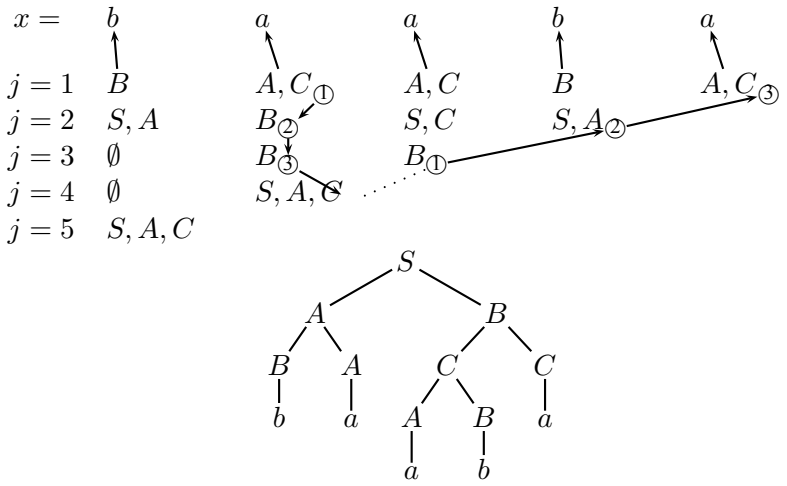
Στο σχήμα 5.3 δίνουμε την εκτέλεση του αλγορίθμου και το αντίστοιχο συντακτικό δένδρο για είσοδο $x = baaba$. Με τα βέλη, δείχνουμε την ακολουθία υπολογισμού για $j = 4$ (μήκος substring) και για το substring $(x)_{2..5}$ (αρχίζει, δηλαδή, από την θέση $i = 2$): Μέσα σε κύκλο και με τον ίδιο αριθμό συμβολίζουμε τα δύο substrings (συνολικού μήκους 4) που συνδυάζονται για να δώσουν το $(x)_{2..5}$.

5.3.3 Αυτόματα στοίβας

Ένα *αυτόματο στοίβας* (push down automaton ή για συντομία PDA) αποτελείται από μία ταινία εισόδου, αλλά επιπλέον, σε σχέση με τα FA, έχει και μία *στοίβα* (μη φραγμένη σε μέγεθος μνήμη, αλλά με περιορισμένες δυνατότητες πρόσβασης σε αυτήν). Η πρόσβαση στην στοίβα γίνεται μόνον στην κορυφή αυτής με τις εξής δύο λειτουργίες:

1. push: Τοποθετεί ένα στοιχείο που δίνεται στην κορυφή της στοίβας.
2. pop: Αφαιρεί ένα στοιχείο για χρήση από την κορυφή της στοίβας.

Είναι προφανές ότι αν κάποιο στοιχείο βρίσκεται βαθιά μέσα στην στοίβα, δηλαδή αφού έχει μπει στην στοίβα με push, έχουν επίσης μπει με push άλλα στοιχεία πάνω



Σχήμα 5.3: Εκτέλεση αλγορίθμου CYK

από αυτό, προκειμένου να έχουμε πρόσβαση σε αυτό θα πρέπει να κάνουμε pop σε όλα τα στοιχεία που είναι από πάνω του.

Θεωρήστε την γλώσσα $L = \{wcw^R \mid w \in (0+1)^*\}$. Για παράδειγμα $110c011 \in L$. Να πώς μπορούμε να αναγνωρίσουμε την παραπάνω γλώσσα με ένα PDA:

1. push(a) στην στοίβα για κάθε 0 που συναντάς στην είσοδο,
   push(b) στην στοίβα για κάθε 1 που συναντάς στην είσοδο,
   μέχρι να συναντήσεις το c

2. διάβασε το c,
   κάνε pop τα στοιχεία της στοίβας και διάβαζε παράλληλα την είσοδο,
   αποδέξου αν υπάρχει συμφωνία των στοιχείων εισόδου με τα στοιχεία της στοίβας (a με 0, b με 1).

Δίνουμε παρακάτω τον τυπικό ορισμό:

Ορισμός 5.14. Ένα αυτόματο στοίβας (push down automaton ή για συντομία PDA) είναι μία πλειάδα $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, όπου:

- Q: πεπερασμένο σύνολο καταστάσεων,
- Σ: αλφάβητο εισόδου,
- Γ: αλφάβητο στοίβας,
- δ: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Pow(Q \times \Gamma^*)$ (πεπερασμένα υποσύνολα), η συνάρτηση μετάβασης (επιτρέπονται ε-κινήσεις και μη ντετερμινισμός),
- $q_0 \in Q$: αρχική κατάσταση,

- $Z_0 \in \Gamma$: αρχικό σύμβολο στην στοίβα,
- $F \subseteq Q$: τελικές καταστάσεις.

Υπάρχουν δύο είδη PDA ως προς το αποδέχεσθαι:

1. αποδέξου όταν βρίσκεσαι σε τελική κατάσταση αφού έχεις διαβάσει όλη την ταινία εισόδου, ανεξάρτητα από το τι υπάρχει στην στοίβα, ή,
2. αποδέξου όταν η στοίβα είναι άδεια αφού έχεις διαβάσει όλη την ταινία εισόδου, ανεξάρτητα από την κατάσταση στην οποία ευρίσκεσαι (σύμβαση: $F = \emptyset$).

Αντίστοιχα ορίζουμε και την γλώσσα που αποδέχεται ένα PDA:

1. Γλώσσα που αποδέχεται σε τελική κατάσταση $L_f(M)$
2. Γλώσσα που αποδέχεται με κενή στοίβα $L_e(M)$

Προκειμένου να γίνει αποδεκτή η $L_1 = \{ww^R \mid w \in (0+1)^*\}$ χωρίς το σημάδι c στην μέση της συμβολοσειράς χρειαζόμαστε απαραιτήτως ένα μη ντετερμινιστικό PDA. Τα μη ντετερμινιστικά PDA είναι γνησίως πιο ισχυρά από τα ντετερμινιστικά.

Τελικά ισχύει το παρακάτω θεώρημα:

Θεώρημα 5.15. *Τα παρακάτω είναι ισοδύναμα για μία γλώσσα L:*

1. *$L = L_f(M_2)$, M_2 είναι PDA.*
2. *$L = L_e(M_1)$, M_1 είναι PDA.*
3. *Η L είναι γλώσσα χωρίς συμφραζόμενα (context free).*

Παραλείπεται η λεπτομερής απόδειξη.

5.4 Γενικές γραμματικές

Τύπου 0: *γενικές γραμματικές (general or unrestricted grammars)*, semi-Thue, phrase structure

Παραγωγές: $\alpha \to \beta$, με $\alpha \neq \varepsilon$

Παράδειγμα 5.16. $L = \{a^{2^n} \mid n \in N\}$
$S \to AaCB$
$CB \to E \mid DB$
$aE \to Ea$
$AE \to \varepsilon$
$aD \to Da$
$AD \to AC$
$Ca \to aaC$

Θεώρημα 5.17. *Τα ακόλουθα είναι ισοδύναμα:*

1. *η L γίνεται αποδεκτή από μία Turing μηχανή (βλέπε κεφάλαιο 7, ενότητα 7.1)*
2. $L = L(G)$, *όπου G είναι γενική γραμματική*

Χωρίς απόδειξη. Μία τέτοια γλώσσα λέγεται και *αναδρομικά αριθμήσιμη* (recursively enumerable)· βλέπε ορισμό 8.3 στην σελίδα 112.

5.5 Γραμματικές με συμφραζόμενα

Τύπου 1: *γραμματικές με συμφραζόμενα (context sensitive grammars, c.s.)*
Παραγωγές: $\alpha \to \beta$, με $\alpha \neq \varepsilon, |\alpha| \leq |\beta|$ (noncontracting grammar, non-decreasing, not ε string)
Η ονομασία context sensitive οφείλεται στην παρακάτω εναλλακτική περιγραφή αυτών των γραμματικών: Κάθε c.s. γραμματική μπορεί να τεθεί σε κανονική μορφή στην οποία όλοι κανόνες παραγωγής είναι της μορφής:

$$\underset{\substack{\nwarrow \quad \nearrow \\ context}}{\alpha_1 A \alpha_2} \to \alpha_1 \beta \alpha_2, \qquad \text{όπου } A\text{: μη τερματικό και } \beta \neq \varepsilon$$

Παράδειγμα 5.18. $1^n 0^n 1^n$. C.s. γραμματική:
$S \to 1Z1$
$Z \to 0 \mid 1Z0A$
$A0 \to 0A$
$A1 \to 11$

'Αλλα παραδείγματα τέτοιων γλωσσών: $\{a^n b^n c^n\}$, $\{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$, $\{ww \mid w \in \Sigma^*\}$, $0^n 1^n 0^n 1^n$.

Γραμμικά φραγμένο αυτόματο (Linearly bounded automaton (LBA)): Είναι μία μη-ντετερμινιστική μηχανή Turing (T.M.) της οποίας η κεφαλή είναι περιορισμένη να κινείται μόνον στο τμήμα της ταινίας που περιέχει την είσοδο.

Θεώρημα 5.19. *Τα ακόλουθα είναι ισοδύναμα (L χωρίς ε):*

1. *η L γίνεται αποδεκτή από LBA*
2. *η L είναι c.s.*

Χωρίς απόδειξη.

Ένα ανοιχτό ερώτημα είναι το εξής: Είναι ισοδύναμη η ντετερμινιστική και η μη ντετερμινιστική εκδοχή του LBA;

Ασκήσεις

1. Σωστό ή λάθος (με εξήγηση);

   (i) Κάθε γλώσσα που παράγεται από κανονική γραμματική αναγνωρίζεται από PDA.

   (ii) Κάθε γλώσσα με συμφραζόμενα αναγνωρίζεται από DFA.

   (iii) Υπάρχει LBA για τη γλώσσα $\{0^{5k} \mid k \in \mathbb{N}\}$.

2. Αποδείξτε ή ανταποδείξτε τις ιδιότητες κλειστότητας της κλάσης των γλωσσών χωρίς συμφραζόμενα (context-free) ως προς ένωση, παράθεση, τομή και άστρο του Kleene. Τι συμπέρασμα προκύπτει για την πράξη του συμπληρώματος;

3. Αποδείξτε ή ανταποδείξτε την ιδιότητα κλειστότητας της κλάσης των γλωσσών χωρίς συμφραζόμενα (context-free) ως προς την πράξη της αναστροφής.

4. Κατατάξτε τις παρακάτω γλώσσες στην Ιεραρχία Chomsky με όσο μεγαλύτερη ακρίβεια μπορείτε (με απόδειξη ή εξήγηση, όσο καλύτερα μπορείτε):

   (i) $\{0^k 1^{k+1} 0^k \mid k \in \mathbb{N}\}$

   (i) $\{0^{3k+1} \mid k \in \mathbb{N}\}$

   (ii) $\{a^k b^m c^n \mid k, m, n \in \mathbb{N}, k < m \leq n\}$

   (i) $\{w \in \{a, b\}^* \mid w \pi\varepsilon\rho\iota\acute{\varepsilon}\chi\varepsilon\iota\text{`}1101'\}$

5. Δώστε περιγραφή υψηλού επιπέδου (όσο καλύτερα μπορείτε) ενός LBA για τις παρακάτω γλώσσες:

   (i) $L = \{a^p : p \text{ είναι δύναμη του } 2\ \}$.

   (ii) $L = \{a^p : p \text{ είναι πρώτος αριθμός}\}$.

6. Αποδείξτε ή ανταποδείξτε τις ιδιότητες κλειστότητας της κλάσης των γλωσσών με συμφραζόμενα (context-sensitive) ως προς ένωση, παράθεση, τομή και άστρο του Kleene. Τι συμπέρασμα προκύπτει για την πράξη του συμπληρώματος;

7. Κατατάξτε τις παρακάτω γλώσσες σε κλάσεις της Ιεραρχίας Chomsky (στην μικρότερη κλάση που ανήκουν) και αποδείξτε τον ισχυρισμό σας δίνοντας κατάλληλα αυτόματα και γραμματικές (όσο καλύτερα μπορείτε). Για όσες θεωρείτε ότι δεν είναι κανονικές, αποδείξτε το γεγονός αυτό με χρήση του Pumping Lemma.

   $L_1 = \{w \in \{0,1\}^* \mid w = u100u^R, \quad u \in \{0,1\}^*\}$, όπου u^R η ανάστροφη συμβολοσειρά της u.

   $L_2 = \{w \in \{0,1\}^* \mid w$ παριστάνει δυαδικό αριθμό που είναι πολλαπλάσιο του 2 αλλά όχι του 4 $\}$

   $L_3 = \{w \in \{a,b,c\}^* \mid w = a^k b^m c^l, \quad k,m,l \in \mathbb{N}, k < m < l\}$

Κεφάλαιο 6

Λογική στην Επιστήμη των Υπολογιστών

6.1 Προτασιακή Λογική

Η γλώσσα της μαθηματικής λογικής στηρίζεται βασικά στις εργασίες του Boole και του Frege. Ο Προτασιακός Λογισμός περιλαμβάνει στο αλφάβητό του, εκτός από τα σύμβολα προτασιακών μεταβλητών, τα λογικά σημάδια ζεύξης: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), …

Στον προτασιακό λογισμό ονομάζουμε ατομικούς τύπους τις σταθερές TRUE και FALSE καθώς και τις προτασιακές μεταβλητές π.χ. $x_1, x_2, \ldots$

Οι **προτασιακοί τύποι** ορίζονται **επαγωγικά**:

1. Οι ατομικοί τύποι είναι τύποι.
2. Αν Φ είναι τύπος τότε και ο $\neg\Phi$ είναι τύπος.
3. Αν οι Φ και Ψ είναι τύποι τότε και οι $(\Phi \wedge \Psi)$ και $(\Phi \vee \Psi)$ είναι τύποι.
4. Ο,τιδήποτε δεν ορίζεται με βάση τα (1)–(3) δεν είναι προτασιακός τύπος.

Παρατηρήσεις:

- Μερικές φορές παραλείπουμε παρενθέσεις και υποθέτουμε αριστερό προσεταιρισμό π.χ. $x_1 \wedge x_2 \wedge \neg x_3$
- Μπορούμε να ορίσουμε νέους τύπους ως **συντομογραφία** άλλων γνωστών π.χ.:

$$(\Phi \rightarrow \Psi) :\equiv (\neg\Phi \vee \Psi)$$
$$(\Phi \leftrightarrow \Psi) :\equiv (\Phi \rightarrow \Psi) \wedge (\Psi \rightarrow \Phi)$$

Οι προτασιακοί τύποι είναι συντακτικές συμβολοσειρές που όμως έχουν κάποια σημασία (**σημασιολογία**) δηλαδή είναι αληθείς ή ψευδείς ανάλογα με τις αληθοτιμές που έχουν απονεμηθεί στις προτασιακές μεταβλητές. Πιο συγκεκριμένα αληθοτιμές των τυπών $\neg\Phi$, $(\Phi \wedge \Psi)$ και $(\Phi \vee \Psi)$ ορίζονται από τις αληθοτιμές των Φ, Ψ όπως φαίνεται στον παρακάτω πίνακα αληθείας (truth table):

Φ	Ψ	$\neg\Phi$	$(\Phi \wedge \Psi)$	$(\Phi \vee \Psi)$
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE		FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE		FALSE	FALSE

Ένας τύπος λέγεται **έγκυρος** (valid) ή **ταυτολογία** αν είναι αληθής για κάθε απονομή αληθοτιμών στις μεταβλητές. Ένας τύπος λέγεται **ικανοποιήσιμος** (satisfiable) αν υπάρχει απονομή αληθοτιμών που τον καθιστά αληθή. 'Αρα Φ είναι ικανοποιήσιμος **εάν και μόνο εάν** ο $\neg\Phi$ δεν είναι ταυτολογία.

Μια προτασιακή μεταβλητή ή άρνηση προτασιακής μεταβλητής ονομάζεται **λέκτημα** (literal). Μια **φράση** (clause) είναι μια διάζευξη από literals (π.χ. $x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$).

Κάθε τύπος της προτασιακής λογικής είναι ισοδύναμος με κάποιον που βρίσκεται σε **συζευκτική κανονική μορφή** (conjunctive normal form) δηλαδή είναι μια σύζευξη από διαζευκτικές φράσεις. Αντίστοιχα είναι επίσης ισοδύναμος με τύπο που βρίσκεται σε **διαζευκτική** κανονική μορφή (disjunctive normal form) δηλαδή είναι μια διάζευξη από συζευκτικές φράσεις.

Μια φράση λέγεται **φράση Horn** αν έχει το πολύ ένα **θετικό** literal δηλαδή είναι της μορφής:

$$(x_0 \vee \neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n) \text{ ή } (x_0) \text{ ή } (\neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n)$$

που γράφεται ισοδύναμα:

$$(x_1 \wedge x_2 \wedge \cdots \wedge x_n \rightarrow x_0), (\text{TRUE} \rightarrow x_0), (x_1 \wedge x_2 \wedge ... \wedge x_n \rightarrow \text{FALSE}),$$

αντίστοιχα.

6.2 Κατηγορηματικός Λογισμός

Ο κατηγορηματικός λογισμός έχει επί πλέον σύμβολα γιά μεταβλητές, σταθερές, κατηγορήματα, συναρτήσεις και τους ποσοδείκτες: $\exists$ (υπαρξιακός) και $\forall$ (καθολικός).

Για να ορίσουμε συντακτικά τις προτάσεις είναι αναγκαίο πρώτα να ορίσουμε τους **όρους** (οι οποίοι όταν τους αποδοθεί σημασία θα ερμηνεύονται σαν **αντικείμενα** από κάποιο σύνολο). Οι όροι ορίζονται επαγωγικά:

1. Οι μεταβλητές και οι σταθερές είναι όροι.
2. Αν $t_1, t_2, ..., t_n$ είναι όροι και f είναι σύμβολο συνάρτησης n θέσεων τότε f($t_1, t_2, ..., t_n$) είναι επίσης όρος.
3. Τίποτε άλλο δεν είναι όρος.

Οι **προτάσεις** του κατηγορηματικού λογισμού ορίζονται επαγωγικά:

1. Αν $t_1, t_2, ..., t_n$ είναι όροι και P είναι σύμβολο κατηγορήματος n θέσεων τότε $P(t_1, t_2, ..., t_n)$ είναι πρόταση (ατομική πρόταση).
2. Αν οι Φ και Ψ είναι προτάσεις και x είναι μεταβλητή τότε και οι $\neg\Phi$, $(\Phi \wedge \Psi)$, $(\Phi \vee \Psi)$, $\forall x\Phi$, $\exists x\Phi$ είναι προτάσεις.
3. Τίποτε άλλο δεν είναι πρόταση.

Οι σταθερές και οι μεταβλητές **ερμηνεύονται** σαν στοιχεία ενός συνόλου A. Τα συναρτησιακά σύμβολα ερμηνεύονται σαν συναρτήσεις: $A^n \to A$. Έτσι κάθε όρος ερμηνεύεται σαν ένα στοιχείο του A.

Τα κατηγορήματα ερμηνεύονται σαν υποσύνολα του A^n. Η πρόταση $P(t_1, t_2, ..., t_n)$ είναι αληθής ανν $(s_1, s_2, ..., s_n) \in R$ όπου $s_1, s_2, ..., s_n$ είναι τα στοιχεία του A με τα οποία ερμηνεύονται οι όροι $t_1, t_2, ..., t_n$ και R το υποσύνολο με το οποίο ερμηνεύεται το P. Οι αληθοτιμές των $\neg\Phi$, $(\Phi \wedge \Psi)(\Phi \vee \Psi)$ ορίζονται από τις αληθοτιμές των Φ και Ψ όπως και στην προτασιακή λογική. Η πρόταση $\forall x\Phi$ είναι αληθής αν η πρόταση Φ είναι αληθής για **οποιαδήποτε** ερμηνεία της μεταβλητής x, ενώ η πρόταση $\exists x\Phi$ είναι αληθής αν η Φ αληθεύει για **κάποια** ερμηνεία της x.
Οι φράσεις Horn για τον κατηγορηματικό λογισμό ορίζονται όπως και στην προτασιακή λογική αν αντί για προτασιακές μεταβλητές χρησιμοποιούμε ατομικές προτάσεις. Ένα πρόγραμμα $Prolog$ είναι βασικά μία σύξευξη από φράσεις Horn.

6.3 Πρωτοβάθμια Λογική

Όπως είπαμε και προηγουμένως η γλώσσα της πρωτοβάθμιας λογικής (ή αλλιώς κατηγορηματικού λογισμού) περιέχει:

- όλα τα σύμβολα που περιέχει ο προτασιακός λογισμός
- επιπλέον σύμβολα για συναρτήσεις και σταθερές, π.χ. $f, g, h, c_1, c_2, ...,$
- σύμβολα για κατηγορήματα π.χ. $P, Q, =, ...$
- και τους ποσοδείκτες : καθολικό $\forall$ και υπαρξιακό $\exists$.

Οι μεταβλητές εδώ ερμηνεύονται σαν στοιχεία κάποιου συνόλου, όχι σαν αληθοτιμές.

Θα ορίσουμε **επαγωγικά** τους **όρους** και τους **τύπους** της πρωτοβάθμιας λογικής.

Όροι:

1. Οι μεταβλητές και οι σταθερές είναι όροι.
2. Αν f είναι σύμβολο συνάρτησης n θέσεων και $t_1, ..., t_n$ είναι όροι τότε όρος είναι και ο $f(t_1, ..., t_n)$.
3. Τίποτα άλλο.

Τύποι:

1. Αν P είναι σύμβολο κατηγορήματος n θέσεων και $t_1, ..., t_n$ είναι όροι τότε $P(t_1, ..., t_n)$ και $t_1 = t_2$ είναι ατομικοί τύποι.
2. Αν οι Φ και Ψ είναι τύποι και x μεταβλητή τότε τύποι είναι και οι: $\neg\Phi$, $(\Phi \vee \Psi)$, $(\Phi \wedge \Psi)$, $\forall x\Phi$, $\exists x\Phi$.
3. Τίποτα άλλο.

Σημειώση: μια σταθερά c μπορεί να θεωρηθεί συνάρτηση 0 θέσεων.

Η **εμβέλεια** του $\forall x$ (ή $\exists x$) στον τύπο $\forall x\Phi$ (ή αντίστοιχα $\exists x\Phi$) είναι ο **υποτύπος** Φ.

Ελεύθερη εμφάνιση της μεταβλητής x στον τύπο Φ λέγεται μια εμφάνιση της μεταβλητής x που δεν είναι μέσα στην εμβέλεια ενός ποσοδείκτη $\forall x$ ή $\exists x$.

Δεσμευμένη εμφάνιση της x είναι μέσα στην εμβέλεια ενός ποσοδείκτη ή και ακριβώς δεξιά του συμβόλου $\forall$ (ή $\exists$).

Ένας τύπος λέγεται **κλειστός** αν δεν περιέχει ελεύθερες εμφανίσεις μεταβλητών.

Η **σημασιολογία** τύπων του κατηγορηματικού λογισμού δίνεται με την βοήθεια των αλγεβρικών δομών A που ονομάζουμε **μοντέλα**. Στην περίπτωση του προτασιακού λογισμού το πεδίο A είναι $\{\text{True}, \text{False}\}$, εδώ όμως μπορεί να είναι οποιοδήποτε μή κενό, πεπερασμένο ή και άπειρο, σύνολο. Εδώ λοιπόν δεν έχουμε **απονομή** αληθοτιμών αλλά **ερμηνεία** (interpretation) των μεν σταθερών και μεταβλητών με στοιχεία του πεδίου A, των δε συναρτησιακών και κατηγορηματικών συμβόλων με πραγματικές **απεικονίσεις** και **σχέσεις** μεταξύ των στοιχείων του πεδίου

A. Με τέτοια σημασιολογία κάθε όρος ερμηνεύεται με στοιχείο του A και κάθε κλειστός τύπος αληθεύει (ή όχι) στο μοντέλο A.

Συμβολίζουμε $\Gamma \vdash \Phi$ το γεγονός ότι ο τύπος Φ αποδεικνύεται **συντακτικά** από τους τύπους του συνόλου Γ.

Συμβολίζουμε $\Gamma \models \Phi$ το γεγονός ότι ο τύπος Φ αληθεύει σε όλα τα μοντέλα όπου αληθεύουν και οι τύποι του συνόλου Γ.

Το περίφημο **θεώρημα πληρότητας** του Gödel λέει:

$$\Gamma \vdash \Phi \quad \text{ανν} \quad \Gamma \models \Phi$$

Αφ' ετέρου το **θεώρημα μη πληρότητας** του Gödel λέει:

Δεν μπορεί να υπάρξει **συνεπής και πλήρης αξιωματικοποίηση** όλων των αληθών τύπων της Αριθμητικής.

Κεφάλαιο 7

Υπολογιστικά μοντέλα

7.1 Μηχανές Turing

Μια μηχανή *Turing* (TM) είναι ένα απλός ιδεατός υπολογιστής, δηλαδή ένα υπολογιστικό μοντέλο. Ας θεωρήσουμε μια πεπερασμένη μηχανική συσκευή με μια ταινία που προεκτείνεται (δυνητικά) μέχρι το άπειρο και προς τις δύο κατευθύνσεις και υποδιαιρείται σε κύτταρα που το καθένα περιέχει 1 ή 0, δηλαδή το αλφάβητο της μηχανής είναι το $\Sigma = \{0, 1\}$. Σε κάθε χρονική στιγμή η κεφαλή της TM βρίσκεται σε ένα κύτταρο, το οποίο θα λέγεται το τρέχον.

Οι βασικές λειτουργίες μιας TM είναι:

- Διάβασε το περιεχόμενο του τρέχοντος κυττάρου
- Γράψε 1 ή 0 στο τρέχον κύτταρο
- Κάνε τρέχον το αμέσως αριστερότερο ή το αμέσως δεξιότερο κύτταρο

Η TM έχει ένα πεπερασμένο αριθμό εσωτερικών καταστάσεων (internal states):

$$Q = \{q_0, q_1, \ldots\}$$

Ένα πρόγραμμα για μια TM είναι ένα σύνολο από τετράδες της μορφής q_i, e, d, q_j όπου $q_i, q_j \in Q, e \in \Sigma$, $d \in A = \Sigma \bigcup \{L, R\}$ με τον εξής συναρτησιακό (ντετερμινιστικό) περιορισμό: Για κάθε $< q_i, e >$ υπάρχει το πολύ ένα $< d, q_j >$ έτσι ώστε η τετράδα $< q_i, e, d, q_j >$ να ανήκει στο πρόγραμμα, δηλαδή πρόκειται για μια **συνάρτηση μετάβασης** (*transition function*) $\delta : Q \times \Sigma \to A \times Q$.

Η συνάρτηση μετάβασης καθορίζει με βάση την παρούσα κατάσταση και το περιεχόμενο του τρέχοντος κυττάρου ποια από τις βασικές λειτουργίες θα εκτελεστεί και ποια θα είναι η επόμενη κατάσταση. Κατά σύμβαση η μηχανή σταματάει στο ζεύγος κατάστασης-συμβόλου $< q_i, e >$ σε περίπτωση που η τιμή $\delta(q_i, e)$ δεν είναι ορισμένη.

$< q_0$	1	0	$q_1 >$
$< q_1$	0	R	$q_2 >$
$< q_2$	1	0	$q_3 >$ | halt για $< q_2 \;\; 0 >$
$< q_3$	0	R	$q_4 >$
$< q_4$	1	R	$q_4 >$
$< q_4$	0	R	$q_5 >$
$< q_5$	1	R	$q_5 >$
$< q_5$	0	1	$q_6 >$
$< q_6$	1	R	$q_6 >$
$< q_6$	0	1	$q_7 >$
$< q_7$	1	L	$q_7 >$
$< q_7$	0	L	$q_8 >$
$< q_8$	1	L	$q_8 >$
$< q_8$	0	R	$q_2 >$

Πίνακας 7.1: TM πρόγραμμα.

7.1.1 Μηχανές με εναδική (unary) αναπαράσταση αριθμών

Σε κάθε TM μπορούμε να αντιστοιχήσουμε μια μερική συνάρτηση από το $\mathbb{N}$ στο $\mathbb{N}$. Η είσοδος $n \in \mathbb{N}$ παριστάνεται με $\underline{n+1}$ συνεχόμενα 1 (έτσι ο αριθμός 0 παριστάνεται με 1). Σαν αρχικό στιγμιότυπο έχουμε την κεφαλή (τρέχον κύτταρο) να δείχνει στο αριστερότερο 1 και να βρίσκεται στην κατάσταση q_0. Σαν έξοδο λαμβάνουμε το συνολικό αριθμό από 1 που βρίσκεται στην ταινία, όταν και εάν η μηχανή σταματήσει.

Παράδειγμα 7.1. Να κατασκευαστεί μια TM που υπολογίζει το $2 * x$. Η TM θα εργάζεται σύμφωνα με το παρακάτω πρόγραμμα (το οποίο γράφεται πάντα πρώτα σε άτυπη γλώσσα υψηλού επιπέδου):

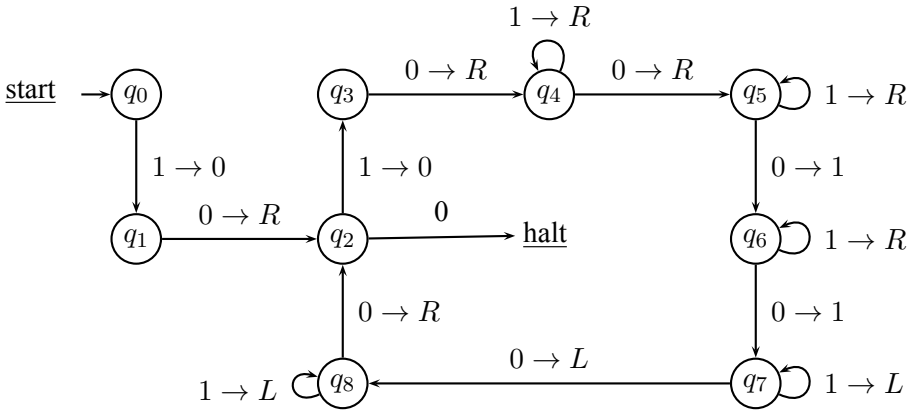
```
αρχικοποίηση; (*διαγραφή του πρώτου 1*)
while είσοδος<>0 do
 begin
   διάγραψε ένα 1 από είσοδο;
   μετακίνησε κεφαλή δεξιά πέρα από είσοδο και έξοδο;
   πρόσθεσε δύο 1 στην έξοδο;
   μετακίνησε κεφαλή αριστερά πέρα από έξοδο και είσοδο
 end
```

Το πρόγραμμα της TM φαίνεται στο πίνακα 7.1. Είναι φανερό ότι η μηχανή για κάθε ομάδα $x+1$ μονάδων με την οποία τροφοδοτείται, πρέπει να σβήνει μια και τις υπόλοιπες να τις διπλασιάζει. Ο διπλασιασμός μπορεί να γίνει με επανειλημμένες παλινδρομήσεις της κεφαλής, όπου κατά την κίνηση προς τα δεξιά σβήνει μια μονάδα από τα αριστερά της εισόδου και προσθέτει δύο στα δεξιά της εξόδου. Η είσοδος και η έξοδος χωρίζονται από ένα κενό. Στην επαναφορά ελέγχει αν έχουν

απομείνει μονάδες στην είσοδο. Η ίδια TM σε μορφή πίνακα φαίνεται στον πίνακα 7.2. Επίσης σε μορφή διαγράμματος καταστάσεων η ίδια TM φαίνεται στο σχήμα 7.1. Είναι δυνατό να απλοποιήσουμε το διάγραμμα αν παραλείψουμε τα ονόματα των καταστάσεων στους κόμβους. Επίσης δεν χρειάζονται οι ενδείξεις L (ή R) στις ακμές αν φέρουμε πάντοτε τις αντίστοιχες ακμές προς τα αριστερά (ή δεξιά αντίστοιχα) και κατακόρυφα όταν η κεφαλή δεν κινείται.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
0		R/q_2	halt	R/q_4	R/q_5	$1/q_6$	$1/q_7$	L/q_8	R/q_2
1	$0/q_1$		$0/q_3$		R/q_4	R/q_5	R/q_6	L/q_7	L/q_8

Πίνακας 7.2: TM σε μορφή πίνακα.



Σχήμα 7.1: TM σε μορφή διαγράμματος καταστάσεων.

Για συναρτήσεις με περισσότερα του ενός ορίσματα χρησιμοποιούμε το 0 σαν διαχωριστικό μεταξύ των διαφόρων εισόδων, π.χ. έχουμε την είσοδο $\ldots 0111011011110 \ldots$ για να παραστήσουμε το $(2, 1, 3)$.

7.1.2 Μηχανές με δυαδική (binary) αναπαράσταση αριθμών

Παράδειγμα 7.2. Μηχανή με αλφάβητο $\Sigma = \{0, 1, \sqcup\}$ (δυαδική αναπαράσταση αριθμού) που υπολογίζει τη συνάρτηση $x \mapsto x + 1$.

Περιγραφή *υψηλού επιπέδου:*

Κάνε τρέχον το κύτταρο με το τελευταίο σύμβολο της εισόδου x;
repeat
  Αν το τρέχον κύτταρο έχει $\sqcup$, γράψε 1 και σταμάτησε;
  Αν το τρέχον κύτταρο έχει 1, γράψε 0, κάνε τρέχον το αμέσως

αριστερότερο κύτταρο, και κρατούμενο $:= 1$;
Αν το τρέχον κύτταρο έχει 0, γράψε 1, κάνε τρέχον το αμέσως
αριστερότερο κύτταρο, και κρατούμενο $:= 0$;
until κρατούμενο $= 0$;
Κάνε τρέχον το κύτταρο με το πρώτο σύμβολο του $x + 1$ και σταμάτησε.

Συνάρτηση μετάβασης:

	0	1	$\sqcup$
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \sqcup, L)$
q_1	$(q_2, 1, L)$	$(q_1, 0, L)$	$(\text{HALT}, 1, S)$
q_2	$(q_2, 0, L)$	$(q_2, 1, L)$	$(\text{HALT}, \sqcup, R)$

Σημείωση: Για συντομία επιτρέπουμε ταυτόχρονα εγγραφή συμβόλου και κίνηση κεφαλής (το μοντέλο είναι ισοδύναμο). Για παράδειγμα, η παραπάνω συνάρτηση καθορίζει ότι αν η μηχανή βρίσκεται στην κατάσταση q_1 και το τρέχον σύμβολο είναι $'1'$ τότε η μηχανή παραμένει στην q_1, το τρέχον σύμβολο γίνεται $'0'$ και η κεφαλή μετακινείται αριστερά (κατά ένα κύτταρο).

Εκτέλεση με είσοδο 1011:

$$\begin{array}{lllll} (q_0, \underline{1}011) & \vdash & (q_0, 1\underline{0}11) & \vdash & (q_0, 10\underline{1}1) \quad \vdash \quad (q_0, 101\underline{1}) \vdash \\ (q_0, 1011\underline{\sqcup}) & \vdash & (q_1, 101\underline{1}) & \vdash & (q_1, 10\underline{1}0) \quad \vdash \quad (q_1, 1\underline{0}00) \vdash \\ (q_2, \underline{1}100) & \vdash & (q_2, \underline{\sqcup}1100) & \vdash & (\text{HALT}, \underline{1}100) \end{array}$$

Επεξηγήσεις: σε κάθε βήμα δίνεται μια περιγραφή της στιγμιαίας *συνολικής κατάστασης (configuration)* της TM, που αποτελείται από την τρέχουσα κατάσταση της μηχανής ακολουθούμενη από το περιεχόμενο της ταινίας. Το σύμβολο $\vdash$ λέγεται *μπάρα* (turnstile) και συμβολίζει τη μετάβαση από μία συνολική κατάσταση στην επόμενη.

Το υπογραμμισμένο σύμβολο δηλώνει το τρέχον κύτταρο, ενώ τα κενά αριστερά και δεξιά του περιεχομένου δεν αναγράφονται εκτός αν το τρέχον κύτταρο περιέχει το κενό.

Ένας υπολογισμός (computation) είναι μια έγκυρη ακολουθία συνολικών καταστάσεων και το αποτέλεσμα του υπολογισμού (έξοδος) είναι το περιεχόμενο της ταινίας όταν ο υπολογισμός σταματήσει.

Αποδεικνύεται ότι το μοντέλο με δυαδική αναπαράσταση αριθμών είναι ισοδύναμο με το μοντέλο της εναδικής αναπαράστασης (συνήθως όμως το "πρόγραμμα" είναι πιο σύντομο).

Κεφάλαιο 8

Υπολογισιμότητα (Computability) και Υπολογιστική Πολυπλοκότητα (Computational Complexity)

8.1 Ιστορία - Εισαγωγή

Η Συλλογιστική του Αριστοτέλη αποτέλεσε την πρώτη προσπάθεια θεμελίωσης της λογικής και των μαθηματικών. Ο *Leibni(t)z* πρότεινε το εξής πρόγραμμα:

1. Να δημιουργηθεί μια τυπική γλώσσα (*formal language*), με την οποία να μπορούμε να περιγράψουμε όλες τις μαθηματικές έννοιες και προτάσεις.

2. Να δημιουργηθεί μια μαθηματική θεωρία (δηλαδή ένα σύνολο από αξιώματα και συμπερασματικούς κανόνες συνεπαγωγής), με την οποία να μπορούμε να αποδεικνύουμε όλες τις ορθές μαθηματικές προτάσεις.

3. Να αποδειχθεί ότι αυτή η θεωρία είναι συνεπής (*consistent*), (δηλαδή ότι η πρόταση "Α και όχι Α" ($A \wedge \neg A$) δεν είναι δυνατόν να αποδειχθεί σ' αυτή τη θεωρία).

Η πραγμάτωση αυτού του προγράμματος άρχισε πολύ αργότερα, προς το τέλος του 19ου αιώνα. Πολλοί επιστήμονες ασχολήθηκαν με τον ορισμό της ενιαίας γλώσσας της μαθηματικής (ή συμβολικής) λογικής (*Boole, Frege*, κ.α). 'Αλλοι ασχολήθηκαν με τον ορισμό της ενιαίας θεωρίας των συνόλων (*Cantor*, κ.α.) και άλλοι με την παραγωγή (*derivation*) όλων των αληθών μαθηματικών προτάσεων με χρήση της Συνολοθεωρίας (*Russel, Whitehead*, κ.α.).

Στην αρχή αυτού του αιώνα ο *Hilbert* βάλθηκε να πραγματοποιήσει το 3ο μέρος του προγράμματος του *Leibni(t)z*, δηλαδή να βρει έναν αλγόριθμο που να αποκρίνεται

(*decides*) για την ορθότητα κάθε μαθηματικής πρότασης. Τελικά, όμως, το 1931 ο *Gödel* απέδειξε ότι:

- Δεν υπάρχει τέτοιος αλγόριθμος.
- Είναι αδύνατον να αποδειχθεί η συνέπεια της Συνολοθεωρίας.
- Επιπλέον, οποιαδήποτε (δηλαδή όχι μόνο η Συνολοθεωρία) αξιωματική θεωρία των Μαθηματικών, που περιλαμβάνει τουλάχιστον την Αριθμοθεωρία, θα περιλαμβάνει και μη αποκρίσιμες (*undecidable*) προτάσεις.
- Κωδικοποιώντας προτάσεις με φυσικούς αριθμούς (αυτή η κωδικοποίηση λέγεται σήμερα "Γκεντελοποίηση": *Gödelization*) μπόρεσε να παρουσιάσει μια συγκεκριμένη πρόταση που είναι μη αποκρίσιμη.

Το αποτέλεσμα αυτό του *Gödel* ήταν η αιτία μιας σημαντικής κρίσης στα κλασσικά μαθηματικά, μα συγχρόνως και η απαρχή των μοντέρνων δυναμικών μαθηματικών. Το κεντρικό ερώτημα δεν είναι πια απλά αν μια πρόταση είναι αληθής η ψευδής, αλλά αν είναι "αποκρίσιμη ή μη αποκρίσιμη", δηλαδή αν είναι "υπολογιστή (*computable*) ή όχι". Αυτό ακριβώς είναι και το αντικείμενο της **Θεωρίας της Υπολογιστότητας** (*computability*). Αν δοθεί ότι μια συνάρτηση f είναι υπολογιστή, ποιο είναι το κόστος ή τα αγαθά (*resources*) που χρειάζονται για να υπολογίσουμε την f; Αυτό είναι το βασικό ερώτημα της **Θεωρίας της Πολυπλοκότητας** (complexity)[1].

Διάφοροι επιστήμονες (*Turing, Church, Kleene, Post, Markov*, κ.α.) βάλθηκαν να ξεκαθαρίσουν τις έννοιες: υπολογιστό ή επιλύσιμο (*solvable*) με αλγόριθμο, υπολογιστή συνάρτηση και αποκρίσιμο πρόβλημα. Κατέληξαν, λοιπόν, σε διαφορετικά υπολογιστικά μοντέλα, τα οποία όμως αποδείχθηκαν όλα ισοδύναμα μεταξύ τους.

Η περίφημη **Θέση (thesis) των Church-Turing** λέει λοιπόν απλουστευμένα: "Όλα τα γνωστά και τα "άγνωστα" μοντέλα της έννοιας "υπολογιστός" είναι μηχανιστικά ισοδύναμα (*effectively equivalent*)". Δηλαδή δοθέντος ενός αλγορίθμου σε ένα μοντέλο για μια συγκεκριμένη συνάρτηση f, μπορούμε μηχανιστικά (με τη βοήθεια μηχανής) να κατασκευάσουμε αλγόριθμο σε ένα άλλο μοντέλο για την ίδια συνάρτηση f.

Ας χρησιμοποιήσουμε εδώ ένα γνωστό μοντέλο υπολογισμού, μια γλώσσα προγραμματισμού υψηλού επιπέδου. Μια συνάρτηση[2] τότε, θα λέγεται υπολογιστή, αν υπάρχει πρόγραμμα που υπολογίζει την τιμή της για κάθε όρισμα. Είναι προφανές ότι υπάρχουν συναρτήσεις μη υπολογιστές, γιατί:

[1] Συχνά χρησιμοποιείται και ο όρος υπολογίσιμος αντί για υπολογιστός

[2] θα πρέπει εδώ να διευκρινίσουμε ότι αναφερόμαστε σε συναρτήσεις $f : \mathbb{N}^n \to \mathbb{N}$, αφού τα δεδομένα σε ένα πραγματικό υπολογιστή κωδικοποιούνται με φυσικούς αριθμούς (ακολουθίες από 0 και 1). Γενικότερα, αναφερόμαστε σε συναρτήσεις από αριθμήσιμα σύνολα σε αριθμήσιμα σύνολα

- Υπάρχουν άπειρα μεν, αλλά μόνο αριθμήσιμα (*countable*) διαφορετικά προγράμματα. Εκτός αυτού μπορούμε χρησιμοποιώντας κωδικοποίηση να τα απαριθμήσουμε μηχανιστικά (*effectively enumerate*)[3]

- Από την άλλη μεριά όμως, ξέρουμε ότι υπάρχουν μη αριθμήσιμες άπειρες (*uncountable*) διαφορετικές συναρτήσεις. Αυτό αποδεικνύεται με διαγωνιοποίηση (*diagonalization*), ανάλογη με αυτή που χρησιμοποιούμε για να δείξουμε ότι το σύνολο $\mathbb{R}$ είναι μη αριθμήσιμο[4]

Ας στραφούμε σε ένα συγκεκριμένο πρόβλημα που είναι μη αποκρίσιμο. Όπως ξέρουμε ο μεταγλωττιστής (*compiler*) μιας γλώσσας προγραμματισμού μπορεί να ελέγξει αν ένα πρόγραμμα είναι συντακτικά ορθό ή όχι. Τι γίνεται όμως με τα λάθη χρόνου εκτέλεσης (*run time errors*); Θα ήταν ωραίο να είχαμε ένα πρόγραμμα που θα μπορούσε να ελέγχει αν ένα συντακτικά ορθό πρόγραμμα θα σταματήσει κάποτε ή αν θα τρέχει για πάντα. Δυστυχώς τέτοιο πρόγραμμα δεν υπάρχει.

Θεώρημα 8.1. *Το **halting problem (HP)** είναι μη αποκρίσιμο.*

Απόδειξη. Έστω ότι $\pi_0, \pi_1, \pi_2, \ldots$ είναι μια μηχανιστική απαρίθμηση (*effective enumeration*) όλων των προγραμμάτων. Ας υποθέσουμε ότι το **HP** είναι επιλύσιμο. Τότε κατασκευάζουμε ένα πρόγραμμα π, που ελέγχει αν το πρόγραμμα π_n με είσοδο n σταματάει ή όχι και ανάλογα με την απάντηση σε αυτόν τον έλεγχο, το πρόγραμμα π σταματάει αν το $\pi_n(n)$ δεν σταματάει, και αντιστρόφως:

$$\pi\colon \text{read}(n); \textbf{if } \pi_n(n) \text{ terminates } \textbf{then } \text{loop\_forever } \textbf{else } \text{halt}$$

Φυσικά αυτό το πρόγραμμα π κάπου θα εμφανίζεται στην παραπάνω αρίθμηση. Ας πούμε ότι ο δείκτης για το π είναι i, δηλαδή $\pi = \pi_i$. Η ιδέα της διαγωνιοποίησης

[3] Απόδειξη: Κάθε πρόγραμμα μιας γλώσσας προγραμματισμού είναι στοιχείο του Σ^*, όπου $\Sigma = \{a_1, a_2, \ldots, a_m\}$ το αλφάβητο της γλώσσας. Το Σ^* όμως αποτελεί την ένωση $\bigcup_{n=0}^{\infty} \Sigma_n$, όπου Σ_n το σύνολο των συμβολοσειρών του αλφάβητου Σ που έχουν μήκος n. Κάθε σύνολο Σ_n είναι πεπερασμένο και έτσι αν διατάξουμε τα στοιχεία του αλφαβητικά μπορούμε να θεωρήσουμε την ακόλουθη αρίθμηση για το Σ^*:

$$\begin{aligned}
\Sigma_0 &: \quad \{\varepsilon\} \\
\Sigma_1 &: \quad \{a_1, a_2, \ldots, a_m\} \\
\Sigma_2 &: \quad \{a_1a_1, a_1a_2, \ldots, a_1a_m, \ldots, a_ma_m\} \\
&\vdots
\end{aligned}$$

Η παραπάνω αρίθμηση του Σ^* είναι μηχανιστική, δηλαδή μπορεί να γίνει με πρόγραμμα. Επομένως, με κατάλληλη χρήση compiler για τον έλεγχο ορθότητας μπορούμε να κατασκευάσουμε μηχανιστική αρίθμηση των συντακτικά ορθών n προγραμμάτων

[4] Απόδειξη: Ας θεωρήσουμε το σύνολο των ολικών συναρτήσεων $\phi\colon \mathbb{N} \to \mathbb{N}$ και έστω $\phi_0, \phi_1, \phi_2, \ldots$ μια αρίθμηση τους (ολικές ονομάζονται οι συναρτήσεις που ορίζονται για κάθε $x \in \mathbb{N}$). Ορίζουμε μια συνάρτηση f ως εξής: $f(x) = \phi_x(x) + 1, \forall x \in \mathbb{N}$. Η f είναι προφανώς ολική συνάρτηση και επομένως θα αντιστοιχίζεται σε κάποιο δείκτη y στην παραπάνω αρίθμηση μας, δηλαδή $f = \phi_y$. Τότε όμως θα ισχύει ότι $\phi_y(y) = f(y) = \phi_y(y) + 1$ που είναι άτοπο. Επομένως το σύνολο των ολικών συναρτήσεων δεν είναι αριθμήσιμο.

είναι να δώσουμε το δείκτη i για input στο π_i. Τότε το $\pi_i(i)$ σταματάει αν και μόνο αν το $\pi(i)$ σταματάει και αυτό συμβαίνει αν και μόνο αν το $\pi_i(i)$ δεν σταματάει. Αντίφαση. □

Τελικά πολλά άλλα προβλήματα είναι επίσης μη επιλύσιμα. Αν και το HP δεν είναι επιλύσιμο, μπορούμε να κατασκευάσουμε με μηχανιστικό τρόπο μια άπειρη λίστα όλων των προγραμμάτων, με την αντίστοιχη είσοδο για την οποία σταματούν. Αυτό δεν σημαίνει φυσικά ότι μπορούμε να επιλύσουμε το HP, γιατί αν για παράδειγμα το $\pi_k(n)$ δεν έχει εμφανισθεί στη λίστα μας, δεν ξέρουμε αν θα προστεθεί στη λίστα αργότερα ή αν δε θα εμφανισθεί ποτέ στη λίστα. Για να ακριβολογούμε λίγο περισσότερο δίνουμε τους παρακάτω ορισμούς.

Ορισμός 8.2. Ένα σύνολο S λέγεται αποκρίσιμο ή υπολογιστό ή επιλύσιμο (*decidable, computable, solvable*) αν και μόνο αν υπάρχει ένας αλγόριθμος που σταματάει ή μια υπολογιστική μηχανή που δίνει έξοδο "ναι" για κάθε είσοδο $a \in S$ και έξοδο "όχι" για κάθε είσοδο $a \notin S$.

Ορισμός 8.3. Ένα σύνολο S λέγεται καταγράψιμο (με μηχανιστική γεννήτρια) (*listable, effectively generatable*) αν και μόνο αν υπάρχει μια γεννήτρια διαδικασία ή μηχανή που καταγράφει όλα τα στοιχεία του S. Στην, πιθανώς άπειρη, λίστα εξόδου επιτρέπονται οι επαναλήψεις και δεν υπάρχει περιορισμός για την διάταξη των στοιχείων.

Μερικές απλές ιδιότητες:

- Αν το S είναι αποκρίσιμο τότε και το $\overline{S}$ είναι αποκρίσιμο.
- Αν το S είναι αποκρίσιμο τότε το S είναι και καταγράψιμο.
- Αν το S και το $\overline{S}$ είναι καταγράψιμα τότε το S είναι αποκρίσιμο.
- Αν το S είναι καταγράψιμο με γνησίως αύξουσα διάταξη τότε το S είναι αποκρίσιμο.

8.2 Υπολογιστικά μοντέλα

Λόγω της θέσης των Church-Turing δεν χρειάζεται να καθορίσουμε ένα συγκεκριμένο υπολογιστικό μοντέλο για τη λύση κάποιου προβλήματος: όλα τα ντετερμινιστικά υπολογιστικά μοντέλα είναι ισοδύναμα μεταξύ τους, με την έννοια ότι αν ένα πρόβλημα λύνεται από κάποιο υπολογιστικό μοντέλο, τότε θα λύνεται και από οποιοδήποτε άλλο, με το πολύ πολυωνυμική απώλεια χρόνου. Μερικά υπολογιστικά μοντέλα είναι τα εξής:

- προγράμματα Pascal

- προγράμματα Pascal χωρίς αναδρομή (αφαίρεση αναδρομής με χρήση στοίβας)
- προγράμματα Pascal χωρίς αναδρομή και χωρίς άλλους τύπους δεδομένων εκτός από τους φυσικούς αριθμούς (επιτυγχάνεται με κωδικοποιήσεις)
- προγράμματα WHILE (μόνη δομή ελέγχου το WHILE)
- προγράμματα GOTO και IF
- Assembler-like RAM (random access machine), URM (universal register machine)
- SRM (single register machine) ένας καταχωρητής
- Μηχανή Turing (πρόσβαση μόνο σε μια κυψέλη "cell" της ταινίας κάθε φορά)

Τα χαρακτηριστικά των παραπάνω μοντέλων είναι:

- ντετερμινιστική πολυπλοκότητα σε διακριτά βήματα
- πεπερασμένο σύνολο εντολών που εκτελούνται από επεξεργαστή
- απεριόριστη μνήμη

'Αλλα μοντέλα είναι:

- παραλλαγές από μηχανές Turing
- Thue: κανόνες επανεγγραφής (re-writing rules)
- Post: κανονικά συστήματα (normal systems)
- Church: λογισμός λ (λ-calculus)
- Curry: συνδυαστική λογική (combinatory logic)
- Markov: M. αλγόριθμοι
- Kleene: γενικά αναδρομικά σχήματα (general recursive schemes)
- Shepherdson-Sturgis, Elgott: URM, SRM, RAM, RASP
- Σχήματα McCarthy (if ... then ... else ... $\Rightarrow$ LISP)

Θεώρημα 8.4. *f είναι TM υπολογιστή ανν*

- *f είναι WHILE-υπολογιστή*
- *f είναι GOTO-υπολογιστή*

- *f είναι PASCAL-υπολογιστή*
- *f είναι μερικά αναδρομική (partial recursive)*

Παραλλαγές Μηχανών Turing που έχουν την ίδια υπολογιστική δυνατότητα, όχι όμως και αποδοτικότητα (*efficiency*) είναι:

- πολλές ταινίες, μνήμη πλέγματος (grid memory), μνήμη περισσοτέρων διαστάσεων
- μεγαλύτερο Σ
- πολλές παράλληλες κεφαλές
- μη ντετερμινιστικές μεταβάσεις
- μίας κατευθύνσεως, απείρου μήκους ταινία
- εγγραφή και κίνηση της κεφαλής σε κάθε βήμα

8.3 Υπολογιστική Πολυπλοκότητα

Μια άλλη (πιο μοντέρνα) ταξινόμηση προβλημάτων (γλωσσών, συνόλων) σε κλάσεις μπορεί να γίνει με κριτήριο το **ποσόν** των αγαθών (χρόνος, χώρος, επεξεργαστές, κ.ο.κ.) που χρειάζεται ένας βέλτιστος αλγόριθμος για να τα επιλύσει (αναγνωρίσει).

Για να χρησιμοποιήσουμε όμως το χρόνο, χώρο, κ.τ.λ. για μέτρο πολυπλοκότητας χρειαζόμαστε επακριβείς ορισμούς του υπολογιστικού μοντέλου καθώς και του μεγέθους που μετράμε. Το κόστος ενός αλγορίθμου δεν είναι φυσικά μια σταθερά τιμή αλλά είναι συνάρτηση του μεγέθους n της εισόδου. Συνήθως μετράμε το κόστος της χειρότερης περίπτωσης για είσοδο μεγέθους n. Έτσι το κόστος του αλγορίθμου (n) είναι το max του κόστους του αλγορίθμου από όλες τις δυνατές εισόδους μεγέθους n.

Το κόστος $C(n)$, τώρα, ενός προβλήματος $\Pi(n)$ είναι το $\min((n))$ από όλους τους αλγορίθμους που λύνουν το πρόβλημα Π. Συνεπώς για να προσδιορίσουμε το κόστος $C(n)$ ενός προβλήματος $\Pi(n)$ χρειαζόμαστε ένα **άνω όριο** (upper bound) δηλαδή έναν αλγόριθμο που έχει κόστος $C(n)$ αλλά και ένα **κάτω όριο** (lower bound), δηλαδή μια απόδειξη ότι το καλύτερο δυνατό κόστος με το τρέχον μοντέλο είναι $C(n)$. Έτσι, π.χ., η χρονική πολυπλοκότητα ταξινόμησης με συγκρίσεις (όπου μοντέλο είναι πρόγραμμα Pascal, και μετράμε τον αριθμό συγκρίσεων) είναι $\Theta(n \log n)$.

Συνήθως ονομάζουμε αποδοτικό ένα αλγόριθμο αν ο χρόνος του είναι πολυωνυμικός ($n^{O(1)}$) ως προς το μέγεθος της εισόδου. **P** λέγεται η κλάση των προβλημάτων

που λύνονται με ντετερμινιστικό αλγόριθμο σε χρόνο πολυωνυμικό. **NP** λέγεται η κλάση των προβλημάτων που λύνονται με μη ντετερμινιστικό αλγόριθμο σε χρόνο πολυωνυμικό. Λέμε ότι ένας μη ντετερμινιστικός αλγόριθμος λύνει ένα πρόβλημα Π εάν υπάρχει τουλάχιστον μια από τις δυνατές εκτελέσεις του A που λύνει το Π. Προφανώς ισχύει **P** $\subseteq$ **NP** αλλά εδώ και τριάντα-πέντε χρόνια παραμένει άλυτο το πρόβλημα "**P** $\neq$ **NP**;". **NP**$-$πλήρη λέγονται τα δυσκολότερα προβλήματα της κλάσης **NP**. Αν πράγματι ισχύει **P** $\neq$ **NP**, τότε για τα **NP**$-$πλήρη προβλήματα δεν υπάρχει αλγόριθμος πολυωνυμικού χρόνου.

PSPACE λέγεται η κλάση των προβλημάτων που λύνονται με (ντετερμινιστικό ή μη ντετερμινιστικό αλγόριθμο) σε πολυωνυμικό χώρο (μνήμη).

Τέλος **NC** λέγεται η κλάση των προβλημάτων που λύνονται με αλγόριθμο που χρησιμοποιεί πολυλογαριθμικό χρόνο ($\log^{O(1)} n$) και πολυωνυμικό αριθμό επεξεργαστών.

Μερικά γνωστά προβλήματα σε αυτές τις κλάσεις:

- Στην κλάση **NP**: το πρόβλημα (SAT) ικανοποιησιμότητας τύπων της προτασιακής λογικής, το πρόβλημα (TSP) του πλανόδιου πωλητή, κ.τ.λ.
- Στην κλάση **PSPACE**: το πρόβλημα (QBF) αποτίμησης τύπων της κατηγορηματικής (boolean) λογικής, το πρόβλημα στρατηγικής σε διάφορα παιχνίδια, κ.τ.λ.
- Στην κλάση **NC**: το πρόβλημα (GAP) πρόσβασης (δηλαδή ύπαρξης μονοπατιού) σε ένα γράφο G μεταξύ δυο κόμβων.
- Το πρόβλημα ικανοποιησιμότητας τύπων της κατηγορηματικής λογικής είναι μη επιλύσιμο αλλά καταγράψιμο.

Ισχύει:

$$\textbf{NC} \subseteq \textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subsetneq \textbf{REC} \subsetneq \textbf{R.E.}$$

Παρατήρηση:
REC = recursive = decidable
R.E. = recursively enumerable = listable

8.4 Αναγωγές μεταξύ προβλημάτων

Με τον όρο *αναγωγή* εννοούμε την μετατροπή ενός υπολογιστικού προβλήματος Π_A σε ένα άλλο πρόβλημα Π_B έτσι ώστε αν μπορούμε να λύσουμε το Π_B να μπορούμε να λύσουμε και το Π_A. Παρά το ότι υπάρχουν διάφορα είδη αναγωγών, εμείς θα ασχοληθούμε με μια απλή μορφή, κατάλληλη για προβλήματα απόφασης (όπου η απάντηση είναι "ναι'' ή "όχι''):

Ορισμός. Λέμε ότι ένα πρόβλημα απόφασης Π_A ανάγεται σε ένα πρόβλημα απόφασης Π, και γράφουμε $\Pi_A \leq \Pi_B$ αν υπάρχει μία υπολογίσιμη (computable) συνάρτηση h που απεικονίζει ένα (οποιοδήποτε) στιγμιότυπο (έγκυρη είσοδο) I_A του Π σε ένα στιγμιότυπο (έγκυρη είσοδο) I_B του Π τέτοια ώστε:

$$\text{answer}_{\Pi_A}(I_A) = \text{“yes''} \iff \text{answer}_{\Pi_B}(I_B) = \text{“yes''}$$

Δηλαδή, η απάντηση για το πρόβλημα Π_A με είσοδο I_A είναι “ναι'' *αν και μόνο αν* η απάντηση για το πρόβλημα Π_B με είσοδο $I_B = h(I_A)$ είναι “ναι''. [5]

Συχνά με τον όρο αναγωγή αναφερόμαστε τόσο στη συνάρτηση h όσο και στον *αλγόριθμο* που την υπολογίζει.

8.4.1 Αναγωγές πολυωνυμικού χρόνου

Εάν η συνάρτηση h μπορεί να υπολογιστεί αποδοτικά, δηλαδή σε πολυωνυμικό χρόνο, τότε λέγεται *αναγωγή πολυωνυμικού χρόνου*. Η ύπαρξη αναγωγής πολυωνυμικού χρόνου του Π_A στο Π_B συμβολίζεται ως εξής:

$$\Pi_A \leq^p \Pi_B$$

Συχνά χρησιμοποιούνται και οι συμβολισμοί $A \leq_P \Pi_B$ και $A \leq_m^p \Pi_B$.

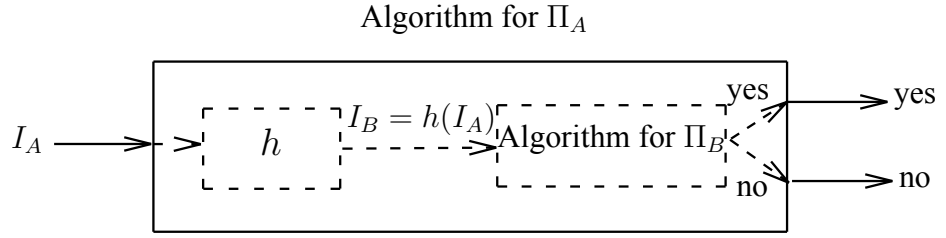
Παράδειγμα: Hamilton Cycle $\leq^p$ TSP Θα περιγράψουμε μια αναγωγή από το πρόβλημα Hamilton Cycle στο TSP.

Η ζητούμενη αναγωγή είναι, όπως αναφέρεται και πιο πάνω, μια συνάρτηση που απεικονίζει οποιοδήποτε στιγμιότυπο του Hamilton Cycle σε ένα στιγμιότυπο του TSP, ώστε η απάντηση για τα δύο στιγμιότυπα να είναι ίδια.

Για ευκολία, θα περιγράψουμε αναγωγή από το πρόβλημα Hamilton Cycle στο πρόβλημα U-TSP (Undirected TSP). [6] Θα δώσουμε τη διαδικασία μετατροπής ενός στιγμιοτύπου του Hamilton Cycle, που είναι απλά ένας μη κατευθυνόμενος γράφος G, σε ένα στιγμιότυπο του U-TSP, που είναι ένας μη κατευθυνόμενος γράφος G' με βάρη στις ακμές του, και ένα όριο κόστους (θυμηθείτε ότι μιλάμε για προβλήματα απόφασης: το ερώτημα στο πρόβλημα U-TSP είναι αν υπάρχει κύκλος Hamilton στον G' με κόστος $\leq B$).

[5] Προσοχή στο ‘αν και μόνο αν’: το συνηθέστερο λάθος είναι να δείχνει κανείς την μία μόνο κατεύθυνση, δηλαδή το ‘μόνο αν’. Αυτό όμως δεν μας εγγυάται ότι γνωρίζοντας την απάντηση για το I_B θα γνωρίζουμε και την απάντηση για το I_A. Πιο συγκεκριμένα, μας εξασφαλίζει μόνο ότι αν η απάντηση για το I_B είναι αρνητική τότε και η απάντηση για το I_A είναι αρνητική. Αν όμως η απάντηση για το I_B είναι καταφατική, μπορεί η απάντηση για το I_A να είναι είτε καταφατική είτε αρνητική.

[6] Αυτό δεν δημιουργεί πρόβλημα αφού οι μη κατευθυνόμενοι γράφοι μπορούν να θεωρηθούν συμμετρικοί κατευθυνόμενοι. Επομένως, το πρόβλημα U-TSP είναι ειδική περίπτωση του προβλήματος TSP.



Σχήμα 8.1: Αναγωγή από Π_A σε Π_B + αλγόριθμος για $\Pi_B \Rightarrow$ αλγόριθμος για Π_A.

Κατασκευή: Ο γράφος G' έχει το ίδιο πλήθος κόμβων με τον G, έστω n, αλλά είναι πλήρης. Σε κάθε ακμή του G' που υπήρχε και στον αρχικό γράφο G δίνουμε βάρος 1, ενώ στις υπόλοιπες δίνουμε βάρος 2. Θέτουμε σαν όριο κόστους $B = n$.

Απόδειξη ορθότητας: Θα δείξουμε ότι υπάρχει κύκλος Hamilton στον αρχικό γράφο G **αν και μόνο αν** υπάρχει κύκλος Hamilton στον G' με κόστος $\leq n$:
'μόνο αν': αν υπάρχει κύκλος Hamilton στον G τότε ο ίδιος κύκλος στον G' αποτελείται από ακμές βάρους 1, οπότε το συνολικό κόστος του κύκλου είναι n.
'αν': αν υπάρχει κύκλος Hamilton στον G' με κόστος $\leq n$, αυτός θα πρέπει να αποτελείται αποκλειστικά από ακμές βάρους 1, αφού κάθε κύκλος Hamilton έχει n ακμές. Επομένως αυτός ο κύκλος υπάρχει και στον G.

Χρειάζεται επιπλέον να δείξουμε ότι η αναγωγή γίνεται σε πολυωνυμικό χρόνο. Πράγματι, ένα πέρασμα του πίνακα γειτνίασης (ή των λιστών γειτνίασης) και αντικατάσταση των (μη διαγωνίων) 0 με 2 αρκεί, επομένως ο απαιτούμενος χρόνος είναι $O(n^2)$ ($O(m)$ με λίστες), δηλαδή πολυωνυμικός ως προς το μέγεθος της εισόδου.

8.4.2 Αναγωγές: δύο τρόποι χρήσης

Όπως είδαμε πιο πάνω, μια ορθή αναγωγή έχει την ιδιότητα οι απαντήσεις για τα δύο στιγμιότυπα (I_A του προβλήματος Π_A και I_B του προβλήματος Π_B) να ταυτίζονται. Αυτό σημαίνει ότι αν έχουμε έναν αποδοτικό αλγόριθμο επίλυσης του Π_B και μια αποδοτική αναγωγή από το Π_A στο Π_B τότε παίρνουμε έναν αποδοτικό αλγόριθμο επίλυσης του Π_A, όπως φαίνεται και στο Σχ. 8.1.

Η ιδιότητα αυτή των αναγωγών μπορεί να χρησιμοποιηθεί με δύο τρόπους, είτε με "θετικό" τρόπο, **προς επίλυση προβλήματος**, είτε με "αρνητικό", προς **απόδειξη δυσκολίας προβλήματος**.

Πιο συγκεκριμένα, μια αναγωγή πολυωνυμικού χρόνου του Π_A στο Π_B μπορεί να χρησιμοποιηθεί είτε για την εύρεση αποδοτικού αλγορίθμου για το πρόβλημα Π_A (εάν γνωρίζουμε αλγόριθμο για το Π_B) είτε για την απόδειξη δυσκολίας του Π_B (εάν διαθέτουμε απόδειξη δυσκολίας για το Π_A). Ο λόγος για το δεύτερο είναι ότι αν γνωρίζουμε (με βεβαιότητα, ή κάτω από βάσιμες υποθέσεις) ότι το Π_A δεν διαθέτει πολυωνυμικό αλγόριθμο, τότε συνάγεται (με (με βεβαιότητα, ή κάτω από τις ίδιες υποθέσεις) ότι και το Π_B δεν διαθέτει πολυωνυμικό αλγόριθμο (αλλιώς

θα διέθετε και το Π_A (αντίφαση).

Παραδείγματα:

- Η επίλυση του Προβλήματος του Βαρκάρη με αναγωγή στο πρόβλημα Reachability (βλ. Ασκήσεις) υπάγεται στην πρώτη περίπτωση.
- Οι αποδείξεις **NP**-πληρότητας υπάγονται στη δεύτερη περίπτωση και παρουσιάζονται πιο αναλυτικά παρακάτω.

Αποδείξεις NP-πληρότητας με χρήση αναγωγής πολυωνυμικού χρόνου

Ορισμός. Ένα πρόβλημα απόφασης Π λέγεται **NP-δύσκολο (NP-hard)** αν για κάθε πρόβλημα $\Pi' \in$ **NP** ισχύει $\Pi' \leq^p \Pi$. Αν επιπλέον $\Pi \in$ **NP** τότε το Π λέγεται **NP-πλήρες (NP-complete)**.

Ισχύει ότι αν ένα πρόβλημα Π είναι **NP**-δύσκολο τότε αν υπήρχε πολυωνυμικός αλγόριθμος S για το πρόβλημα αυτό θα συνεπαγόταν ότι **P**=**NP** (γιατί θα μπορούσαμε να 'συνθέσουμε' τον αλγόριθμο της αναγωγής $\Pi' \leq^p \Pi$ με τον υποθετικό αλγόριθμο S και να πάρουμε πολυωνυμικό αλγόριθμο για το Π', όπου Π' οποιοδήποτε πρόβλημα στην κλάση **NP**). Παρ'ότι αυτό είναι ακόμη ανοιχτό πρόβλημα, θεωρείται εξαιρετικά απίθανο να συμβαίνει, οπότε αν γνωρίζουμε ότι ένα πρόβλημα είναι **NP**-δύσκολο συμπεραίνουμε ότι είναι μάλλον αδύνατο να έχει πολυωνυμικό αλγόριθμο. Τα ίδια ισχύουν βέβαια αν ένα πρόβλημα είναι **NP**-πλήρες (αφού **NP**-πλήρες είναι ειδική περίπτωση **NP**-δύσκολου).

Το παρακάτω θεώρημα δίνει τον (συνήθη) τρόπο απόδειξης ότι ένα πρόβλημα είναι **NP**-δύσκολο (ή και **NP**-πλήρες).

Θεώρημα. Αν $\Pi_A \leq^p \Pi_B$ και το Π_A είναι **NP**-πλήρες τότε το Π_B είναι **NP**-δύσκολο. Αν επιπλέον $\Pi_B \in$ **NP** τότε το Π_B είναι **NP**-πλήρες.

Η ιδέα της απόδειξης (δεν θα την παρουσιάσουμε αναλυτικά εδώ) είναι ότι η σύνθεση αναγωγών πολυωνυμικού χρόνου είναι αναγωγή πολυωνυμικού χρόνου, επομένως ισχύει ότι:

$$\Pi \leq^p \Pi_A \text{ και } \Pi_A \leq^p \Pi_B \Rightarrow \Pi \leq^p \Pi_B$$

Παράδειγμα: η αναγωγή που δώσαμε παραπάνω Hamilton Cycle $\leq^p$ TSP, επειδή γνωρίζουμε ότι το Hamilton Cycle είναι **NP**-πλήρες οδηγεί στο συμπέρασμα ότι και το πρόβλημα απόφασης TSP είναι **NP**-δύσκολο. Επειδή επιπλέον το πρόβλημα απόφασης TSP ανήκει στην κλάση **NP** (αποδεικνύεται σχετικά εύκολα: μια πιθανή λύση επαληθεύεται σε πολυωνυμικό χρόνο) συμπεραίνουμε ότι το πρόβλημα απόφασης TSP είναι **NP**-πλήρες.

Ασκήσεις

1. *Κάλυψη κόμβων* σε έναν γράφο $G(V, E)$ λέγεται ένα υποσύνολο V' των κόμβων του γράφου τέτοιο ώστε κάθε ακμή του γράφου έχει έναν τουλάχιστον κόμβο της στο σύνολο, δηλαδή $\forall\{u, v\} \in E : u \in V' \vee v \in V'$. *Ανεξάρτητο σύνολο* σε έναν γράφο $G(V, E)$ λέγεται ένα υποσύνολο κόμβων του γράφου V' που δεν έχουν καμμία ακμή μεταξύ τους, δηλαδή $\forall u, v \in V' : \{u, v\} \notin E$.

   (α) Αποδείξτε ότι ένα υποσύνολο V' των κόμβων του γράφου είναι κάλυψη κόμβων αν και μόνο αν το σύνολο $V \setminus V'$ είναι ανεξάρτητο.

   (β) *Κλίκα* σε έναν γράφο $G(V, E)$ λέγεται ένα υποσύνολο κόμβων V' του γράφου που συνδέονται όλοι ανά δύο μεταξύ τους, δηλαδή $\forall u, v \in V' : (u, v) \in E$.

   Περιγράψτε αναγωγή από το πρόβλημα Vertex Cover (δίνεται γράφος και αριθμός k, υπάρχει κάλυψη κόμβων μεγέθους το πολύ k στο γράφο;) στο πρόβλημα Clique (δίνεται γράφος και αριθμός m, υπάρχει κλίκα μεγέθους τουλάχιστον m στο γράφο;). Τι συμπέρασμα προκύπτει εάν γνωρίζουμε ότι το πρόβλημα Vertex Cover είναι NP-πλήρες; Τι έχετε να πείτε για το πρόβλημα Independent Set (δίνεται γράφος και αριθμός t, υπάρχει ανεξάρτητο σύνολο μεγέθους τουλάχιστον t στο γράφο;);

2. Γενικεύοντας το Πρόβλημα του Βαρκάρη (βλ. και ασκήσεις γράφων) μας δίνονται n αντικείμενα και χωρητικότητα βάρκας k (εκτός του βαρκάρη). Οι ασυμβατότητες μεταξύ των αντικειμένων δίνονται από μια σχέση R: $R(a_i, a_j)$ σημαίνει ότι απαγορεύεται τα a_i και a_j να βρίσκονται στην ίδια όχθη αφύλακτα (επιτρέπεται όμως να είναι στην ίδια όχθη με τον βαρκάρη παρόντα, ή μέσα στη βάρκα πηγαίνοντας από την μια όχθη στην άλλη).

   (α) Έχει πάντοτε λύση το Πρόβλημα του Βαρκάρη;

   (β) Επεκτείνετε την αναγωγή που περιγράψαμε στο μάθημα σε μια αναγωγή του Γενικευμένου Προβλήματος του Βαρκάρη στο Πρόβλημα Προσβασιμότητας σε γράφο (Reachability).

   (γ) Διατυπώστε έναν αλγόριθμο επίλυσης του Γενικευμένου Προβλήματος του Βαρκάρη. Ποια είναι η πολυπλοκότητα του αλγορίθμου σας;

3. Ορίζουμε το πρόβλημα της Διαδρομής Ίππου ως εξής: είσοδος είναι οι διαστάσεις ορθογώνιας σκακιέρας n, m, καθώς και κάποια απαγορευμένα τετράγωνα που δίνονται σαν ζεύγη (i, j). Θέλουμε να βρούμε αν ένα άλογο (ίππος) που ξεκινά από το τετράγωνο (x_1, y_1) μπορεί να φτάσει στο τετράγωνο (x_2, y_2) χωρίς να πατήσει πάνω σε απαγορευμένα τετράγωνα, και αν ναι με ποιον τρόπο.

   Σημείωση: η κίνηση του αλόγου είναι 2 τετράγωνα προς μία κατεύθυνση, οριζόντια ή κάθετα, και μετά 1 τετράγωνο αριστερά ή δεξιά, κάθετα στην

αρχική κατεύθυνση. Το άλογο επιτρέπεται να περάσει πάνω από απαγορευμένα τετράγωνα, αλλά όχι να σταματήσει σε αυτά.

(α) Έχει λύση το πρόβλημα για τη σκακιέρα 4×5 με απαγορευμένο τετράγωνο το $(2, 3)$, αρχικό τετράγωνο $(1, 1)$ και τελικό τετράγωνο $(3, 1)$; Αν ναι, ζωγραφίστε τη διαδρομή στο παρακάτω σχήμα. Αν όχι εξηγήστε γιατί.

	1	2	3	4	5
1	♞				
2			×		
3					
4					

(β) Περιγράψτε με σαφήνεια έναν όσο το δυνατόν πιο αποδοτικό αλγόριθμο που να επιλύει το πρόβλημα της Διαδρομής Ίππου στη γενική περίπτωση (δηλ. για οποιαδήποτε σκακιέρα δοθεί). Ποια είναι η πολυπλοκότητα του αλγορίθμου σας σε σχέση με τα n, m και το πλήθος k των απαγορευμένων τετραγώνων;

(γ) Θεωρήστε επιπλέον την παραλλαγή όπου δίνεται επιπλέον ένας ακέραιος k και ζητείται να βρείτε εάν υπάρχει διαδρομή από το αρχικό στο τελικό τετράγωνο με k το πολύ κινήσεις. Βρείτε την απάντηση για την παραπάνω σκακιέρα, για $k = 3$.

Πώς πρέπει να τροποποιήσετε / συμπληρώσετε τον αλγόριθμό σας ώστε να επιλύει αυτή την εκδοχή στη γενική περίπτωση; Αλλάζει η πολυπλοκότητα του αλγορίθμου σας; Εξηγήστε.

4. (α) Ορίστε τις κλάσεις Reg (Regular), P (Polynomial Time), CF (Context Free), NP (Non-deterministic Polynomial Time), REC (Recursive), RE (Recursively Enumerable).

(β) Σχεδιάστε τις παραπάνω κλάσεις σε διάγραμμα Hasse, με σύντομη αιτιολόγηση.

(γ) Ορίστε το πρόβλημα Κύκλου Hamilton. Σε ποια από τις παραπάνω κλάσεις ανήκει και γιατί;

(δ) Αντιστοιχήστε προβλήματα με κλάσεις πολυπλοκότητας (με σύντομη αιτιολόγηση):

(i) Προβλήματα:

- Ελάχιστο Συνδετικό Δένδρο
- $L_1 = \{w \in \{a, b\}^* \mid w$ περιέχει ίσο αριθμό a και $b\}$
- Ισοδυναμία αυτομάτων
- $L_2 = \{ww \mid w \in \{a, b, c, d\}^*\}$

- Satisfiability (SAT)
- $L_3 = \{w \in \{0,1\}^* \mid w$ περιέχει ακριβώς τρία $'0'\}$
- Halting Problem

(ii) Κλάσεις: NP, REC, Context Free, Regular, RE, Context Sensitive, P