Αλγόριθμοι
και
Πολυπλοκότητα

Στάθης Ζάχος

Αθήνα 2009

Αλγόριθμοι
και
Πολυπλοκότητα

Στάθης Ζάχος

Πρόλογος

Στις σημειώσεις αυτές παρουσιάζεται περιληπτικά η ύλη που καλύπτεται στο μάθημα «Αλγόριθμοι και Πολυπλοκότητα» που διδάσκεται στους φοιτητές του 7ου εξαμήνου της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου.

ΠΡΟΣΟΧΗ: ΟΙ ΣΗΜΕΙΩΣΕΙΣ ΔΕΝ ΥΠΟΚΑΘΙΣΤΟΥΝ ΤΗΝ ΠΑΡΑΚΟΛΟΥΘΗΣΗ ΤΟΥ ΜΑΘΗΜΑΤΟΣ.

Σκοπός του μαθήματος αυτού είναι η εισαγωγή στις έννοιες των αλγορίθμων και της πολυπλοκότητας, η παρουσίαση βασικών τεχνικών σχεδίασης και ανάλυσης αλγορίθμων και η εφαρμογή των τεχνικών αυτών για την επίλυση υπολογιστικών προβλημάτων.

Στο μάθημα περιλαμβάνονται: τεχνικές για ασυμπτωτική ανάλυση πολυπλοκότητας προγραμμάτων, κριτήρια για την επιλογή αλγορίθμων, μέθοδοι σχεδιασμού καλών αλγορίθμων («διαίρει και βασίλευε», δυναμικός προγραμματισμός, «άπληστη» στρατηγική), εφαρμογές στη θεωρία γραφημάτων (αναζήτηση σε βάθος, αναζήτηση σε πλάτος, ελάχιστο συνδετικό δέντρο, διαδρομή ελαχίστου κόστους), επεξεργασία δεδομένων (διάταξη και αναζήτηση), αλγεβρικά προβλήματα (υπολογισμός πολυωνύμων, πολλαπλασιασμός πινάκων), αφηρημένη θεωρία πολυπλοκότητας (ντετερμινιστικοί και μη-ντετερμινιστικοί αλγόριθμοι πολυωνυμικού χρόνου, NP-πλήρη προβλήματα), τέλος προσεγγιστικοί αλγόριθμοι.

Θέλω να ευχαριστήσω τους παρακάτω μεταπτυχιακούς σπουδαστές που βοήθησαν στην προετοιμασία αυτών των σημειώσεων: Χρήστο Νομικό, Άρη Παγουρτζή, Ευριπίδη Μάρκου, Κατερίνα Ποτίκα, Πέτρο Ποτίκα, Χριστόδουλο Φραγκουδάκη, Παναγιώτη Χείλαρη, Σταύρο Βάσσο (νυν διδάκτορες), Αντώνη Καβαρνό, Γεωργία Καούρη και Γιώργο Πιερράκο.

Στάθης Ζάχος, 2009

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Κατάλογος αλγορίθμων

Κεφάλαιο 1

Εισαγωγή

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*).

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.

Ο τρόπος που προσεγγίζει κανείς την πολυπλοκότητα οδηγεί σ'έναν αρχικό διαχωρισμό της έννοιας **πολυπλοκότητα**, σε **συγκεκριμένη** (*concrete*) πολυπλοκότητα και **μη συγκεκριμένη, περισσότερο θεωρητική** (*abstract*) **πολυπλοκότητα**.

Ο κλάδος της συγκεκριμένης (*concrete*) πολυπλοκότητας ασχολείται με την περιγραφή συστηματικών τεχνικών αξιολόγησης των μετρήσιμων αγαθών (*resources*) που χαρακτηρίζουν την αποδοτικότητα ενός συγκεκριμένου αλγορίθμου (κυρίως του χρόνου και του χώρου που απαιτούνται απ'τον αλγόριθμο) σ'ένα συγκεκριμένο **υπολογιστικό μοντέλο**.

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 το input. Λογικά το κόστος αυξάνεται με την αύξηση του μεγέθους της εισόδου. Από την άλλη μεριά το κόστος μπορεί να διαφέρει για διαφορετικά inputs 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. Η εκτίμηση για την πολυπλοκότητα του αλγορίθμου ουσιαστικά σημαίνει εκτίμηση της **αυξητικής τάσης** της αντίστοιχης συνάρτησης.

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

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

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

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

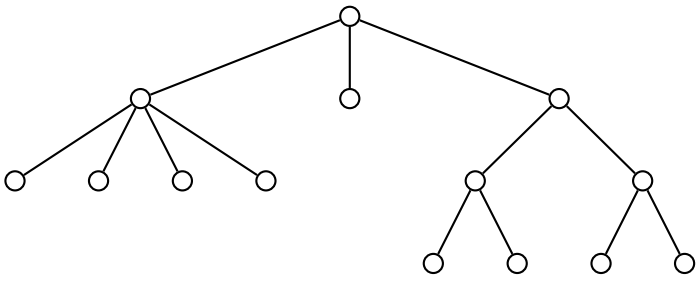
Ένας αλγόριθμος είναι **ντετερμινιστικός** (*deterministic*) ή **μη ντετερμινιστικός** (*nondeterministic*). Ο ντετερμινιστικός αλγόριθμος διακρίνεται από τα παρακάτω στοιχεία:

- Ο υπολογισμός που προτείνει είναι γραμμικός. Για κάθε υπολογιστική διαμόρφωση (configuration) υπάρχει ακριβώς μία νόμιμη επόμενη διαμόρφωση.
- Η υπολογιστική διαδικασία προχωρεί βήμα προς βήμα και είναι σε θέση να σταματήσει για οποιαδήποτε δυνατή είσοδο.

Σχήμα 1.1: Ντετερμινιστικός αλγόριθμος

Σχηματικά ο ντετερμινιστικός αλγόριθμος φαίνεται στο σχήμα 1.1. Ο μη ντετερμινιστικός αλγόριθμος παριστάνεται στο σχήμα 1.2. Όπως φαίνεται στο σχήμα 1.2 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 «μαντεύει» μια ακολουθία από υπολογιστικές διαμορφώσεις (δηλαδή ένα μονοπάτι στο δέντρο) και στη δεύτερη φάση, κάτω από 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.

Εκτός από την διάκριση σε ντετερμινιστικούς και μη ντετερμινιστικούς αλγορίθμους μπορούμε τους διακρίνουμε και σε διάφορα άλλα είδη, όπως π.χ.



Σχήμα 1.2: Μη ντετερμινιστικός αλγόριθμος

σε **παράλληλους** (*parallel*) αλγορίθμους, σε **πιθανοτικούς** (*probabilistic*) αλγορίθμους, σε **εναλλασσόμενους** (*alternating*) και άλλους.

Οι αλγόριθμοι μπορούν να ταξινομηθούν ανάλογα με κάποιο χαρακτηριστικό τους. Έτσι έχουμε ταξινομήσεις ανάλογα με:

- τις δομές δεδομένων που χρησιμοποιούν
- το είδος των δεδομένων που χρησιμοποιούν (πραγματικούς αριθμούς–αριθμητική ανάλυση, γράφους, κ.τ.λ.)
- την πολυπλοκότητά τους
- την στρατηγική σχεδιασμού τους

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, R. 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 του Church»**:

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

ή διατυπωμένη αλλιώς:

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

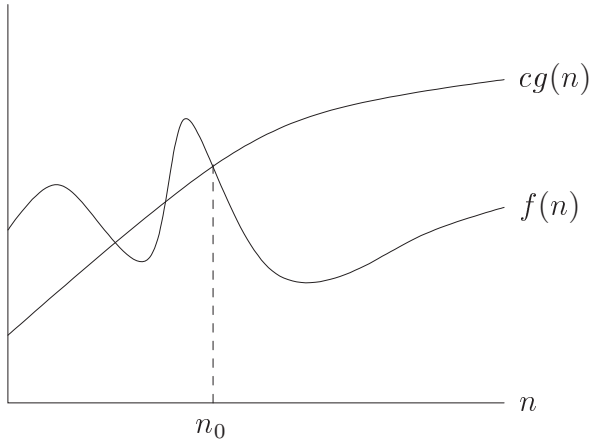
δηλαδή:

> **«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».**

1.1 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) 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 1.3: $f = O(g)$

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).

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

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

ή

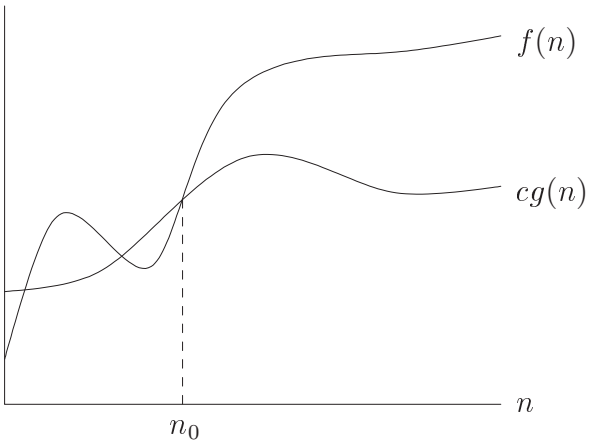
$$o(g) = \{f \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$$

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

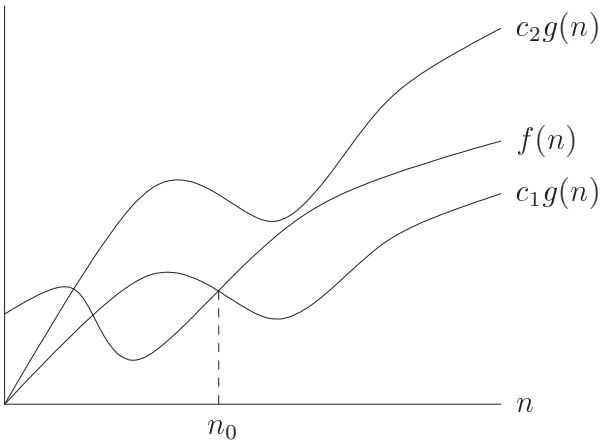
$$\omega(g) = \{f \mid \forall c > 0,\ \exists n_0 : \forall n > n_0 \ \ f(n) \geq cg(n)\}$$

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

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



Σχήμα 1.4: $f = \Omega(g)$



Σχήμα 1.5: $f = \Theta(g)$

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: $O(1) < O(\alpha(n)) < O(\log^* n) < O(\log(n)) < O(\sqrt{n}) < O(n) < O(n\log(n)) < O(n^2) < \ldots < O(poly) < O(2^n) < O(n!) < O(n^n) < O(A(n))$

Σημείωση: γράφουμε «$<$» αντί «$\subset$».

Ορισμός 1.1.1 (*Πολυλογαριθμική συνάρτηση*)**.** $\log^* n$ δίνει πόσες φορές πρέπει να λογαριθμήσουμε το n για να πάρουμε 1.

Ορισμός 1.1.2 (*Συνάρτηση Ackermann*)**.** Ορίζουμε μία συνάρτηση ως εξής:

$$A(x,y) = \begin{cases} 1, & \text{όταν } x = 0, y \geq 0 \\ 2, & \text{όταν } x = 1, y = 0 \\ x + 2, & \text{όταν } x \geq 2, y = 0 \\ A(A(x-1,y), y-1), & \text{όταν } x, y \geq 1 \end{cases}$$

Μπορούμε εύκολα να παρατηρήσουμε ότι:

$$\begin{aligned} A(x,0) &= 2 + x \\ A(x,1) &= 2x \\ A(x,2) &= 2^x \\ A(x,3) &= 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}x \end{aligned}$$

Η συνάρτηση Ackermann είναι μια συνάρτηση η οποία αυξάνεται πολύ γρήγορα. Η συνάρτηση Ackermann με ένα όρισμα μπορεί να οριστεί ως εξής: $A(n) = A(n,n)$.

Η συνάρτηση $\alpha(n)$ είναι ψευδοαντίστροφη της συνάρτησης $A(n)$. Δηλαδή το $\alpha(n)$ είναι το ελάχιστο x για το οποίο $n \leq A(x)$. Για παράδειγμα $A(1) = 2$ άρα $\alpha(1) = \alpha(2) = 1$. $A(2) = 4$ άρα $\alpha(3) = \alpha(4) = 2$. $A(3) = 8$ άρα $\alpha(5) = \ldots = \alpha(8) = 3$.

$A(4) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}65536$, δηλαδή $\alpha(n) \leq 4$, όπου n είναι ένας πάρα πολύ μεγάλος ακέραιος. Συνεπώς η συνάρτηση $\alpha(n)$ είναι πολύ αργά αυξανόμενη συνάρτηση (σχεδόν σταθερή).

Ορισμός 1.1.3. Δύο συναρτήσεις f_1, f_2 έχουν πολυωνυμική σχέση μεταξύ τους (*polynomially related*) ανν υπάρχουν πολυώνυμα $p_1(x)$, $p_2(x)$ τέτοια ώστε

$$\forall n : f_1(n) \leq p_1(f_2(n)) \ \wedge \ f_2(n) \leq p_2(f_1(n))$$

Παράδειγμα 1.1.4. Οι συναρτήσεις $f_1(n) = n^3$ και $f_2(n) = n^{17}$ είναι polynomially related, ενώ οι n^3 και 2^n δεν είναι.

Κεφάλαιο 2

Αλγόριθμοι και πολυπλοκότητα

Η ονομασία Αλγόριθμος προέρχεται από το όνομα του Άραβα 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 Άλγεβρα).

Υπενθυμίζουμε ότι σκοπός ενός αλγορίθμου είναι η επίλυση ενός προβλήματος. Πριν ξεκινήσουμε ας θέσουμε ορισμένες ερωτήσεις για τα προβλήματα: Ποιος παράγει ένα πρόβλημα; Τι είναι πρόβλημα; Πώς μοντελοποιούμε ένα πρόβλημα; Τι είναι λύση; Πώς αναπαριστούμε μια λύση (γλώσσα); Είναι πράγματι λύση του προβλήματος (ορθότητα); Ποια είναι η αποδοτικότητα της 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 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.

2.1.1 Ένας απλός αλγόριθμος για το gcd

$z := min(a, b)$;
while $(a \textbf{ mod } z \neq 0)$ **or** $(a \textbf{ mod } z \neq 0)$ **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.

2.1.2 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.

2.1.3 Αλγόριθμος του Ευκλείδη

$i := a$; $j := b$;
while $(i > 0)$ **and** $(j > 0)$ **do**
  if $i > j$ **then** $i := i \textbf{ mod } j$ **else** $j := j \textbf{ mod } i$;
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 $a \textbf{ mod } b$.

Στην χειρότερη περίπτωση ο αλγόριθμος έχει πολυπλοκότητα $O(\log(a+b))$. 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 $F_k = (\varphi^k - \hat{\varphi}^k)/\sqrt{5}$, όπου $\varphi = (1+\sqrt{5})/2 \approx 1.618$ (γνωστή και ως η *χρυσή τομή*) και $\hat{\varphi} = (1-\sqrt{5})/2$. Ας σημειώσουμε ότι επειδή $|\hat{\varphi}| < 1$, το F_k είναι ίσο με το $\varphi^k/\sqrt{5}$ στρογγυλοποιημένο στον πλησιέστερο ακέραιο, οπότε προκύπτει το παρακάτω πόρισμα:

Πόρισμα 2.1.1. $\log_\varphi(F_k) + 1 \leq k \leq \log_\varphi(F_k) + 2$

Έχουμε τα παρακάτω αποτελέσματα σχετικά με την πολυπλοκότητα:

Λήμμα 2.1.2. *Ο αλγόριθμος του Ευκλείδη για $a = F_{k+1}$ και $b = F_k$ έχει χρονική πολυπλοκότητα $O(k)$.*

Πόρισμα 2.1.3. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι $\Omega(\log(a+b))$.*

Λήμμα 2.1.4. *Τα ζεύγη διαδοχικών αριθμών Fibonacci είναι η χειρότερη περίπτωση από άποψη χρονικής πολυπλοκότητας για τον αλγόριθμο του Ευκλείδη.*

Πόρισμα 2.1.5. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι $O(\log(a+b))$.*

Θεώρημα 2.1.6. *Η χρονική πολυπλοκότητα του αλγορίθμου του Ευκλείδη είναι $\Theta(\log(a+b))$.*

2.2 Το πρόβλημα αναζήτησης σε ταξινομημένο πίνακα

Το γενικό πρόβλημα αναζήτησης (searching problem) είναι:

Searching
Είσοδος: Ένας ταξινομημένος πίνακας a από n διακριτά κλειδιά:

$$a[1] < a[2] < \ldots < a[n]$$

και ένα κλειδί k
Έξοδος: Εμφανίζεται το k στον a; ναι ή όχι. Αν ναι, τότε επιστρέφεται η θέση i, όπου $a[i] = k$

Η αναζήτηση διατυπωμένη ως παιχνίδι ανάμεσα σε 2 παίκτες.

Αντί του παραπάνω προβλήματος αναζήτησης θα συζητήσουμε το εξής σχετικό παιχνίδι αναζήτησης ανάμεσα σε δύο παίκτες:

- Ο *παίκτης 1* επιλέγει έναν αριθμό x στο πεδίο $[1..n]$.
- Ο *παίκτης 2* αναζητά το x με όσο το δυνατό λιγότερες ερωτήσεις τύπου: είναι το $x \leq i$; (για κάποιο $1 \leq i \leq n$).

Ορισμός 2.2.1. Η ερώτηση «$x \leq i$?» ονομάζεται σύγκριση.

2.2.1 Σειριακή Αναζήτηση (Sequential Search)

function Sequential_Search(n,x);
begin
  $i := 0$;
  repeat $i := i + 1$
  until $x \leq i$; *(∗ σύγκριση ∗)*
  return (i)
end

Η ορθότητα (correctness) είναι προφανής. Για την πολυπλοκότητα έχουμε να παρατηρήσουμε, ότι μπορεί να είναι μόνο 1 σύγκριση όταν $x = 1$, n συγκρίσεις στη χειρότερη περίπτωση όταν $x = n$ και $n/2$ συγκρίσεις στη μέση περίπτωση αν x επιλέγεται ομοιόμορφα στο πεδίο $[1..n]$.

2.2.2 Δυαδική αναζήτηση (Binary Search)

function Binary_Search(n,x);
begin
  $l := 1; u := n$;
  while $l < u$ **do**
  begin
    mid := $(l + u)$ **div** 2;
    if $x \leq mid$ *(∗ σύγκριση ∗)* **then** $u := mid$ **else** $l := mid + 1$
  end;
  return (l)
end

Για την ορθότητα, δείχνουμε με επαγωγή ότι πάντα $l \leq x \leq u$. Στο τέλος, $l = u$ και για αυτό $x = l$.

Για την πολυπλοκότητα δείχνουμε:

- $k = \log_2 n$ συγκρίσεις για $n = 2^k$.
  - * Με επαγωγή, $u - l = 2^{k-h} - 1$ μετά από h συγκρίσεις.
  - * $u - l = 2^{k-k} - 1 = 0$ μετά από k συγκρίσεις (στο τέλος).
- $\lceil \log_2 n \rceil$ συγκρίσεις στη χειρότερη περίπτωση.
- Μπορεί να είναι $\lfloor \log_2 n \rfloor$ σε μερικές περιπτώσεις.
- Ποτέ λιγότερες από $\lfloor \log_2 n \rfloor$ συγκρίσεις.

2.2.3 Δεν μπορούμε να αναζητήσουμε γρηγορότερα

Απόδειξη με *επιχείρημα αντιπάλου*(adversary argument):

Έστω πως ο αντίπαλος παίκτης 1 δεν επιλέγει το x στην αρχή του παιχνιδιού αλλά διατηρεί ένα σύνολο S (όσο μπορεί μεγαλύτερο) από υποψήφιους για x. [1]

Έστω ότι δίνει ο παίκτης 2 μια ερώτηση αναζήτησης, και $S(Y)$ = το σύνολο των υποψηφίων για τους οποίους η απάντηση του παίκτη 1 είναι ναι, και $S(N)$ = το σύνολο των υποψηφίων για τους οποίους η απάντηση του παίκτη 1 είναι όχι. Ισχύει $S = S(Y) \cup S(N)$.

Ο κανόνας απάντησης του παίκτη 1 είναι (για να διατηρήσει όσο το δυνατόν μεγάλο S):

- απαντάει ναι, αν $|S(Y)| \geq |S(N)|$.
- απαντάει όχι, αν $|S(Y)| < |S(N)|$.

Στην πρώτη περίπτωση θέτει $S := S(Y)$ ενώ στη δεύτερη $S := S(N)$.

Θεώρημα 2.2.2. *Ο αντίπαλος 1 μπορεί να αναγκάσει να γίνουν τουλάχιστον* $\lceil \log_2 n \rceil$ *συγκρίσεις.*

Proof.

- Υποθέτουμε ότι ο παίκτης 2 κάνει k ερωτήσεις-συγκρίσεις. Έστω S_0, S_1, $S_2, \ldots, S_k$ ότι είναι το σύνολο των υποψηφίων. Συγκεκριμένα, $|S_0| = n$ και $|S_k| = 1$.
- Με τον προηγούμενο κανόνα απάντησης: $|S_{i+1}| \geq |S_i|/2$ για $1 \leq i \leq k-1$. Για αυτό, $k \geq \lceil \log_2 n \rceil$.

□

[1] Ουσιαστικά ο παίκτης 2 είναι ο αλγόριθμος αναζήτησης ενώ ο αντίπαλος παίκτης 1 προσπαθεί να επιλέξει τη χειρότερη δυνατή περίπτωση για τον αλγόριθμο.

2.3 Εύρεση ελαχίστου και μεγίστου

2.3.1 Εύρεση ελαχίστου (minimum)

Εύρεση ελαχίστου (minimum)
Είσοδος: Ένας αταξινόμητος πίνακας a από n κλειδιά: $a[1..n]$.
Έξοδος: Στη θέση 1 το κλειδί $a[1]$ ώστε $a[1] \leq a[i]$ για $1 \leq i \leq n$.
Στόχος: Ελαχιστοποίησε τον αριθμό των συγκρίσεων ανάμεσα στα κλειδιά.

function Trivial_Find_Min($a[1..n]$);
begin
  for $i := 2$ **to** n **do if** $a[1] > a[i]$ *(* σύγκριση *)* **then** swap$(a[1], a[i])$;
  return $(a[1])$
end

Για την ορθότητα: Μετά από το γύρο i, έχουμε $a[1] = min\{a[1], \ldots, a[i]\}$.
Για την πολυπλοκότητα έχουμε: Ακριβώς $n-1$ συγκρίσεις.

2.3.2 Μπορούμε καλύτερα; (adversary argument)

Έστω $B =$ το σύνολο αυτών που μπορούν ακόμα να είναι ο μικρότερος και $R =$ το σύνολο αυτών που δεν μπορούν να είναι. Αρχικά, $B = \{a[1], \ldots, a[n]\}$ και $R = \emptyset$. Στο τέλος, $|B| = 1$ και $|R| = n - 1$. Η στρατηγική αντιπάλου αποσκοπεί στη διατήρηση (όσο το δυνατόν) μεγαλύτερου B.

Ο κανόνας απάντησης είναι:
[εδώ $(x : y)$ είναι η σύγκριση των στοιχείων x, y]

- δίνει μία συνεπή απάντηση αν ερωτάται: $(r_1 : r_2)$.
- δίνει απάντηση $b < r$ αν ερωτάται: $(b : r)$.
- δίνει απάντηση $b_1 < b_2$ αν ερωτάται: $(b_1 : b_2)$ και μεταφέρει το b_2 από το B στο R.

Θεώρημα 2.3.1. *Ο αντίπαλος μπορεί να αναγκάσει να γίνουν $n-1$ συγκρίσεις.*

Proof.

- Μόνο $(b_1 : b_2)$ είναι χρήσιμη σύγκριση.
- Κάθε χρήσιμη σύγκριση ελαττώνει το μέγεθος του B κατά 1.

- Χρειάζονται $n-1$ χρήσιμες συγκρίσεις για να ελαττωθεί το μέγεθος του B από n σε 1.

□

2.3.3 Παράλληλη εύρεση ελαχίστου (ή σειριακά με divide & conquer)

Ορισμός 2.3.2. Ένα *στάδιο* (stage, round) περιέχει πολλές συγκρίσεις όπου όμως κάθε κλειδί συγκρίνεται το πολύ μια φορά.

Σκοπός:

- ελαχιστοποίηση αριθμού σταδίων
- ελαχιστοποίηση αριθμού συγκρίσεων

Στο Trivial_Find_Min είχαμε $n-1$ στάδια με μια σύγκριση ανά στάδιο.

function Parallel_Find_Min($a[1..n]$); *(*$n = 2^k$*)*
begin
  if $n = 1$ **then return** ($a[1]$) **else**
    for $i := n$ **div** 2 **do** *(*γύρος*)*
      if $a[i] \geq a[i + n$ **div** $2]$ *(*σύγκριση*)* **then** swap($a[i], a[i + n$ **div** $2]$);
  return (Parallel_Find_Min($a[i..n$ **div** $2]$))
end

Αριθμός συγκρίσεων: $\frac{n}{2} + \frac{n}{4} + \cdots + 1 = n - 1$.
Αριθμός σταδίων (αναδρομικών κλήσεων): $\log_2 n$.

Μπορούμε να βρούμε τον ελάχιστο με λιγότερα στάδια;

- το πολύ $|B|$ **div** 2 χρήσιμες συγκρίσεις ανά στάδιο
- άρα χρειάζονται $\lceil \log_2 n \rceil$ στάδια για μείωση του μεγέθους υποψηφίων B από n στο 1.

2.3.4 Εύρεση ελαχίστου και μεγίστου

Απλός

procedure Parallel_Find_Min_and_Max($a[1..n]$);
begin
  min := Parallel_Find_Min($a[1..n]$);
  max := Parallel_Find_Max($a[1..n]$)
end

Η πολυπλοκότητα για $n = 2^k$:

- $2(n-1)$ συγκρίσεις
- $\log_2 n$ στάδια

Επιδέξιος

procedure Parallel_Find_Min_and_Max($a[1..n]$);
begin
  for $i := 1$ **to** n **div** 2 **do**
    if $a[i] > a[i + n$ **div** $2]$ **then** swap$(a[i], a[i + n$ **div** $2])$;
  min := Parallel_Find_Min($a[1 \quad . \quad . \quad n$ **div** $2]$);
  max := Parallel_Find_Max($a[n$ **div** $2 + 1 \quad . \quad . \quad n]$)
end

Η πολυπλοκότητα για $n = 2^k$:

- $\frac{n}{2} + 2(\frac{n}{2} - 1) = \frac{3n}{2} - 2$ συγκρίσεις
- $1 + \log_2(n/2) = \log_2 n$ στάδια

2.3.5 Δεν γίνεται καλύτερα (adversary argument)

Έστω $N =$ το σύνολο αυτών που μπορούν να είναι είτε ο μεγαλύτερος είτε ο μικρότερος, $H =$ μπορούν να είναι μόνο ο μεγαλύτερος, $B =$ μπορούν να είναι μόνο ο μικρότερος, $R =$ δεν μπορούν να είναι ούτε ο μεγαλύτερος ούτε ο μικρότερος. Αρχικά, $N = \{a[1], \ldots, a[n]\}, H = B = R = \emptyset$. Στο τέλος $|H| = 1, |B| = 1, |N| = 0, |R| = n - 2$.

Ο κανόνας απάντησης είναι:

- δίνει μία συνεπή απάντηση αν ερωτάται: $(r_1 : r_2)$
- δίνει απάντηση $h > r$ αν ερωτάται: $(r : h)$.
- δίνει απάντηση $b < r$ αν ερωτάται: $(b : r)$.
- δίνει απάντηση $n < r$ αν ερωτάται: $(n : r)$ και μεταφέρει το n από το N στο B.
- δίνει απάντηση $b < n$ αν ερωτάται: $(b : n)$ και μεταφέρει το n από το N στο H.
- δίνει απάντηση $n < h$ αν ερωτάται: $(n : h)$ και μεταφέρει το n από το N στο B.

- δίνει απάντηση $n_1 < n_2$ αν ερωτάται: $(n_1 : n_2)$ και μεταφέρει το n_1 από το N στο B και το n_2 από το N στο H.
- δίνει απάντηση $b < h$ αν ερωτάται: $(b : h)$.
- δίνει απάντηση $b_1 < b_2$ αν ερωτάται: $(b_1 : b_2)$ και μεταφέρει το b_2 από το B στο R.
- δίνει απάντηση $h_1 < h_2$ αν ερωτάται: $(h_1 : h_2)$ και μεταφέρει το h_1 από το H στο R

Θεώρημα 2.3.3. *Ο αντίπαλος μπορεί να αναγκάσει να γίνουν τουλάχιστον* $\lceil \frac{3n}{2} \rceil - 2$ *συγκρίσεις.*

Proof.

- Κάθε μη-μέγιστο, μη-ελάχιστο κλειδί ακολουθεί την πορεία:

$$N \to \{B, H\} \to R.$$

- $(n_1 : n_2), (b_1 : b_2)$ και $(h_1 : h_2)$ είναι χρήσιμες συγκρίσεις.
- $(n_1 : n_2)$ είναι καλύτερη από $(n : r), (b : n), (n : h)$.
- Οι υπόλοιπες συγκρίσεις δεν είναι χρήσιμες.
- Ο γρηγορότερος τρόπος να αδειάσει το N απαιτεί τουλάχιστον $\lceil \frac{n}{2} \rceil$ συγκρίσεις.
- Ο γρηγορότερος τρόπος να μείνει ένα κλειδί και στο B και στο H απαιτεί τουλάχιστον $n - 2$ συγκρίσεις.

□

2.3.6 Παράλληλη εύρεση ελαχίστου σε πίνακα μεγέθους n, για οποιοδήποτε n

procedure Parallel_Find_Min($a[1..n]$);
begin

- σύγκρινε μόνο n **div** 2 ζεύγη ($\lfloor \frac{n}{2} \rfloor$)
- συνέχισε αναδρομικά με $(n + 1)$ **div** 2 υποψηφίους ($\lceil \frac{n}{2} \rceil$)(δηλαδή τους $n/2$ μικρότερους, αν n άρτιο, ή $\lfloor \frac{n}{2} \rfloor$ μικρότερους συν το μη- συγκριθέντα, αν n περιττό).

end

Αριθμός σταδίων: $R(n) = \lceil \log_2 n \rceil$

Proof.

- $R(i) \leq R(i+1)$, για $i \geq 1$
- $R(2^k) = \log_2(2^k) = k$
- $R(n) = k$ για κάθε n με: $2^{k-1} < n \leq 2^k$
- $\lceil \log_2 n \rceil = k$ για κάθε n με: $2^{k-1} < n \leq 2^k$

□

Αριθμός συγκρίσεων: $C(n) = n - 1$, διότι δεν υπάρχει άχρηστη σύγκριση.

2.3.7 Παράλληλη εύρεση ελαχίστου και μεγίστου σε πίνακα μεγέθους n γενικώς

procedure Parallel_Find_Min_and_Max($a[1..n]$);
begin

- σύγκρινε μόνο n **div** 2 ζεύγη ($\lfloor \frac{n}{2} \rfloor$)
- max := Parallel_Find_Max με $\lceil n/2 \rceil$ υποψηφίους (δηλαδή τους $n/2$ μεγαλύτερους για n άρτιο, $\lfloor n/2 \rfloor$ συν το μη- συγκριθέντα για n περιττό).
- min := Parallel_Find_Min με $\lceil n/2 \rceil$ υποψηφίους (δηλαδή τους $n/2$ μικρότερους για n άρτιο, ή $\lfloor \frac{n}{2} \rfloor$ μικρότερους συν το μη- συγκριθέντα για n περιττό).

end

Παρατήρηση: Αν n περιττός υπάρχει ένας υποψήφιος για max και για min.
Αριθμός σταδίων: $RR(n) = \lceil \log_2 n \rceil$

Proof. Απόδειξη όπως και στην 2.3.6 □

Αριθμός συγκρίσεων: $CC(n) = \lceil 3n/2 \rceil - 2$

Proof. $CC(n) = 2C(\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor) = 2\lceil \frac{n}{2} \rceil - 2 + \lfloor \frac{n}{2} \rfloor$ □

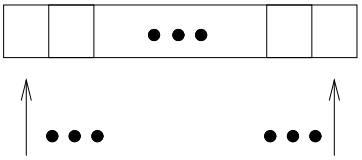
Υλοποίηση

- Ζευγάρωμα:

  - Σύγκρινε $a[1]$ με $a[n]$.
  - Σύγκρινε $a[2]$ με $a[n-1]$.

    $\ldots$
  - Σύγκρινε $a[i]$ με $a[n+1-i]$.

    $\ldots$
  - Σύγκρινε $a[\lfloor \frac{n}{2} \rfloor]$ με $a[n+1-\lfloor \frac{n}{2} \rfloor]$.



- **if** $a[i] < a[n+1-i]$ **then** $\text{swap}(a[i], a[n+1-i])$
- αν n άρτιο τότε όλα ζευγαρώνονται
- αν n περιττό τότε ο $a[\lceil \frac{n}{2} \rceil]$ δεν ζευγαρώνεται, διότι

$$\lfloor \frac{n}{2} \rfloor < \lceil \frac{n}{2} \rceil < n+1-\lfloor \frac{n}{2} \rfloor.$$

- Οι υποψήφιοι ελάχιστοι είναι:
  - για άρτιο n : $a[1], \ldots, a[\frac{n}{2}]$.
  - για περιττό n : $a[1], \ldots, a[\lceil \frac{n}{2} \rceil]$.
- Οι υποψήφιοι μέγιστοι είναι:
  - για άρτιο n : $a[\frac{n}{2}+1], \ldots, a[n]$.
  - για περιττό n : $a[\lceil \frac{n}{2} \rceil], \ldots, a[n]$

2.3.8 Εύρεση του πρώτου και του δεύτερου μικρότερου

Find First and Second
Είσοδος: Ένας πίνακας από n κλειδιά: $a[1..n]$.
Έξοδος: Ο πίνακας a με αλλαγές ώστε:

- $a[1] \leq a[i]$ για $2 \leq i \leq n$.
- $a[2] \leq a[i]$ για $3 \leq i \leq n$.

Στόχος: Να ελαχιστοποιήσουμε τον αριθμό των συγκρίσεων μεταξύ των κλειδιών.

procedure Trivial_Find_First_and_Second($a[1..n]$); *(*$n = 2^k$*)*
begin
  First := Trivial_Find_Min($a[1..n]$);
  First := Trivial_Find_Min($a[2..n]$)
end

Η ο είναι προφανής και η πολυπλοκότητα είναι ακριβώς $2n-3$ συγκρίσεις.

Μια καλύτερη λύση
Ποιος μπορεί να είναι δεύτερος; Μόνο εκείνοι που συγκρίθηκαν με τον πρώτο. Πόσοι συγκρίθηκαν με τον πρώτο; Στην Trivial λύση μερικές φορές $n-1$, άρα αν χρησιμοποιήσουμε την Parallel_Find_Min($a[1..n]$), η απάντηση είναι $k = \log_2 n$.
Άρα αλγόριθμος: Πρώτος := ο ελάχιστος του αρχικού πίνακα.
Δεύτερος := ο ελάχιστος από τους $\log n$ συγκριθέντες με τον πρώτο.
Η πολυπλοκότητα είναι: $n + \log n - 2$ συγκρίσεις.

Βελτιστότητα: Κανένας αλγόριθμος δεν χρησιμοποιεί λιγότερες συγκρίσεις.

Άσκηση: Υλοποιήστε τον.

2.4 Το πρόβλημα της ταξινόμησης (sorting)

Sorting Problem
Είσοδος: Ένας αταξινόμητος πίνακας με n διακριτά κλειδιά: $a[1..n]$
Έξοδος: Ταξινομημένος πίνακας $a : a[1] < a[2] < \ldots < a[n]$
Στόχος: Να ελαχιστοποιήσουμε τον αριθμό των συγκρίσεων μεταξύ των κλειδιών.
Κάτω φράγμα: Τουλάχιστον $\Omega(n \log n)$ συγκρίσεις.

2.4.1 Αλγόριθμοι Ταξινόμησης

- Μέθοδος της Φυσαλλίδας (Bubble-Sort): $O(n^2)$ χειρότερη και μέση περίπτωση.
- Μέθοδος Εισαγωγής (Insertion-Sort): $O(n^2)$ χειρότερη και μέση περίπτωση.
- Μέθοδος Συγχώνευσης (Merge-Sort): $O(n \log n)$ χειρότερη και μέση περίπτωση.
- «Γρήγορη» Ταξινόμηση (Quick-Sort): $O(n \log n)$ μέση περίπτωση. $O(n^2)$ χειρότερη περίπτωση.
- Μέθοδος Σωρού (Heap-Sort): $O(n \log n)$ χειρότερη και μέση περίπτωση.
- Ταξινόμηση Δυαδικού Δέντρου (Binary-Tree-Sort): $O(n \log n)$ μέση περίπτωση, $O(n^2)$ χειρότερη περίπτωση.
- Ταξινόμηση Ισοζυγισμένου Δέντρου (Balanced-Tree-Sort) $O(n \log n)$ χειρότερη και μέση περίπτωση.

Σε άλλο μοντέλο (όχι με συγκρίσεις, γιατί το μέγεθος όλων των κλειδιών είναι φραγμένο):

- Ταξινόμηση Μέτρησης (Counting-Sort), Ταξινόμηση Βάσης (Radix - Sort), Ταξινόμηση Δοχείων (Bucket-Sort): $O(n)$ λειτουργίες.

2.4.2 Ταξινόμηση Φυσαλλίδας

procedure Bubble_Sort(a[1..n]);
begin
  for i := 1 **to** n−1 **do**
    for $j := n$ **downto** $i + 1$ **do**
      if $a[j] < a[j-1]$ **then** swap$(a[j], a[j-1])$
end

Για την ορθότητα, με επαγωγή, μετά το στάδιο i έχουμε:

- $a[1] < a[2] < \ldots < a[i]$.
- $a[i] < a[j]$ για όλα τα $i < j \leq n$.

Για την πολυπλοκότητα: Ο αριθμός των συγκρίσεων είναι *πάντα*

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = O(n^2).$$

2.4.3 Ταξινόμηση με Εισαγωγή

procedure Insertion_Sort(a[1..n]);
begin
  for $j := 2$ **to** n **do**
  begin
    $key := a[j]$; $i := j - 1$;
    while $(i > 0)$ **and** $(key < a[i])$ **do** *(∗ ολίσθηση ∗)*
    begin $a[i + 1] := a[i]$; $i := i - 1$
    end;
    $a[i + 1] := key$ *(∗τοποθέτηση∗)*
  end
end

Για τη ορθότητα έχουμε: με επαγωγή, μετά από το γύρο j: $a[1] < a[2] < \ldots < a[j]$. Το άνω φράγμα στον αριθμό των συγκρίσεων: $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$. Το κάτω φράγμα στον αριθμό των συγκρίσεων: $1+1+\ldots+1 = n-1$. Μέσος αριθμός συγκρίσεων: $\frac{n-1}{2}+\frac{n-2}{2}+\ldots+\frac{1}{2} = \frac{n(n-1)}{4}$. Πολυπλοκότητα: $O(n^2)$ χειρότερη και μέση περίπτωση.

Άλλους αλγόριθμους ταξινόμησης θα δούμε στα επόμενα κεφάλαια.

Κεφάλαιο 3

Βασικές Δομές Δεδομένων

3.1 Εισαγωγή

Όπως είναι γνωστό, για να εκτελεστεί ένας αλγόριθμος με Η/Υ θα πρέπει να γραφτεί σε μία αυστηρά ορισμένη γλώσσα Η/Υ. Αυτή η υλοποίηση του αλγόριθμου σε γλώσσα προγραμματισμού Η/Υ, λέγεται **πρόγραμμα**. Ένα πρόγραμμα Η/Υ επενεργεί σε σύνολα δεδομένων που αποθηκεύονται στη μνήμη του Η/Υ. Είναι λοιπόν φανερό πως η επιλογή των δομών με τις οποίες θα οργανωθούν τα δεδομένα στη μνήμη του Η/Υ επηρεάζουν την απόδοση του προγράμματος, άρα την απόδοση γενικότερα, του αλγορίθμου. **Δομές δεδομένων**, ονομάζουμε τις διάφορες διμελείς σχέσεις μεταξύ των στοιχείων ενός συνόλου δεδομένων. Οι σχέσεις αυτές μπορεί να είναι γραμμικές ή μη γραμμικές.

Έστω ένα μη κενό σύνολο δεδομένων και μια διμελής σχέση που διατάσσει τα στοιχεία του έτσι ώστε ένα στοιχείο που ονομάζεται αρχή να έχει ένα επόμενο, ένα στοιχείο που ονομάζεται τέλος ένα προηγούμενο και κάθε άλλο στοιχείο να έχει ένα μόνο προηγούμενο και ένα μόνο επόμενο. Τότε λέμε ότι τα στοιχεία του συνόλου αυτού των δεδομένων, είναι **ολικώς ή γραμμικώς διατεταγμένα** (*totally or linearly ordered*) και η δομή που ορίζεται απ' αυτή τη σχέση ονομάζεται γραμμική δομή δεδομένων (*linear data structure*). Κάθε άλλη δομή δεδομένων που δεν είναι γραμμική, ονομάζεται **μη γραμμική δομή δεδομένων** (*non-linear*). Σημειωτέον ότι η διάταξη αυτή δεν έχει σχέση με τυχόν άλλη διάταξη των τιμών των κόμβων π.χ. λεξικογραφική ή αριθμητική.

Στις γραμμικές δομές δεδομένων, ανήκουν οι πίνακες, οι εγγραφές, τα σύνολα, τα αρχεία και οι γραμμικές λίστες. Απ'τις γραμμικές λίστες μπορούμε να ξεχωρίσουμε τις **στοίβες** (*stacks*) και τις **ουρές** (*queues*).

Στις μη γραμμικές δομές δεδομένων, ανήκουν οι **γράφοι** (ή **γραφήματα**) και τα δέντρα.

Τις γραμμικές δομές δεδομένων έχουμε δύο τρόπους να τις αποθηκεύσουμε,

τον **σειριακό** (σε array) και τον **δυναμικό** (με pointers). Ο χρόνος που απαιτείται για να κάνουμε εισαγωγή ενός στοιχείου (*insertion*) ή ανάκτηση (*retrieval*) από μία γραμμική δομή δεδομένων είναι σταθερός και ανεξάρτητος του **μεγέθους του πίνακα ή της λίστας**, δηλαδή οι πράξεις αυτές έχουν πολυπλοκότητα χρόνου $O(1)$.

3.2 Γράφοι

3.2.1 Γενικά

Ορισμός 3.2.1. 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 3.2.2. 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** ή **πλευρές** (*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, καταχρηστικές θεωρούμε και ακμές-βρόχους, δηλαδή $e = \{x, x\}$.

Αν $e = \{v_1, v_2\}$ είναι πλευρά 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 πλευράς e επειδή δε η πλευρά e της συνδέει είναι **γειτονικές** (*adjacent*) κορυφές στο G.

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

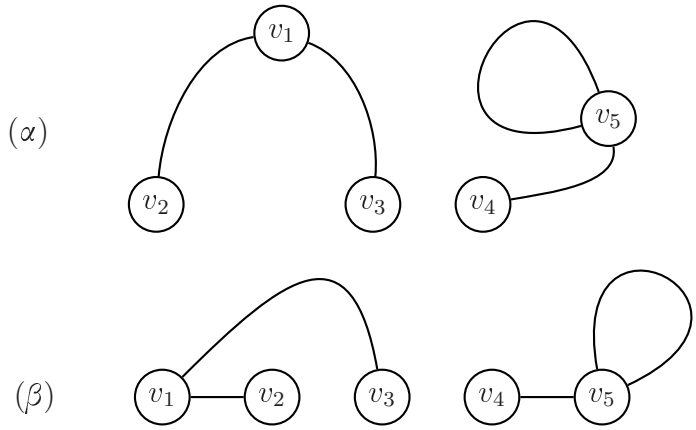
Ο ορισμός του γράφου όπως δόθηκε παραπάνω, δεν διευκολύνει την εποπτική αντίληψη του όρου. Είναι δυνατόν και πολλές φορές επιβάλλεται, για την αναγνώριση και τη μελέτη ιδιοτήτων των γράφων, η απεικόνιση αυτών με τη βοήθεια διαγράμματος.

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

Παράδειγμα 3.2.3. 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 πλευρές 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 πλευρών του, **μέγεθος** (*size*) του G και συμβολίζεται με $|E|$. Στην Πληροφορική όμως, συνήθως ονομάζουμε μέγεθος το $n = |V|$.

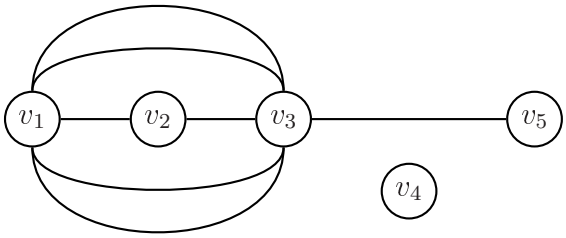
Παράδειγμα 3.2.4. Στο γράφο G του Παραδείγματος 2 η τάξη του ισούται με 5 και το μέγεθος με 4. Ο γράφος G μπορεί επίσης να συμβολιστεί και με $G(5, 4)$.

Παρατήρηση 3.2.5. Από τον ορισμό του γράφου προκύπτει ότι μία πλευρά δεν μπορεί να έχει ως άκρα την ίδια κορυφή. Συχνά όμως στην Πληροφορική, όπως αναφέραμε και πιο πάνω, χρειαζόμαστε μια τέτοια πλευρά. Η πλευρά τότε λέγεται **βρόχος** (*loop*). Επίσης από τον ορισμό του γράφου προκύπτει ότι, δεν είναι δυνατή η ύπαρξη περισσοτέρων πλευρών με ίδια άκρα, δηλαδή δεν είναι δυνατή η ύπαρξη παράλληλων πλευρών. Στο γράφο του Παραδείγματος 2, εφόσον υπάρχει η πλευρά 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.

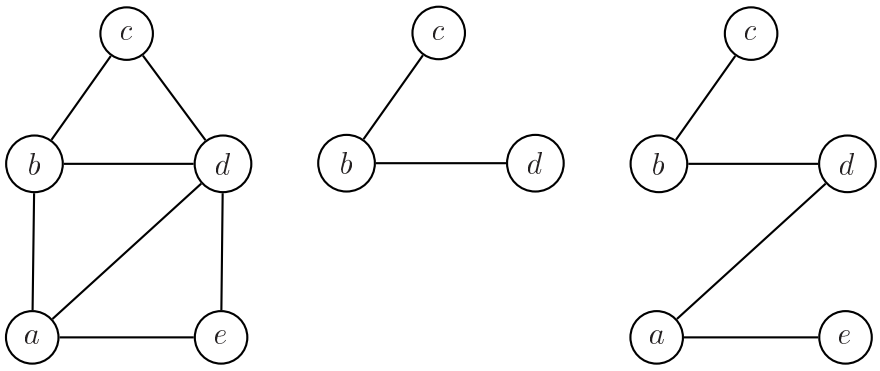
Υπάρχουν όμως και **πολυγραφήματα** (*multigraphs*). Το διάγραμμα ενός πολυγραφήματος μπορεί να περιέχει πολλές ακμές που συνδέουν τις ίδιες κορυφές.

Παράδειγμα 3.2.6. Στο σχήμα 3.2 φαίνεται ένα πολυγράφημα.



Σχήμα 3.2: Πολυγράφημα

3.2.2 Υπογράφος



Γράφος G Γράφος G' Γράφος $G >>$

Σχήμα 3.3: Ο γράφος G και δύο υπογράφοι αυτού

Ορισμός 3.2.7. Ένας γράφος $G' = (V', E')$ είναι **υπογράφος** (*subgraph*) ενός άλλου γράφου $G = (V, E)$, αν ισχύει $V' \subseteq V$ και $E' \subseteq E$.

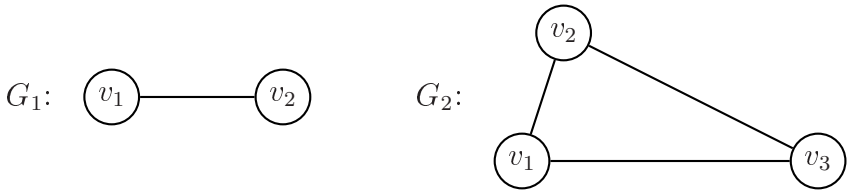
Παράδειγμα 3.2.8. Στο σχήμα 3.3 ο G' είναι ένας υπογράφος του G.

Ορισμός 3.2.9. Έστω $G' = (V', E')$ υπογράφος ενός γράφου $G = (V, E)$. Αν ισχύει $V' = V$ τότε ο υπογράφος λέγεται **παράγων υπογράφος** του γράφου G.

Παράδειγμα 3.2.10. Στο σχήμα 3.3 ο $G >>$ είναι παράγων υπογράφος του G.

3.2.3 Βαθμός κορυφής

Ορισμός 3.2.11. Έστω ένας γράφος $G = (V, E)$. **Βαθμός** (*degree, valence*) μιας κορυφής $v \in V$ ονομάζεται ο αριθμός των πλευρών 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 πλευρών του. Δηλαδή σε ένα γράφο $G = (V, E)$ έχουμε ότι

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

Παράδειγμα 3.2.12. 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.

3.2.4 Δρόμος - Μονοπάτι - Κύκλος

Ορισμός 3.2.13. Σ'ένα γράφο G, μια πεπερασμένη ακολουθία, εναλλάξ κορυφών και πλευρών του G που αρχίζει και τελειώνει σε κορυφή και που κάθε πλευρά που περιέχεται στην ακολουθία προσπίπτει στην κορυφή που προηγείται και σ' αυτήν που έπεται, λέγεται **δρόμος** ή **διαδρομή** (*walk*) στο G.

Παράδειγμα 3.2.14. Στο γράφο G στο σχήμα 3.3 η ακολουθία κορυφών και πλευρών του γράφου

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

είναι δρόμος στο G.

Ορισμός 3.2.15. Αν σ'έναν δρόμο ενός γράφου κάθε πλευρά του δρόμου εμφανίζεται μόνο μία φορά, ο δρόμος λέγεται **δρομίσκος** ή **μονοπάτι** (*trail*).

Παράδειγμα 3.2.16. 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 3.2.17. Ένας δρόμος στον οποίο κάθε κορυφή και κάθε πλευρά του εμφανίζονται ακριβώς μία φορά, λέγεται **απλό μονοπάτι** (*path*).

Παράδειγμα 3.2.18. 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.

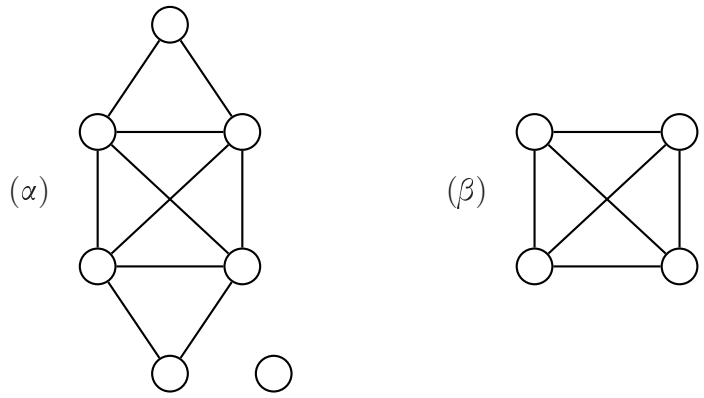
Ορισμός 3.2.19. Ένας δρόμος με αρχή και τέλος την ίδια κορυφή, λέγεται **κλειστός δρόμος**, αλλιώς λέγεται **ανοικτός**.

Ορισμός 3.2.20. Ένας δρόμος που είναι κλειστό μονοπάτι λέγεται **κύκλος** (*cycle*). Ένας δρόμος που είναι απλό κλειστό μονοπάτι λέγεται **απλός κύκλος** (*simple cycle*).

Παράδειγμα 3.2.21. 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 3.2.22. Ένας κύκλος που περνά ακριβώς μια φορά από κάθε πλευρά 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)).



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

Ορισμός 3.2.23. Ένας κύκλος που περνά ακριβώς μια φορά από κάθε κορυφή ενός γράφου G (χωρίς απαραίτητα να περνά και από όλες τις πλευρές) ονομάζεται **κύκλος Hamilton**. Ένας γράφος που έχει κύκλο Hamilton ονομάζεται **γράφος Hamilton** (σχήμα 3.5(β)).

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

Παράδειγμα 3.2.24. Στο παράδειγμα 3.2.21 τα μήκη των δρόμων με τη σειρά που εμφανίζονται είναι 2, 5 και 3.

3.2.5 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*).

Ορισμός 3.2.25 (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 ($a_{i,j} = a_{j,i}$).

Πίνακας πρόσπτωσης

Ορισμός 3.2.26. Έστω ένας γράφος $G = (V, E)$ με $V = \{v_1, v_2, ..., v_n\}$ και $E = \{l_1, l_2, ..., l_m\}$. Τότε ο γράφος μπορεί να παρασταθεί με τη βοήθεια ενός $n \times m$ πίνακα $B(G)$, που ονομάζεται **πίνακας πρόσπτωσης** (*incidence matrix*), όπου

$$B(G) = [b_{ij}], \quad b_{ij} = \begin{cases} 1, & \text{αν } l_j \text{ προσπίπτει στο } v_i \\ 0, & \text{αλλιώς} \end{cases}$$

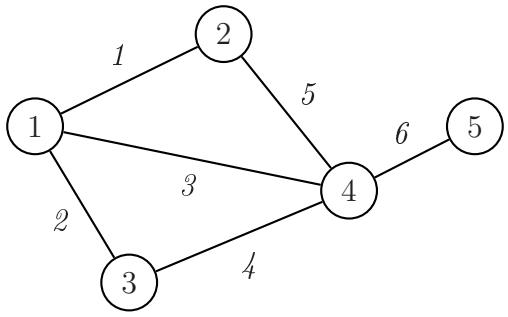
Αποδεικνύεται ότι ισχύει:

$$B(G)B(G)^T = A(G) + \begin{pmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_n \end{pmatrix}$$

όπου d_i είναι ο βαθμός του κόμβου 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 3.2.27. Οι αραιοί γράφοι έχουν $O(n)$ ακμές ενώ οι πυκνοί γράφοι έχουν $\Omega(n^2)$.



Σχήμα 3.6: Γράφος με αριθμημένες ακμές

Παράδειγμα 3.2.28. Η αναπαράσταση του γράφου που φαίνεται στο σχήμα 3.6 με τον πίνακα γειτνίασης είναι η παρακάτω:

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

Για να χρησιμοποιήσουμε τον πίνακα πρόσπτωσης πρέπει πρώτα να αριθμήσουμε με κάποιο τρόπο τις πλευρές του γράφου. Μια αρίθμηση για το γράφο του σχήματος 3.6 είναι και η ακόλουθη:

$$1\colon (1,2),\ 2\colon (1,3),\ 3\colon (1,4),\ 4\colon (3,4),\ 5\colon (2,4),\ 6\colon (4,5).$$

Χρησιμοποιώντας αυτή την αρίθμηση η αναπαράσταση του γράφου με τον πίνακα πρόσπτωσης είναι η παρακάτω:

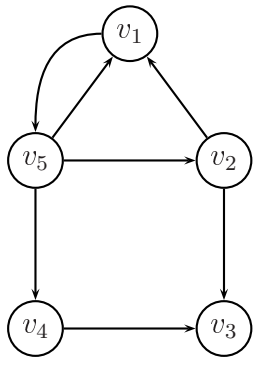
$$B(G) = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Τέλος, η αναπαράσταση με τις λίστες γειτνίασης είναι η παρακάτω:

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

3.2.6 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 3.2.29. 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 3.2.30. Έστω x μια κορυφή ενός προσανατολισμένου γράφου. Ο αριθμός των πλευρών που προσπίπτουν (εισέρχονται) σ' αυτήν την κορυφή ονομάζεται **προς-βαθμός** (*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 πλευρών 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 3.2.31. Στον προσανατολισμένο γράφο στο σχήμα 3.7 έχουμε:

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

3.2.7 Συνεκτικός Γράφος

Ορισμός 3.2.32. 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:

$$V_i \subseteq V \quad , 1 \leq i \leq k$$
$$V_i \bigcap V_j = \emptyset \quad , \forall i, j$$
$$V_1 \bigcup V_2 \bigcup \ldots \bigcup V_k = V$$

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

Ορισμός 3.2.33. Έστω ένας γράφος $G = (V, E)$ και $V' \subseteq V$. Ο υπογράφος που έχει σύνολο κορυφών το V' και σύνολο πλευρών όλες τις πλευρές του G, των οποίων και τα δύο άκρα ανήκουν στο V', λέγεται **παραγόμενος υπογράφος** (*induced subgraph*) από τον V' και συμβολίζεται $G[V']$.

Ορισμός 3.2.34. 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 $\Omega(G)$.

Ορισμός 3.2.35. 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 3.2.36. Ο γράφος του Παραδείγματος 2 είναι μη συνεκτικός με $\omega(G) = 2$. Ο γράφος του Παραδείγματος 12 είναι συνεκτικός με $\omega(G) = 1$.

Παρατήρηση 3.2.37. Στην περίπτωση προσανατολισμένου γράφου οι ακμές σε μονοπάτια (άρα και σε κύκλο) πρέπει να έχουν όλες τον ίδιο προσανατολισμό.

Ορισμός 3.2.38. 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 3.2.39. Ένας απλός γράφος $G = (V, E)$ (χωρίς βρόχους και παράλληλες πλευρές) 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 3.2.40. 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 πλευρές 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 $m = |Y|$.

Ένας γράφος ονομάζεται επίπεδος (planar) αν μπορεί να σχεδιαστεί στο επίπεδο έτσι ώστε όλες οι πλευρές 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 $K_{3,3}$.

3.3 Δέντρα

3.3.1 Γενικά

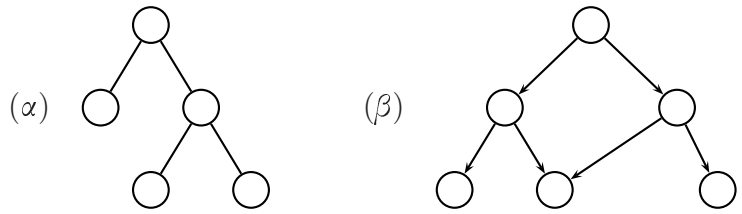
Ορισμός 3.3.1. Δένδρο λέγεται ένας συνεκτικός γράφος που δεν περιέχει κύκλους. **Δάσος** λέγεται κάθε γράφος που δεν περιέχει κύκλους. Οι συνεκτικές

συνιστώσες ενός δάσους, είναι δέντρα. Ένα δέντρο στο οποίο ξεχωρίζουμε μια κορυφή, την οποία ονομάζουμε **ρίζα**, λέγεται δέντρο με ρίζα (*rooted tree*).

Πρόταση 3.3.2. *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.



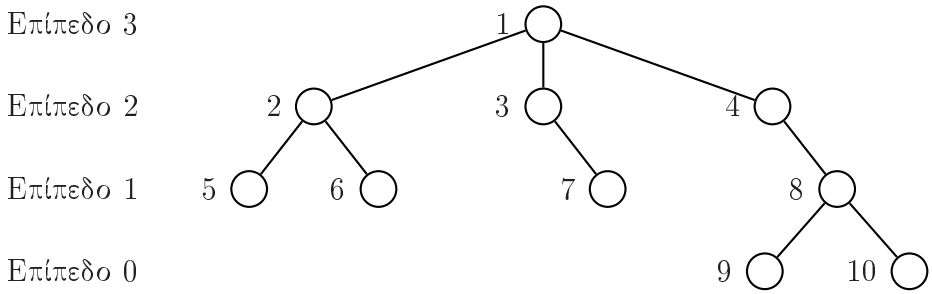
Σχήμα 3.8: (α) Δένδρο, (β) Κατευθυνόμενος ακυκλικός γράφος

Ορισμός 3.3.3. 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 3.3.4. 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.



Σχήμα 3.9: Δένδρο με αριθμημένες κορυφές

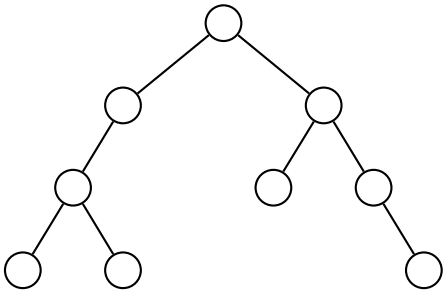
Ορισμός 3.3.5. Ύψος (*height*) ενός κόμβου είναι η μέγιστη απόστασή του από απόγονό του. Ύψος δέντρου είναι το ύψος της ρίζας. Βάθος (*depth*) ενός κόμβου είναι η απόστασή του από τη ρίζα. Επίπεδο (*level*) κόμβου είναι το ύψος του δέντρου πλην το βάθος του κόμβου.

Ορισμός 3.3.6. 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.

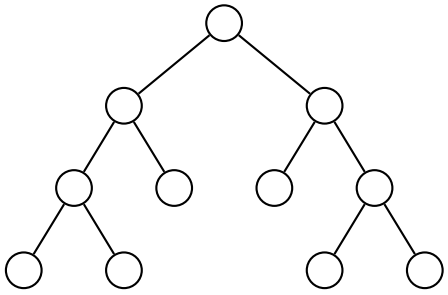
Ένα δυαδικό δέντρο στο οποίο κάθε κορυφή του είναι είτε τερματική είτε έχει ακριβώς δύο παιδιά, λέγεται **γεμάτο δυαδικό δέντρο** (*full binary tree*). Δηλαδή, σε ένα γεμάτο δυαδικό δέντρο δεν υπάρχουν **εκφυλισμένοι εσωτερικοί κόμβοι** (*no degenerate branch nodes*) (σχήμα 3.10).

Ένα γεμάτο δυαδικό δέντρο του οποίου όλα τα φύλλα βρίσκονται στο επίπεδο 0 ονομάζεται **πλήρες δυαδικό δέντρο** (complete binary tree). **Σχεδόν πλήρες** ονομάζεται ένα δυαδικό δέντρο του οποίου όλα τα φύλλα βρίσκονται στο επίπεδο 0 ή στο επίπεδο 1, και όλα τα φύλλα του επιπέδου 0 είναι συγκεντρωμένα αριστερά (ορισμένοι ονομάζουν αυτό το δέντρο πλήρες - δες σχήμα 3.10). Ενα πλήρες δυαδικό δέντρο είναι γεμάτο, το αντίθετο δεν ισχύει.

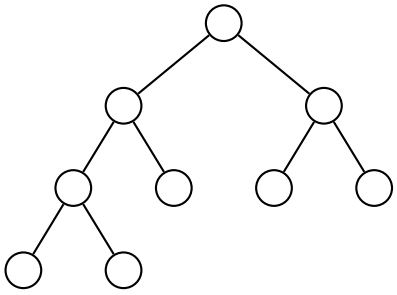
Μια κωδικοποίηση ενός δυαδικού δέντρου στον υπολογιστή μπορεί να γίνει πολύ εύκολα ως εξής: χρησιμοποιούμε ένα μονοδιάστατο πίνακα A και στη



Δυαδικό δένδρο (Binary Tree)



Γεμάτο δυαδικό δένδρο
(Full Binary Tree)



Πλήρες δυαδικό δένδρο
(Complete Binary Tree)

Σχήμα 3.10: Δυαδικά δένδρα

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.

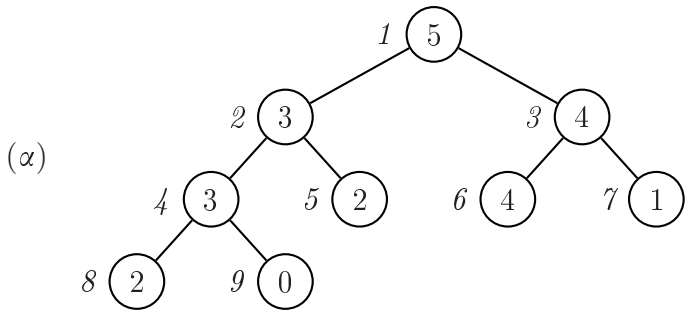
3.4 Σωροί-Ουρά Προτεραιότητας-Heapsort

Ορισμός 3.4.1. Έστω ότι μας δίνονται κάποια στοιχεία για τα οποία ισχύει μια ολική σχέση διάταξης ($<$). Μια δομή δεδομένων η οποία μας επιτρέπει να κάνουμε με αποδοτικό τρόπο εισαγωγή ενός καινούργιου στοιχείου και εξαγωγή (δηλαδή εύρεση και διαγραφή) του μεγαλύτερου στοιχείου, λέγεται **ουρά με προτεραιότητα** (*priority queue*). [ADT: abstract data type, αφηρημένη δομή δεδομένων].

Ένας τρόπος για να παραστήσουμε τις ουρές με προτεραιότητα είναι η

υλοποίηση με σωρό.

Ορισμός 3.4.2. Σωρός (*heap tree*) είναι ένα πλήρες δυαδικό δέντρο στο οποίο η τιμή της ρίζας είναι μεγαλύτερη ή ίση (ή μικρότερη ή ίση) από τις τιμές των υπολοίπων κόμβων και αυτό ισχύει αναδρομικά για την ρίζα κάθε υποδέντρου (σχήμα 3.11).



(β)

1	2	3	4	5	6	7	8	9
5	3	4	3	2	4	1	2	0

Σχήμα 3.11: (α) Σωρός (Heap tree) και (β) ο πίνακας που αντιστοιχεί σε αυτόν

Έστω ότι έχουμε n ακεραίους αποθηκευμένους με τυχαία σειρά σε ένα μονοδιάστατο πίνακα $a[n]$. Θα περιγράψουμε δύο στρατηγικές με τις οποίες μπορούμε να διατάξουμε τα στοιχεία του πίνακα a σε σωρό.

Η πρώτη στρατηγική είναι η παρακάτω: Θεωρούμε το πρώτο στοιχείο του πίνακα ($a[1]$) σαν σωρό και εισάγουμε ένα προς ένα τα υπόλοιπα στοιχεία του πίνακα διατηρώντας κάθε φορά τη δομή του σωρού. Δηλαδή, όπως φαίνεται στον αλγόριθμο 3.1, κάθε στοιχείο ξεκινά από το τέλος του μέχρι στιγμής δέντρου και βρίσκει τη σωστή θέση του ανεβαίνοντας προς την ρίζα. Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n \log n)$. Αυτό συμβαίνει διότι στη χειρότερη περίπτωση (δηλαδή όταν τα στοιχεία βρίσκονται αποθηκευμένα στον πίνακα με αύξουσα σειρά), κάθε στοιχείο πρέπει να διανύσει όλη την απόσταση ως τη ρίζα, συνεπώς αρκεί χρόνος $O(\log n)$.

Παρατήρηση 3.4.3. Η μέση χρονική πολυπλοκότητα είναι $O(n)$ (Άσκηση).

Η δεύτερη στρατηγική κατασκευής ενός σωρού είναι η εξής: Θεωρούμε τον πίνακα a σαν δέντρο και εξετάζουμε ένα προς ένα όλα τα υποδέντρα του

Αλγόριθμος 3.1 Κατασκευή σωρού (insert)

```
procedure insert(var a:array; n:integer);
var item:integer; k:integer;
begin
  item:=a[n]; k:=n div 2; (* εύρεση του γονέα *)
  while ((k>0) and (a[k]<item)) do
    begin a[n]:=a[k]; n:=k; k:=k div 2; end;
  a[n]:=item
end

procedure ConstructHeapInsert(var a:array; n:integer);
var i:integer;
begin
  for i:=2 to n do insert(a,i)
end
```

αρχίζοντας από το τέλος. Κάθε υποδέντρο το μετατρέπουμε σε σωρό και το ενώνουμε με την τελική δομή. Στη χειρότερη περίπτωση, ο χρόνος που χρειάζεται για την κατασκευή ενός σωρού με τη βοήθεια του αλγορίθμου 3.2, είναι $O(n)$[1].

Όμως η χρήση της διαδικασίας *CostructHeapCombine* απαιτεί, όλα τα στοιχεία που θα φτιάξουν το σωρό να είναι διαθέσιμα από την αρχή της διαδικασίας, σε αντίθεση με τη χρήση της διαδικασίας *ConstructHeapInsert* όπου ένα καινούργιο στοιχείο μπορεί να εισαχθεί στο δέντρο οποιαδήποτε χρονική στιγμή.

Παρατήρηση 3.4.4. Η διαδικασία *combine* μπορεί να χρησιμοποιηθεί κι όταν θέλουμε να διαγράψουμε οποιοδήποτε στοιχείο ενός σωρού (όχι μόνο τη ρίζα) χωρίς να χαλάσουμε την ιδιότητα σωρού.

Μια από τις εφαρμογές του σωρού είναι η ταξινόμηση (*sorting*). Στην ταξινόμηση η απλή στρατηγική επιβάλλει συνεχώς να διαλέγουμε από τα στοιχεία που απομένουν, το μεγαλύτερο (ή το μικρότερο). Ένας αλγόριθμος που θα χρησιμοποιούσε αυτή τη στρατηγική όπως είναι, χωρίς καμία βελτίωση της αρχικής σκέψης, θα απαιτούσε στη χειρότερη περίπτωση χρόνο $O(n^2)$ $(n-1$

[1]Αυτό προκύπτει ως εξής: Για επίπεδο i ο αριθμός επαναλήψεων είναι $k-i$, όπου $k = \lceil \log n \rceil$. Οι κόμβοι που υπάρχουν στο επίπεδο i είναι το πολύ 2^{i-1}. Συνεπώς :

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

Αλγόριθμος 3.2 Κατασκευή σωρού (combine)

```
procedure combine(var a:array; i,n:integer);
(* Μετατρέπει το υποδένδρο με ρίζα a[i] σε σωρό, υπό την
προϋπόθεση ότι τα υποδένδρα με
ρίζες τα παιδιά του a[i] είναι σωροί *)
var left, right, largest_child:integer;

begin
  while 2*i ≤ n do (* ο κόμβος i έχει τουλάχιστον ένα παιδί *)
  begin
    left:=2*i; (* αριστερό παιδί *)
    right:=2*i+1; (* δεξί παιδί *)
    largest_child:=left;
    if right ≤ n and a[right]>a[left] then largest_child:=right;
          (* largest_child: η θέση του παιδιού με τη μεγαλύτερη τιμή *)
    if a[i]<a[largest_child] then
    begin
      swap(a[i],a[largest_child]);
      i:=largest_child
    end
    else i:= n div 2 + 1 (*exit while*)
  end
end

procedure CostructHeapCombine(var a:array; n:integer);
var i:integer;
begin
  for i:=n div 2 downto 1 do combine(a,i,n)
end
```

συγκρίσεις για κάθε στοιχείο). Η χρήση σωρού επιτρέπει την εύρεση του μεγαλύτερου στοιχείου και τη διαγραφή του από τα υπόλοιπα σε χρόνο τάξης $O(\log n)$. Έτσι επιτυγχάνεται για όλη τη διαδικασία της ταξινόμησης χρόνος στη χειρότερη περίπτωση της τάξης $O(n \log n)$.

Η μέθοδος που κάνει ταξινόμηση είναι η παρακάτω (αλγόριθμος 3.3): Κατασκευάζουμε από τον δοσμένο πίνακα $a[1..n]$, ένα σωρό με κάποια από τις μεθόδους που ήδη περιγράφηκαν. Έπειτα ανταλλάσσουμε (*swap*) το πρώτο στοιχείο της δομής ($a[1]$) με το τελευταίο ($a[n]$). Έτσι το τελευταίο στοιχείο του πίνακα τώρα είναι το μεγαλύτερο. Στη συνέχεια κάνουμε σωρό τον πίνακα $a[1..n-1]$,

παίρνουμε πάλι το πρώτο στοιχείο και το βάζουμε στη θέση $a[n-1]$, κ.ο.κ. Τελικά, και μετά από χρόνο $O(n \log n)$, ο πίνακας a είναι ταξινομημένος σε αύξουσα σειρά (ascending order).

Αλγόριθμος 3.3 Ταξινόμηση HeapSort

```
procedure HeapSort(var a:array; n:integer);
var i:integer;
begin
  ConstructHeap(a,n); (* Αλγόριθμος 1 ή Αλγόριθμος 2 *)
  for i:=n downto 2 do
    begin swap(a[1],a[i]); combine(a,1,i−1) end
end
```

Ένα εύλογο ερώτημα είναι το εξής: Υπάρχει αλγόριθμος ο οποίος να ταξινομεί με συγκρίσεις στοιχεία σε χρόνο μικρότερο από αυτόν της τάξης $O(n \log n)$ (μιλώντας πάντα για τη χειρότερη περίπτωση); Η απάντηση είναι πως δεν είναι δυνατόν να υπάρξει τέτοιος αλγόριθμος. Κάθε αλγόριθμος που ταξινομεί με συγκρίσεις, μπορεί να παρασταθεί με τη βοήθεια ενός **δέντρου απόφασης** (*decision tree*), δηλαδή ενός δυαδικού δέντρου του οποίου οι εσωτερικές κορυφές αντιπροσωπεύουν μία σύγκριση (μία απόφαση). Το δέντρο που θα χρησιμοποιηθεί για την ταξινόμηση n στοιχείων θα έχει οπωσδήποτε $n!$ φύλλα (όλες οι δυνατές μεταθέσεις των n στοιχείων). Το μήκος του δρόμου απ' τη ρίζα στο φύλλο ενός δέντρου απόφασης μας δίνει τον αριθμό των συγκρίσεων που χρειάστηκαν για την ταξινόμηση που παριστάνει το φύλλο. Το μήκος του μεγαλύτερου απ' τα μονοπάτια, δηλαδή το ύψος του δέντρου μας δίνει το αριθμό των συγκρίσεων στη χειρότερη περίπτωση. Μπορούμε να προσδιορίσουμε το ελάχιστο ύψος ενός δέντρου απόφασης n στοιχείων το οποίο έχει $n!$ φύλλα ως εξής:

$$\left.\begin{array}{ll} h \geq & \log(n!) \\ n! \geq & (\frac{n}{2})^{\frac{n}{2}} \end{array}\right\} \Rightarrow h \geq \frac{n}{2} \log(\frac{n}{2})$$

Σαν συνέπεια έχουμε το ακόλουθο:

Θεώρημα 3.4.5. *Κάθε δέντρο απόφασης για ταξινόμηση n στοιχείων, έχει πολυπλοκότητα $\Omega(n \log n)$.*

Παρατήρηση 3.4.6. Πολυπλοκότητα δέντρου απόφασης λέγεται το ύψος του.

Πόρισμα 3.4.7. *Η ταξινόμηση με συγκρίσεις n στοιχείων, έχει χρονική πολυπλοκότητα $\Theta(n \log n)$.*

Proof. Από το πιο πάνω θεώρημα είδαμε ότι χρειαζόμαστε για την ταξινόμηση n στοιχείων με συγκρίσεις, χρόνο $\Omega(n \log n)$. Όπως είδαμε ο αλγόριθμος 3.3 έχει πολυπλοκότητα $O(n \log n)$. Συνεπώς το πρόβλημα της ταξινόμησης με συγκρίσεις λύνεται σε χρόνο $\Theta(n \log n)$, άρα ο αλγόριθμος 3.3 είναι **βέλτιστος**. □

3.5 Σύνολα - Συστήματα Εισαγωγής και Ανάκτησης Πληροφοριών - Πράξεις σε Σύνολα

3.5.1 Γενικά

Στην σχεδίαση αλγορίθμων τα σύνολα είναι η βάση πολλών σπουδαίων και χρήσιμων **γενικευμένων δομών δεδομένων** (*abstract data types*). Έχουν αναπτυχθεί πολλές τεχνικές υλοποίησης τέτοιων γενικευμένων δομών δεδομένων που βασίζονται σε σύνολα.

Η δομή του συνόλου είναι η βάση πολλών προβλημάτων στα οποία μας ενδιαφέρει η γρήγορη ανάκτηση πληροφοριών (*information retrieval*). Σε αυτά τα προβλήματα, συνήθως έχουμε ένα σύμπαν, καθολικό σύνολο (*universe*) από το οποίο μπορούν να πάρουν στοιχεία όλα τα σύνολα (*sets*) που χρησιμοποιούνται. Οι περιπτώσεις που συναντάμε είναι:

- | sets | $\sim$ | universe |, δηλαδή οι πληθικοί αριθμοί των συνόλων που χρησιμοποιούνται είναι της τάξης του πληθικού αριθμού (cardinality) του καθολικού συνόλου.

- | sets | << |universe|, # operations $\sim$ | universe |, δηλαδή οι πληθικοί αριθμοί των συνόλων που χρησιμοποιούνται είναι πολύ μικρότεροι από τον πληθικό αριθμό του καθολικού συνόλου και επιπλέον ο αριθμός των πράξεων ανάμεσα στα σύνολα είναι πολύ μεγάλος.

- | sets | << | universe |, # operations << |universe|, όμοια με την προηγούμενη περίπτωση, με τη διαφορά ότι ο αριθμός των πράξεων με τα σύνολα είναι μικρός.

Έστω ότι έχουμε ένα σύνολο αναφοράς U με n στοιχεία από το οποίο μπορούμε να κατασκευάσουμε άλλα σύνολα S, τα οποία είναι υποσύνολα του U.

Ένας τρόπος να παραστήσουμε τα σύνολα αυτά S, είναι με τη βοήθεια ενός διανύσματος μήκους n, $S[1..n]$ τέτοιου ώστε $S[i] = 1$ αν το i-οστό στοιχείο του U ανήκει στο S και $S[i] = 0$ σε άλλη περίπτωση.

Άλλος τρόπος για να παραστήσουμε σύνολα S, ξένα μεταξύ τους, είναι με τη βοήθεια των δέντρων.

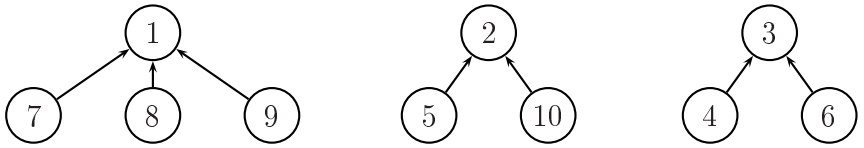
Παράδειγμα 3.5.1. Έστω

$$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\},$$

$$S_1 = \{1, 7, 8, 9\}, S_2 = \{2, 5, 10\}, S_3 = \{3, 4, 6\}.$$

Τα S_1, S_2, S_3 μπορούμε να τα παραστήσουμε όπως φαίνεται στο σχήμα 3.12. Δηλαδή κάθε κορυφή συνδέεται με τον γονέα της. Π.χ.

$PARENT[7]$	$= 1,$	στοιχείο του ίδιου συνόλου
$PARENT[9]$	$= 1,$	στοιχείο του ίδιου συνόλου
$PARENT[10]$	$= 2,$	στοιχείο του ίδιου συνόλου
$PARENT[1]$	$= 0,$	το 1 είναι ρίζα του δέντρου και χρησιμεύει σαν όνομα του συνόλου



$S_1 = \{1, 7, 8, 9\}$ $S_1 = \{2, 5, 10\}$ $S_1 = \{3, 4, 6\}$

Σχήμα 3.12: Παράσταση συνόλων ξένων μεταξύ τους με χρήση δένδρων

Έστω στοιχείο $a \in universe$ και σύνολα $S, S_1, S_2 \subseteq universe$. Οι πιο χαρακτηριστικές πράξεις μεταξύ συνόλων είναι οι παρακάτω:

- Member(a, S): Ελέγχει αν το στοιχείο a ανήκει στο σύνολο S και αν αυτό ισχύει επιστρέφει true αλλιώς false. Search(a, S): Επιστρέφει ένα δείκτη στο στοιχείο a αν $a \in S$ αλλιώς επιστρέφει nil.

- Insert(a, S): Εισάγει το στοιχείο a στο σύνολο S και επιστρέφει το σύνολο $S \bigcup \{a\}$.

- Delete(a, S): Διαγράφει το στοιχείο a από το σύνολο S και επιστρέφει το σύνολο $S \setminus \{a\}$.

- Union(S_1, S_2): Επιστρέφει το σύνολο $S_1 \bigcup S_2$. Υποθέτουμε ότι τα σύνολα S_1, S_2 είναι ξένα (disjoint) για να αποφύγουμε τον έλεγχο για διπλά στοιχεία.

- Find(a): Επιστρέφει το όνομα του συνόλου στο οποίο ανήκει το a. Υποθέτουμε ότι έχουμε διαμέριση σε ξένα σύνολα, άρα το στοιχείο a, θα ανήκει σε ένα ακριβώς σύνολο.

- Split(a, S): Χωρίζει το σύνολο S σε δύο σύνολα S_1, S_2 τέτοια ώστε:

$$S_1 = \{b \mid b \leq a \bigwedge b \in S\} \text{ και } S_2 = \{b \mid b > a \bigwedge b \in S\}$$

  Εδώ υποθέτουμε ότι το σύνολο S είναι ένα σύνολο του οποίου τα στοιχεία έχουν μια γραμμική διάταξη ($\leq$).

- Max(S), Min(S): Επιστρέφει το μεγαλύτερο ή το μικρότερο των στοιχείων του S. Υποθέτουμε πάλι γραμμική διάταξη των στοιχείων του S.

- Successor(a, S): Επιστρέφει το μικρότερο από τα στοιχεία του S που είναι μεγαλύτερο του a (επόμενο στοιχείο). Predecessor(a, S): Ομοίως επιστρέφει το μεγαλύτερο από τα στοιχεία του S που είναι μικρότερο του a (προηγούμενο στοιχείο).

Ανάλογα λοιπόν με το ποιες λειτουργίες από τις παραπάνω χρησιμοποιούμε συχνά, φτιάχνουμε και τις κατάλληλες δομές, υλοποιώντας τα σύνολα με διάφορες τεχνικές.

3.5.2 Δομή λεξικού (Dictionary)

Ορισμός 3.5.2. Ας υποθέσουμε ότι έχουμε ένα σύνολο S και θέλουμε να εκτελούνται γρήγορα οι λειτουργίες της εισαγωγής καινούργιου στοιχείου, διαγραφής παλαιού στοιχείου και ελέγχου για την ύπαρξη κάποιου στοιχείου στο S. Θέλουμε δηλαδή να εκτελείται αποδοτικά μία ακολουθία από διαδικασίες *Insert, Delete* και *Member*. Αυτή τη γενικευμένη αφηρημένη δομή δεδομένων (ADT) την ονομάζουμε **λεξικό** (*Dictionary*).

Η υλοποίηση ενός λεξικού μπορεί να γίνει π.χ. με μια συνάρτηση

$$h\colon universe \to \{0, \ldots, m-1\},$$

που ονομάζεται **συνάρτηση κατακερματισμού** (*hashing function*). Φροντίζουμε ώστε η συνάρτηση h που διαλέγουμε να υπολογίζεται γρήγορα για οποιοδήποτε στοιχείο του καθολικού συνόλου, σε χρόνο δηλαδή $O(1)$. Θεωρούμε ένα πίνακα A (hash table). Κάθε στοιχείο $A[i]$ του πίνακα δείχνει σε μια λίστα η οποία περιέχει εκείνα τα στοιχεία a του καθολικού συνόλου για τα οποία ισχύει $h(a) = i$. Συνεπώς για να κάνουμε *Insert, Delete* και *Member* ένα στοιχείο a, αρκεί να ψάξουμε μόνο τη λίστα στην οποία δείχνει το στοιχείο $A[h(a)]$.

Βέβαια στη χειρότερη περίπτωση, είναι πιθανόν μετά από n εισαγωγές στοιχείων, να έχουμε μια λίστα μήκους σχεδόν n (δηλαδή σχεδόν όλα τα στοιχεία να έχουν πάει στην ίδια λίστα). Σε αυτή την περίπτωση, αν χρειαστεί

να εκτελέσουμε n φορές τις διαδικασίες *Delete* ή *Member*, ο χρόνος που απαιτείται είναι $O(n^2)$. Αν όμως φροντίσουμε ώστε η εκλογή της συνάρτησης h να εξασφαλίζει μια όσο το δυνατό ομοιόμορφη κατανομή των στοιχείων στις λίστες, έτσι ώστε να μην υπάρχει συσσώρευση στοιχείων σε μια λίστα, τότε ο χρόνος αναζήτησης μπορεί να καλυτερεύσει σημαντικά. Για παράδειγμα, αν πρόκειται να εισαχθούν περίπου $n \in O(m)$ στοιχεία, τότε τη στιγμή που εισάγεται το i-οστό στοιχείο, η λίστα στην οποία θα πρέπει να μπει θα έχει αναμενόμενο μήκος $\frac{i-1}{m} < 1$ συνεπώς η κάθε διαδικασία που θα πρέπει να διατρέξει κάποια λίστα θα χρειάζεται περίπου σταθερό χρόνο $O(1)$ και έτσι n διαδικασίες θα χρειάζονται $O(n)$ χρόνο περίπου.

Είναι συνηθισμένο να μην γνωρίζουμε από πριν τον πληθικό αριθμό που μπορεί να έχει το σύνολό μας. Στην περίπτωση αυτή διαλέγουμε μια τιμή m για τον πίνακα (*hash table, bucket table*) και όταν ο αριθμός των στοιχείων γίνει μεγαλύτερος από m, δημιουργούμε ένα καινούργιο πίνακα-στήλη μεγέθους $2m$ και με rehashing (δηλαδή ορίζοντας μια νέα συνάρτηση με πεδίο τιμών αυτή τη φορά το $[0, 2m-1]$), βάζουμε τα στοιχεία στον καινούργιο πίνακα, καταστρέφοντας τον παλιό. Όταν τα στοιχεία γίνουν περισσότερα από $2m$, δημιουργούμε ένα άλλο πίνακα μεγέθους $4m$ κ.ο.κ. Είναι σαφές ότι κάθε φορά η κατάλληλη επιλογή της συνάρτησης κατακερματισμού παίζει σπουδαίο ρόλο προκειμένου να διατηρήσουμε τους χρόνους προσπέλασης μικρούς.

Παράδειγμα 3.5.3. Έστω ότι το σύνολό μας αποτελείται από ακεραίους που μπορούν να πάρουν τιμές στο διάστημα $[0, r], r > n$. Τότε αν χρησιμοποιήσουμε τη συνάρτηση $h(a) = a \bmod m$, όπου m το μέγεθος του τρέχοντα πίνακα-στήλη έχουμε τα παρακάτω: Έστω ότι εισάγουμε τους αριθμούς $1, 5, 8, 3, 9, 6$. Αρχικά επιλέγουμε $m = 2$ και έχουμε :

$$\begin{array}{ll} 0: & \\ 1: & 1, 5 \end{array}$$

Σε αυτό το σημείο επιλέγουμε $m = 4$:

$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 5 \\ 2: & \\ 3: & 3 \end{array}$$

Τέλος με $m = 8$ προκύπτει το παρακάτω:

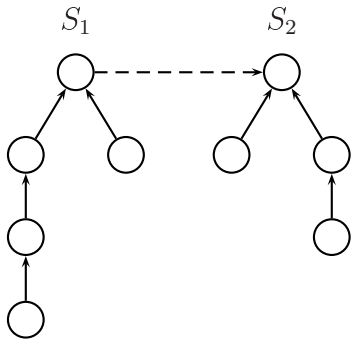
$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 9 \\ 2: & \\ 3: & 3 \\ 4: & \\ 5: & 5 \\ 6: & 6 \\ 7: & \end{array}$$

3.5.3 Δομή UNION - FIND

Ορισμός 3.5.4. Έστω ότι έχουμε διάφορα ξένα μεταξύ τους σύνολα και μας ενδιαφέρει η αποδοτική υλοποίηση μιας **ακολουθίας** από διαδικασίες ένωσης (*Union*) συνόλων και εύρεσης του συνόλου στο οποίο ανήκει κάποιο στοιχείο (*Find*). Μια τέτοια δομή δεδομένων ονομάζεται δομή Union-Find.

Η αναπαράσταση των συνόλων μπορεί να γίνει π.χ. με δέντρα, όπου κάθε κόμβος περιέχει ένα στοιχείο και έχει ένα pointer προς τον γονέα του. Κατά σύμβαση το όνομα του συνόλου είναι το στοιχείο που τυχαίνει να βρίσκεται στη ρίζα.

Η array Parent[i], μας δίνει τον γονέα του κόμβου i και θέτουμε $Parent[root] = 0$. Για να βρούμε την ένωση δύο συνόλων S_1, S_2 αρκεί να μεταβάλλουμε την εγγραφή της Parent σε μια απ' τις δύο ρίζες των S_1, S_2, έτσι ώστε αντί να δείχνει 0, να δείχνει στη ρίζα του άλλου δέντρου (σχήμα 3.13). Η υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 3.4.

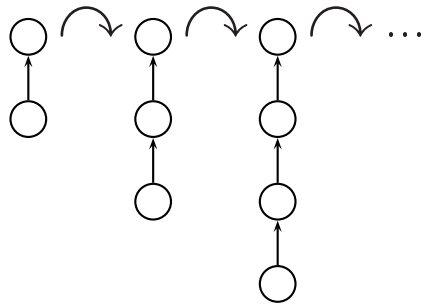


Σχήμα 3.13: Η ένωση των συνόλων S_1 και S_2

Η εκτέλεση της συνάρτησης Union γίνεται σε σταθερό χρόνο $O(1)$, ενώ για την εύρεση του συνόλου S στο οποίο ανήκει το στοιχείο i, αρκεί να βρούμε τη

ρίζα του δέντρου που αναπαριστά το σύνολο S. Η υλοποίηση της συνάρτησης Find φαίνεται στον αλγόριθμο 3.5.

Στη χειρότερη περίπτωση, ένα εκφυλισμένο (*degenarate*) δέντρο μπορεί να προκύψει από την ένωση πολλών συνόλων όπως στο σχήμα 3.14. Έτσι λοιπόν ο χρόνος που χρειάζεται η Find για να διανύσει ένα μονοπάτι του συνόλου-δέντρου με n-στοιχεία-κορυφές είναι στην χειρότερη περίπτωση $O(n)$. Συνεπώς n εκτελέσεις της Find χρειάζονται χρόνο $O(n^2)$.



Σχήμα 3.14: Εκφυλισμένο δένδρο μετά από την ένωση πολλών συνόλων

Μπορούμε να βελτιώσουμε αυτό το χρόνο αν λάβουμε υπόψη μας τον αριθμό των στοιχείων που έχει κάθε σύνολο και συνδέουμε κάθε φορά το σύνολο-δέντρο με τα λιγότερα στοιχεία-κόμβους σε εκείνο με τα περισσότερα, αλλάζοντας τη συνάρτηση Union (Balancing).

Θέτουμε την πληροφορία του πληθικού αριθμού κάθε συνόλου σαν τιμή του γονέα της ρίζας του ($Parent[root] := -\#elements$). Ο αρνητικός αριθμός ξεχωρίζει την τιμή του γονέα της ρίζας από τα υπόλοιπα στοιχεία του συνόλου. Η νέα υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 3.6.

Λήμμα 3.5.5. *Ένα δέντρο που κατασκευάστηκε με τη βοήθεια του αλγόριθμου 3.6 έχει ύψος μικρότερο από* $\lfloor \log n \rfloor + 1$.

Proof. Επαγωγική βάση: $n = 1$. Προφανές.
Επαγωγικό βήμα: Έστω αληθές για όλα τα δέντρα με αριθμό κόμβων $\leq n-1$. Έστω ότι η τελευταία εφαρμογή της διαδικασίας ήταν $union(k, j)$ και ότι το δέντρο j είχε m κόμβους. Χωρίς βλάβη της γενικότητας $1 \leq m \leq \frac{n}{2}$.

Αλγόριθμος 3.4 Διαδικασία ένωσης (Union)

```
function Union (i,j:integer(*set*)):integer;(*set*)
begin
  Parent[i]:=j; return(j)
end
```

Αλγόριθμος 3.5 Διαδικασία εύρεσης (Find)

```
function Find (i:integer(*element*)):integer;(*set*)
begin
  while Parent[i]>0 do i := Parent[i];
  return(i)
end
```

Αλγόριθμος 3.6 Βελτιωμένη διαδικασία ένωσης (Balancing)

```
function Union (i,j:integer(*set*)): integer;(*set*)
var x:integer;(*αρνητικός αριθμός στοιχείων του νέου συνόλου *)
begin
  x:=parent[i]+parent[j];
  if |parent[i]| < |parent[j]| then
    begin parent[i]:=j; parent[j]:=x; return(j) end
  else begin parent[j]:=i; parent[i]:=x; return(i) end
end
```

Περίπτωση 1: Νέο ύψος = ύψος του δέντρου k:

$$\text{ύψος} \leq \lfloor \log(n-m) \rfloor + 1 \leq \lfloor \log n \rfloor + 1.$$

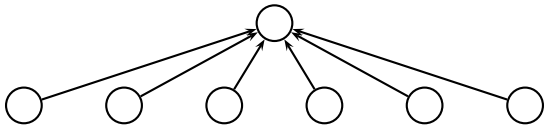
Περίπτωση 2: Νέο ύψος = ύψος του δέντρου $j+1$:

$$\text{ύψος} \leq \lfloor \log m \rfloor + 2 \leq \lfloor \log \frac{n}{2} \rfloor + 2 \leq \lfloor \log n \rfloor + 1.$$

□

Σαν αποτέλεσμα έχουμε ότι ο χρόνος που χρειάζεται μια εκτέλεση της συνάρτησης *Find* είναι $O(\log n)$, δηλαδή n εκτελέσεις χρειάζονται $O(n \log n)$ χρόνο. Μπορούμε να πετύχουμε μια ακόμη καλυτέρευση του χρόνου, αλλάζοντας αυτή τη φορά τη συνάρτηση *Find*. (*Path Compression*) : 'Ενα προς ένα τα στοιχεία-κορυφές που συναντά ο αλγόριθμος στο δρόμο για τη ρίζα, τα «ξεκρεμά» από τη θέση τους και τα «κρεμάει» από τη ρίζα, με αποτέλεσμα το σύνολο-δέντρο να τείνει προοδευτικά να μετασχηματιστεί όπως στο σχήμα 3.15. Η νέα υλοποίηση της συνάρτησης *Find* φαίνεται στον αλγόριθμο 3.7.

Είναι προφανές ότι αυτός ο τρόπος συμφέρει όταν πρόκειται να εκτελεστούν πολύ περισσότερα του ενός *Find*. Αποδεικνύεται μάλιστα ότι n εκτελέσεις της συνάρτησης *Find* χρειάζονται χρόνο $O(n\alpha(n))$, όπου $\alpha(n)$ είναι ψευδοαντίστροφη της συνάρτησης Ackermann(σχεδόν σταθερά). Η ιδέα αυτή είναι παράδειγμα



Σχήμα 3.15: Path compression

αποσβεστικής αποδοτικότητας (*amortization*). Ένα βήμα της *Path Compression* κοστίζει αλλά έτσι εξοικονομείται χρόνος μετά από πολλές εφαρμογές της *Find*.

Αλγόριθμος 3.7 Βελτιωμένη διαδικασία εύρεσης (Path compression)

```
function Find(i:integer(*element*)):integer;
var j,t:integer(*element*);
begin
  j:=i; (* Εύρεση της ρίζας *)
  while parent[j]>0 do j:=parent[j];
  (* Ξεκρέμασμα των φύλλων και κρέμασμα από τη ρίζα *)
  while (i<>j) do
    begin t:=parent[i]; parent[i]:=j; i:=t end;
  return(j)
end
```

Αναφορά R. Tarjan: Efficiency of a good but not linear set union algorithm, JACM, 1975.

3.5.4 Δυαδικά δέντρα αναζήτησης (binary search trees)

Ένας άλλος τρόπος αναπαράστασης συνόλων με μια γραμμική διάταξη, είναι με **δυαδικά δέντρα**. Αυτή η δομή είναι χρήσιμη όταν έχουμε μεγάλα σύνολα και είναι ασύμφορη η χρησιμοποίηση των προηγούμενων μεθόδων. Ένα **δυαδικό δέντρο αναζήτησης** (*binary search tree*) μπορεί να υποστηρίξει αποδοτικά πράξεις όπως *Insert, Delete, Member* και *Min* απαιτώντας κατά μέσο όρο $O(\log n)$ χρόνο για κάθε διαδικασία (όπου n ο πληθάριθμος του συνόλου).

Τα στοιχεία του συνόλου διατάσσονται στο δυαδικό δέντρο ως εξής: Όλα τα στοιχεία-κορυφές που βρίσκονται στο αριστερό υποδέντρο μιας κορυφής x, είναι μικρότερα από το στοιχείο που βρίσκεται στην κορυφή x ενώ τα στοιχεία του δεξιού υποδέντρου είναι μεγαλύτερα. Αυτή η συνθήκη (*binary search tree property*) ισχύει για όλες τις κορυφές του δέντρου (σχήμα 3.16).

Ο έλεγχος για το αν ένα στοιχείο x ανήκει στο σύνολο γίνεται όπως παρακάτω:

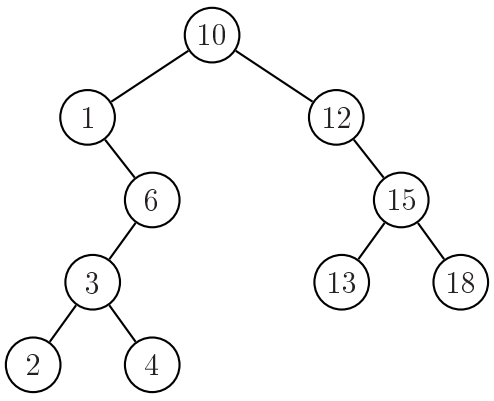
Συγκρίνουμε το στοιχείο x με τη ρίζα του δέντρου

- Αν τα στοιχεία είναι ίσα η διαδικασία τελειώνει.
- Αν το x είναι μικρότερο προχωράμε στο αριστερό υποδέντρο του x και επαναλαμβάνουμε τη διαδικασία.
- Αν το x είναι μεγαλύτερο προχωράμε στο δεξιό υποδέντρο του x και επαναλαμβάνουμε τη διαδικασία.

Η συνάρτηση Member φαίνεται στο αλγόριθμο 3.8. Κάθε κόμβος περιέχει κάποιο στοιχείο του συνόλου και δύο δείκτες, ένα στο αριστερό παιδί και ένα στο δεξιό. Οι διαδικασίες Insert, RetrieveMin και Delete είναι εντελώς ανάλογες και φαίνονται στους αλγόριθμους 3.9, 3.10 και 3.11.

Αλγόριθμος 3.8 Συνάρτηση Member σε δυαδικό δένδρο αναζήτησης

```
function Member(x:elementtype; A:^node):boolean;
begin
  if A=Nil then return(false)
  else if x=A^.element then return(true)
  else if x<A^.element then return(Member(x,A^.leftchild))
  else return(Member(x,A^.rightchild))
end
```



Σχήμα 3.16: Δυαδικό δένδρο αναζήτησης

3.5.5 Ισοζυγισμένα Δέντρα (Balanced Trees)

Όπως είδαμε στην προηγούμενη παράγραφο, χρησιμοποιώντας ένα δυαδικό δέντρο αναζήτησης σαν αναπαράσταση ενός συνόλου, έχουμε έναν αναμενόμενο χρόνο (*average-case complexity*) $O(\log n)$ για κάθε προσπέλαση. Στη χειρότερη περίπτωση όμως, αν εισάγουμε συνεχώς στοιχεία στο σύνολό μας, μπορεί να καταλήξουμε σε εκφυλισμένο δέντρο του οποίου το ύψος προσεγγίζει τον αριθμό των κορυφών του. Έτσι μια προσπέλαση σε αυτό το δέντρο χρειάζεται χρόνο $O(n)$.

Έχουν αναπτυχθεί διάφορες τεχνικές που φροντίζουν να διατηρούν το δέντρο **ισοζυγισμένο** (*balanced*) κατά τη διάρκεια διαφόρων διαδικασιών που το μεταβάλλουν (Insert, Delete κ.α.). Δύο από αυτές τις τεχνικές είναι τα $2-3$ trees και τα AVL trees[2].

Ορισμός 3.5.6. Το $2-3$ tree είναι ένα δέντρο στο οποίο κάθε κορυφή που δεν είναι φύλλο έχει δύο ή τρία παιδιά και κάθε μονοπάτι από τη ρίζα σε ένα φύλλο έχει το ίδιο μήκος.

Τα στοιχεία εισάγονται σε ένα $2-3$ tree συνήθως ως εξής: Κάθε εσωτερική κορυφή έχει δύο θέσεις.

- Στην αριστερή θέση εισάγεται το μεγαλύτερο από τα στοιχεία του υποδέντρου που έχει ρίζα το αριστερό παιδί της κορυφής αυτής.
- Στη δεξιά θέση εισάγεται το μεγαλύτερο από τα στοιχεία του δεξιού υποδέντρου.

[2]Το όνομα AVL προέρχεται από τα αρχικά των δημιουργών του Adel'son-Vel'skii-Landis[1962]

Αλγόριθμος 3.9 Συνάρτηση Insert σε δυαδικό δένδρο αναζήτησης

```
procedure Insert(x:elementtype; var A:^node);
begin
  if A=Nil then
  begin
    new(A); A^.element:=x;
    A^.leftchild:=Nil; A^.rightchild:=Nil
  end
  else if x<A^.element then Insert(x,A^.leftchild)
  else if x>A^.element then Insert(x,A^.rightchild)
  (* Αν x=A^.element, τότε ήδη x in A *)
end
```

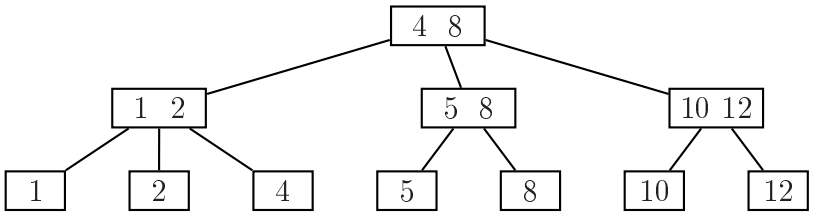
Αλγόριθμος 3.10 Συνάρτηση RetrieveMin σε δυαδικό δένδρο αναζήτησης

```
function RetrieveMin(var A:^node):elementttype; (* Επιστρέφει,
διαγράφοντας από το σύνολο A
   το μικρότερο στοιχείο του *)
begin
  if A^.leftchild=Nil then
    (* Το A δείχνει στο μικρότερο στοιχείο *)
    begin return(A^.element); A:=A^.rightchild end
  else return(RetrieveMin(A^.leftchild))
end
```

Αλγόριθμος 3.11 Συνάρτηση Delete σε δυαδικό δένδρο αναζήτησης

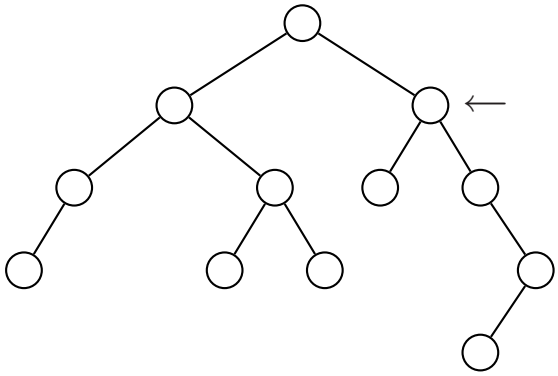
```
procedure Delete(x:elementtype; var A:^node);
begin
  if A<>Nil then
    if x<A^.element then Delete(x, A^.leftchild)
    else if x>A^.element then Delete(x,A^.rightchild)
    else (* x=A^.element *)
         if A^.leftchild=Nil then A:=A^.rightchild
    else if A^.rightchild=Nil then A:=A^.leftchild
% else if A^.leftchild=Nil then A:=A^.rightchild
    else (* ο κόμβος με το x έχει 2 παιδιά*)
      A^.element := RetrieveMin(A^.rightchild)
end
```

Τα φύλλα του δέντρου περιέχουν τα στοιχεία του συνόλου (σχήμα 3.17).



Σχήμα 3.17: 2–3 δένδρο

Ορισμός 3.5.7. Το **AVL tree** είναι ένα δυαδικό δέντρο αναζήτησης με την εξής ιδιότητα, το ύψος του αριστερού και του δεξιού υποδέντρου κάθε κορυφής, διαφέρουν το πολύ κατά 1.



Σχήμα 3.18: Το δένδρο δεν είναι AVL εξαιτίας της σημειωμένης κορυφής

Παράδειγμα 3.5.8. Το δέντρο στο σχήμα 3.18 δεν είναι AVL tree διότι η σημειωμένη κορυφή έχει ένα αριστερό υποδέντρο ύψους 0 και ένα δεξιό υποδέντρο ύψους 2. Παρ' όλα αυτά η ιδιότητα του AVL tree ισχύει σε κάθε άλλη κορυφή.

Οι διαδικασίες αναπτύσσονται έτσι ώστε να διατηρούν τη μία ή την άλλη δομή, με συνέπεια η προσπέλαση στα δέντρα να γίνεται στη χειρότερη περίπτωση (*worst-case complexity*) σε χρόνο $O(\log n)$.

Κεφάλαιο 4

Η τεχνική DIVIDE AND CONQUER για την σχεδίαση αλγορίθμων

4.1 Γενικά

Με δεδομένο ένα πρόβλημα με είσοδο (*input*) μεγέθους n, η τεχνική *DIVIDE AND CONQUER* χωρίζει το πρόβλημα σε μικρότερου μεγέθους υποπροβλήματα, με τέτοιο τρόπο, έτσι ώστε από τις λύσεις των υποπροβλημάτων, να μπορεί εύκολα να κατασκευασθεί η λύση του αρχικού προβλήματος. Μια εφαρμογή αυτής της τεχνικής έχει ήδη αναπτυχθεί, στους αλγορίθμους για τον χειρισμό των δέντρων δυαδικής αναζήτησης.

Στον αλγόριθμο 4.1 δίνεται ένα γενικό σχήμα της μεθόδου για την περίπτωση που το πρόβλημα χωρίζεται σε δύο υποπροβλήματα.

Παρατηρούμε τα εξής:

- small($p..q$) είναι μια boolean συνάρτηση, η οποία παίρνει την τιμή true μόνο στην περίπτωση που το μήκος $(q-p+1)$ της εισόδου είναι αρκετά μικρό, έτσι ώστε η λύση του προβλήματος να συμφέρει να υπολογιστεί απ' ευθείας από τη συνάρτηση g.

- PartitionPoint(p, q) είναι μια συνάρτηση που επιστρέφει ένα σημείο m στο διάστημα $[p, q]$. Στο σημείο αυτό χωρίζονται τα δεδομένα εισόδου κι έτσι δημιουργούνται δύο υποπροβλήματα με αντίστοιχες εισόδους $a[p..m]$ και $a[m+1..q]$.

- Combine(..) είναι μια συνάρτηση που συνδυάζοντας τις λύσεις των υποπροβλημάτων επιστρέφει τη λύση του αρχικού προβλήματος.

Αλγόριθμος 4.1 Γενικό σχήμα μεθόδου Διαίρει και Βασίλευε (Divide and Conquer)

```
function divconq (a:array[p..q] of item): info;
  var m: index;
begin
  if small(p..q) then return(g(a[p..q]))
  else
  begin
    m := PartitionPoint(p,q);
    return(Combine(divconq(a[p..m]), divconq(a[m+1..q])))
  end
end
```

Αν $T(n)$ είναι η συνάρτηση που δίνει τη χρονική πολυπλοκότητα του αλγορίθμου 4.1 και το μέγεθος των υποπροβλημάτων συνεχώς υποδιπλασιάζεται, τότε έχουμε την παρακάτω αναδρομική σχέση:

$$T(n) = \begin{cases} g(n_0) & \text{για } n \leq n_0 \\ 2T(n/2) + f(n) & \text{αλλιώς} \end{cases}$$

όπου $f(n)$ είναι η πολυπλοκότητα των PartitionPoint και Combine. Για $n > 1$ έχουμε:

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

Αν $n = 2^k$, τότε όταν $i = k$ έχουμε:

$$T(n) \quad = \quad ng(1) + \frac{n}{2}f(2) + \frac{n}{4}f(4) + \cdots + \frac{n}{2^j}f(2^j) + \cdots + 2f(\frac{n}{2}) + f(n)$$

Όταν το n δεν είναι δύναμη του 2, χρησιμοποιούμε την αμέσως μεγαλύτερη δύναμη του 2 και το αποτέλεσμα ως προς τάξη μεγέθους O δεν αλλάζει.

Παράδειγμα 4.1.1.

$$T(n) = \begin{cases} a & \text{για } n = 1 \\ 2T(n/2) + cn & \text{για } n > 1 \end{cases} \quad \Rightarrow$$

$$T(n) = n \cdot a + \sum (\frac{n}{2^j}) \cdot c \cdot 2^j = n \cdot a + cn \log_2 n = O(n \log n)$$

Παράδειγμα 4.1.2.

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 2T(n/2) + c & , \text{για } n > 1 \end{cases} \Rightarrow T(n) = n \cdot a + c \cdot (n-1) = O(n)$$

4.2 Δυαδική αναζήτηση (Binary Search)

Στη 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 4.2.

Αλγόριθμος 4.2 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 4.2, 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:

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

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 4.2, σε ταξινομημένο πίνακα στοιχείων χρειάζεται χρόνο τάξης $\Theta(\log n)$.

4.3 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:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 2 = \\ &= 4T(\frac{n}{4}) + 4 + 2 = \\ &= 8T(\frac{n}{8}) + 8 + 4 + 2 = \end{aligned}$$

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

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

$$T(n) = \frac{n}{2}T(2) + 2^{k+1} - 2 = \frac{3}{2}n - 2 \tag{4.1}$$

4.4 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 $\frac{n}{2}$-bit. Αν υποθέσουμε ότι το n είναι μια δύναμη του 2 τότε έχουμε:

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

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

Τότε το γινόμενο των X και Y εκφράζεται ως εξής:

$$X\,Y = ac \cdot 2^n + (ad + bc) \cdot 2^{\frac{n}{2}} + bd \tag{4.2}$$

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 με τάξη 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 εφαρμογές καταλήγουμε στο:

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

Δηλαδή η πράξη του πολλαπλασιασμού αν γίνει με τη χρήση της 4.1, 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:

$$(ad + bc) \;=\; [(a-b)(d-c) + ac + bd] \tag{4.3}$$

Συνδυάζοντας τις 4.1 και 4.2 έχουμε:

$$X\,Y = ac \cdot 2^n + [(a-b)(d-c) + ac + bd]\; 2^{\frac{n}{2}} + bd \tag{4.4}$$

Για τον υπολογισμό της 4.3 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:

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

4.5 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.

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:

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

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.

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.). Για να οδηγηθούμε από το ένα πρόβλημα στο άλλο, αρκεί να πάρουμε υπόψιν μας ότι ισχύουν τα παρακάτω (*Schur*):

1. $\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \cdot \begin{pmatrix} A & 0 \\ 0 & X \end{pmatrix} \cdot \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}$

2. $\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} I & -AC^{-1} \\ 0 & I \end{pmatrix} \cdot \begin{pmatrix} A^{-1} & 0 \\ 0 & X^{-1} \end{pmatrix} \cdot \begin{pmatrix} I & 0 \\ -B^{-1}A & I \end{pmatrix}$

όπου $X = D - CA^{-1}B$. Και για το αντίστροφο:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix}$$

4.6 Ταξινόμηση MergeSort

Άλλη μία εφαρμογή της τεχνικής DIVIDE AND CONQUER είναι στο πρόβλημα της ταξινόμησης n στοιχείων με συγχώνευση. Τα n στοιχεία χωρίζονται σε δύο υποπίνακες και κάθε υποπίνακας ταξινομείται χωριστά. Τελικά οι δύο ταξινομημένοι υποπίνακες συγχωνεύονται σε ένα πίνακα, που έτσι είναι ταξινομημένος. Η ταξινόμηση των υποπινάκων γίνεται αναδρομικά. Η υλοποίηση της αυτής της διαδικασίας (*mergesort*) φαίνεται στον αλγόριθμο 4.3.

Η διαδικασία *merge* έχει πολυπλοκότητα τάξης $O(n)$ ενώ μπορούμε να γράψουμε την παρακάτω αναδρομική σχέση για να περιγράψουμε τον αριθμό συγκρίσεων (χρονική πολυπλοκότητα) της διαδικασίας *Mergesort*:

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

Υποθέτοντας ότι $n = 2^k$ από την παραπάνω σχέση παίρνουμε:

$$T(n) = n \log n - n + 1 = O(n \log n)$$

Αλγόριθμος 4.3 Ταξινόμηση MerseSort

```
procedure Merge(a[p..mid] and b[mid+1..q] into c[p..q]);
(* Συγχωνεύει τα ταξινομημένα a και b δίνοντας το c *)
begin
  i:=p; j:=mid+1; k:=i;
  while (i<=mid) and (j<=q) do
   begin
     if a[i]<b[j] then
       begin c[k]:=a[i]; i:=i+1 end
     else
       begin c[k]:=b[j]; j:=j+1 end;
     k:=k+1
   end
  if i>mid then for l:=j to q do
    begin c[k]:=b[l]; k:=k+1 end
  else for l:=i to mid do
    begin c[k]:=a[l]; k:=k+1 end
end

procedure MergeSort (a[p..q]: list);
begin
  if p=q then return a[p]
  else
  begin
    mid:=(p+q) div 2;
    mergesort(a[p..mid]); mergesort(a[mid+1..q]);
    merge( (a[p..mid] and a[mid+1..q]) into a[p..q] )
  end
end
```

4.7 Ταξινόμηση QuickSort

Στην ταξινόμηση *QuickSort*, με δεδομένη μία ακολουθία n στοιχείων, η τεχνική DIVIDE AND CONQUER χωρίζει την ακολουθία σε δύο μέρη έτσι ώστε κάθε στοιχείο του πρώτου μέρους να είναι μικρότερο από κάθε στοιχείο του δεύτερου. Ο χωρισμός γίνεται από μία διαδικασία Partition ως εξής: Αρχικά επιλέγεται (με κάποιο τρόπο) ένα στοιχείο m της ακολουθίας σαν οδηγός (*pivot*). Έπειτα, με βάση το στοιχείο-οδηγό m, η ακολουθία διασπάται (*split*) σε δύο υποακολουθίες, από τις οποίες η μια περιέχει όλα τα στοιχεία

που είναι μικρότερα από το m και η άλλη περιέχει το m και όλα τα στοιχεία που είναι μεγαλύτερα ή ίσα από αυτό.

Η υλοποίηση αυτής της διάσπασης γίνεται ως εξής: χρησιμοποιούμε δύο δείκτες-δρομείς. Ο ένας από αυτούς (i) ξεκινά από το αριστερό άκρο της ακολουθίας και όσο το στοιχείο στο οποίο δείχνει είναι μικρότερο από τον οδηγό m μετακινείται προς τα δεξιά. Ο άλλος δρομέας (j) ξεκινά από το δεξιό άκρο της ακολουθίας και όσο το στοιχείο στο οποίο δείχνει είναι μεγαλύτερο ή ίσο από τον οδηγό m μετακινείται προς τα αριστερά. Όταν οι δρομείς σταματήσουν τότε αν ισχύει $i < j$ τα στοιχεία στα οποία δείχνουν εναλλάσσονται. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να συναντηθούν οι δρομείς. Τότε ο δρομέας j θα έχει σταματήσει μία θέση αριστερά από τον δρομέα i, διότι το στοιχείο στο οποίο δείχνει ο i είναι μεγαλύτερο ή ίσο του m (αφού έχει σταματήσει σ' αυτό) και συνεπώς ο j προσπερνά και σταματά στη θέση $i-1$ το στοιχείο της οποίας είναι μικρότερο του m. Στην οριακή περίπτωση, που όλα τα στοιχεία της ακολουθίας είναι μεγαλύτερα ή ίσα του στοιχείου-οδηγού, οι δρομείς θα σταματήσουν στο ίδιο σημείο το οποίο είναι το αριστερό άκρο της ακολουθίας.

Αλγόριθμος 4.4 Ταξινόμηση QuickSort

```
function Partition (a: array; p, q: indexdomain): indexdomain;
begin
   i:= p; j:= q; choose m element from a[p..q];
   repeat
      while a[i] < m do i := i+1;
      while a[j] >= m do j := j−1;
      if i < j then swap(a[i], a[j]);
   until i >= j;
   return(i)
end

procedure Quicksort (a, p, q);
begin
  if p<q then begin k:= Partition (a, p, q);
                    Quicksort(a, p, k−1); Quicksort(a, k, q);
              end
end
```

Τελικά, όλα τα στοιχεία που βρίσκονται αριστερά του i είναι μικρότερα από το x και όλα τα στοιχεία που βρίσκονται από τη θέση i μέχρι το τέλος

είναι μεγαλύτερα ή ίσα του x. Τις δύο υποακολουθίες που προκύπτουν τις επεξεργαζόμαστε αναδρομικά με τον ίδιο τρόπο, ώσπου να φτάσουμε σε υποακολουθία με ένα στοιχείο. Η υλοποίηση των παραπάνω φαίνεται στον αλγόριθμο 4.4.

Παρατήρηση 4.7.1. Στην υλοποίηση πρέπει να γίνεται κάποιος έλεγχος του δρομέα j έτσι ώστε αν βγει στη θέση 0, να μην χρησιμοποιηθεί η τιμή $a[0]$ που δεν υπάρχει π.χ. στην περίπτωση που όλα τα στοιχεία της ακολουθίας είναι $\geq m$. Επίσης στην περίπτωση αυτή, δηλαδή που έχει επιλεγεί τέτοιο m ώστε όλα τα στοιχεία να είναι μεγαλύτερα ή ίσα από αυτό, από τις δύο υποακολουθίες που θα προκύψουν, η μία θα έχει μήκος 0 και η άλλη μήκος ίσο με εκείνο της αρχικής. Αυτό μπορούμε να το αποφύγουμε αν, για παράδειγμα, επιλέξουμε σαν στοιχείο-οδηγό το μεγαλύτερο από τα δύο πρώτα στοιχεία, ή (αν αυτά είναι ίσα) από τα τρία πρώτα, κ.ο.κ.

1	2	3	4	5	6	7	8	9	10
$\overrightarrow{\text{Τ}}$	Α	Ξ	Ι	Ν	Ο	Μ	Η	Σ	$\overleftarrow{\text{Η}}$
Η	Α	$\overrightarrow{\Xi}$	Ι	Ν	Ο	Μ	$\overleftarrow{\text{Η}}$	Σ	Τ
Η	Α	Η	Ι	$\overrightarrow{\text{Ν}}$	Ο	$\overleftarrow{\text{Μ}}$	Ξ	Σ	Τ
Η	Α	Η	Ι	$\overleftarrow{\text{Μ}}$	$\overrightarrow{\text{Ο}}$	Ν	Ξ	Σ	Τ
Η	Α	Η	Ι	Μ	Ο	Ν	Ξ	Σ	Τ

Table 4.1: Πρώτη εφαρμογή της Partition σε πίνακα 10 στοιχείων

Στον πίνακα 4.1 φαίνεται η πρώτη εφαρμογή της διαδικασίας Partition σε μια ακολουθία δέκα στοιχείων. Σαν οδηγός έχει επιλεγεί το στοιχείο «N». Οι δρομείς i, j συμβολίζονται με τα $\rightarrow, \leftarrow$ αντίστοιχα.

Στην καλύτερη περίπτωση η επιλογή του στοιχείου-οδηγού είναι τέτοια ώστε η ακολουθία να χωρίζεται σε δύο ίσα μέρη, ενώ στην χειρότερη περίπτωση η ακολουθία χωρίζεται σε δύο μέρη μήκους 1 και $n-1$ στοιχείων. Στην χειρότερη περίπτωση η πολυπλοκότητα είναι της τάξης $O(n^2)$ ενώ στη μέση περίπτωση είναι $O(n)$.

Στον πίνακα 4.2 φαίνεται όλη η διαδικασία της ταξινόμησης της ακολουθίας του πίνακα 4.1. Το στοιχείο-οδηγός σε κάθε βήμα είναι υπογραμμισμένο και οι θέσεις που σταματούν οι δρομείς είναι σημειωμένες με τα αντίστοιχα σύμβολα.

1	2	3	4	5	6	7	8	9	10
$\overrightarrow{\text{T}}$	A	Ξ	I	$\underline{\text{N}}$	O	M	H	Σ	$\overleftarrow{\text{H}}$
H	A	$\overrightarrow{\Xi}$	I	$\underline{\text{N}}$	O	M	$\overleftarrow{\text{H}}$	Σ	T
H	A	H	I	$\overrightarrow{\text{N}}$	O	$\overleftarrow{\text{M}}$	Ξ	Σ	T
H	A	H	I	$\overleftarrow{\text{M}}$	$\overrightarrow{\text{O}}$	$\underline{\text{N}}$	Ξ	Σ	T
H	A	H	I	M	O	N	Ξ	Σ	T
H	A	$\overleftarrow{\text{H}}$	$\overrightarrow{\text{I}}$	M	$\overrightarrow{\text{O}}$	N	$\overleftarrow{\Xi}$	Σ	T
H	A	H	I	M	Ξ	$\overleftarrow{\text{N}}$	$\overrightarrow{\text{O}}$	Σ	T
$\overrightarrow{\text{H}}$	$\overleftarrow{\text{A}}$	H	$\overleftarrow{\text{I}}$	$\overrightarrow{\text{M}}$	Ξ	N	O	Σ	T
$\overrightarrow{\text{A}}$	$\overleftarrow{\text{H}}$	H	I	M	$\overrightarrow{\Xi}$	$\overleftarrow{\text{N}}$	$\overleftarrow{\text{O}}$	$\overrightarrow{\Sigma}$	T
A	H	H	I	M	$\overleftarrow{\text{N}}$	$\overrightarrow{\Xi}$	O	Σ	T
A	$\overleftrightarrow{\text{H}}$	$\underline{\text{H}}$			N	Ξ	$\overleftarrow{\text{O}}$	$\overrightarrow{\Sigma}$	T
	H	H			N	Ξ		Σ	T
	H	H						Σ	T
A	H	H	I	M	N	Ξ	O	Σ	T

Table 4.2: Ταξινόμηση QuickSort σε πίνακα 10 στοιχείων

4.8 Εύρεση του k-οστού μικρότερου στοιχείου

Ένα πρόβλημα που είναι σχετικό με αυτό της ταξινόμησης είναι το πρόβλημα της εύρεσης του k-οστού μικρότερου στοιχείου μιας ακολουθίας n στοιχείων και η τοποθέτησή του στην k-οστή θέση. Με άλλα λόγια ζητάμε το στοιχείο που θα βρισκόταν στην k-οστή θέση της ακολουθίας, αν αυτή ήταν ταξινομημένη κατά αύξουσα σειρά. Ο προφανής τρόπος λύσης, δηλαδή η ταξινόμηση και η εξαγωγή του k-οστού στοιχείου της, έχει πολυπλοκότητα $O(n \log n)$. Χρησιμοποιώντας τον αλγόριθμο 4.5 έχουμε στην χειρότερη περίπτωση πολυπλοκότητα $O(n^2)$, όμως στην μέση περίπτωση η πολυπλοκότητα είναι $O(n)$. Η πολυπλοκότητα χώρου είναι $O(1)$. Ο αλγόριθμος 4.5 χρησιμοποιεί μια παραλλαγή της διαδικασίας Partition της προηγούμενης παραγράφου. Σε αυτήν την παραλλαγή θεωρούμε ότι τα στοιχεία της a είναι διαφορετικά και το αποτέλεσμα είναι: αριστερά του partition point όλα $< m$, δεξιά του partition point όλα τα $> m$ και στο partition point το pivot m. Αν η θέση t στην οποία γίνεται διαμέριση (partition) είναι μεγαλύτερη από k, τότε το k-οστό στοιχείο βρίσκεται στην υποακολουθία $a[p..t-1]$, ειδάλλως το k-οστό μικρότερο στοιχείο της αρχικής ακολουθίας, βρίσκεται στην υποακολουθία $a[t..q]$. Όταν τελειώσει η διαδικασία Selection, στη θέση k του πίνακα a υπάρχει το k-οστό μικρότερο στοιχείο της ακολουθίας.

Αλγόριθμος 4.5 Εύρεση k–οστού μικρότερου στοιχείου

```
function Partition1 (a: array; p, q: indexdomain): indexdomain;
begin
   i:= p; j:= q; choose m element from a[p..q];
   repeat
      while a[i] < m do i := i+1;
      while a[j] > m do j := j−1;
      if i < j then swap(a[i], a[j]);
   until i >= j;
   return(i)
end

procedure Selection1 (a, p, q, k);
begin
    t:= Partition1 (a, p, q);
    if k < t then Selection1(a, p, t−1, k)
    else if k > t then Selection1(a, t+1, q, k−t)
end
```

Μπορούμε να πετύχουμε πολυπλοκότητα της τάξης $O(n)$, για την χειρότερη

περίπτωση, αν επιλέξουμε κατάλληλα το pivot m ως προς το οποίο θα γίνει το partition. Αυτό επιτυγχάνεται με τον παρακάτω αλγόριθμο: χωρίζουμε την αρχική ακολουθία S, σε λ ακολουθίες των 5 στοιχείων οπότε $\lambda = \lfloor \frac{n}{5} \rfloor$ και θα περισσεύουν το πολύ 4 στοιχεία. Στη συνέχεια ταξινομούμε κάθε πεντάδα σε σταθερό χρόνο c και βρίσκουμε το μεσαίο στοιχείο τους. Όλες οι πεντάδες ταξινομούνται σε χρόνο

$$T(n) = \sum_{1}^{\lambda} c = c\lambda = O(n).$$

Αντί να ταξινομήσουμε τις πεντάδες αρκεί να βρούμε το εκάστοτε μεσαίο στοιχείο τους. Μετά παίρνουμε το μεσαίο στοιχείο κάθε πεντάδας και σχηματίζουμε νέα ακολουθία, τη M, η οποία έχει λ στοιχεία. Είναι εύκολο να δούμε ότι η ακολουθία M έχει το $\frac{1}{5}$ των στοιχείων της S διότι $\lambda = \lfloor \frac{n}{5} \rfloor \leq \frac{n}{5}$. Η εύρεση του μεσαίου στοιχείου m της ακολουθίας M γίνεται αναδρομικά σε χρόνο $T(\frac{n}{5})$. Τώρα ισχύει ότι τουλάχιστον το $\frac{1}{4}$ των στοιχείων της S είναι μικρότερα του m και επίσης τουλάχιστον το $\frac{1}{4}$ είναι μεγαλύτερα του m (σχήμα 4.1).

Ιδέα της απόδειξης:
Υπάρχουν τουλάχιστον $\lfloor \frac{n}{10} \rfloor$ στοιχεία (τα στοιχεία της ακολουθίας M), τα οποία είναι μικρότερα από το m. Καθένα απ' αυτά τα $\lfloor \frac{n}{10} \rfloor$ είναι μεγαλύτερο από τουλάχιστον 2 στοιχεία (από τις πεντάδες των 5 στοιχείων). Συνεπώς συνολικά έχουμε τουλάχιστον $3 \cdot \lfloor \frac{n}{10} \rfloor$ στοιχεία τα οποία είναι μικρότερα από το m. Έστω ότι η υποακολουθία S_1 περιέχει όλα τα στοιχεία που είναι μικρότερα του m. Έχουμε:

$$|S_1| \geq 3\lfloor \frac{n}{10} \rfloor \geq 3(\frac{n}{10} - 1) \geq 3\frac{n}{12} = \frac{n}{4}, \forall n \geq 60$$

Στην πραγματικότητα

$$|S_1| \geq \frac{n}{4}, \forall n \geq 35.$$

Άρα η υποακολουθία S_2 περιέχει όλα τα στοιχεία που είναι μεγαλύτερα του m, τότε:

$$|S_2| \leq \frac{3n}{4}$$

Όμοια $|S_1| \leq \frac{3n}{4}$. Στην περίπτωση που $k \leq |S_1|$ το k-οστό μικρότερο στοιχείο βρίσκεται στην ακολουθία S_1 ενώ αν $|S_1| < k$ το ζητούμενο στοιχείο βρίσκεται στην ακολουθία S_2. Σε οποιοδήποτε περίπτωση το πρόβλημα λύνεται σε χρόνο $T(\frac{3n}{4})$. Συνεπώς:

$$T(n) = \left\{ \begin{array}{ll} d & , \text{για } n \leq i \\ T(\frac{n}{5}) + T(\frac{3n}{4}) + cn & , \text{για } n > i \end{array} \right\} = O(n) \text{ (Δες θεώρημα 4.8.1)}$$

Στην παραπάνω σχέση, i είναι μια σταθερά που καθορίζει το μέγιστο μέγεθος των δεδομένων εισόδου για τον οποίο μας συμφέρει να χρησιμοποιήσουμε τον προφανή τρόπο λύσης που αναφέρθηκε στην αρχή. Στον αλγόριθμο 4.8 φαίνεται η επιλογή του k-οστού στοιχείου από την ακολουθία S.

Σημείωση: Το αποτέλεσμα της παραγράφου 4.8 είναι ειδική περίπτωση των παρακάτω γενικών θεωρημάτων περί αναδρομικών σχέσεων.

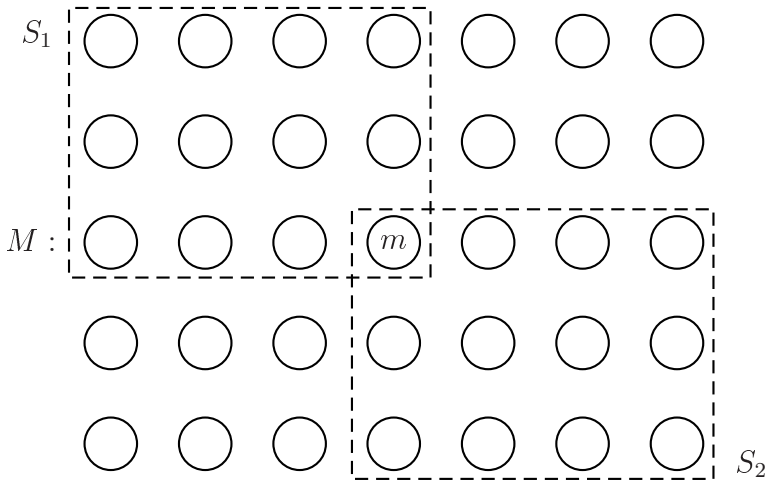
Θεώρημα 4.8.1. *Αν ισχύει ότι* $T(1) = d$ *και* $T(n) = T(an) + T(bn) + cn$ *με* $a + b < 1$ *τότε η συνάρτηση* $T(n)$ *είναι γραμμική, ισχύει δηλαδή ότι* $T(n) = O(n)$.

Proof. Επαγωγή. □

Θεώρημα 4.8.2. *Αν ισχύει* $T(1) = d$ *και* $T(n) = aT(\frac{n}{b}) + cn$ *τότε:*

$$T(n) = \begin{cases} O(n) & , \text{ για } a < b \\ O(n \log_2 n) & , \text{ για } a = b \\ O(n^{\log_b a}) & , \text{ για } a > b \end{cases}$$

Proof. Χρησιμοποιώντας το θεώρημα 4.8.1 και προηγούμενα παραδείγματα. □



Σχήμα 4.1: Κατάλληλη επιλογή του σημείου ως προς το οποίο γίνεται η Partition

Αλγόριθμος 4.6 Βελτιωμένη εύρεση k-οστού μικρότερου στοιχείου

```
function Partition2 (a: array; p, q: indexdomain): indexdomain;
begin
   i:= p; j:= q; (*m is the median of the medians in M*)
   repeat
      while a[i] < m do i := i+1;
      while a[j] > m do j := j−1;
      if i < j then swap(a[i], a[j]);
   until i >= j;
   return(i)
end

procedure Selection2(a, p, q, k);
begin
  if |a|<=50 then sort a;(* με κάποια μέθοδο *)
  else begin (* choose m *)
      preprocess(a); (* 5−tuples, medians, M *)
      Selection2(M, 1, ⌊n/5⌋, ⌊n/10⌋);
      m := M[⌊n/10⌋];
      t:=Partition2(a,p,q);
      if k<t then Selection2(a,p,t−1,k)
      else if k>t then Selection2(a,t+1,q,k−t)
  end
end
```

4.9 To master theorem

Η κύρια μέθοδος παρέχει έναν τρόπο για να λύνουμε αναδρομικές συναρτήσεις του τύπου:

$$T(n) = aT(n/b) + f(n)$$

όπου $a \geq 1$ και $b > 1$ είναι σταθερές και η $f(n)$ είναι μια ασυμπτωτικά θετική συνάρτηση. Η παραπάνω αναδρομική συνάρτηση περιγράφει τον χρόνο εκτέλεσης ενός αλγορίθμου που υποδιαιρεί ένα πρόβλημα μεγέθους n σε a υποπροβλήματα, το καθ' ένα με μέγεθος n/b, όπου a και b είναι θετικές σταθερές. Τα a υποπροβλήματα λύνονται αναδρομικά, σε χρόνο $T(n/b)$ το καθένα. Το κόστος της διαίρεσης του προβλήματος σε υποπροβλήματα και ο συνδυασμός των επιμέρους αποτελεσμάτων περιγράφεται από την συνάρτηση $f(n)$.

Master Theorem

Το θεώρημα 4.8.2 αποτελεί απλή μορφή του γενικότερου θεωρήματος που ακολουθεί. (Σημείωση: όπως στην απλή περίπτωση του θεωρήματος το $T(n)$ εξαρτάται από τη σύγκριση του a με το b έτσι στο γενικό θεώρημα εξαρτάται από την σύγκριση του $f(n)$ με το $n^{\log_b a}$):

Θεώρημα 4.9.1. Master Theorem *Έστω $a \geq 1$ και $b > 1$ σταθερές, $f(n)$ μια συνάρτηση, και η $T(n)$ ορίζεται στους μη αρνητικούς ακεραίους από την αναδρομή*

$$T(n) = aT(n/b) + f(n)$$

(το n/b σημαίνει είτε $\lfloor n/b \rfloor$ είτε $\lceil n/b \rceil$). Τότε η $T(n)$ μπορεί να φραχτεί ασυμπτωτικά ως εξής:

- $T(n) = \Theta(f(n))$, *αν* $f(n) = \Omega(n^{\log_b a+\varepsilon})$ *για κάποια σταθερά* $\varepsilon > 0$,*και αν* $af(n/b) \leq cf(n)$ *για κάποια σταθερά* $c < 1$ *και όλα τα αρκετά μεγάλα* n
- $T(n) = \Theta(f(n) \log_2 n)$, *αν* $f(n) = \Theta(n^{\log_b a})$
- $T(n) = \Theta(n^{\log_b a})$, *αν* $f(n) = O(n^{\log_b a-\varepsilon})$ *για κάποια σταθερά* $\varepsilon > 0$

Σχηματικά, έχουμε

$$T(n) = \begin{cases} \Theta(f(n)) & , \text{για } f(n) = \Omega(n^{\log_b a+e}) \wedge af(\frac{n}{b}) \leq cf(n) \\ \Theta(f(n) \log_2 f(n)) & , \text{για } f(n) = \Theta(n^{\log_b a}) \\ \Theta(n^{\log_b a}) & , \text{για } f(n) = O(n^{\log_b a-e}) \end{cases}$$

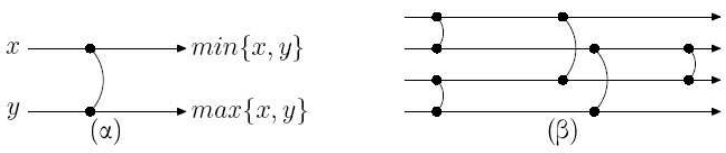
Κεφάλαιο 5

Δίκτυα Ταξινόμησης

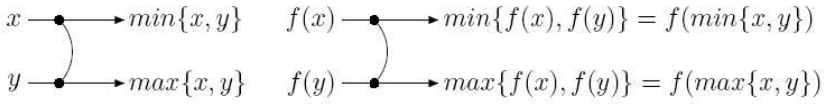
5.1 Εισαγωγή

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 5.1(α) παρουσιάζεται ένας συγκριτής ενώ στο σχήμα 5.1(β) δίνεται ένα παράδειγμα δικτύου ταξινόμησης τεσσάρων 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 5.1(β) είναι 3 και 5 αντίστοιχα.



Σχήμα 5.1: (α) Συγκριτής (Comparator) (β) Δίκτυο ταξινόμησης (Sorter) 4 εισόδων



Σχήμα 5.2: Για αύξουσα $f(x)$, η διάταξη διατηρείται

Στη συνέχεια παρουσιάζεται ένας συστηματικός τρόπος κατασκευής δικτύων ταξινόμησης n εισόδων και δείχνεται η ορθότητά του, με τη βοήθεια της αρχής $0-1$.

5.2 Αρχή 0 – 1

Η αρχή $0-1$ υποδεικνύει ότι αν ένα δίκτυο ταξινομεί οποιαδήποτε ακολουθία n δυαδικών αριθμών, τότε μπορεί να ταξινομήσει και οποιαδήποτε ακολουθία n αριθμών. Μας επιτρέπει να επιβεβαιώσουμε τη λειτουργία του δικτύου χρησιμοποιώντας μόνο δυαδικές εισόδους.

Η αρχή $0-1$ βασίζεται στο παρακάτω λήμμα.

Λήμμα 5.2.1. Αν ένα δίκτυο με είσοδο την ακολουθία $\langle a_1, a_2, \cdots, a_n\rangle$ παράγει ως έξοδο την ακολουθία $\langle b_1, b_2, \cdots, b_n\rangle$, τότε για κάθε αύξουσα συνάρτηση $f(x)$, το δίκτυο με είσοδο την ακολουθία $\langle f(a_1), f(a_2), \cdots, f(a_n)\rangle$ παράγει την ακολουθία $\langle f(b_1), f(b_2), \cdots, f(b_n)\rangle$.

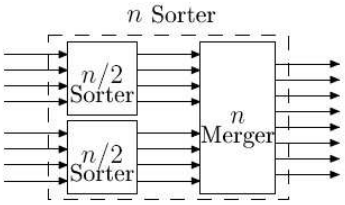
Το λήμμα ισχύει γιατί κάθε συγκριτής διατάσσει δύο εισόδους x, y με τον ίδιο τρόπο που διατάσσει τις εισόδους $f(x), f(y)$ (σχήμα 5.2).

Θεώρημα 5.2.2 (Αρχή 0–1). *Αν ένα δίκτυο σύγκρισης με n εισόδους ταξινομεί οποιαδήποτε ακολουθία δυαδικών αριθμών, τότε ταξινομεί οποιαδήποτε ακολουθία αριθμών.*

Proof. Έστω ότι το δίκτυο ταξινομεί οποιαδήποτε ακολουθία δυαδικών αριθμών, αλλά υπάρχει μη δυαδική ακολουθία αριθμών που δεν ταξινομείται σωστά. Δηλαδή υπάρχει μια ακολουθία εισόδου $\langle a_1, a_2, \cdots, a_n\rangle$ που έχει αριθμούς a_i και a_j για τους οποίους ενώ ισχύει $a_i < a_j$, το δίκτυο τοποθετεί τον a_j πριν τον a_i στην ακολουθία εξόδου. Ορίζουμε την αύξουσα συνάρτηση

$$f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i \end{cases}$$

Σύμφωνα όμως με την παραπάνω πρόταση, αφού το δίκτυο ταξινομεί τον a_j πριν τον a_i στην ακολουθία εξόδου, σημαίνει ότι ο $f(a_j)$ τοποθετείται πριν τον $f(a_i)$ στην ακολουθία εξόδου του δικτύου με είσοδο την δυαδική ακολουθία



Σχήμα 5.3: Δίκτυο Ταξινόμησης n εισόδων (Sorter)

$\langle f(a_1), f(a_2), \cdots, f(a_n)\rangle$. Ξέρουμε όμως πως $f(a_j) = 1$ και $f(a_i) = 0$, άρα το δίκτυο δεν ταξινομεί σωστά την δυαδική ακολουθία $\langle f(a_1), f(a_2), \cdots, f(a_n)\rangle$. Άτοπο. □

5.3 Κατασκευή δικτύου ταξινόμησης

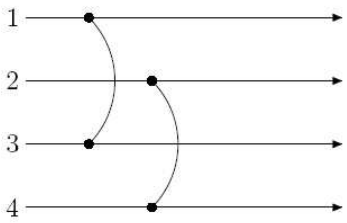
Το δίκτυο ταξινόμησης (Sorter) που παρουσιάζεται, βασίζεται στη φιλοσοφία του αλγόριθμου Merge-Sort (σχήμα 5.3). Το υποδίκτυο Merger δέχεται δύο ταξινομημένες ακολουθίες $\langle a_1, a_2, \cdots, a_{n/2}\rangle$ και $\langle a_{1+n/2}, a_{2+n/2}, \cdots, a_n\rangle$ και παράγει μία ταξινομημένη ακολουθία $\langle b_1, b_2, \cdots, b_n\rangle$.

Αν περιορίσουμε τις τιμές των a_i σε 0 και 1, τότε η ακολουθία $\langle a_1, a_2, \cdots, a_{n/2}, a_n, a_{n-1}, \cdots, a_{1+n/2}\rangle$ είναι της μορφής $0^*1^*0^*$. Ακολουθίες της μορφής $0^*1^*0^*$ και $1^*0^*1^*$ ονομάζονται bitonic και ταξινομούνται με τη βοήθεια των δικτύων Half Cleaner (σχήμα 5.4).

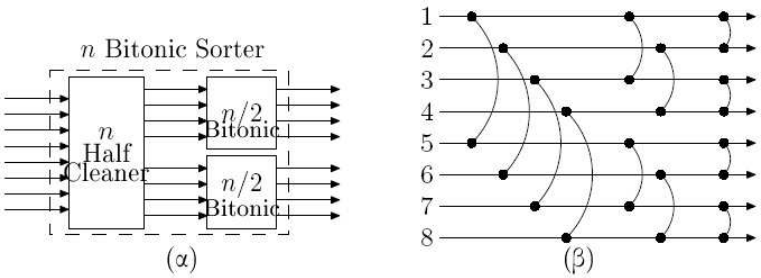
Θα κατασκευάσουμε διαδοχικά τα εξής δίκτυα: Half Cleaner, Bitonic Sorter, Merger, Sorter. Το δίκτυο Half Cleaner αποτελείται από ζυγό αριθμό γραμμών $n = 2k$ και k συγκριτές, καθένας από τους οποίους συνδέει τη γραμμή i με την γραμμή $i+k$ (σχήμα 5.4). Όταν δέχεται ως είσοδο μία bitonic ακολουθία n αριθμών, τότε δίνει στην έξοδό του δύο bitonic ακολουθίες $n/2$ αριθμών και επιπλέον, είτε η πρώτη είναι η $\langle 0, \cdots, 0\rangle$, είτε η δεύτερη είναι η $\langle 1, \cdots, 1\rangle$. Αφήνουμε την απόδειξη για άσκηση. Ο Half Cleaner έχει βάθος 1, μέγεθος k και ταυτίζεται με ένα συγκριτή όταν $n = 2$.

Με τη βοήθεια των Half Cleaners, η κατασκευή ενός Bitonic Sorter ακολουθιών γίνεται αναδρομικά και παρουσιάζεται στο σχήμα 5.5. Αν το βάθος και το μέγεθος του Bitonic Sorter n εισόδων είναι $D(n)$ και $S(n)$ αντίστοιχα, τότε ισχύει

$$\begin{aligned} D(n) &= 1 + D(\frac{n}{2}) \\ &= \log_2 n \end{aligned}$$



Σχήμα 5.4: Παράδειγμα Half Cleaner 4 εισόδων



Σχήμα 5.5: (α) Bitonic Sorter (β) 8 εισόδων

και

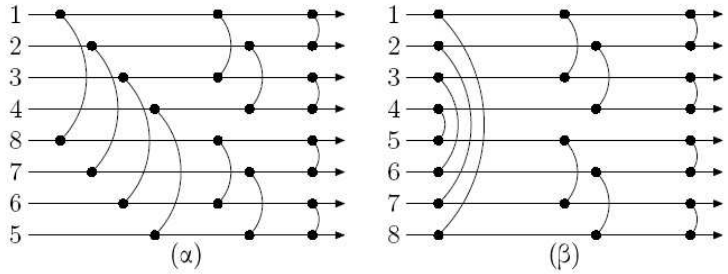
$$\begin{aligned} S(n) &= \frac{n}{2} + 2S(\frac{n}{2}) \\ &= \frac{n}{2} + 2\frac{n}{4} + 4\frac{n}{8} + \cdots + \frac{n}{2} \\ &= \frac{n}{2}\log_2 n \end{aligned}$$

Το δίκτυο Merger προκύπτει από ένα δίκτυο Bitonic Sorter αν αντιστρέψουμε τη σειρά των τελευταίων μισών γραμμών του. Στο σχήμα 5.6 δίνεται παράδειγμα δικτύου Merger, 8 εισόδων, που προκύπτει από τον Bitonic Sorter του σχήματος 5.5(β).

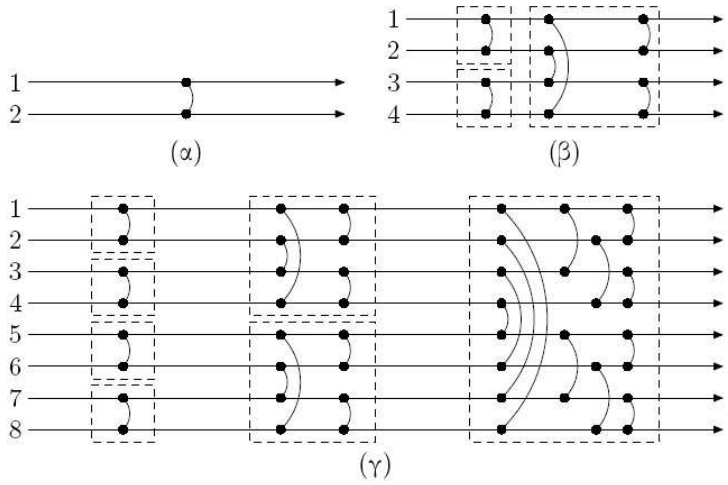
Όπως φαίνεται και στο σχήμα 5.3, ένα δίκτυο ταξινόμησης (Sorter) αποτελείται από υποδίκτυα Merger. Στο σχήμα 5.7 δίνονται παραδείγματα δικτύων ταξινόμησης (Sorter) 2, 4 και 8 εισόδων.

Το βάθος $D(n)$ ενός δικτύου ταξινόμησης (Sorter) είναι:

$$\begin{aligned} D(n) &= D(\frac{n}{2}) + \log_2 n \\ &= 1 + \cdots + \log_2 \frac{n}{4} + \log_2 \frac{n}{2} + \log_2 n \\ &= \frac{\log_2 n(\log_2 n + 1)}{2} \\ &= O(\log^2 n) \end{aligned}$$



Σχήμα 5.6: Merger 8 εισόδων (ισοδύναμες αναπαραστάσεις)



Σχήμα 5.7: Δίκτυα ταξινόμησης (α) 2, (β) 4 και (γ) 8 εισόδων

Το μέγεθος του $S(n)$ είναι:

$$\begin{aligned} S(n) &= 2S(\frac{n}{2}) + \frac{n}{2}\log_2 n \\ &= \frac{n}{2}(1 + \cdots + \log_2 \frac{n}{4} + \log_2 \frac{n}{2} + \log_2 n) \\ &= \frac{n}{2}\frac{\log_2 n(\log_2 n + 1)}{2} \\ &= O(n \log^2 n) \end{aligned}$$

Υπάρχουν και δίκτυα ταξινόμησης μεγέθους $O(n \log n)$ και βάθους $O(\log n)$, τα οποία όμως είναι περισσότερο περίπλοκα από τα Merge-Sort δίκτυα ταξινόμησης που παρουσιάσαμε.

Κεφάλαιο 6

Η Τεχνική GREEDY (άπληστη)

6.1 Γενικά

Τα περισσότερα προβλήματα που επιδέχονται λύση με τη χρησιμοποίηση ενός *greedy* (άπληστου) αλγόριθμου απαιτούν να βρεθεί ένα υποσύνολο των δεδομένων εισόδου το οποίο να ικανοποιεί κάποιους **περιορισμούς** (*constraints*) και να είναι βέλτιστο ως προς κάποιο αντικειμενικό κριτήριο. Κάθε υποσύνολο του συνόλου δεδομένων εισόδου που ικανοποιεί τους περιορισμούς λέγεται **εφικτή** (*feasible*) λύση. Η απαίτηση του προβλήματος είναι να βρεθεί όχι απλώς μια εφικτή λύση, αλλά μια εφικτή λύση η οποία ελαχιστοποιεί ή μεγιστοποιεί μια δεδομένη **αντικειμενική συνάρτηση** (*objective function*). Συνήθως είναι εύκολο να βρεθεί μια εφικτή λύση, όχι όμως και μια βέλτιστη.

Ένας άπληστος αλγόριθμος κάνει πάντα την επιλογή που φαίνεται καλύτερη τη δεδομένη χρονική στιγμή. Δηλαδή κάνει την τοπικά βέλτιστη επιλογή με την ελπίδα ότι αυτή η επιλογή θα τον οδηγήσει στην ολικά βέλτιστη λύση. Οι greedy αλγόριθμοι δεν δίνουν πάντα βέλτιστες λύσεις. Επειδή ο άπληστος αλγόριθμος δουλεύει με ένα κριτήριο τοπικής βελτιστότητας (local optimality measure), για να δίνει βέλτιστη λύση για κάποιο πρόβλημα, θα πρέπει στο συγκεκριμένο πρόβλημα να ισχύει ότι η τοπικά βέλτιστη επιλογή είναι πράγματι και ολικά βέλτιστη.

Ο αλγόριθμος ξεκινά με την κενή λύση και σε κάθε βήμα την μεγαλώνει με τέτοιο τρόπο ώστε η επιλογή που κάνει να είναι τοπικά η καλύτερη (ανάλογα με το ορισμό της αντικειμενικής συνάρτησης) και συγχρόνως εφικτή (να ικανοποιεί τους περιορισμούς). Τα δεδομένα είναι διατεταγμένα σύμφωνα με μια διαδικασία επιλογής, η οποία βασίζεται σε κάποιο κανόνα βελτιστοποίησης. Ο κανόνας μπορεί να είναι η αντικειμενική συνάρτηση, μπορεί όμως και όχι. Σε κάθε πρόβλημα μπορεί να βρεθούν πολλοί κανόνες βελτιστοποίησης οι οποίοι να μην οδηγούν όλοι σε βέλτιστη λύση. Στον αλγόριθμο 6.1 παρουσιάζεται η

Αλγόριθμος 6.1 Γενικό σχήμα άπληστου (greedy) αλγόριθμου

```
function Greedy (a:set of elements) : solution;
var
  x:item;
begin
  solution:=empty;
  for all elements of a do
  begin
    (* χρήση των κανόνων βελτιστοποίησης *)
    x:=OptSelectRemove(a);
    (* έλεγχος αν πληρούνται οι περιορισμοί *)
    if feasible(solution+{x}) then solution:=solution+{x}
    (* η λύση μεγαλώνει και ενημερώνεται η αντικειμενική συνάρτηση *)
  end;
  return(solution)
end
```

ουσία της μεθόδου.

6.2 Βέλτιστη αποθήκευση, optimal storing

Στο πρόβλημα της βέλτιστης αποθήκευσης έχουμε μια μαγνητική ταινία μήκους L και n αρχεία για να τα αποθηκεύσουμε. Το αρχείο i έχει μήκος l_i και ισχύει ότι

$$\sum_{i=1}^{n} l_i \leq L$$

(δηλαδή χωράνε όλα τα αρχεία στην ταινία). Με t_j συμβολίζουμε το χρόνο που χρειάζεται για να προσπελάσουμε το αρχείο i_j από την αποθηκευμένη μετάθεση των αρχείων την οποία συμβολίζουμε με $P : i_1, i_2, \ldots, i_n$. Ισχύει ότι

$$t_j \sim \sum_{k=1}^{j} l_{i_k}$$

δηλαδή ότι ο χρόνος που χρειάζεται για να προσπελάσουμε το i_j αρχείο είναι ανάλογος του συνολικού μήκους των αρχείων $i_1, \ldots, i_j$.

Αν όλα τα αρχεία αναζητούνται με την ίδια περίπου συχνότητα, τότε ο αναμενόμενος μέσος χρόνος ανάκτησης (*mean retrieval time*, MRT) ενός

αρχείου είναι:

$$\text{MRT} = \frac{1}{n}\sum_{j=1}^{n} t_j$$

Θέλουμε να προσδιορίσουμε μια μετάθεση αποθήκευσης των αρχείων έτσι ώστε ο μέσος χρόνος ανάκτησης (MRT) να γίνει ελάχιστος. Ελαχιστοποίηση του MRT ισοδυναμεί με ελαχιστοποίηση της συνάρτησης

$$Z(P) = \sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_k}$$

Είναι φανερό ότι $Z(P) = l_{i_1} + (l_{i_1} + l_{i_2}) + \cdots + (l_{i_1} + l_{i_2} + \cdots + l_{i_n})$. Η εφαρμογή της άπληστης μεθόδου εντοπίζεται στην επέκταση της ακολουθίας $i_1, i_2, \ldots, i_{s-1}$ έτσι ώστε να είναι βέλτιστο το $Z(i_1, i_2, \ldots, i_s)$. Αυτό επιτυγχάνεται αν στο ήδη σχηματισμένο $\sum_{k=1}^{s-1} l_{i_k}$ το l_{i_s} που θα προστεθεί είναι αυτό του μικρότερου αρχείου από αυτά που απομένουν. Συνεπώς ταξινομούμε τα αρχεία κατά το μήκος τους και τα αποθηκεύουμε στην ταινία με αύξουσα διάταξη μήκους.

Θεώρημα 6.2.1. *Η αποθήκευση των αρχείων με αύξουσα διάταξη ως προς το μήκος τους ελαχιστοποιεί την ποσότητα $Z(P)$.*

Proof. Έστω ότι η βέλτιστη διάταξη των αρχείων, που ελαχιστοποιεί το $Z(P)$, δεν είναι αύξουσα ως προς το μήκος τους. Δηλαδή υπάρχει κάποιο k τέτοιο ώστε $l_{ik} > l_{i_{k+1}}$. Ισχύει:

$$Z(P') = nl_{i_1} + (n-1)l_{i_2} + \ldots + (n-k+1)l_{i_k} + (n-k)l_{i_{k+1}} + \ldots + l_{i_n}$$

Εναλλάσσουμε τις θέσεις των αρχείων i_k και i_{k+1} και έχουμε:

$$Z(P') = nl_{i_1} + (n-1)l_{i_2} + \ldots + (n-k+1)l_{i_{k+1}} + (n-k)l_{i_k} + \ldots + l_{i_n}$$

$$Z(P') = Z(P) + l_{i_{k+1}} - l_{i_k} < Z(P)$$

Αυτό όμως είναι αδύνατο αφού το $Z(P)$ είναι ελάχιστο. Ο χρόνος που χρειάζεται είναι $O(n \log n)$. □

6.3 Το πρόβλημα του σακιδίου, Knapsack Problem

Το πρόβλημα είναι το εξής: Έχουμε ένα σακίδιο το οποίο χωράει αντικείμενα συνολικού βάρους το πολύ M, και n «συνεχή» αντικείμενα, από τα οποία

μπορούμε να πάρουμε ένα κλάσμα της ποσότητάς τους. Κάθε αντικείμενο i έχει βάρος w_i και κόστος p_i ενώ x_i είναι το κλάσμα του αντικειμένου που μπαίνει στο σακίδιο. Απαιτείται φυσικά $p_i > 0$ και $w_i > 0$, για όλα τα i. Ο στόχος είναι να γεμίσουμε το σακίδιο με τέτοιες ποσότητες έτσι ώστε να πετύχουμε το μέγιστο κόστος.

Η αντικειμενική συνάρτηση που πρέπει να μεγιστοποιηθεί είναι αυτή του ολικού κόστους του σακιδίου, δηλαδή του $\sum_n p_i x_i$. Είναι σαφές ότι πρέπει να ικανοποιούνται και οι περιορισμοί:

$$\begin{cases} \sum_n w_i x_i \leq M \\ 0 \leq x_i \leq 1 \end{cases}$$

$$p_i > 0 \wedge w_i > 0$$

- Περίπτωση 1: Ισχύει ότι $\sum_{i=1}^{n} w_i \leq M$.

  Τότε το πρόβλημα έχει την παρακάτω προφανή λύση: Για κάθε αντικείμενο i θέτουμε $x_i = 1$, παίρνουμε δηλαδή όλες τις ποσότητες όλων των αντικειμένων. Είναι επίσης προφανές ότι η λύση αυτή είναι βέλτιστη.

- Περίπτωση 2: Ισχύει ότι $\sum_{i=1}^{n} w_i > M$.

  Στην περίπτωση αυτή ζητάμε τη βέλτιστη εφικτή λύση για την οποία μεγιστοποιείται το $\sum p_i x_i$ και ισχύει ότι $\sum w_i x_i = M$. Ταξινομούμε λοιπόν τα αντικείμενα ως προς το κλάσμα $\frac{p_i}{w_i}$ κατά φθίνουσα διάταξη και σύμφωνα με αυτή τοποθετούμε τα αντικείμενα στο σακίδιο. Όταν δεν χωράει ολόκληρη η ποσότητα ενός αντικειμένου επιλέγουμε τέτοιο x_i έτσι ώστε $\sum w_i x_i = M$.

Ο αλγόριθμος αυτός έχει πολυπλοκότητα $O(n \log n)$ και ισχύει το εξής:

Θεώρημα 6.3.1. *Ο άπληστος αλγόριθμος για το πρόβλημα του σακιδίου δίνει τη βέλτιστη λύση.*

Proof. Έστω ότι ο άπληστος αλγόριθμος δίνει την παρακάτω λύση:

1	2	$\ldots$	k	$\ldots$	n
$\frac{p_1}{w_1}$	$\frac{p_2}{w_2}$	$\ldots$	$\frac{p_k}{w_k}$	$\ldots$	$\frac{p_n}{w_n}$
x_1	x_2	$\ldots$	x_k	$\ldots$	x_n

και έστω ότι μια βέλτιστη λύση είναι:

1	2	$\ldots$	k	$\ldots$	n
$\frac{p_1}{w_1}$	$\frac{p_2}{w_2}$	$\ldots$	$\frac{p_k}{w_k}$	$\ldots$	$\frac{p_n}{w_n}$
x'_1	x'_2	$\ldots$	x'_k	$\ldots$	x'_n

Έστω ελάχιστο k τέτοιο ώστε $x_i = x'_i, \forall i < k$ και $x_k \neq x'_k$. Ισχύουν:

$$\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} x'_i w_i = M \Rightarrow \sum_{i=k}^{n} x_i w_i = \sum_{i=k}^{n} x'_i w_i (= M - \sum_{i=1}^{k-1} x_i w_i) \quad (6.1)$$

(α) $x_k < x'_k$. Αν $x_k = 1$ τότε θα έπρεπε $x'_k > 1$ το οποίο είναι αδύνατο. Αν $x_k < 1$, σημαίνει ότι ο άπληστος αλγόριθμος δεν πήρε ολόκληρο το αντικείμενο διότι δεν χωρούσε στο σάκο και μάλιστα πήρε το μεγαλύτερο δυνατό x_k ώστε να γεμίσει ο σάκος. Άρα δεν μπορεί $x_k < x'_k$.

(β) $x_k > x'_k$. Από τη σχέση 6.1 έχουμε:

$$\sum_{i=k+1}^{n} (x'_i - x_i) w_i = (x_k - x'_k) w_k \Rightarrow \sum_{i=k+1}^{n} (x'_i - x_i) w_i \frac{p_k}{w_k} = (x_k - x'_k) w_k \frac{p_k}{w_k}$$

Τα $\frac{p_i}{w_i}$ είναι σε φθίνουσα διάταξη και συνεπώς ισχύει $\frac{p_k}{w_k} \geq \frac{p_i}{w_i}, \forall i > k$. Συνεπώς

$$\sum_{i=k+1}^{n} (x'_i - x_i) w_i \frac{p_k}{w_k} \geq \sum_{i=k+1}^{n} (x'_i - x_i) p_i$$

Δηλαδή

$$(x_k - x'_k) p_k \geq \sum_{i=k+1}^{n} (x'_i - x_i) p_i \quad \Rightarrow \sum_{i=k}^{n} x_i p_i \geq \sum_{i=k}^{n} x'_i p_i$$
$$\Rightarrow \sum_{i=1}^{n} x_i p_i \geq \sum_{i=1}^{n} x'_i p_i$$

και επειδή $\sum_{i=1}^{n} x'_i p_i = MAXIMUM$, στην τελευταία σχέση ισχύει η ισότητα. Δηλαδή η λύση που μας δίνει ο άπληστος αλγόριθμος είναι επίσης βέλτιστη.

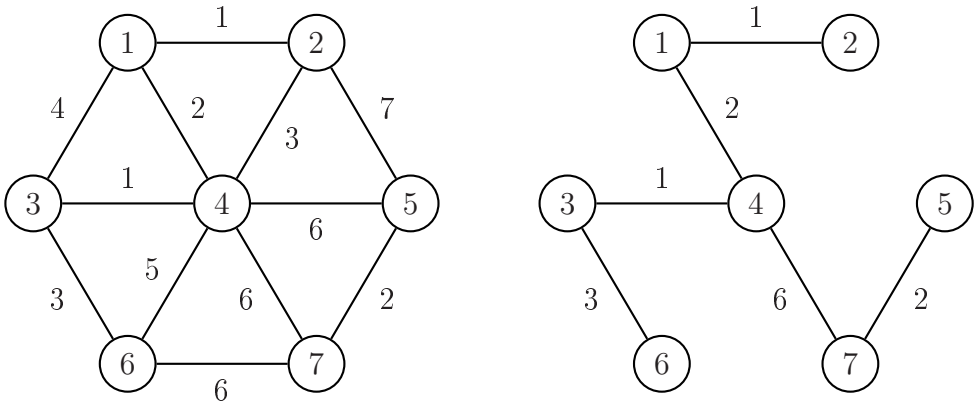
□

6.4 Ελαχίστου κόστους συνδετικό δέντρο, minimum cost spanning tree

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

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

δηλαδή ένα δέντρο που συνδέει όλες τις κορυφές του V και το συνολικό κόστος των πλευρών του είναι το ελάχιστο δυνατό. Το δέντρο αυτό λέγεται ελαχίστου κόστους συνδετικό δέντρο του γράφου G (σχήμα 6.1).



Σχήμα 6.1: Ο γράφος G και ένα ελαχίστου κόστους συνδετικό δένδρο αυτού

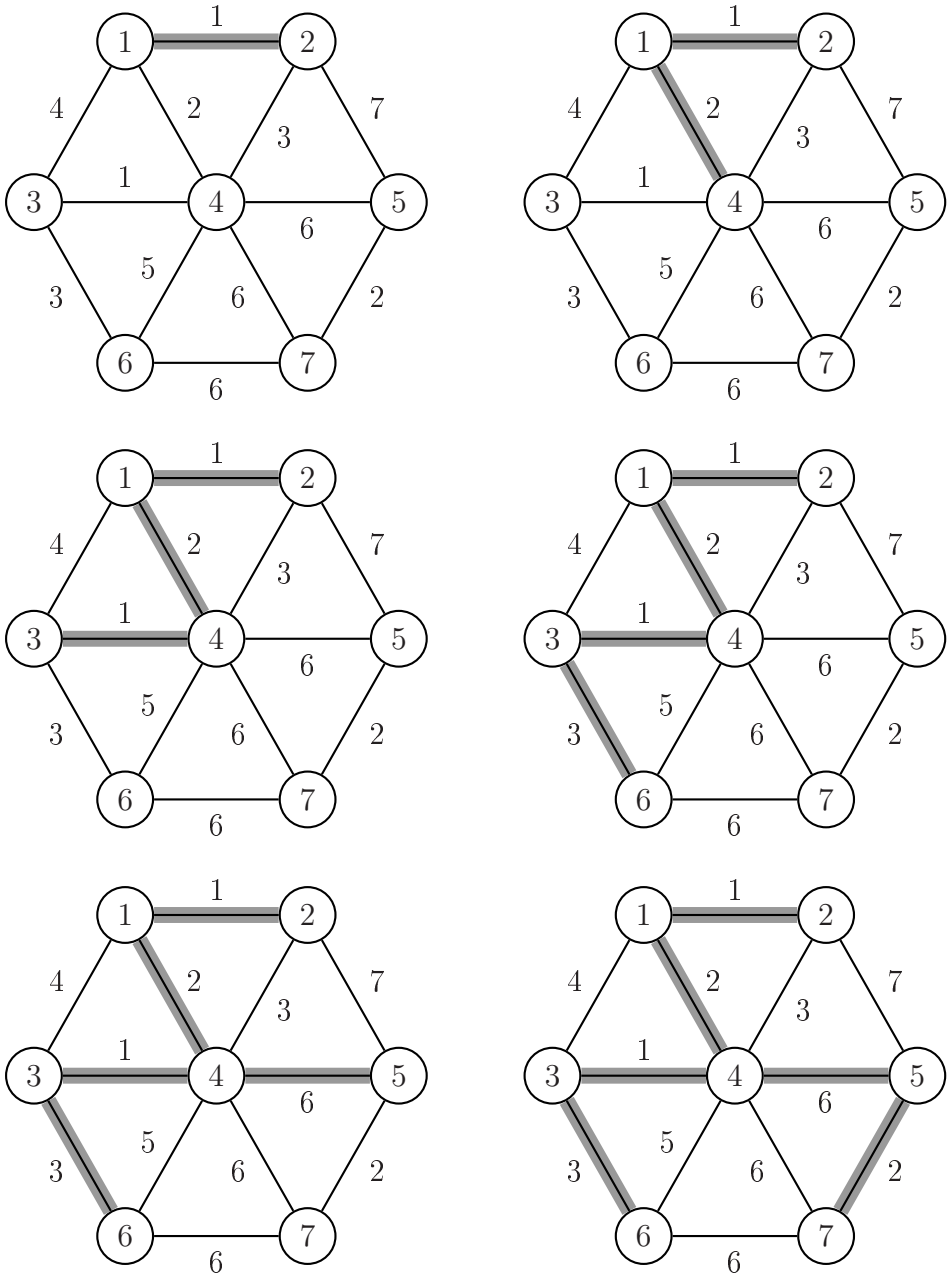
Τα ελάχιστα συνδετικά δέντρα βρίσκουν εφαρμογή στο σχεδιασμό τηλεπικοινωνιακών δικτύων. Οι κόμβοι του γράφου αναπαριστούν πόλεις και οι ακμές αναπαριστούν πιθανούς τηλεπικοινωνιακούς συνδέσμους μεταξύ των πόλεων. Το κόστος κάθε ακμής είναι το κόστος κατασκευής του συγκεκριμένου συνδέσμου για το τελικό 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 πλευρά-πλευρά (*edge-by-edge*) ακολουθώντας είτε το κριτήριο του *Prim* είτε το κριτήριο του *Kruskal*, είτε άλλα κριτήρια.

Κριτήριο του Prim: Διαλέγουμε κάθε φορά την πλευρά που έχει το ελάχιστο κόστος έτσι ώστε ο νέος υπογράφος να παραμένει δέντρο. Η υλοποίηση που χρησιμοποιεί αυτό το κριτήριο φαίνεται στον αλγόριθμο 6.2 ενώ στο σχήμα 6.2 φαίνεται η ακολουθία των πλευρών που προστίθενται στο ελάχιστο συνδετικό δέντρο μέχρι αυτό να πάρει την τελική μορφή του.

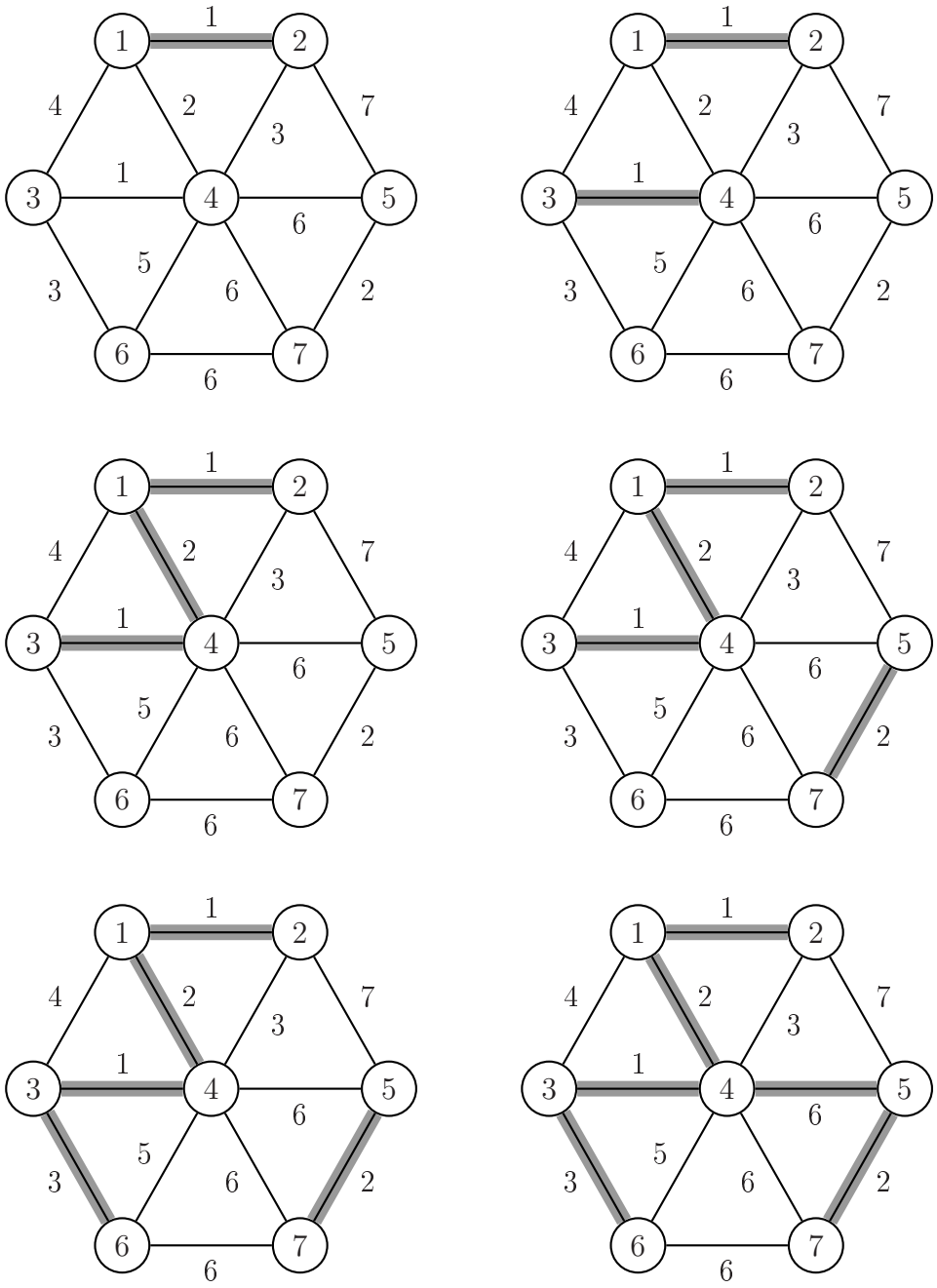
Κριτήριο του Kruskal: Διαλέγουμε κάθε φορά την πλευρά ελαχίστου κόστους έτσι ώστε ο νέος υπογράφος να μην έχει κύκλους. Σχηματίζουμε έτσι ένα δάσος το οποίο τελικά γίνεται δέντρο. Η υλοποίηση που χρησιμοποιεί αυτό το κριτήριο φαίνεται στον αλγόριθμο 6.3 ενώ στο σχήμα 6.3 φαίνεται η ακολουθία των πλευρών που προστίθενται στο ελάχιστο συνδετικό δέντρο μέχρι αυτό να πάρει την τελική μορφή του.

Αλγόριθμος 6.2 Άπληστη μέθοδος εύρεσης ελαχίστου κόστους συνδετικού δέντρου (minimum cost spanning tree) με το κριτήριο του Prim

```
procedure Prim (E: set of edges;
                cost: array[1..n,1..n] of real;
                var tree: array[1..n−1] of edges;
                var mincost : real);
  var DistFromTree:array[1..n] of real;
      Edge:array[1..n] of edges; i,j,k,l : integer;
  (* To DistFromTree[i] περιέχει το ελάχιστο από τα κόστη των πλευρών
     που συνδέουν τον κόμβο i με το μέχρι στιγμής κατασκευασμένο
     συνδετικό δέντρο και το Edge[i] περιέχει την πλευρά με αυτό
     το κόστος *)
begin
  tree[1]:=(k,l); mincost:=cost[k,l]; (* (k,l)= edge with minimum cost *)
  for i:=1 to n do
  begin DistFromTree[i]:=min(cost[k,i],cost[l,i]); update Edge[i] end;
  for i:=2 to n−1 do
  begin
    select j such that DistFromTree[j] is minimum but<>0;
    tree[i]:=Edge[j]; mincost:=mincost+DistFromTree[j];
    DistFromTree[j]:=0;
    for k:=1 to n do begin update DistFromTree[k]; update Edge[k] end
  end
end
```



Σχήμα 6.2: Εφαρμογή του κριτηρίου του Prim για την εύρεση ελαχίστου κόστους συνδετικού δένδρου



Σχήμα 6.3: Εφαρμογή του κριτηρίου του Kruskal για την εύρεση ελαχίστου κόστους συνδετικού δένδρου

Αλγόριθμος 6.3 Άπληστη μέθοδος εύρεσης ελαχίστου κόστους συνδετικού δέντρου (minimum cost spanning tree) με το κριτήριο του Kruskal

```
procedure Kruskal (...);
(*θεωρεί ακμές ταξινομημένες κατά βάρος, αρκεί όμως να είναι σε heap*)
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
```

Αν $n = |V|$ και $e = |E|$ τότε ο αλγόριθμος 6.2 (με το κριτήριο του Prim) έχει πολυπλοκότητα $O(n^2)$ ενώ ο αλγόριθμος 6.3 (με το κριτήριο του Kruskal) έχει πολυπλοκότητα $O(e \log e)$. Για συνεκτικούς γράφους ισχύει:

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

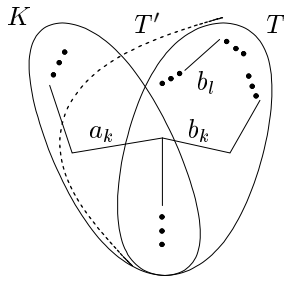
Βλέπουμε λοιπόν ότι ο αλγόριθμος του *Kruskal* έχει καλύτερη απόδοση στους αραιούς (*sparse*) γράφους όπου η πολυπλοκότητά του είναι $O(n \log n)$, σε αντίθεση με τον αλγόριθμο του *Prim* που έχει πολυπλοκότητα $O(n^2)$.

Θεώρημα 6.4.1. *Η λύση που δίνει ο αλγόριθμος του Kruskal στο πρόβλημα του συνδετικού δέντρου είναι η βέλτιστη.*

Proof. Έστω $a_1, a_2, \ldots, a_n$ οι ακμές σε αύξουσα σειρά βάρους στο δέντρο K, το οποίο προκύπτει από τον αλγόριθμο του *Kruskal*. Έστω $b_1, b_2, \ldots, b_n$ οι ακμές σε αύξουσα σειρά βάρους στο δέντρο T το οποίο είναι το βέλτιστο συνδετικό δέντρο ελαχιστού κόστους. Έστω ότι τα δύο δέντρα δεν ταυτίζονται, δηλαδή ότι υπάρχει i για το οποίο $a_i \neq b_i$. Έστω k το ελάχιστο τέτοιο i, δηλαδή:

$$\begin{cases} a_j = b_j, \forall j < k \\ a_k \neq b_k \end{cases}$$

Περίπτωση 1: Το κόστος της a_k είναι μεγαλύτερο από το κόστος της b_k. Τότε στο $k-$οστό βήμα ο αλγόριθμος θα προτιμούσε την πλευρά b_k από την πλευρά a_k, διότι, αφού το μέχρι στιγμής δέντρο είναι το ίδιο δεν σχηματίζεται κύκλος ούτε με το b_k. Άτοπο.



Σχήμα 6.4: Για την απόδειξη ορθότητας αλγορίθμου Kruskal

Περίπτωση 2: Το κόστος της a_k είναι μικρότερο ή ίσο από το κόστος της b_k. Στην περίπτωση αυτή προσθέτουμε στο δέντρο T την ακμή a_k κι έτσι σχηματίζεται ένας κύκλος. Ο κύκλος αυτός πρέπει να περιέχει μια ακμή b_l με $l \geq k$ η οποία δεν ανήκει στο δέντρο K, αλλιώς ο κύκλος θα υπήρχε στο δέντρο K. Είναι φανερό ότι το κόστος της b_l είναι μεγαλύτερο ή ίσο από το κόστος της a_k, διότι ο αλγόριθμος Kruskal προτίμησε την a_k από την b_l. Αν αφαιρέσουμε από το T' (βλέπε σχήμα 6.4) την b_l τότε προκύπτει ένα δέντρο με μικρότερο ή ίσο βάρος από το ελάχιστο. Αν το βάρος είναι μικρότερο έχουμε καταλήξει σε άτοπο. Αν είναι ίσο τότε επαναλαμβάνουμε την ίδια διαδικασία έχοντας τώρα ένα βέλτιστο δέντρο για το οποίο το k είναι μεγαλύτερο. Μετά από έναν αριθμό επαναλήψεων ή καταλήγουμε σε άτοπο, ή τα δέντρα ταυτίζονται. □

Εκτός από τους αλγόριθμους *Prim* και *Kruskal* υπάρχουν και άλλοι αλγόριθμοι για εύρεση συνδετικού δέντρου ελαχίστου κόστους, όπως ο αλγόριθμος του Yao $O(e \log(\log n))$, ο οποίος αρχίζει δέντρα παράλληλα σε κάθε κόμβο και τα συνδέει στις επόμενες φάσεις. Ο αριθμός των εναπομεινάντων δέντρων υποδιπλασιάζεται σε κάθε φάση.

6.5 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 πλευράς $i \rightarrow j$. Αν η πλευρά $i \rightarrow j$ 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 6.4.

Αλγόριθμος 6.4 Εύρεση συντομότερων μονοπατιών (single source shortest paths) με τη μέθοδο του Dijkstra

---

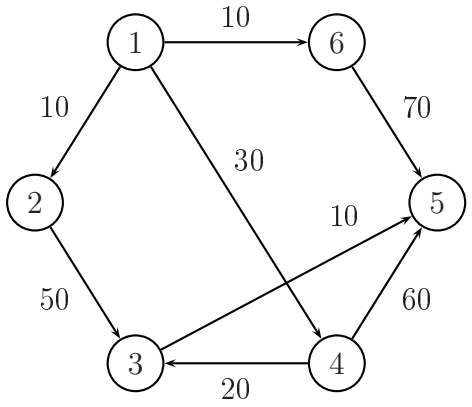
```
procedure Dijkstra;
begin (* Αρχικοποίηση *)
  S := {1};
  for i:=2 to n do begin D[i]:=cost[1,i]; P[i]:=1 end;
  for i:=2 to n−1 do
  begin
    Select w from V − S such that D[w] is minimum;
    S := S + {w};
    for all v in V − S do
    begin
      if D[v] > D[w] + C[w,v] then
      begin
   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

Από τον αλγόριθμο 6.4 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 6.5.1. Έστω ότι μας δίνεται ο γράφος που φαίνεται στο σχήμα 6.5. Η εκτέλεση του αλγορίθμου φαίνεται στον πίνακα 6.1. Για να τυπώσουμε το συντομότερο μονοπάτι από ένα κόμβο σε ένα άλλο ανιχνεύουμε τους προκατόχους του 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 4. Άρα το συντομότερο μονοπάτι από τον κόμβο 1 στον 5 είναι το $1 \to 4 \to 3 \to 5$.



Σχήμα 6.5: Κατευθυνόμενος γράφος με βάρη

			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										

Table 6.1: Εφαρμογή της μεθόδου του Dijkstra για το γράφο G

Κεφάλαιο 7

Δυναμικός Προγραμματισμός (DYNAMIC PROGRAMMING)

7.1 Γενικά

Ο δυναμικός προγραμματισμός είναι μια γενίκευση της άπληστης (*greedy*) μεθόδου. Χρησιμοποιείται για την επίλυση προβλημάτων, των οποίων η λύση μπορεί να θεωρηθεί σαν μια ακολουθία αποφάσεων. Τα προβλήματα είναι συνήθως προβλήματα βελτιστοποίησης, συνεπώς αναζητάμε την ακολουθία των αποφάσεων που τελικά δίνει τη βέλτιστη λύση. Σ' αυτή την αλυσίδα των αποφάσεων οι πληροφορίες που λαμβάνονται υπόψη είναι «τοπικές», περιγράφουν δηλαδή την κατάσταση του προβλήματος όπως αυτή έχει διαμορφωθεί μετά τη λήψη της τελευταίας απόφασης. Σε πολλά προβλήματα οι αποφάσεις που εξαρτώνται μόνο από τοπικές πληροφορίες οδηγούν πάντα σε μια βέλτιστη σειρά αποφάσεων, όπως για παράδειγμα τα προβλήματα που λύνονται με την τεχνική greedy. Σε άλλα προβλήματα όμως αυτό δεν ισχύει. Θα έπρεπε λοιπόν κανείς να δημιουργήσει όλες τις σειρές αποφάσεων, να τις δοκιμάσει και να επιλέξει την καλύτερη. Η τεχνική του δυναμικού προγραμματισμού επεμβαίνει ακριβώς εδώ, αποκλείοντας σειρές αποφάσεων, χαρακτηρίζοντάς τις ως μη βέλτιστες. Έτσι αυτές οι σειρές αποφάσεων δεν δοκιμάζονται.

Η αρχή πάνω στην οποία βασίζεται ο δυναμικός προγραμματισμός είναι η αρχή της βελτιστότητας (*principle of optimality*):

> «Οποιαδήποτε υπακολουθία της βέλτιστης ακολουθίας αποφάσεων είναι επίσης βέλτιστη για το υποπρόβλημα που αντιστοιχεί στη συγκεκριμένη υπακολουθία»

Παράδειγμα 7.1.1. Έστω ότι έχουμε ένα γράφο και το συντομότερο μονοπάτι

από τον κόμβο u στον κόμβο w:

$$u \to w : u, u_1, u_2, \ldots, u_{k-1}, u_k, \ldots, w$$

Τότε είναι σαφές ότι η λύση $u_2, \ldots, u_{k-1}, u_k$ είναι η βέλτιστη για το πρόβλημα που ζητά το συντομότερο μονοπάτι από τον κόμβο u_2 στον u_k $(u_2 \to u_k)$.

Η εφαρμογή της αρχής της βελτιστότητας στα προβλήματα δυναμικού προγραμματισμού οδηγεί σε σχεδιασμό αλγορίθμων που εκφράζονται σαν αναδρομικές σχέσεις. Ο τρόπος που ορίζεται μια αναδρομική σχέση είναι είτε «προς τα εμπρός» (*forward approach*) είτε «προς τα πίσω» (*backward approach*). Αυτό σημαίνει ότι για να εκφράσουμε μια απόφαση x_i από τη βέλτιστη σειρά $x_1, x_2, \ldots, x_n$ είτε θα την εκφράσουμε με την βοήθεια των $x_{i+1}, \ldots, x_n$ («προς τα εμπρός») είτε με τη βοήθεια των $x_1, x_2, \ldots, x_{i-1}$ («προς τα πίσω»).

Παράδειγμα 7.1.2. Στο προηγούμενο παράδειγμα, και χρησιμοποιώντας τον «προς τα εμπρός» ορισμό της αναδρομικής σχέσης έχουμε:

$$MinPathCost_{u \to w} = \min_{\forall k \ adjacent \ to \ u} (Cost(u, k) + MinPathCost_{k \to w})$$

ενώ με τον «προς τα πίσω» ορισμό έχουμε:

$$MinPathCost_{u \to w} = \min_{\forall k \ adjacent \ to \ w} (MinPathCost_{u \to k} + Cost(k, w))$$

7.2 Εύρεση συντομότερων μονοπατιών σε γράφο: αλγόριθμος Bellman-Ford

Μια εφαρμογή της backward approach είναι και ο παρακάτω αλγόριθμος που βρίσκει τα συντομότερα μονοπάτια από έναν κόμβο σε όλους τους άλλους κόμβους του γράφου.

Μπορεί να αποδειχθεί με επαγωγή ότι μετά την εκτέλεση του σταδίου (stage) k, για κάθε κόμβο v (που βρίσκεται σε απόσταση το πολύ k ακμών από τον αρχικό) το $D[v]$ περιέχει το μήκος του συντομότερου μονοπατιού από τον 1 στον v που έχει το πολύ k ακμές και το $P[v]$ περιέχει τον προηγούμενο κόμβο στο μονοπάτι αυτό. Επομένως, μετά από $|V| - 1$ στάδια θα έχει υπολογιστεί το συντομότερο μονοπάτι από τον κόμβο 1 σε κάθε κόμβο του γράφου. Η πολυπλοκότητα του αλγορίθμου είναι $O(|V||E|)$.

Παρατήρηση: σε αντίθεση με τον αλγόριθμο του Dijkstra, ο αλγόριθμος δουλεύει σωστά ακόμη και αν υπάρχουν ακμές αρνητικού βάρους, αρκεί να μην υπάρχουν κύκλοι αρνητικού βάρους. Ακόμη όμως και αν υπάρχουν τέτοιοι κύκλοι μπορεί να τους εντοπίζει με εκτέλεση ενός επιπλέον σταδίου (*Άσκηση: βρείτε τι ακριβώς πρέπει να κάνει ο αλγόριθμος σε αυτό το πρόσθετο στάδιο*).

Αλγόριθμος 7.1 Εύρεση συντομότερων μονοπατιών (single source shortest paths) με τη μεθόδο Bellman-Ford

$D[1] := 0$; *(* ο αρχικός κόμβος *)*
for $i := 2$ **to** $|V|$ **do** $D[i] := \infty$;

for *stage* $:= 1$ **to** $|V| - 1$ **do**
  for all edges $(w, v) \in E$ **do**
    if $D[v] > D[w] + C[w, v]$ **then**
    begin
      $P[v] := w$;
      $D[v] := D[w] + C[w, v]$
    end

7.3 Το πρόβλημα του σακιδίου με τις ακέραιες ποσότητες (Discrete Knapsack Problem)

Έστω μια συλλογή αντικειμένων $U = \{u_1, u_2, \ldots, u_n\}$. Για κάθε αντικείμενο $u \in U$ είναι ορισμένοι δύο θετικοί ακέραιοι, ένα βάρος w και ένα κέρδος p. Δίνεται επίσης και ένας θετικός ακέραιος M. Ζητάμε τη βέλτιστη ακολουθία αποφάσεων $x_1, x_2, \ldots, x_n$ με $x_i \in \{0, 1\}$, που αντιστοιχούν στα $u_1, u_2, \ldots, u_n$ έτσι ώστε:

$$\begin{cases} \sum w_i x_i & \leq & M \\ \sum p_i x_i & = & MAXIMUM \end{cases}$$

Με άλλα λόγια, θέλουμε να μεγιστοποιήσουμε το κέρδος του σακιδίου παίρνοντας ($x_i = 1$) ή όχι ($x_i = 0$) το κάθε αντικείμενο u_i.

Είναι φανερό ότι ισχύει η αρχή της βελτιστότητας γιατί, για παράδειγμα, ένα τελικό τμήμα $x_i, \ldots, x_n$ της βέλτιστης ακολουθίας $x_1, \ldots, x_n$ είναι βέλτιστο για το υποπρόβλημα με τον αντίστοιχο θετικό ακέραιο $M' = M - \sum_{k=1}^{i-1} w_k x_k$.

Χρησιμοποιούμε το συμβολισμό

$$Knap(X, Y) \begin{cases} X : \text{το σύνολο των δεικτών του } u \\ Y : \text{ο εκάστοτε θετικός } M \end{cases}$$

για να συμβολίσουμε τα διάφορα στιγμιότυπα του προβλήματος. (Θα χρησιμοποιούμε τον ίδιο συμβολισμό και για το κέρδος της βέλτιστης λύσης ενός στιγμιοτύπου). Αν η ακολουθία $x_1, x_2, \ldots, x_n$ είναι η βέλτιστη λύση για το $Knap(1..n, M)$ τότε αν πάρουμε οποιοδήποτε τελικό τμήμα αυτής της ακολουθίας, έστω το

$x_i, \ldots, x_n$, το τμήμα αυτό είναι βέλτιστο για το στιγμιότυπο

$$Knap(i..n, M - \sum_{k=1}^{i-1} w_k x_k).$$

Αντίστοιχα, για ένα αρχικό τμήμα της ακολουθίας, έστω το $x_1, \ldots, x_i$ έχουμε ότι είναι βέλτιστο για το

$$Knap(1..i, M - \sum_{k=i+1}^{n} w_k x_k).$$

Η αναδρομική λύση μπορεί να ακολουθήσει τις γνωστές κατευθύνσεις (Forward Approach):

$$Knap(1..n, M) = \max\{0 + Knap(2..n, M), p_1 + Knap(2..n, M - w_1)\}$$

ή (Backward Approach):

$$Knap(1..n, M) = \max\{0 + Knap(1..n-1, M), p_n + Knap(1..n-1, M - w_n)\}$$

Γενικότερα:

$$Knap(1..i, X) = \max\{0 + Knap(1..i-1, X), p_i + Knap(1..i-1, X - w_i)\}$$

όπου

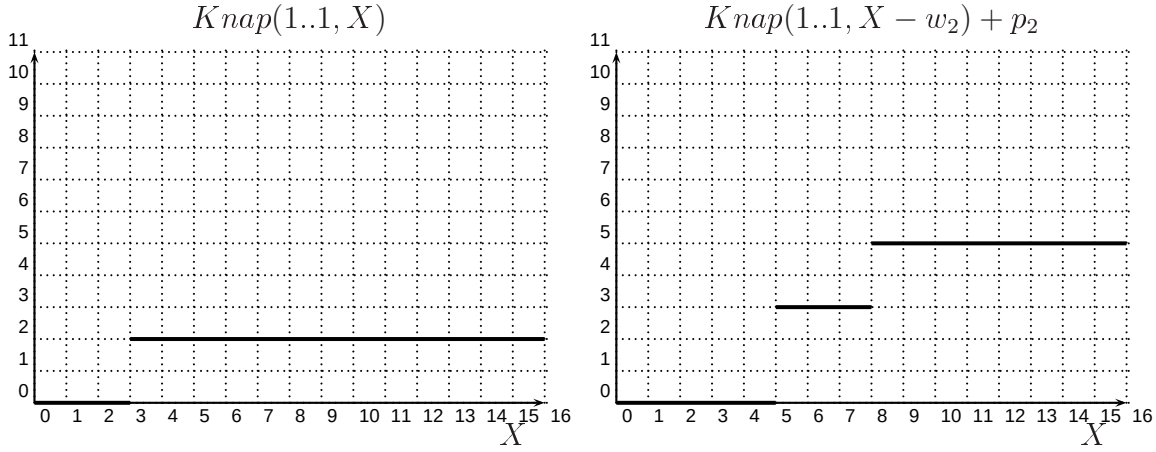
- $0 + Knap(1..i-1, X)$, είναι το μέγιστο κέρδος που έχουμε αν δεν πάρουμε το u_i αντικείμενο και
- $p_i + Knap(1..i-1, X - w_i)$, είναι το μέγιστο κέρδος που έχουμε αν πάρουμε το u_i αντικείμενο.

Παράδειγμα 7.3.1. Χρησιμοποιώντας τους συμβολισμούς που ορίσαμε παραπάνω έστω ότι $U = \{u_1, u_2, u_3\}$,
$\vec{w} = (3, 5, 6)$ και $\vec{p} = (2, 3, 5)$. Να λυθεί το πρόβλημα για M=10.

Σύμφωνα με τη γενική προσέγγιση που αναφέραμε παραπάνω έχουμε:

$$Knap(1..3, X) = \max\{Knap(1..2, X), Knap(1..2, X - w_3) + p_3\}$$
$$Knap(1..2, X) = \max\{Knap(1..1, X), Knap(1..1, X - w_2) + p_2\}$$
$$Knap(1..2, X - w_3) + p_3 =$$
$$\max\{Knap(1..1, X - w_3) + p_3, Knap(1..1, X - w_3 - w_2) + p_3 + p_2\}$$

Αναλύοντας την κάθε περίπτωση ξεχωριστά, έχουμε:



Σχήμα 7.1: Γραφικές παραστάσεις (α) $Knap(1..1, X)$ και (β) $Knap(1..1, X - w_2) + p_2$

Στο σχήμα 7.1(α):

$$Knap(1..1, X) = \begin{cases} \text{αδύνατο}, & X < 0 \\ 0, & 0 \leq X < 3 \\ 2, & 3 \leq X \end{cases}$$

Στο σχήμα 7.1(β):

$$Knap(1..1, X - w_2) + p_2 = \begin{cases} \text{αδύνατο}, & X < 5 \\ 3, & 5 \leq X < 8 \\ 5, & 8 \leq X \end{cases}$$

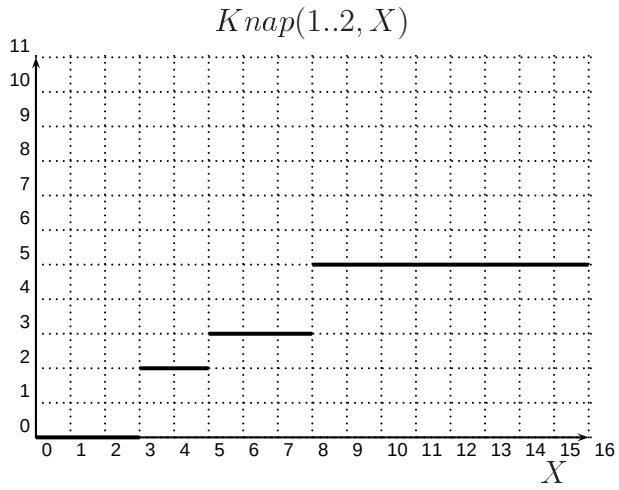
Οι δύο γραφικές παραστάσεις (σχήμα 7.1) μετά την επικάλυψή τους δίνουν τη γραφική παράσταση που φαίνεται στο σχήμα 7.2.

Οι γραφικές παραστάσεις των Σχημάτων 7.2 και 7.3, μετά από την επικάλυψή τους δίνουν τη γραφική παράσταση που φαίνεται στο σχήμα 7.4. Από την παράσταση αυτή προκύπτει ότι για $M = 10$ το μέγιστο κέρδος που μπορεί να έχει το σακίδιο είναι 7 και αυτό γίνεται παίρνοντας τα αντικείμενα u_1 και u_3.

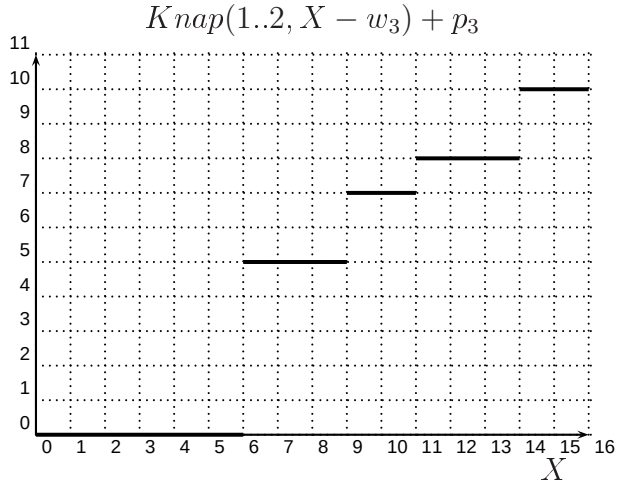
7.3.1 Η μέθοδος του πίνακα

Η αναδρομική σχέση της backward approach

$$Knap(1..i, X) = \max\{0 + Knap(1..i-1, X), p_i + Knap(1..i-1, X - w_i)\}$$

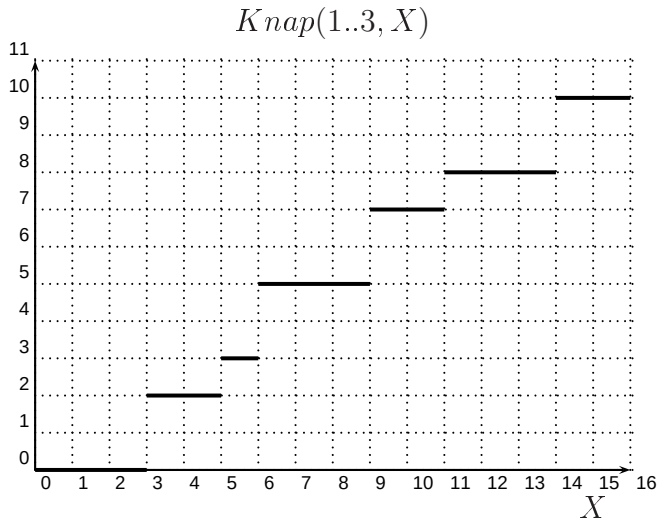


Σχήμα 7.2: Το γράφημα $Knap(1..2, X)$ όπως προκύπτει από την επικάλυψη των $Knap(1..1, X)$ και $Knap(1..1, X - w_2) + p_2$



Σχήμα 7.3: Το γράφημα $Knap(1..2, X - w_3) + p_3$

οδηγεί άμεσα σε έναν bottom-up αλγόριθμο για την επίλυση του Discrete Knapsack. Ο αλγόριθμος αυτός υπολογίζει τις τιμές $Knap(1..i, X)$ για όλα τα i και για όλα τα X αρχίζοντας από τα μικρότερα i και συνεχίζοντας προς τα μεγαλύτερα. Συγκεκριμένα, μπορούμε να φτιάξουμε έναν διδιάστατο πίνακα $K[0..n, 0..M]$ ο οποίος στη θέση $[i, X]$ περιέχει την τιμή $Knap[1..i, X]$, δηλαδή τη μέγιστη συνολική αξία που μπορούμε να πάρουμε χρησιμοποιώντας στοιχεία από το $\{u_1, \ldots, u_i\}$ που έχουν συνολικό βάρος το πολύ X. Η 0-στή γραμμή και η 0-στή στήλη αρχικοποιούνται με 0, δηλαδή $K[0, X] = 0, 0 \leq X \leq M$,



Σχήμα 7.4: Το γράφημα $Knap(1..3, X)$ όπως προκύπτει από την επικάλυψη των $Knap(1..2, X)$ και $Knap(1..2, X - w_3) + p_3$

και $K[i, 0] = 0, 1 \leq i \leq n$. Ο πίνακας γεμίζει γραμμή-γραμμή, ξεκινώντας από $i = 1$. Κάθε γραμμή του πίνακα υπολογίζεται από την προηγούμενη βάσει του παρακάτω τύπου:

$$K[i, X] = \max\{K[i-1, X], p_i + K[i-1, X - w_i]\}$$

όπου ο δεύτερος όρος $p_i + K[i-1, X-w_i]$ λαμβάνεται υπ'όψη μόνο αν $X \geq w_i$.

Ο πίνακας που αντιστοιχεί στο Παράδειγμα 7.3.1 είναι ο Πίνακας 7.1.

i											
(u_3) 3	0	0	0	2	2	3	**5**	5	5	**7**	7
(u_2) 2	0	0	0	2	2	**3**	3	3	**5**	5	5
(u_1) 1	0	0	0	**2**	2	2	2	2	2	2	2
0	**0**	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10
											X

Table 7.1: Πίνακας επίλυσης Discrete Knapsack

Η τιμή της βέλτιστης λύσης είναι η τιμή του πίνακα στη θέση $K[n, M]$. Από τον πίνακα αυτόν μπορούμε να κατασκευάσουμε τη λύση καθεαυτή, δηλαδή να βρούμε το υποσύνολο των στοιχείων που το άθροισμά των αξιών τους ισούται με $K[n, M]$ και το άθροισμα των βαρών τους είναι $\leq M$, εκτελώντας την παρακάτω διαδικασία για το πρόβλημα $Knap[1..n, M]$.

Διαδικασία εύρεσης βέλτιστης λύσης για το $Knap[1..i, X]$ μέσω του πίνακα K: Παρατηρήστε ότι η τιμή της βέλτιστης λύσης για για το υποπρόβλημα $Knap[1..i, X]$ είναι η $K[i, X]$. Για να δούμε αν το αντικείμενο u_i συμμετέχει στη βέλτιστη λύση συγκρίνουμε τις τιμές στις θέσεις $K[i, X]$ και $K[i-1, X]$: αν αν $K[i, X] > K[i-1, X]$ τότε στη βέλτιστη λύση συμμετέχει οπωσδήποτε το αντικείμενο u_i, αλλιώς (αν δηλαδή $K[i, X] = K[i-1, X]$) το u_i δεν συμμετέχει στη βέλτιστη λύση (ή υπάρχει βέλτιστη λύση στην οποία δεν συμμετέχει το u_i). Στην πρώτη περίπτωση η βέλτιστη λύση αποτελείται από το u_i μαζί με τη βέλτιστη λύση του υποπροβλήματος $Knap[1..i-1, X-w_i]$ η οποία βρίσκεται επαναλαμβάνοντας τη διαδικασία από τη θέση $K[i-1, X-w_i]$, κ.ο.κ. Στη δεύτερη περίπτωση μια βέλτιστη λύση είναι η βέλτιστη λύση για το υποπρόβλημα $Knap[1..i-1, X]$. Η διαδικασία σταματάει αν στην τρέχουσα θέση η τιμή είναι 0.

Από τα παραπάνω γίνεται φανερό ότι η διαδικασία υπολογισμού της βέλτιστης λύσης για το $Knap[1..n, M]$ θα σταματήσει σε n το πολύ βήματα.

Ο παραπάνω αλγόριθμος έχει πολυπλοκότητα $O(nM)$, είναι επομένως *ψευδοπολυωνυμικός* (βλ. συζήτηση στο τέλος της ενότητας). Ένας πιο αποδοτικός αλγόριθμος προκύπτει αν παρατηρήσουμε ότι μας ενδιαφέρουν ορισμένες μόνο θέσεις του πίνακα αυτού, συγκεκριμένα οι θέσεις όπου $K[i, X] > K[i, X-1]$. Παρατηρήστε ότι στις θέσεις αυτές το X ισούται με το άθροισμα των βαρών κάποιων από τα αντικείμενα με δείκτη $\leq i$ (δηλαδή το βάρος ενός υποσυνόλου του $\{u_1, \ldots, u_i\}$). Οι τιμές στις υπόλοιπες θέσεις του πίνακα προκύπτουν άμεσα από τις τιμές στις «χαρακτηριστικές» αυτές θέσεις. Στον Πίνακα 7.1 οι τιμές στις (νέες) χαρακτηριστικές θέσεις που προκύπτουν σε κάθε γραμμή σημειώνονται με έντονους (bold) χαρακτήρες. Ένας βελτιωμένος αλγόριθμος που ουσιαστικά υπολογίζει τις τιμές μόνο αυτών των θέσεων περιγράφεται παρακάτω.

7.3.2 Βελτιωμένος αλγόριθμος για το Discrete Knapsack

Παρατηρούμε ότι η τελική βαθμωτή συνάρτηση (*step function*) του σχ. 7.4 μπορεί να οριστεί πλήρως από τα παρακάτω ζεύγη βάρους-κέρδους (w, p) (που αντιστοιχούν στις «χαρακτηριστικές» θέσεις του πίνακα K που αναφέραμε παραπάνω):

$$(0,0),\ (3,2),\ (5,3),\ (6,5),\ (9,7),\ (11,8),\ (14,10)$$

Τα ζεύγη αυτά μπορούμε να τα βρούμε χωρίς να χρησιμοποιήσουμε πίνακα ή αναδρομή, σχηματίζοντας τα σύνολα S_0^i και S_1^i ως εξής:

- $S_0^0 = \{(0,0)\}$
- $S_1^i = \{(w + w_i, p + p_i) \mid (w, p) \in S_0^{i-1}\}, 1 \leq i < n$

- $S_0^i = \text{merge-discard}(S_0^{i-1}, S_1^i), 1 \leq i < n$

όπου

$$\text{merge-discard}(S_0^{i-1}, S_1^i) = \\ S_0^{i-1} \cup S_1^i - \{(w,p) \mid \exists (w',p') \colon w' \leq w \wedge p' \geq p\} - \{(w,p) \mid w > M\}$$

Με άλλα λόγια η συνάρτηση *merge-discard* επιστρέφει την ένωση των δύο συνόλων διαγράφοντας τα επικαλυπτόμενα ζεύγη (*dominated pairs*) και τα ζεύγη με βάρος μεγαλύτερο του M. Τα σύνολα S_0^i και S_1^i για το παράδειγμα φαίνονται στον παρακάτω πίνακα 7.2.

i	S_0^i	S_1^i
0	{(0,0)}	
1		{(3,2)}
1	{(0,0),(3,2)}	
2		{(5,3),(8,5)}
2	{(0,0),(3,2),(5,3),(8,5)}	
3		{(6,5),(9,7),(11,8),(14,10)}
3	{(0,0),(3,2),(5,3),~~(8,5)~~,(6,5),(9,7),~~(11,8)~~,~~(14,10)~~}	

Table 7.2: Παράδειγμα Discrete Knapsack-σύνολα S_0^i και S_1^i

Διαγράφουμε τα ζεύγη που φαίνονται στον πίνακα και το ολικό βάρος που θα έχει το σακίδιο όπως και το μέγιστο κέρδος του εκφράζονται από το τελευταίο ζεύγος του S_0^3. Δηλαδή $W_{total} = 9$ και $P_{\max} = 7$.

Τα αντικείμενα που πρέπει να πάρουμε στο σακίδιο τα βρίσκουμε ως εξής: Επειδή το τελευταίο ζεύγος του $S_0^3, (9,7)$, ανήκει στο S_1^3 σημαίνει πως πρέπει να πάρουμε το αντικείμενο u_3 ($x_3 = 1$). Έχουμε $(9,7) - (6,5) = (3,2)$. Το ζεύγος $(3,2)$ δεν ανήκει στο S_1^2 συνεπώς δεν πρέπει να πάρουμε το αντικείμενο u_2 ($x_2 = 0$). Το ζεύγος $(3,2)$ ανήκει στο S_1^1 άρα πρέπει να πάρουμε το αντικείμενο u_1 ($x_1 = 1$). Έχουμε $(3,2) - (3,2) = (0,0)$ και συνεπώς τελειώσαμε. Η λύση είναι $(x_1, x_2, x_3) = (1,0,1)$.

Η μέθοδος λύσης του γενικού προβλήματος φαίνεται στον αλγόριθμο 7.2 ενώ στον αλγόριθμο 7.3 φαίνεται η συνάρτηση *merge-discard*. Τέλος η διαδικασία *Traceback* που θέτει τιμές στα $x_1, x_2, \ldots, x_n$ φαίνεται στον αλγόριθμο 7.4.

Η πληθικότητα του συνόλου S^n είναι:

$$|S^n| \leq 2 \cdot |S^{n-1}| \leq \cdots \leq 2^n$$

δηλαδή $O(2^n)$. Παρατηρήστε ότι η συνολική πληθικότητα όλων των S_i είναι και αυτή $O(2^n)$. Άρα η χρονική πολυπλοκότητα είναι $O(2^n)$ (με αποδοτική υλοποίηση της merge-discard – ο χρόνος για την Traceback είναι μόνο $O(n^2)$).

Αλγόριθμος 7.2 Επίλυση του προβλήματος σακιδίου με ακέραιες ποσότητες (Discrete Knapsack)

```
procedure Discrete Knapsack (...);
begin
  S_0^0 := {(0,0)};
  for i := 1 to n do
  begin
    S_1^i := {(w + w_i, p + p_i) | (w,p) in S_0^{i-1}};
    S_0^i:=merge−discard(S_0^{i-1}, S_1^i);
  end;
  Traceback; (* to find x_n, x_{n-1}, ..., x_1 *)
end
```

Αλγόριθμος 7.3 Η συνάρτηση merge-discard

```
function Merge−Discard (R, Q: set of items):set of items;
var
  L: set of items;
begin
  L := R + Q;
  for all (w,p) in L do
    if exists (w',p') in L: (w' ≤ w and p' ≥ p) then
      L := L − {(w,p)};
    for all (w,p) in L do if w > M then L := L − {(w,p)};
    Merge−Discard:= L;
end;
```

Αλγόριθμος 7.4 Η διαδικασία Traceback

```
procedure Traceback;
begin
  A:=(the last pair of S_0^n that is the pair with maximum cost);
  for i := n downto 1 do
    if A in S_1^i then begin x_i := 1; A := (w_A − w_i, p_A − p_i) end
    else x_i := 0
end
```

Δηλαδή ο αλγόριθμος που περιγράψαμε δεν είναι αποδοτικός. Όπως θα δούμε

σε επόμενο κεφάλαιο, το Discrete Knapsack ανήκει σε μια κλάση προβλημάτων για τα οποία, μέχρι σήμερα, δεν έχει βρεθεί αποδοτικός αλγόριθμος.

Δηλαδή, θεωρητικά η πολυπλοκότητα του παραπάνω αλγορίθμου είναι εκθετική. Προσέξτε όμως ότι για όλα τα ζεύγη p, w, το w φράσσεται ως εξής: $w \leq M$. Επίσης ισχύει $p \leq \sum p_j$ και $M \leq \sum w_j$ (διαφορετικά μπορούμε να πάρουμε όλα τα αντικείμενα στο σακίδιο). Συνεπώς, μπορεί κανείς να φράξει την συνολική πληθικότητα (και άρα τη χρονική πολυπλοκότητα) των συνόλων S_i ως εξής:

$$O(nM) = O(n \sum w_j) \qquad O(n \sum p_j)$$

Ας σημειωθεί ότι το άνω φράγμα $O(nM)$ προκύπτει και από το γεγονός ότι ο αλγόριθμος 7.2 αποτελεί βελτίωση της μεθόδου του πίνακα όπως αναφέρεται στο τέλος της Ενότητας 7.3.1.

Πρακτικά επομένως, αν π.χ. $M << 2^n$ (πολύ μικρότερο) ή $\sum p_j << 2^n$, τότε έχουμε έναν αποδοτικό αλγόριθμο, ίσως και πολυωνυμικό. Στην πραγματικότητα ο παραπάνω αλγόριθμος είναι ψευδοπολυωνυμικός. Ένας αλγόριθμος λέγεται *ψευδοπολυωνυμικός* όταν είναι πολυωνυμικός (ως προς το μήκος της εισόδου), με την προϋπόθεση οι αριθμοί που εμφανίζονται στην είσοδο να είναι κωδικοποιημένοι σε unary. Δηλαδή όταν είναι πολυωνυμικός ως προς τις **τιμές** των αριθμών που υπάρχουν στην είσοδό του και ως προς το πλήθος και το μέγεθος των δεδομένων. Συνήθως, για τα προβλήματα που υπάρχει ψευδοπολυωνυμικός αλγόριθμος υπάρχει και καλός προσεγγιστικός αλγόριθμος.

7.4 Συντομότερο μονοπάτι ανάμεσα σε κάθε ζευγάρι κορυφών ενός γράφου (All Pairs Shortest Paths Problem)

Έχουμε έναν προσανατολισμένο (κατευθυνόμενο) γράφο $G(V, E)$ στον οποίο κάθε πλευρά $v \rightarrow w$ έχει ένα κόστος $C[v, w]$. Μπορούν να υπάρχουν και αρνητικά κόστη όχι όμως κύκλοι με αρνητικό συνολικό κόστος. Ζητάμε το μικρότερο μήκος από όλα τα μονοπάτια που ενώνουν τον κόμβο v με τον κόμβο w, για κάθε διατεταγμένο ζεύγος κόμβων (v, w).

Μπορούμε να λύσουμε το πρόβλημα αν εφαρμόσουμε n φορές τον αλγόριθμο του *Dijkstra*, με συνολικό κόστος $O(n^3)$. Θα εφαρμόσουμε εδώ δυναμικό προγραμματισμό με τον αλγόριθμο του *Floyd*.

Αριθμούμε τους κόμβους του γράφου σαν $1, 2, 3 \ldots$ και χρησιμοποιούμε έναν πίνακα $A_{n \times n}$ για τον οποίο ισχύει:

$$A[i,j] = \begin{cases} 0 & , i = j \\ C[i,j] & , i \neq j \quad \& \quad (i,j) \in E \\ \infty & , \text{αλλιώς} \end{cases}$$

Έστω $i \to j$ το συντομότερο μονοπάτι από την κορυφή i στην κορυφή j και ας συμβολίσουμε με K το σύνολο των μεταξύ i και j κορυφών, από τις οποίες διέρχεται αυτό το μονοπάτι. Θεωρούμε, χωρίς βλάβη της γενικότητας, ότι στο μονοπάτι αυτό δεν περιέχεται κύκλος[1] και παίρνουμε μια κορυφή k από το σύνολο K. Είναι φανερό ότι το μονοπάτι $i \to k$ $(k \to j)$ είναι το συντομότερο ανάμεσα στις κορυφές $i, k (k, j)$ ειδάλλως ούτε το μονοπάτι από την i στην j (που περνά από την κορυφή k) θα ήταν το συντομότερο. Ισχύει λοιπόν η αρχή της βελτιστότητας άρα μπορούμε να χρησιμοποιήσουμε την τεχνική του δυναμικού προγραμματισμού.

Αλγόριθμος 7.5 Επίλυση του προβλήματος All Pairs Shortest Paths

```
procedure AllShortestPaths (...);
begin
  for i := 1 to n do
    for j := 1 to n do A[i,j] := Cost[i,j];
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        A[i,j] := min(A[i,j], A[i,k] + A[k,j])
end
```

Συμβολίζοντας με $A^k[i,j]$ το κόστος του συντομότερου μονοπατιού από την κορυφή i στην κορυφή j, το οποίο διέρχεται μόνο από κορυφές με δείκτες μικρότερους ή ίσους του k έχουμε:

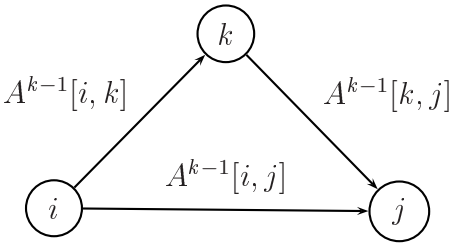
$$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$$

Η παραπάνω σχέση έχει την απλή ερμηνεία που φαίνεται στο σχήμα 7.5. Για να υπολογίσουμε το $A^k[i,j]$ συγκρίνουμε το $A^{k-1}[i,j]$, δηλαδή το κόστος του $i \to j$ που δεν περνά από τον κόμβο με αρίθμηση k (ή μεγαλύτερη), με το $A^{k-1}[i,k] + A^{k-1}[k,j]$, δηλαδή το κόστος του $i \to j$ που περνά από τον κόμβο

[1] Στην περίπτωση που υπάρχει κύκλος τον αφαιρούμε χωρίς να αυξάνουμε το συνολικό μήκος του μονοπατιού, αφού στον γράφο δεν υπάρχουν αρνητικοί κύκλοι.

με αρίθμηση k (αλλά όχι μεγαλύτερη). Το πρόβλημα μας λοιπόν θα έχει λυθεί αν για κάθε i, j υπολογίσουμε το:

$$A[i,j] = \min\{\min_{1\le k\le n}\{A^{k-1}[i,k] + A^{k-1}[k,j]\}, C[i,j]\}$$



Σχήμα 7.5: Υπολογισμός του $A^k[i,j]$ ως το ελάχιστο του κόστους των δύο διαδρομών

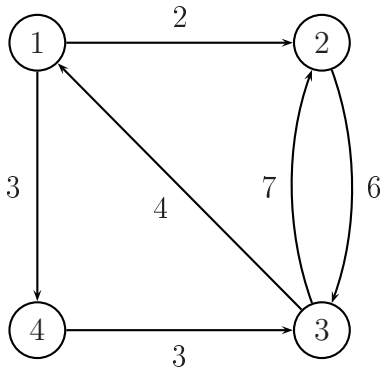
Παρατηρούμε ότι:

$$\begin{aligned} A^0[i,j] &= C[i,j] \\ A^k[i,k] &= A^{k-1}[i,k] \\ A^k[k,j] &= A^{k-1}[k,j] \end{aligned}$$

Τελικά το συντομότερο μονοπάτι από την κορυφή i στην j θα είναι εκείνο που επιτρέπεται να περνά από όλους τους n κόμβους, δηλαδή το $A^n[i,j]$.

Συνεπώς όλοι οι υπολογισμοί μπορούν να γίνουν μέσα στον ίδιο πίνακα $A_{n\times n}$. Η υλοποίηση της μεθόδου φαίνεται στον αλγόριθμο 7.5. Η πολυπλοκότητα του αλγορίθμου είναι της τάξης $O(n^3)$.

Παράδειγμα 7.4.1. Δίνεται ο προσανατολισμένος γράφος που φαίνεται στο σχήμα 7.6. Ζητάμε το συντομότερο μονοπάτι ανάμεσα σε κάθε ζεύγος κορυφών του γράφου.



Σχήμα 7.6: Αναζήτηση των συντομότερων μονοπατιών στο γράφο

Αρχικά σχηματίζουμε τον πίνακα[2]

$$A^0 = C = \begin{bmatrix} 0 & 2 & \infty & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 7 & 0 & \infty \\ \infty & \infty & 3 & 0 \end{bmatrix}$$

και στη συνέχεια προχωράμε σε βήματα, σχηματίζοντας κάθε φορά τον πίνακα $A^k (A^0 = C)$:

$$A^1 = \begin{bmatrix} 0 & 2 & \infty & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 6 & 0 & 7 \\ \infty & \infty & 3 & 0 \end{bmatrix}, A^2 = \begin{bmatrix} 0 & 2 & 8 & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 6 & 0 & 7 \\ \infty & \infty & 3 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 2 & 8 & 3 \\ 10 & 0 & 6 & 13 \\ 4 & 6 & 0 & 7 \\ 7 & 9 & 3 & 0 \end{bmatrix}, A^4 = \begin{bmatrix} 0 & 2 & 6 & 3 \\ 10 & 0 & 6 & 13 \\ 4 & 6 & 0 & 7 \\ 7 & 9 & 3 & 0 \end{bmatrix}$$

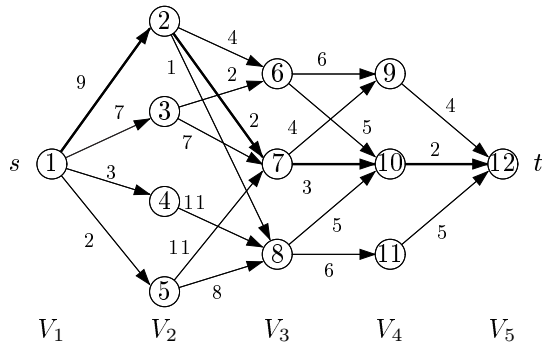
Παρατηρούμε ότι στον πίνακα A^1 έχει αλλάξει το στοιχείο στη θέση $[3, 2]$, όπως και στη θέση $[3, 4]$. Αυτό συμβαίνει γιατί ο πίνακας A^1 υπολογίζει τα συντομότερα ανάμεσα σε όλα τα ζεύγη κορυφών του γράφου που όμως περνούν το πολύ από την κορυφή 1 (και όχι κάποια άλλη κορυφή με μεγαλύτερη αρίθμηση). Αρχικά λοιπόν το συντομότερο μονοπάτι ανάμεσα στις κορυφές 3 και 2 ήταν η πλευρά που απ' ευθείας τις ένωνε, δηλαδή η $3 \to 2$ η οποία είχε κόστος 7. Παρόμοια, δεν υπήρχε μονοπάτι που να ένωνε την κορυφή 3 με την κορυφή 4 (δηλαδή η πλευρά $3 \to 4$). Κατά το σχηματισμό του πίνακα A^1 διαπιστώνουμε ότι το μονοπάτι $3 \to 1 \to 2$ έχει μικρότερο συνολικά κόστος από την πλευρά $3 \to 2$ και ότι υπάρχει μονοπάτι που ενώνει τις 3 και 4, το $3 \to 1 \to 4$. Ενημερώνουμε λοιπόν τον πίνακα με τα κόστη αυτών των μονοπατιών. Ανάλογα σχηματίζονται και οι υπόλοιπες πίνακες.

7.5 Συντομότερο μονοπάτι σε multistage γράφο

Έστω $G(V, E)$ ένας προσανατολισμένος γράφος του οποίου το σύνολο των κορυφών διαμερίζεται σε $k > 1$ υποσύνολα $V_1, V_2, \ldots, V_k$ που είναι ξένα μεταξύ τους. Για τους υπογράφους

$$G(V_1, E_1),\ G(V_2, E_2),\ \ldots, G(V_k, E_k)$$

[2]Μπορούμε να αντικαταστήσουμε το ∞ του πίνακα C με κάποια τιμή μεγαλύτερη του $(n-1) \cdot M$, όπου M είναι το $\max\{C[i, j] \mid (i, j) \in E\}$ αφού κανένα μονοπάτι δεν μπορεί να ξεπερνά την τιμή αυτή.



Σχήμα 7.7: Αναζήτηση συντομότερου μονοπατιού στο multistage γράφο

ισχύει ότι $E_1 = E_2 = \ldots = E_k = \emptyset$ και αν $(u, v) \in E$ και $u \in V_i$ τότε $v \in V_{i+1}$ για $1 \leq i \leq k-1$. Ακόμη $|V_1| = |V_k| = 1$. Την μοναδική κορυφή του συνόλου V_1 την ονομάζουμε αφετηρία και τη συμβολίζουμε με s, ενώ την μοναδική κορυφή του συνόλου V_k την ονομάζουμε τέρμα και την συμβολίζουμε με t. Ο γράφος $G(V, E)$ ονομάζεται *multistage* γράφος (π.χ. σχήμα 7.7). Ζητάμε το συντομότερο μονοπάτι που ενώνει την αφετηρία με το τέρμα.

Αριθμούμε τις κορυφές του γράφου, αντιστοιχώντας στην κορυφή s τον αριθμό 1 και στη συνέχεια στις κορυφές του V_2 τους αριθμούς $2, 3, \ldots$ μέχρι να εξαντληθούν. Συνεχίζουμε με το σύνολο $V_3, \ldots$και καταλήγουμε στην κορυφή t στην οποία αντιστοιχούμε τον αριθμό n. Με την αρίθμηση αυτή κάθε κορυφή αποκτά και ένα δείκτη. Όλοι οι δείκτες του V_{i+1} είναι μεγαλύτεροι από όλους τους δείκτες του V_i για κάθε $1 \leq i \leq n$.

Για να δημιουργηθεί το συντομότερο μονοπάτι από την κορυφή s στην t πρέπει να παρθούν $k-2$ αποφάσεις. Κάθε απόφαση, π.χ. η i , συνεπάγεται επιλογή μιας από τις κορυφές του V_i για να συμπεριληφθεί σε αυτό το ελαχίστου κόστους μονοπάτι. Ο αλγόριθμος 7.6 που βρίσκει το μονοπάτι ελαχίστου κόστους από την κορυφή 1 στην n βασίζεται στην παρακάτω αναδρομική σχέση:

$$OptCost[1, n] = \min_{r \in V_2} \{cost[1, r] + OptCost[r, n]\}$$

όπου $OptCost[1, n]$ σημαίνει μονοπάτι ελαχίστου κόστους από την κορυφή 1 στην n και $cost[1, r]$ σημαίνει το κόστος της ακμής $1 \rightarrow r$.

Η διαδικασία της επιλογής της κορυφής r έχει κόστος $O(out\_deg(j))$, η δε συνολική διαδικασία έχει κόστος $O(n + e)$, όπου n οι κορυφές του γράφου και οι e οι ακμές του. Ένα από τα συντομότερα μονοπάτια του γράφου στο σχήμα 7.7 είναι το $s \rightarrow 2 \rightarrow 7 \rightarrow 10 \rightarrow t$ με μήκος 16 (ένα άλλο είναι το $s \rightarrow 3 \rightarrow 6 \rightarrow 10 \rightarrow t$ όπως εύκολα μπορεί να διαπιστώσει κανείς).

Αλγόριθμος 7.6 Εύρεση συντομότερου μονοπατιού σε multistage γράφο

```
procedure ShortestPathInMultistageGraph (...)
begin
  DistFromN[n]:=0;
  for j:=n−1 downto 1 do
  begin
    select r with min{cost[j,r]+DistFromN[r]};
    DistFromN[j]:=cost[j,r]+DistFromN[r]; next[j]:=r;
  end
end
```

7.6 Το πρόβλημα του πλανόδιου πωλητή (The Traveling Salesman Problem)

Στο πρόβλημα του πλανόδιου πωλητή μας δίνεται ένας (συνήθως πλήρης) γράφος με βάρη και το ζητούμενο είναι ένας κύκλος ελαχίστου κόστους (minimum cost tour) που να περνά μία ακριβώς φορά από κάθε κόμβο και να καταλήγει στον κόμβο από τον οποίο ξεκίνησε. Ο γράφος μπορεί να είναι προσανατολισμένος ή όχι.

Μπορούμε εύκολα να δούμε ότι ισχύει η αρχή της βελτιστότητας, αφού αν διαλέξουμε, χωρίς βλάβη της γενικότητας, σαν αρχή τον κόμβο 1:

$$MinCostTour_{1\to 1} = \min_{k\neq 1}\{cost[1,k] + MinCostTour_{k\to 1}\}$$

Ορίζουμε $g(i,S)$ να είναι το κόστος του συντομότερου μονοπατιού από τον κόμβο i στον κόμβο 1, που περνά από όλους τους κόμβους του S. Τότε

$$g(1, V-\{1\}) = \min_{2\leq k\leq n}\{cost[1,k] + g(k, V-\{1,k\})\}$$

και γενικά ισχύει

$$g(i,S) = \min_{j\in S}\{cost[i,j] + g(j, S-\{j\})\}$$

Επίσης $g(i,\emptyset) = cost[i,1]$.

Παράδειγμα 7.6.1. Έστω κατευθυνόμενος γράφος με τον παρακάτω πίνακα κόστους:

$$C = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Είναι φανερό ότι αν υπολογίσουμε το $g(1,\{2,3,4\})$, το πρόβλημά μας θα έχει λυθεί. Αρχίζουμε λοιπόν να υπολογίζουμε τα διάφορα g για $|V| = 0$, $|V| = 1$, $|V| = 2$ και$|V| = 3$.

Στην περίπτωση που $|V| = 0$ έχουμε :

$$\begin{array}{lcl} g(2,\emptyset) & = & Cost[2,1] = 5 \\ g(3,\emptyset) & = & Cost[3,1] = 6 \\ g(4,\emptyset) & = & Cost[4,1] = 8 \end{array}$$

Στην περίπτωση που $|V| = 1$ έχουμε:

$$\begin{array}{lcl} g(2,\{3\}) & = & Cost[2,3] + g(3,\emptyset) = 9+6 = 15 \\ g(2,\{4\}) & = & Cost[2,4] + g(4,\emptyset) = 10+8 = 18 \\ g(3,\{2\}) & = & Cost[3,2] + g(2,\emptyset) = 13+5 = 18 \\ g(3,\{4\}) & = & Cost[3,4] + g(4,\emptyset) = 12+8 = 20 \\ g(4,\{2\}) & = & Cost[4,2] + g(2,\emptyset) = 8+5 = 13 \\ g(4,\{3\}) & = & Cost[4,3] + g(3,\emptyset) = 9+6 = 15 \end{array}$$

Στην περίπτωση που $|V| = 2$ έχουμε:

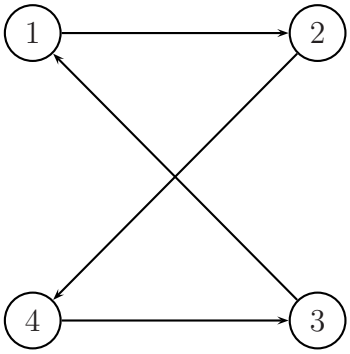
$$\begin{array}{rcl} g(2,\{3,4^{\swarrow}\}) & = & \min(Cost[2,3] + g(3,\{4\}), Cost[2,4] + g(4,\{3\})) \\ & = & \min(9+20, 10+15) = 25 \\ g(3,\{2,4^{\swarrow}\}) & = & \min(Cost[3,2] + g(2,\{4\}), Cost[3,4] + g(4,\{2\})) \\ & = & \min(13+18, 12+13) = 25 \\ g(4,\{2^{\swarrow},3\}) & = & \min(Cost[4,2] + g(2,\{3\}), Cost[4,3] + g(3,\{2\})) \\ & = & \min(8+15, 9+18) = 23 \end{array}$$

Τελικά για $|V| = 3$ έχουμε:

$$\begin{array}{rcl} g(1,\{2^{\swarrow},3,4\}) & = & \min(Cost[1,2] + g(2,\{3,4\}), Cost[1,3] + g(3,\{2,4\}), \\ & & Cost[1,4] + g(4,\{2,3\})) \\ & = & \min(10+25, 15+25, 20+23) = 35 \end{array}$$

Τα βέλη «$\swarrow$» δηλώνουν τον κόμβο που κάθε φορά διαλέξαμε. Έτσι λοιπόν, στο συγκεκριμένο γράφο, αρχίζοντας από το 1 ο δείκτης μας οδηγεί στο 2. Από το 2 πηγαίνουμε στο 4 και από εκεί αναγκαστικά οδηγούμαστε στο 3 αφού αυτό είναι το μόνο σημείο που δεν έχουμε επισκεφθεί και τελικά ο κύκλος κλείνει στο 1. Τελικά ο ζητούμενος κύκλος είναι ο $1 \to 2 \to 4 \to 3$ (σχήμα 7.8) και φυσικά δεν παίζει ρόλο ο κόμβος από τον οποίο θα ξεκινήσουμε.

Σε κάθε στάδιο k υπολογίζουμε τα $g(i,S)$ για $n-1$ διαφορετικά i. Όλα τα $n-2$ στοιχεία που απομένουν (εκτός του $(1,j)$) συνδυάζονται ανά $|S| = k$,



Σχήμα 7.8: Κύκλος ελαχίστου κόστους

δηλαδή υπάρχουν $\binom{n-2}{k}$ δυνατοί συνδυασμοί (διαφορετικές τιμές της g). Ο χώρος λοιπόν που χρειαζόμαστε είναι:

$$Space = (n-1)\sum_{k=0}^{n-2}\binom{n-2}{k} = (n-1)(1+1)^{n-2} = (n-1)\cdot 2^{n-2}$$

Άρα η πολυπλοκότητα χώρου είναι της τάξης $O(n2^n)$. Παρατηρούμε ότι αυτός ο αλγόριθμος λύσης του προβλήματος δεν είναι αποδοτικός. Όπως θα δούμε σε επόμενο κεφάλαιο, το πρόβλημα TSP (όπως και το πρόβλημα του διακριτού σακιδίου) ανήκει επίσης σε μια κλάση προβλημάτων για τα οποία έως σήμερα δεν έχει βρεθεί αποδοτικός αλγόριθμος.

Κεφάλαιο 8

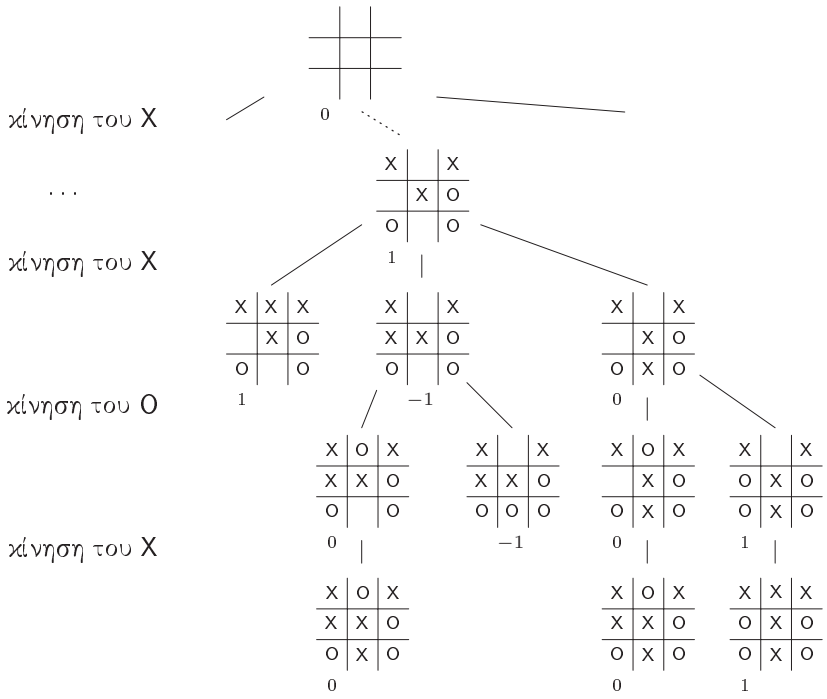
Η τεχνική της οπισθοδρόμησης (BACKTRACKING)

8.1 Γενικά

Είναι γεγονός ότι αρκετά συχνά βρισκόμαστε αντιμέτωποι με προβλήματα, για τη λύση των οποίων δεν μπορούμε να εφαρμόσουμε κάποια γνωστή τεχνική. Η μόνη διέξοδος τότε, είναι η εξαντλητική αναζήτηση (*exhaustive search*) όλων των μερικών λύσεων που φαίνεται να οδηγούν στην τελική λύση. Σ' αυτή την παράγραφο θα αναφερθούμε σε μια τεχνική, που συστηματοποιώντας την εξαντλητική αναζήτηση μας βοηθάει στην επίλυση τέτοιου τύπου προβλημάτων. Σε γενικές γραμμές, καθώς από τις μερικές λύσεις, οι οποίες διαμορφώνονται σε βήματα, προσπαθούμε να φτάσουμε στη τελική λύση, είναι πιθανό σε κάποιο σημείο να διαπιστώσουμε ότι οι επιλογές που κάναμε δεν πρόκειται να μας οδηγήσουν πλέον σε αυτή. Τότε σταματάμε και «οπισθοχωρούμε» σε κάποιο προηγούμενο βήμα, απ' όπου φαίνεται ότι αξίζει να συνεχίσουμε προς άλλη κατεύθυνση την προσπάθεια ανεύρεσης της τελικής λύσης. Συχνά δε, μπορούμε να εφαρμόσουμε και τεχνικές που μειώνουν κατά πολύ το κόστος της εξαντλητικής αναζήτησης (*A-B prunning, branch and bound*).

8.2 Δέντρα παιχνιδιών (Game Trees)

Ας φανταστούμε δύο παίκτες που παίζουν το γνωστό παιχνίδι της τρίλιζας (*tic-tac-toe*). Το ταμπλό του παιχνιδιού είναι ένας πίνακας 3×3 και οι δύο παίχτες παίζουν διαδοχικά ο ένας ένα X, σε κάποια θέση του πίνακα, και ο άλλος ένα O. Είναι φανερό ότι υπάρχει ένας πεπερασμένος αριθμός από διαφορετικές καταστάσεις του πίνακα και επίσης είναι σίγουρο ότι μετά από έναν ορισμένο αριθμό κινήσεων το παιχνίδι θα τελειώσει. Μπορούμε



Σχήμα 8.1: Τμήμα από πιθανό παιχνίδι τρίλιζας

να αντιστοιχίσουμε στο παιχνίδι αυτό ένα δέντρο, στο οποίο κάθε κόμβος αναπαριστά κάποια κατάσταση του πίνακα. Η ρίζα του δέντρου αντιστοιχεί στην αρχική κατάσταση του παιχνιδιού (τον κενό πίνακα) και γενικά, αν η κατάσταση x του παιχνιδιού αντιστοιχεί στον κόμβο n του δέντρου, τότε όλα τα παιδιά του n αντιστοιχούν σε όλες τις δυνατές και επιτρεπόμενες καταστάσεις στις οποίες μπορεί να περιέλθει το παιχνίδι, μετά την κατάσταση x.

Στο σχήμα 8.1 φαίνεται ένα μέρος από ένα πιθανό παιχνίδι τρίλιζας. Τα φύλλα του δέντρου αντιστοιχούν σε καταστάσεις του πίνακα όπου η επόμενη κίνηση είναι αδύνατη, είτε γιατί κάποιος παίκτης νίκησε, είτε γιατί υπάρχει ισοπαλία. Οι τιμές -1, 0, ή 1 που φαίνονται στα φύλλα, αντιστοιχούν σε ήττα, ισοπαλία ή νίκη του πρώτου παίχτη (που σημαίνει X).Οι τιμές αυτές μεταδίδονται προς τα πάνω στο δέντρο, με τέτοιο τρόπο, ώστε αν ένας κόμβος αντιστοιχεί σε κατάσταση του πίνακα όπου είναι η σειρά του πρώτου παίχτη να κινηθεί, ο κόμβος αυτός να παίρνει τη μέγιστη τιμή από εκείνες που έχουν τα παιδιά του. Αν ο κόμβος αντιστοιχεί σε κατάσταση του πίνακα όπου είναι η σειρά του δεύτερου παίχτη να κινηθεί, τότε η τιμή που παίρνει αυτός ο κόμβος είναι η ελάχιστη από εκείνες που έχουν τα παιδιά του. Υποθέτουμε δηλαδή ότι ο πρώτος παίχτης διαλέγει την καλύτερη γι' αυτόν κίνηση ενώ ο δεύτερος

διαλέγει την καλύτερη κίνηση που θα καταλήξει, αν είναι δυνατό, σε ήττα του αντιπάλου, με την ισοπαλία σαν δεύτερη επιλογή.

Η απονομή τιμών στους κόμβους συνεχίζεται μέχρι να φτάσουμε στη ρίζα. Αν τελικά η ρίζα πάρει την τιμή 1 τότε ο πρώτος παίχτης έχει μια στρατηγική που τον οδηγεί σίγουρα σε νίκη. Ακολουθώντας αυτή την στρατηγική, όταν είναι η σειρά του δεύτερου παίχτη, δεν υπάρχει γι' αυτόν άλλη επιλογή παρά να κινηθεί έτσι ώστε τελικά να χάσει. Το γεγονός ότι το παιχνίδι είναι πεπερασμένο οδηγεί τον πρώτο παίχτη στην τελική νίκη. Αντίστοιχα αν η ρίζα πάρει την τιμή -1, τότε ο δεύτερος παίχτης έχει την νικηφόρα στρατηγική. Πάντοτε όμως, στο παιχνίδι της τρίλιζας, η ρίζα παίρνει την τιμή 0, πράγμα που σημαίνει ότι κανένας παίχτης δεν έχει σίγουρη στρατηγική, παρά μόνο μπορεί να οδηγήσει το παιχνίδι σε ισοπαλία, κάνοντας τις καλύτερες για τον εαυτό του κινήσεις.

Η ιδέα του δέντρου παιχνιδιών με κόμβους που έχουν τιμές -1, 0, και 1 μπορεί να γενικευθεί και σε δέντρα που στα φύλλα δίνεται οποιαδήποτε τιμή και το υπόλοιπο δέντρο υπολογίζεται με τον ίδιο κανόνα που περιγράψαμε προηγούμενα. Η γενίκευση αυτή είναι χρήσιμη για παιχνίδια πιο περίπλοκα από την τρίλιζα, όπως για παράδειγμα το σκάκι. Το δέντρο του σκακιού, αν και πεπερασμένο, είναι τόσο τεράστιο ώστε η αποτίμησή του από τα φύλλα προς τη ρίζα να είναι αδύνατη. Τα σκακιστικά προγράμματα, κάθε φορά που χρειάζεται να κάνουν μια κίνηση, φτιάχνουν το δέντρο του παιχνιδιού με ρίζα την τρέχουσα κατάσταση της σκακιέρας. Στη συνέχεια επεκτείνουν αυτό το δέντρο μερικά επίπεδα, ο αριθμός των οποίων εξαρτάται από τις δυνατότητες του υπολογιστή που δουλεύουν. Είναι φανερό ότι για τα περισσότερα φύλλα του δέντρου δεν θα ισχύει ότι αντιστοιχούν σε νίκη, ήττα ή ισοπαλία. Γι' αυτό το λόγο το πρόγραμμα χρησιμοποιεί μια συνάρτηση καταστάσεων που προσπαθεί να υπολογίσει την πιθανότητα να κερδίσει ο υπολογιστής. Για παράδειγμα, η υλική υπεροχή ενός παίκτη παίζει σημαντικό ρόλο στον υπολογισμό αυτής της πιθανότητας, όπως και άλλοι παράγοντες σαν την οργανωμένη δύναμη άμυνας γύρω από τους δύο βασιλιάδες. Χρησιμοποιώντας αυτή τη συνάρτηση, ο υπολογιστής μπορεί να προσεγγίσει την πιθανότητα της νίκης, μετά από τις επόμενες δυνατές του κινήσεις, με την παραδοχή ότι και οι δύο πλευρές κάνουν τις καλύτερες δυνατές κινήσεις.

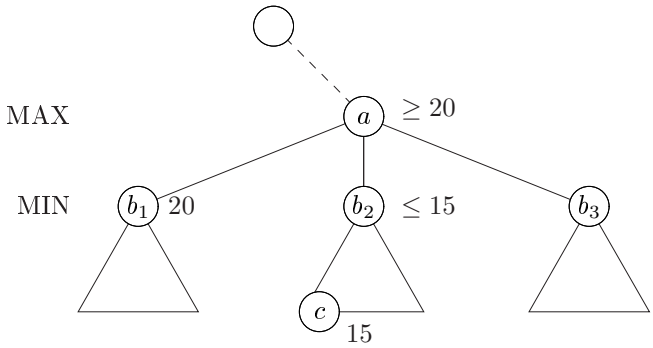
8.3 Υλοποίηση της οπισθοδρόμησης

Ας υποθέσουμε ότι μας δίνεται ένα παιχνίδι με τους κανόνες του, όπως οι επιτρεπτές κινήσεις και οι όροι τερματισμού του. Το πρόβλημά μας είναι να κατασκευάσουμε το δέντρο του και να αποτιμήσουμε τη ρίζα του. Η κατασκευή του δέντρου θα γίνει με τον προφανή τρόπο και μετά η διάσχισή

του θα γίνει μεταδιατεταγμένα (postorder). Η μεταδιατεταγμένη διάσχιση μας εξασφαλίζει ότι επισκεπτόμαστε έναν κόμβο n μετά από όλα τα παιδιά του. Έτσι λοιπόν μπορούμε να αποτιμήσουμε τον εκάστοτε κόμβο λαμβάνοντας κάθε φορά το μέγιστο ή το ελάχιστο των τιμών των κόμβων των παιδιών του. Ο χώρος στον οποίο θα χρειαστεί να αποθηκεύσουμε το δέντρο μπορεί να καταλήξει να είναι απαγορευτικά μεγάλος, όμως με προσεκτικούς χειρισμούς μπορούμε να εξετάζουμε μόνο ένα μονοπάτι από τη ρίζα προς κάποιο κόμβο.

Σε αυτό το σημείο πρέπει να επισημάνουμε ότι δεν είναι μόνο στρατηγικές για παιχνίδια που μπορούν να βρεθούν με τις τεχνικές που περιγράφουμε. Ένα παιχνίδι, μπορεί π.χ. να αναπαριστά μια λύση ενός πρακτικά χρήσιμου προβλήματος.

8.4 Alpha - Beta prunning



Σχήμα 8.2: Alpha - Beta prunning

Κατά τη διάρκεια υπολογισμού ενός δέντρου μπορούμε συχνά να κάνουμε μια απλή παρατήρηση που θα μας επιτρέψει να αγνοήσουμε, ένα μεγάλο μέρος του. Στο σχήμα 8.2 έστω ότι έχουμε ήδη αποτιμήσει τον κόμβο b_1 με την τιμή 20. Αφού ο κόμβος a αποτιμάται με το μέγιστο των τιμών των παιδιών του, είναι φανερό ότι η τιμή του θα είναι τουλάχιστον το 20. Αν υποθέσουμε ότι συνεχίζοντας βρίσκαμε ότι ο κόμβος c έχει την τιμή 15, τότε αφού ο c είναι απόγονος του b_2 και ο b_2 αποτιμάται με το ελάχιστο των τιμών των παιδιών του, καταλήγουμε στο ότι, η τιμή του b_2 δεν μπορεί να ξεπερνά το 15. Οποιαδήποτε τιμή λοιπόν έχει ο b_2 δεν μπορεί να επηρεάσει την τιμή του a, ούτε οποιουδήποτε προγόνου του.

Στην παραπάνω περίπτωση μπορούμε να αγνοήσουμε τόσο τον κόμβο b_2, όσο και όλους τους απογόνους του που δεν έχουν ακόμη εξετασθεί. Οι γενικοί κανόνες που μας βοηθούν να «κλαδεύουμε» (*prune*) κόμβους εξετάζουν

τις τιμές των φύλλων του δέντρου όπως επίσης και τα πάνω και κάτω όρια (ανάλογα με το πώς αποτιμάται κάποιος κόμβος) των τιμών των εσωτερικών κόμβων του δέντρου.

8.5 Branch and Bound

Χρησιμοποιώντας μια ιδέα παρόμοια με αυτή του α-β-prunning μπορούμε να ελαττώσουμε ακόμη περισσότερο τους κόμβους του δέντρου που πρέπει να εξετάσουμε. Ας υποθέσουμε ότι έχουμε μια μέθοδο που μας επιτρέπει να πάρουμε ένα κάτω όριο του κόστους οποιασδήποτε μερικής λύσης από αυτές που αναπαριστά κάποιος κόμβος n. Αν η καλύτερη μερική λύση που έχουμε μέχρι στιγμής κοστίζει λιγότερο από το κάτω όριο του κόμβου, τότε μπορούμε να παραλείψουμε από την εξέτασή μας όλους τους κόμβους κάτω από το n.

Έστω λοιπόν ότι εξετάζουμε τα παιδιά ενός κόμβου και βρίσκουμε ότι το κάτω όριο ενός παιδιού είναι ίσο ή μεγαλύτερο από την καλύτερη μερική λύση που έχουμε βρει μέχρι εκείνη τη στιγμή. Μπορούμε τότε να κλαδέψουμε αυτό το παιδί και να μην μας απασχολήσουν οι απόγονοί του. Ενδιαφέρουσες είναι οι περιπτώσεις όπου το κάτω όριο ενός κόμβου είναι μικρότερο από την καλύτερη μερική λύση που υπάρχει μέχρι στιγμής, και συνάμα μπορούμε να κλαδέψουμε όλα τα παιδιά του γιατί τα κάτω όριά τους είναι μεγαλύτερα από την καλύτερη μερική λύση. Αν δεν μπορούμε να κλαδέψουμε κανένα παιδί, τότε επιλέγουμε πρώτα αυτό με το μικρότερο κάτω όριο, με την ελπίδα να φτάσουμε γρηγορότερα σε λύση με φθηνότερο κόστος από το καλύτερο που έχουμε. Μετά από την εξέταση ενός παιδιού, πρέπει να επανεξετασθεί το αν τα αδέλφια του μπορούν να κλαδευτούν, μιας και μπορεί να έχει βρεθεί μια νέα καλύτερη μερική λύση.

Κεφάλαιο 9

Αναζήτηση και διάσχιση σε δένδρα και γράφους

9.1 Γενικά

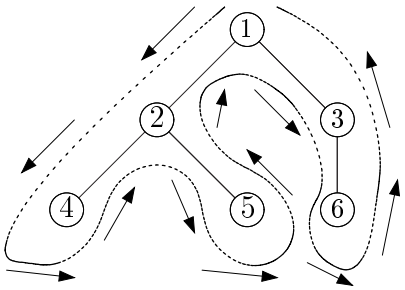
Ένα πρόβλημα που προκύπτει αρκετά συχνά, είναι να διασχίσουμε (*traverse*) μια δομή δεδομένων έτσι ώστε να επισκεφθούμε όλα τα στοιχεία της με κάποιο συστηματικό τρόπο.

Αυτό που ζητάμε είναι να διασχίσουμε τη δομή δεδομένων με έναν τρόπο που βασίζεται στη δομή της. Ειδικότερα για γράφους ως διάσχιση εννοούμε την επίσκεψη όλων των κόμβων με μετακίνηση από κόμβο σε κόμβο μέσω ακμών του γράφου. Αυτό μπορούμε να το πετύχουμε με μια ποικιλία μεθόδων από τις οποίες θα ξεχωρίσουμε τις σπουδαιότερες που αφορούν τα δέντρα και τους γράφους.

9.2 Διάσχιση δέντρων

Οι σπουδαιότεροι τρόποι διάσχισης ενός δέντρου είναι η προδιατεταγμένη (*preorder*), η ενδοδιατεταγμένη (*inorder*) και η μεταδιατεταγμένη (*postorder*). Οι διασχίσεις αυτές παράγουν τις διατάξεις (*orderings*) των κόμβων με τα αντίστοιχα ονόματα.

Έστω μια διαδρομή γύρω από το δέντρο που ξεκινάει από τη ρίζα και προχωρά με φορά αντίθετη από αυτή των δεικτών του ρολογιού μένοντας όσο το δυνατόν πλησιέστερα στο δέντρο, όπως αυτή στο σχήμα 9.1. Η προδιατεταγμένη διάταξη παράγεται όταν κατά τη διάρκεια της διαδρομής καταγράψουμε κάθε κόμβο την πρώτη φορά που τον συναντάμε, ενώ η μεταδιατεταγμένη όταν τον καταγράψουμε την τελευταία φορά. Η ενδοδιατεταγμένη διάταξη προκύπτει όταν καταγράψουμε τα φύλλα τη πρώτη φορά που τα συναντάμε, αλλά τους



Σχήμα 9.1: Διάσχιση δένδρου

άλλους κόμβους τη δεύτερη φορά. Για το δέντρο του σχήματος 9.1 οι τρεις διατάξεις των κόμβων είναι οι ακόλουθες:

- προδιατεταγμένη: 1 2 4 5 3 6
- μεταδιατεταγμένη: 4 5 2 6 3 1
- ενδοδιατεταγμένη: 4 2 5 1 6 3

Ο αλγόριθμος 9.1 είναι η υλοποίηση της ενδοδιατεταγμένης διάσχισης σε ένα δυαδικό δέντρο. Στην υλοποίηση αυτή υποθέτουμε ότι έχουν γίνει οι κατάλληλες δηλώσεις για την μεταβλητή T, ενώ η πολυπλοκότητα χώρου και χρόνου είναι φανερό ότι είναι $O(n)$. Μπορούμε να εξαλείψουμε τη δεύτερη αναδρομή με την υλοποίηση που φαίνεται στον αλγόριθμο 9.2. Μια μη αναδρομική υλοποίηση χρησιμοποιεί μια στοίβα (*stack*) και υποθέτει ότι όλο το δέντρο είναι το δεξί παιδί μιας εικονικής (*dummy*) μεταβλητής T. Η υλοποίηση αυτή φαίνεται στον αλγόριθμο 9.3.

Οι αναδρομικές υλοποιήσεις για την προδιατεταγμένη και μεταδιατεταγμένη διάσχιση ενός δυαδικού δέντρου φαίνονται στον αλγόριθμο 9.4 και στον αλγόριθμο 9.5. Οι μη αναδρομικές υλοποιήσεις γι' αυτές τις διασχίσεις γίνονται με τρόπο ανάλογο της ενδοδιατεταγμένης.

9.3 Διάσχιση γράφων

9.3.1 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;

Αλγόριθμος 9.1 Ενδοδιατεταγμένη διάσχιση

```
procedure Inorder (T: bintree);
begin
  if T<>empty then
  begin
    Inorder(leftchlild(T)); Visit(T);
    Inorder(rightchild(T))
  end
end
```

Αλγόριθμος 9.2 Ενδοδιατεταγμένη διάσχιση χωρίς δεύτερη αναδρομή

```
procedure Inorder (T);
begin
  while T<>empty do
  begin
    Inorder(leftchild(T)); Visit(T);
    T:=rightchild(T)
  end
end
```

Αλγόριθμος 9.3 Ενδοδιατεταγμένη διάσχιση χωρίς αναδρομή

```
procedure Inorder (T);
(* Υποθέτουμε ότι έχουμε μια εικονική μεταβλητή T για την
   οποία ισχύει rightchild(T)=tree *)
begin
  stack:=empty; push(T);
  repeat
    while rightchild(T)<>empty do
    begin
      T:=rightchild(T);
      while leftchild(T)<>empty do
        begin push(T); T:=leftchild(T) end;
      Visit(T)
    end;
    pop(T); Visit(T)
  until stack=empty
end
```

Αλγόριθμος 9.4 Προδιατεταγμένη διάσχιση

```
procedure Preorder (T:bintree);
begin
  if T<>empty then
  begin
    Visit(T); Preorder(leftchlild(T));
    Preorder(rightchild(T))
  end
end
```

Αλγόριθμος 9.5 Μεταδιατεταγμένη διάσχιση

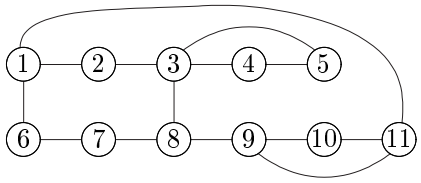
```
procedure Postorder (T);
begin
  while T<>empty do
  begin
    Postorder(leftchild(T)); Postorder(rightchild(T));
    Visit(T)
  end
end
```

- 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 πλευρές τους, χωρίζουν το γράφο σε δύο ή περισσότερες συνεκτικές συνιστώσες.

9.3.2 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*)



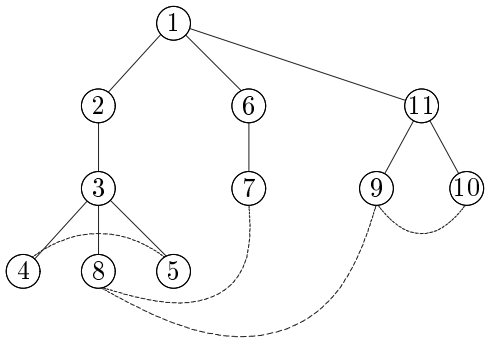
Σχήμα 9.2: Παράδειγμα γράφου στον οποίο θα γίνει αναζήτηση

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 9.6.

Αλγόριθμος 9.6 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
```

Παράδειγμα 9.3.1. Έστω ο γράφος που φαίνεται στο σχήμα 9.2. Αν καλέσουμε τη διαδικασία **bfs** με **bfs**(1) έχουμε τα βήματα που φαίνονται στο πίνακα 9.1. Το συνδετικό δέντρο με προτεραιότητα πλάτους, είναι αυτό που φαίνεται στο σχήμα 9.3. Οποιαδήποτε υλοποίηση και να χρησιμοποιηθεί, ο αλγόριθμος 9.6 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 9.3: Αναζήτηση κατά πλάτος στον γράφο

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
8	7-9-10-4-8
5	7-9-10-4-8-5
-	9-10-4-8-5
-	10-4-8-5
-	4-8-5
-	8-5
-	5
-	Ø

Table 9.1: Εκτέλεση αναζήτησης κατά πλάτος

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 9.7. Στον αλγόριθμο αυτό είναι φανερό ότι αν ο γράφος είναι συνεκτικός, τότε με την πρώτη κλήση της bfs θα επισκεφθούμε όλους τους κόμβους του.

Αλγόριθμος 9.7 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
```

9.3.3 Αναζήτηση κατά βάθος (Depth First Search)

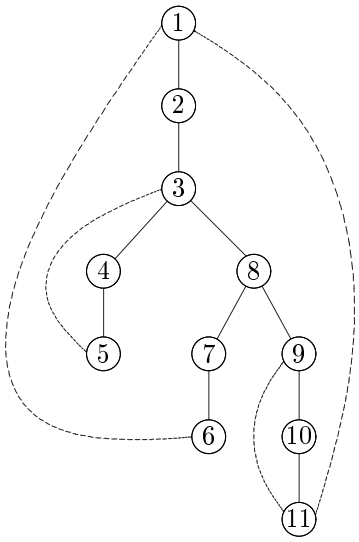
Στην αναζήτηση κατά βάθος από κάθε κόμβο v που επισκεπτόμαστε αναζητάμε άλλους κόμβους όσο το δυνατό βαθύτερα. Μετά το τέλος της διαδικασίας προκύπτει το συνδετικό δέντρο με προτεραιότητα βάθους (*depth first spanning tree*). Η υλοποίηση της διαδικασίας φαίνεται στον αλγόριθμο 9.8, ενώ το συνδετικό δέντρο με προτεραιότητα βάθους είναι αυτό που φαίνεται στο σχήμα 9.4.

Αλγόριθμος 9.8 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
```

Η πολυπλοκότητα του **dfs** είναι ίδια με το **bfs**.

Ο Αλγόριθμος **dft** είναι ακριβώς σαν τον **bft**, όπου όμως αντικαθιστούμε την κλήση **bfs** με την κλήση της **dfs**. Εκτός από το **bfs** και το **dfs** υπάρχει και το λεγόμενο **D-search** που είναι ακριβώς η ίδια διαδικασία με το **bfs** εκτός από το ότι χρησιμοποιεί μια στοίβα (*stack*) αντί ουράς.



Σχήμα 9.4: Αναζήτηση κατά βάθος στον γράφο

9.4 Το πρόβλημα της Μέγιστης ροής (Max Flow)

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 συνάρτηση 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:

[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.

Θεώρημα 9.4.1. *(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 9.9 Αλγόριθμος Ford-Fulkerson

---

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)*

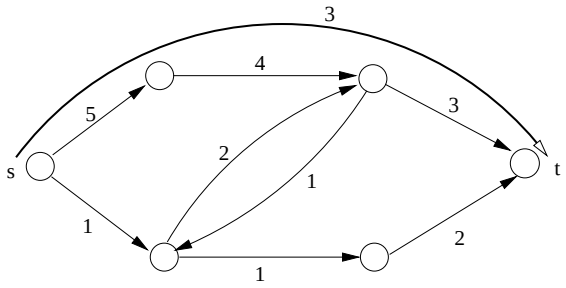
3. Επανάληψη της διαδικασίας στο παραμένον δίκτυο (residual network) εφ'όσον υπάρχει μονοπάτι από τον s στον t.

Χρειάζεται να 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 τις προσθέτουμε). Το δίκτυο που προκύπτει είναι το παραμένον δίκτυο.

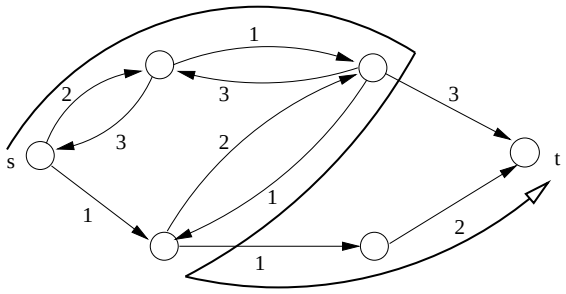
Η τελική ροή είναι το άθροισμα όλων των f_i. Αν σε κάποιο ζεύγος αντίθετων ακμών υπάρχει ροή και στις δύο ακμές, τότε η διαφορά τους αποδίδεται στην ακμή που είχε τη μεγαλύτερη ροή, ενώ η ροή της αντίθετης ακμής μηδενίζεται. Ένα παράδειγμα της εκτέλεσης του αλγορίθμου δίνεται στα Σχήματα 9.5, 9.6, 9.7.

Τα μονοπάτια που χρησιμοποιεί ο αλγόριθμος λέγονται συνήθως *μονοπάτια επαύξησης (augmenting paths)*. Πολυπλοκότητα του αλγορίθμου: για ακέραιες χωρητικότητες η πολυπλοκότητα είναι $O(|f^*||E|)$, όπου f^* η μέγιστη ροή, καθώς σε κάθε επανάληψη η ροή αυξάνεται τουλάχιστον κατά 1 μονάδα, και η εύρεση μονοπατιού από τον s στον t γίνεται σε χρόνο $O(|E|)$. 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).

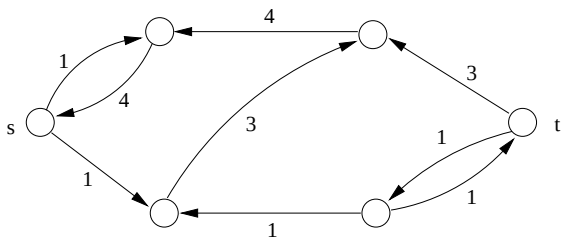
Αξίζει να σημειωθεί ότι ένα άλλο γνωστό πρόβλημα, το πρόβλημα Perfect Matching (Τέλειο Ταίριασμα) λύνεται με αναγωγή στο πρόβλημα Maximum Flow.



Σχήμα 9.5: Το αρχικό δίκτυο και η πρώτη ροή μεγέθους 3 από τον κόμβο s στον t.



Σχήμα 9.6: Το παραμένον δίκτυο και η δεύτερη ροή μεγέθους 1 από τον κόμβο s στον t.



Σχήμα 9.7: Το παραμένον δίκτυο. Δεν υπάρχει μονοπάτι από τον κόμβο s στον t.

Κεφάλαιο 10

Υπολογισιμότητα (Computability)

10.1 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 (δηλ. 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*), (δηλ. 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). Άλλοι 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 (δηλ. 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:

- Υπάρχουν άπειρα μεν, αλλά μόνο αριθμήσιμα (*countable*) διαφορετικά προγράμματα. Εκτός αυτού μπορούμε χρησιμοποιώντας κωδικοποίηση

[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

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 10.1.1. *Το* **halting problem (HP)** *είναι μη αποκρίσιμο.*

Proof. 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:

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

[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^*:

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

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.

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 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 10.1.2. 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 10.1.3. 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.

10.2 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*) $\delta : Q \times \Sigma \rightarrow A \times Q$. 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.

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 10.2.1. 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
```

$$
\begin{array}{llll}
< q_0 & 1 & 0 & q_1 > \\
< q_1 & 0 & \mathrm{R} & q_2 > \\
< q_2 & 1 & 0 & q_3 > \mid \text{halt για} < q_2 \;\; 0 > \\
< q_3 & 0 & \mathrm{R} & q_4 > \\
< q_4 & 1 & \mathrm{R} & q_4 > \\
< q_4 & 0 & \mathrm{R} & q_5 > \\
< q_5 & 1 & \mathrm{R} & q_5 > \\
< q_5 & 0 & 1 & q_6 > \\
< q_6 & 1 & \mathrm{R} & q_6 > \\
< q_6 & 0 & 1 & q_7 > \\
< q_7 & 1 & \mathrm{L} & q_7 > \\
< q_7 & 0 & \mathrm{L} & q_8 > \\
< q_8 & 1 & \mathrm{L} & q_8 > \\
< q_8 & 0 & \mathrm{R} & q_2 >
\end{array}
$$

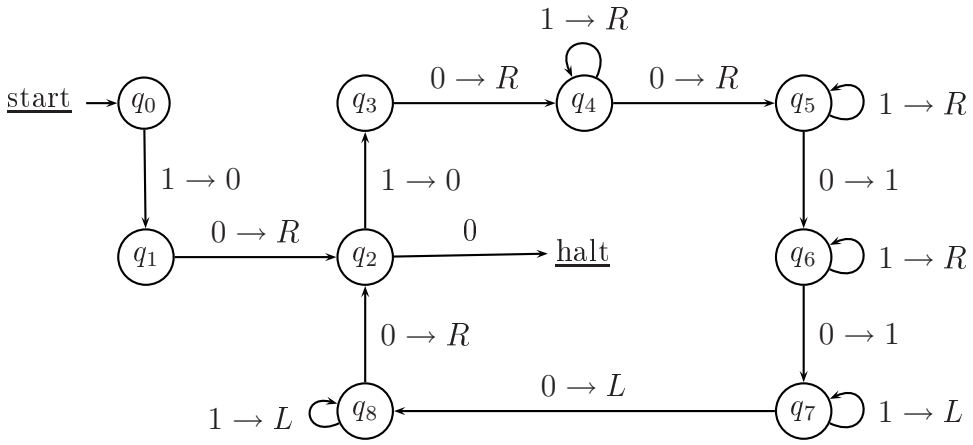
Table 10.1: TM πρόγραμμα.

Το πρόγραμμα της TM φαίνεται στο πίνακα 10.1. 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 10.2. Επίσης σε μορφή διαγράμματος καταστάσεων η ίδια TM φαίνεται στο σχήμα 10.1. 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

Table 10.2: TM σε μορφή πίνακα.

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 $(2,1,3)$.



Σχήμα 10.1: TM σε μορφή διαγράμματος καταστάσεων.

10.3 Μηχανή τυχαίας προσπέλασης (RANDOM ACCESS MACHINE)

Η μηχανή τυχαίας προσπέλασης (RAM) είναι ένα υπολογιστικό μοντέλο, που μοιάζει με μια τυπική γλώσσα *assembly*. Ορίστηκε αρχικά το 1963 από τους *Sheperdson* και *Sturgis* (URM και SRM) και αργότερα προσαρμόστηκε στις ανάγκες της θεωρίας Πολυπλοκότητας από τον *Elgott* και *Maclane*. Η RAM αποτελείται από μια ταινία εισόδου, μια ταινία εξόδου, πρόγραμμα και μνήμη. Η ταινία εισόδου είναι μια σειρά από κελιά, το καθένα από τα οποία μπορεί να περιέχει ένα ακέραιο (πιθανόν αρνητικό). Κάθε φορά που διαβάζεται ένα σύμβολο από την ταινία εισόδου η κεφαλή μετακινείται στο αμέσως δεξιότερο κελί. Η ταινία εξόδου αποτελείται επίσης από κελιά τα οποία αρχικά είναι όλα κενά. Κάθε φορά που εκτελείται μια εντολή για γράψιμο, τυπώνεται ένας ακέραιος στο κελί που δείχνει η κεφαλή εξόδου και η κεφαλή μετακινείται μια θέση προς τα δεξιά. Από την στιγμή που γράφεται ένα σύμβολο στην ταινία εξόδου, δεν μπορεί να αλλάξει.

Η μνήμη αποτελείται από ένα πίνακα καταχωρητών (*registers*) καθένας από τους οποίους μπορεί να περιέχει έναν ακέραιο. Το πρόγραμμα είναι μια ακολουθία εντολών. Οι εντολές είναι αριθμητικές, εισόδου-εξόδου, διακλάδωσης κ.α. Όλοι οι υπολογισμοί γίνονται σε ένα συσσωρευτή (*accumulator*). Ένα τυπικό σύνολο εντολών φαίνεται στον πίνακα 10.3. Κάθε εντολή αποτελείται από ένα κωδικό και μια διεύθυνση. Πρόγραμμα RAM με εξήγηση σε μια αντίστοιχη «high level» ψευδογλώσσα και identifier-labels: *pos while continue endwhile endelse*.

Κώδικας εντολής	Διεύθυνση
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JUMP	operand
10. JGTZ	label
11. JZERO	label
12. HALT	label

Table 10.3: Τυπικό σύνολο εντολών RAM.

Υπάρχει επίσης ένας μετρητής (*program counter*) που αρχικοποιείται στην πρώτη εντολή του προγράμματος και κάθε στιγμή καθορίζει την επόμενη εντολή που θα εκτελεστεί. Μετά την εκτέλεση της k-οστής εντολής ο μετρητής παίρνει αυτόματα την εντολή $k+1$ (δηλαδή την επόμενη εντολή) εκτός αν η k-οστή εντολή είναι JUMP,HALT,JGTZ ή ZERO.

Ας υποθέσουμε ότι ένα πρόγραμμα διαβάζει ακεραίους από την ταινία εισόδου και γράφει το πολύ έναν ακέραιο n στην ταινία εξόδου. Έστω $x_1, x_2, \ldots, x_n$ οι ακέραιοι που βρίσκονται στα πρώτα n κελιά της ταινίας εισόδου και ότι το πρόγραμμα γράφει τον ακέραιο y στο πρώτο κελί της ταινίας εξόδου και μετά σταματά. Τότε λέμε ότι το πρόγραμμα υπολογίζει την συνάρτηση $f(x_1, x_2, \ldots, x_n) = y$. Ένα τυπικό πρόγραμμα RAM φαίνεται στον πίνακα 10.4.

10.4 Σχέση μεταξύ TM και RAM

Θεώρημα 10.4.1. *Δίνεται μια multitape (με πολλαπλές ταινίες) TM η οποία λύνει κάποιο πρόβλημα σε χρόνο $T(n)$. Τότε υπάρχει RAM που λύνει το ίδιο πρόβλημα σε $O(T(n))$, αν υποθέσουμε ότι κάθε καταχωρητής μπορεί να αποθηκεύσει αριθμούς οσουδήποτε μήκους και σε χρόνο $O(T(n)\log T(n))$, αν λάβουμε υπόψη μας και το μήκος των αριθμών σε bits.*

Θεώρημα 10.4.2. *Δίνεται μια RAM που λύνει κάποιο πρόβλημα σε χρόνο $T(n)$ (με περιορισμό του μήκους των αριθμών που μπορούν να αποθηκευθούν στους καταχωρητές). Τότε υπάρχει multitape (με πολλαπλές ταινίες) TM που λύνει το ίδιο πρόβλημα σε $O(T^2(n))$ χρόνο.*

Πρόγραμμα RAM		Αντίστοιχη ψευδογλώσσα
	READ 1	read r1
	LOAD 1	
	JGTZ pos	if $r1 <= 0$ then write 0
	WRITE =0	
	JUMP endelse	
pos:	LOAD 1	else begin
	STORE 2	
	LOAD 1	$r2 \leftarrow r1$
	SUB =1	
	STORE 3	$r3 \leftarrow r1 - 1$
while:	LOAD 3	
	JGTZ continue	
	JUMP endwhile	while $r3 > 0$ do begin
continue:	LOAD 2	
	MULT 1	
	STORE 2	
	LOAD 3	$r2 \leftarrow r1 * r1$
	SUB =1	
	STORE 3	$r3 \leftarrow r3 - 1$
	JUMP while	
endwhile:	WRITE 2	write r2
endelse:	HALT	

Table 10.4: Τυπικό πρόγραμμα RAM.

Πόρισμα 10.4.3. Τα υπολογιστικά μοντέλα RAM και multitape TM έχουν πολυωνυμική σχέση μεταξύ τους (polynomial related - δες Ορισμό 1.1.3)

10.5 Υπολογιστικά Μοντέλα

Λόγω της θέσης του Church 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

Άλλα 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 10.5.1. *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

Κεφάλαιο 11

Αφηρημένη θεωρία πολυπλοκότητας (Abstract Theory of Complexity)

11.1 Γενικά

Η χρονική πολυπλοκότητα ενός αλγορίθμου που επιλύει κάποιο υπολογιστικό πρόβλημα είναι μία αύξουσα συνάρτηση $T(n)$ όπου n το μέγεθος της εισόδου (*input*). Συγκεκριμένα:

$$T(n) = \max\{\#\text{steps for input}x \mid |x| = n\}$$

Το μέγεθος του input εξαρτάται φυσικά από την αναπαράστασή του. Όμως αν θεωρήσουμε ότι:

- Στην αναπαράστασή του δεν χρησιμοποιούνται σύμβολα που να μην εκφράζουν τίποτα ή να εκφράζουν πληροφορίες που δεν χρειάζονται.
- Κατά την κωδικοποίηση (αντιστοίχηση των εισόδων σε αριθμούς), οι αριθμοί που μένουν αχρησιμοποίητοι είναι «λίγοι». Συγκεκριμένα υπάρχει πολυώνυμο $p(n)$ ώστε για κάθε n, ο n-οστός μεγαλύτερος κωδικός να είναι μικρότερος από $p(n)$.
- Για την αναπαράσταση αριθμών χρησιμοποιείται το δυαδικό, το δεκαδικό ή οποιοδήποτε άλλο σύστημα εκτός από το εναδικό, τότε:

Κάθε αναπαράσταση (*encoding*) της εισόδου μπορεί να διαφέρει μόνο πολυωνυμικά από μία άλλη. Αυτό σημαίνει ότι αν η χρονική πολυπλοκότητα ενός προβλήματος είναι πολυωνυμική, τότε οποιαδήποτε αναπαράσταση για την είσοδο ενός στιγμιοτύπου (*instance*) του προβλήματος και να χρησιμοποιήσουμε, η $T(n)$ παραμένει πολυωνυμική, αφού η σύνθεση πολυωνύμων είναι πολυώνυμο.

Ορισμός 11.1.1. Αν υπάρχει κάποιο πολυώνυμο P τέτοιο ώστε: $\forall n, T(n) \leq p(n)(T(n) = O(poly))$ τότε λέμε ότι ο αλγόριθμος είναι αποδοτικός (*efficient*), δηλαδή επιλύει το πρόβλημα σε πολυωνυμικό χρόνο (*Edmonds '68*).

Έτσι λοιπόν, όταν εξετάζουμε την «αφηρημένη» πολυπλοκότητα ενός προβλήματος, πρέπει να έχουμε κατά νου τα εξής:

- Δεν μας ενδιαφέρει κάποιο συγκεκριμένο υπολογιστικό μοντέλο.
- Δεν μας ενδιαφέρει κάποια συγκεκριμένη κωδικοποίηση της εισόδου.
- Δεν μας ενδιαφέρει ο συγκεκριμένος βαθμός του πολυωνύμου.

Δυστυχώς υπάρχουν πολλά ενδιαφέροντα προβλήματα για τα οποία δεν είναι γνωστό αν είναι επιλύσιμα σε πολυωνυμικό χρόνο (*tractable*), προβλήματα για τα οποία όλοι οι γνωστοί αλγόριθμοι είναι εκθετικοί. Τα υπολογιστικά προβλήματα μπορούμε να τα εντάξουμε σε κατηγορίες με βάση διάφορα χαρακτηριστικά τους. Έτσι λοιπόν, έχουμε προβλήματα αναζήτησης, βελτιστοποίησης, απόφασης, κ.ά.

11.2 Προβλήματα βελτιστοποίησης (Optimization Problems)

Στα προβλήματα βελτιστοποίησης ζητάμε τη λύση που ελαχιστοποιεί (ή μεγιστοποιεί) κάποια αντικειμενική συνάρτηση. Για κάθε στιγμιότυπο x ενός προβλήματος βελτιστοποίησης A, υπάρχει ένα σύνολο εφικτών (*feasible*) λύσεων $F(x)$. Σε κάθε λύση $s \in F(x)$, αντιστοιχούμε μέσω μιας αντικειμενικής συνάρτησης c ένα θετικό ακέραιο $c(s)$. Ζητάμε εκείνη τη λύση $s \in F(x)$, για την οποία το $c(s)$ είναι ελάχιστο (ή μέγιστο).

Παράδειγμα: Το πρόβλημα του Πλανόδιου Πωλητή (*Traveling Salesman Problem*): Δίνεται ένα πεπερασμένο σύνολο $C = \{c_1, c_2, \ldots, c_n\}$ από πόλεις και μια απόσταση $d(c_i, c_j) \in Z^+, \forall (c_i, c_j) \in C^2$. Ζητάμε μια διαδρομή (*tour*) που να περνά ακριβώς μια φορά από κάθε πόλη και να επιστρέφει στην αρχική, με ελάχιστο συνολικό μήκος. Δηλαδή θέλουμε να ορίσουμε μια μετάθεση της ακολουθίας $< c_{\pi_1}, c_{\pi_2}, \ldots, c_{\pi_n} >$ τέτοια ώστε να ελαχιστοποιήσουμε την παρακάτω έκφραση:

$$\sum_{i=1}^{n-1} d(c_{\pi_i}, c_{\pi_{i+1}}) + d(c_{\pi_n}, c_{\pi_1})$$

11.3 Προβλήματα απόφασης (Decision Problems)

Τα προβλήματα απόφασης επιδέχονται απαντήσεις της μορφής «ναι» ή «όχι». Αν θεωρήσουμε το σύνολο A όλων των εισόδων x για τις οποίες η απάντηση είναι «ναι» τότε κάθε πρόβλημα απόφασης ισοδυναμεί με ένα πρόβλημα της μορφής «$x \in A$;».

Έστω ένα οποιοδήποτε πρόβλημα βελτιστοποίησης στο οποίο για είσοδο x ζητάμε μια εφικτή λύση s που βελτιστοποιεί μια αντικειμενική συνάρτηση c. Σ' αυτά το πρόβλημα μπορούμε να αντιστοιχίσουμε το εξής πρόβλημα απόφασης: «Για είσοδο x, n υπάρχει εφικτή λύση s για το x με $c(s)$ καλύτερο του n».

Παράδειγμα 11.3.1. Δίνεται ένας γράφος $G(V, E)$. Ποια είναι η μέγιστη κλίκα; Αυτό είναι ένα πρόβλημα βελτιστοποίησης. Το αντίστοιχο πρόβλημα απόφασης είναι: «Δίνεται γράφος $G(V, E)$ και αριθμός n. Υπάρχει στον $G(V, E)$ κλίκα με n (ή περισσότερα) στοιχεία;», ή αλλιώς, «Ανήκει το $< G(V, E), n >$, στο A;», όπου:

$$A = \{< G(V, E), n > |\ G(V, E) \text{ γράφος που περιέχει κλίκα με } n \text{ στοιχεία}\}$$

Πιο τυπικά, ένα πρόβλημα απόφασης Π αποτελείται από ένα σύνολο D_Π από στιγμιότυπα του προβλήματος και από κάποιο υποσύνολο $Y_\Pi \subseteq D_\Pi$ από «ναι» στιγμιότυπα. Η περιγραφή του προβλήματος απόφασης γίνεται ως εξής: Κατ' αρχάς ορίζουμε μια γενική μορφή του προβλήματος (που περιγράφει όλα τα στιγμιότυπα αυτού), η οποία περιέχει διάφορους μεταβλητούς όρους, όπως σύνολα, γράφους, συναρτήσεις, αριθμούς, κλπ. Το σύνολο D_Π ορίζεται ως εξής:

- Ένα στιγμιότυπο ανήκει στο D_Π αν και μόνο αν μπορούμε να το κατασκευάσουμε από το «γενικό» στιγμιότυπο αντικαθιστώντας συγκεκριμένα αντικείμενα (π.χ. συγκεκριμένα σύνολα, κ.λ.π.) σε όλους τους μεταβλητούς όρους του γενικού στιγμιοτύπου.

Έπειτα θέτουμε μια ερώτηση σε σχέση με τους όρους του «γενικού»' στιγμιοτύπου (*generic instance*), η οποία μπορεί να απαντηθεί με ένα «ναι» ή με ένα «όχι». Το σύνολο Y_Π ορίζεται ως εξής:

- Ένα στιγμιότυπο ανήκει στο Y_Π όταν και μόνο όταν η απάντηση στην ερώτηση γι'αυτό το συγκεκριμένο πλέον στιγμιότυπο είναι «ναι».

Παράδειγμα 11.3.2. Το πρόβλημα ύπαρξης ισομορφικού υπογράφου (*subgraph isomorphism*).
Γενικό στιγμιότυπο: Δύο γράφοι $G_1(V_1, E_1)$, $G_2(V_2, E_2)$ ($D_\Pi = \{(G_1, G_2) \mid G_1, G_2 \text{: γράφοι}\}$).

Ερώτηση: Υπάρχει υπογράφος του G_1 ισομορφικός με τον G_2; Δηλαδή υπάρχουν $V' \subseteq V_1$ και $E' \subseteq E_1$ έτσι ώστε $|V'| = |V_2|$ και $|E'| = |E_2|$ και συνάρτηση $f : V_2 \to V', 1{-}1$ και τέτοια ώστε $(u, v) \in E_2$ αν και μόνο αν $(f(u), f(v)) \in E'$; $(Y_\Pi = \{G_1, G_2\} \mid \exists V' \, \exists E' \ldots\})$.

Το επόμενο παράδειγμα αφορά το πρόβλημα του πλανόδιου πωλητή (TSP) το οποίο ορίσαμε ήδη σαν πρόβλημα βελτιστοποίησης. Ας ορίσουμε τώρα το αντίστοιχο πρόβλημα απόφασης.

Παράδειγμα 11.3.3. Το πρόβλημα TSP.
Γενικό στιγμιότυπο: Ένα πεπερασμένο σύνολο $C = \{c_1, c_2, \ldots, c_n\}$ από πόλεις, ένας πίνακας αποστάσεων $d(c_i, c_j) \in Z^+, \forall (c_i, c_j) \in C$ και κάποιο όριο $B \in Z^+$.
Ερώτηση: Υπάρχει διαδρομή που περνά από όλες τις πόλεις που ανήκουν στο C, με συνολικό μήκος $\leq B$; Δηλαδή υπάρχει διάταξη της ακολουθίας $< c_{\pi_1}, c_{\pi_2}, \ldots, c_{\pi_n} >$ τέτοια ώστε:

$$\sum_{i=1}^{n-1} d(c_{\pi_i}, c_{\pi_{i+1}}) + d(c_{\pi_n}, c_{\pi_1}) \leq B;$$

Στο προηγούμενο παράδειγμα φαίνεται ο τρόπος αντιστοίχησης ενός προβλήματος βελτιστοποίησης σε πρόβλημα απόφασης. Εδώ βλέπουμε ότι το πρόβλημα απόφασης δε μπορεί να είναι πιο δύσκολο απ' το πρόβλημα βελτιστοποίησης. Δηλαδή αν μπορούσαμε να λύσουμε το πρόβλημα βελτιστοποίησης εύκολα (δηλαδή σε πολυωνυμικό χρόνο, ντετερμινιστικά) τότε σίγουρα θα μπορούσαμε να αποφανθούμε για το αντίστοιχο πρόβλημα απόφασης, συγκρίνοντας απλά το ελάχιστο συνολικό μήκος της διαδρομής με το B.

Από την άλλη πλευρά αν αποδεικνύαμε ότι το πρόβλημα απόφασης είναι δύσκολο, τότε το πρόβλημα βελτιστοποίησης θα είναι τουλάχιστον τόσο δύσκολο. Σε πολλές περιπτώσεις το πρόβλημα απόφασης είναι το ίδιο δύσκολο με το αντίστοιχο πρόβλημα βελτιστοποίησης.

11.4 Προβλήματα απόφασης και γλώσσες

Ένας από τους λόγους που μετατρέπουμε όλα τα προβλήματα σε προβλήματα απόφασης και μελετούμε αυτά, είναι ότι τα προβλήματα απόφασης έχουν άμεση σχέση με τις τυπικές γλώσσες.

Για ένα οποιοδήποτε πεπερασμένο σύνολο Σ από σύμβολα (αλφάβητο), ορίζουμε ως Σ^* το σύνολο από όλες τις πεπερασμένου μήκους συμβολοσειρές (*strings*) που αποτελούνται από τα σύμβολα του Σ. Για παράδειγμα, αν $\Sigma =$

$\{0,1\}$, τότε $\Sigma^* = \{\varepsilon, 0, 1, 01, 001, 101, \ldots\}$. Δηλαδή $|\Sigma^*| = \infty$. Αν $L \subseteq \Sigma^*$, τότε λέμε ότι η L είναι μια γλώσσα (*language*) στο αλφάβητο Σ.

Η αντιστοιχία ανάμεσα στα προβλήματα απόφασης και στις γλώσσες φαίνεται από το ότι για να εισαχθεί ένα στιγμιότυπο κάποιου προβλήματος σε ένα υπολογιστικό μοντέλο πρέπει να περιγραφεί σαν μια ακολουθία συμβόλων (*string*) από κάποιο πεπερασμένο σύνολο συμβόλων (αλφάβητο).

Έτσι λοιπόν ένα πρόβλημα Π και ο τρόπος κωδικοποίησής του e χωρίζουν το Σ^*, σε τρεις κλάσεις από *strings*:

- Η πρώτη κλάση περιέχει εκείνα τα *strings* τα οποία δεν είναι κωδικοποιήσεις στιγμιοτύπων του προβλήματος.
- Η δεύτερη κλάση περιέχει τα *strings* που είναι κωδικοποιήσεις στιγμιοτύπων για τα οποία η απάντηση στο πρόβλημα απόφασης είναι «όχι».
- Τέλος η τρίτη κλάση περιέχει τα *strings* που είναι κωδικοποιήσεις στιγμιοτύπων για τα οποία η απάντηση είναι «ναι».

Αυτή η τελευταία κλάση, ορίζουμε ότι είναι μία γλώσσα, $L \sqsubseteq \Sigma^*$, η οποία εξαρτάται κάθε φορά από το πρόβλημα Π και τον τρόπο κωδικοποίησης e. Δηλαδή:

$$L(\Pi, e) = \{x \in \Sigma^* \mid x \text{ αναπαριστά (μέσω του } e) \text{ ένα στιγμιότυπο } I \in Y_\Pi\}$$

Έτσι λοιπόν, ένα *string* x ανήκει στη γλώσσα L ανν το πρόβλημα απόφασης για το στιγμιότυπο του Π το οποίο αναπαρίσταται (μέσω του e) με το string x δίνει απάντηση «ναι».

11.5 Οι κλάσεις P και NP

Ονομάζουμε P (*Polynomial*) την κλάση των προβλημάτων τα οποία επιλύονται σε πολυωνυμικό χρόνο από κάποιο ντετερμινιστικό αλγόριθμο. Ο ντετερμινιστικός αλγόριθμος θα υλοποιείται σε κάποιο υπολογιστικό μοντέλο. Το υπολογιστικό μοντέλο στο οποίο θα αναφερόμαστε από εδώ και στο εξής είναι η ντετερμινιστική μηχανή Turing (*Deterministic Turing Machine, DTM*).

Λέμε ότι ένα DTM πρόγραμμα M με ένα αλφάβητο Σ **αποδέχεται** (*accepts*) το string $x \in E$ αν και μόνο αν το M σταματά σε κατάσταση αποδοχής q_Y για το x. Συνεπώς η γλώσσα L_M την οποία αναγνωρίζει το πρόγραμμα M είναι:

$$L_M = \{x \in \Sigma^* : M \text{αποδέχεται το } x\}$$

Αυτός ο ορισμός δεν απαιτεί από το M να σταματά για όλες τις εισόδους, παρά μόνο για εκείνες που ανήκουν στην L_M. Δηλαδή αν $x \in \Sigma^* - L_M$, τότε

το M με είσοδο x μπορεί να σταματά σε κατάσταση απόρριψης q_N, ή μπορεί να συνεχίζει επ' άπειρον.

Μια γλώσσα είναι **αναδρομικά αριθμήσιμη** (*recursively enumerable, RE*) ανν υπάρχει DTM πρόγραμμα M που την αποδέχεται. Αν επιπλέον για κάθε $x \in \Sigma^* - L_M$ το M σταματά στην κατάσταση q_N τότε λέμε ότι το DTM πρόγραμμα M **αποκρίνεται** (*decides*) για τη γλώσσα L. Μία γλώσσα L είναι **αναδρομική** (*Recursive*) αν υπάρχει DTM πρόγραμμα M που αποκρίνεται για την L.

Ένα DTM πρόγραμμα M που αποκρίνεται για μία γλώσσα L_M, λέμε ότι λύνει ένα πρόβλημα απόφασης Π το οποίο αντιστοιχεί σε μία γλώσσα $L(\Pi, e)$, αν και μόνο αν $L_M = L(\Pi, e)$.

Ο χρόνος μιας DTM (*Deterministic Turing Machine*) M για κάποιο $n =$ μήκος του input ορίζεται ως ο μέγιστος χρόνος για να αποκριθεί η M για οποιοδήποτε x μήκους n. Δηλαδή:

$$T(n) = \max_{|x|=n}\{\text{steps to decide } x\}$$

Ένα DTM πρόγραμμα M καλείται πρόγραμμα πολυωνυμικού χρόνου αν υπάρχει ένα πολυώνυμο p τέτοιο ώστε: $\forall n \in Z^+ : T_M(n) \leq p(n)$.

Τώρα μπορούμε να ορίσουμε την κλάση P ως εξής: Η κλάση P (*Polynomial Time*) περιέχει εκείνες τις γλώσσες που είναι αποκρίσιμες σε πολυωνυμικό χρόνο από μία μηχανή.

$$P = \{L \mid \exists \text{ πολυωνυμικού χρόνου DTM που αποκρίνεται για την γλώσσα } L\}$$

Ένας ισοδύναμος ορισμός είναι:

$$P = \{L \mid \exists \text{ πολυωνυμικού χρόνου DTM που αποδέχεται την γλώσσα } L\}$$

Για να δείξουμε την ισοδυναμία των δύο ορισμών αρκεί να δείξουμε ότι: Αν υπάρχει πολυωνυμικός ντετερμινιστικός αλγόριθμος που αποδέχεται μία γλώσσα L, τότε υπάρχει αλγόριθμος που μπορεί να αποκριθεί για την L σε πολυωνυμικό χρόνο (το αντίστροφο είναι προφανές). Η απόδειξη γίνεται ως εξής: Έστω ότι έχουμε μία γλώσσα L για την οποία ξέρουμε ότι υπάρχει ένας ντετερμινιστικός αλγόριθμος A που την αποδέχεται σε πολυωνυμικό χρόνο, $O(n^k)$. Αυτό σημαίνει ότι υπάρχει σταθερά c τέτοια ώστε, ο αλγόριθμος A να αποδέχεται την L το πολύ σε $T = cn^k$ βήματα. Φτιάχνουμε έναν άλλο αλγόριθμο A', ο οποίος (με το ίδιο input string x) εξομοιώνει τον A για χρόνο T. Αν στο τέλος του χρόνου T ο A αποδέχεται το x τότε ο A' το αποδέχεται επίσης. Αν στο τέλος του χρόνου T ο A δεν έχει αποδεχθεί το x, τότε ο A' το απορρίπτει. Δηλαδή έχουμε έναν ντετερμινιστικό αλγόριθμο ο οποίος σε πολυωνυμικό χρόνο αποκρίνεται (*decides*) για τη γλώσσα L.

Ονομάζουμε NP (*Non-deterministic Polynomial Time*) την κλάση των προβλημάτων τα οποία επιλύονται σε πολυωνυμικό χρόνο από κάποιο μη-ντετερμινιστικό αλγόριθμο. Ο μη-ντετερμινιστικός αλγόριθμος είναι ένα ιδεατό μοντέλο το οποίο σε κάθε βήμα μπορεί να κάνει επιλογές. Αποδέχεται την είσοδο x αν υπάρχει ένα μονοπάτι στο δέντρο επιλογών, στο τέλος του οποίου αποδέχεται το x και απορρίπτει το x αν όλα τα μονοπάτια οδηγούν σε απόρριψη του x. Όπως και παραπάνω, επειδή υπάρχει περιορισμός στο διατιθέμενο χρόνο, «αποδέχεται» είναι ισοδύναμο με «αποκρίνεται».

Εναλλακτικά μπορούμε να θεωρήσουμε ότι, ένας μη-ντετερμινιστικός αλγόριθμος δουλεύει σε δύο στάδια: Δεδομένου ενός στιγμιότυπου I ενός προβλήματος, στο πρώτο στάδιο ο αλγόριθμος μαντεύει κάποια πιθανή λύση S. Στο δεύτερο στάδιο, ο αλγόριθμος επαληθεύει ότι η S είναι πράγματι λύση του προβλήματος (αν αυτό ισχύει).

Παράδειγμα 11.5.1. Αναζήτηση σε μη-ταξινομημένο πίνακα $a[n]$. Ντετερμινιστικά χρειαζόμαστε χρόνο $\Theta(n)$. Μη-ντετερμινιστικά χρειαζόμαστε χρόνο $\Theta(1)$. Ο μη-ντετερμινιστικός αλγόριθμος δουλεύει ως εξής:

```
choose a[i]
verify : if a[i]=x then found
```

Παράδειγμα 11.5.2. Ταξινόμηση n στοιχείων. Ντετερμινιστικά χρειαζόμαστε χρόνο $\Theta(n \log n)$. Μη-ντετερμινιστικά χρειαζόμαστε χρόνο $\Theta(n)$. Ο μη-ντετερμινιστικός αλγόριθμος δουλεύει ως εξής:

```
choose a permutation
verify : a[i]<a[i+1] for all i
```

Η ύπαρξη πολυωνυμικού, μη-ντετερμινιστικού αλγορίθμου χαρακτηρίζει προβλήματα για τα οποία η εύρεση της λύσης είναι χρονοβόρα (εκθετική) ενώ ο έλεγχος της ορθότητας αυτής (π.χ. μια απόδειξη) είναι γρήγορος (πολυωνυμικός).

Παράδειγμα 11.5.3. Το πρόβλημα SAT. Έστω x_i προτασιακές μεταβλητές που παίρνουν τιμή True ή False. Ονομάζουμε:

- literals, τους όρους $x_i, \neg x_i$,
- clauses, τις διαζεύξεις (disjunctions) από literals $literal_1 \vee literal_2 \vee \ldots \vee literal_m$
- CNF (Conjuctive Normal Form), την ακόλουθη μορφή:

$$clause_1 \wedge clause_2 \wedge \ldots \wedge clause_n$$

Ορίζουμε σαν πρόβλημα της Ικανοποιησιμότητας (*SATisfiability*) το εξής:

SAT
Δεδομένα: Μία boolean έκφραση σε CNF
Ερώτηση: Υπάρχει ανάθεση τιμών στις μεταβλητές που να ικανοποιεί την έκφραση (δηλαδή η έκφραση να αποτιμάται σε True);

Ντετερμινιστικά οι γνωστοί τρόποι χρειάζονται $O(2^n)$ χρόνο (εκθετικό). Μη-ντετερμινιστικά χρειαζόμαστε πολυωνυμικό χρόνο. Ο μη-ντετερμινιστικός αλγόριθμος δουλεύει ως εξής:

- Διάλεξε μια απονομή αλήθειας
- Έλεγξε αν αυτή η απονομή ικανοποιεί την boolean έκφραση

Είναι φανερό ότι η επαλήθευση γίνεται εύκολα σε γραμμικό χρόνο ως προς το μήκος του τύπου.

Ο χρόνος εκτέλεσης μίας NDTM (*Non-Deterministic Turing Machine*) για ένα input x είναι το μήκος του συντομότερου μονοπατιού στο τέλος του οποίου αποδέχεται το x. Δηλαδή ένα NDTM πρόγραμμα M κάνει κάθε φορά εκείνη την επιλογή η οποία το οδηγεί στο να αποδεχθεί το input x (αν αυτό γίνεται) όσο το δυνατόν γρηγορότερα. Η χρονική πολυπλοκότητα του NDTM προγράμματος M ορίζεται ως εξής:

$$T(n) = \begin{cases} \max\limits_{|x|=n}\{\min \#\ \text{βημάτων για αποδοχή } x\}, & \text{αν κάποιο τέτοιο } x \text{ αποδεκτό} \\ 1, & \text{αλλιώς} \end{cases}$$

Ένα NDTM πρόγραμμα M είναι πολυωνυμικού χρόνου αν:

$$\exists \text{ πολυώνυμο } p:\ T(n) \leq p(n), \forall n$$

Έτσι μπορούμε τώρα να ορίσουμε και τυπικά την κλάση NP ως εξής:

$$NP = \{L \mid \exists\ NDTM \text{ πρόγραμμα } M \text{ το οποίο μέσα σε πολυωνυμικό χρόνο αποδέχεται τη γλώσσα } L\}$$

Συνεπώς τα προβλήματα που ανήκουν στο NP έχουν την ιδιότητα ότι το μήκος του μονοπατιού που ακολουθεί ο μη-ντετερμινιστικός αλγόριθμος για να καταλήξει σε αποδοχή του input x είναι πολυωνυμικό ως προς το μέγεθος του x. Ισοδύναμα μπορούμε να πούμε ότι ένα πρόβλημα ανήκει στο NP αν η επαλήθευση της λύσης του μπορεί να γίνει σε πολυωνυμικό χρόνο.

11.5.1 Σχέση μεταξύ P και NP

Προφανώς ισχύει $P \subseteq NP$. Δηλαδή οποιοδήποτε πρόβλημα λύνεται σε πολυωνυμικό χρόνο από έναν ντετερμινιστικό αλγόριθμο μπορεί να λυθεί σε πολυωνυμικό χρόνο επίσης και από έναν μη- ντετερμινιστικό αλγόριθμο, επειδή ο ντετερμινιστικός αλγόριθμος είναι ειδική περίπτωση μη- ντετερμινιστικού αλγορίθμου (σε κάθε βήμα έχουμε μία μόνο επιλογή). Το ερώτημα είναι αν τα προβλήματα που λύνονται σε πολυωνυμικό χρόνο με ένα NDTM μπορούν να λυθούν σε πολυωνυμικό χρόνο και με ένα DTM, δηλαδή αν $P = NP$.

Γνωρίζουμε πάντως, ότι ένας μη-ντετερμινιστικός αλγόριθμος μπορεί να εξομοιωθεί από έναν ντετερμινιστικό με εκθετική όμως απώλεια χρόνου. Δηλαδή ισχύει:

$$NP \subseteq DEXPTIME$$

Οι περισσότεροι ερευνητές πιστεύουν πως το P είναι διαφορετικό από το NP. Παρ' όλα αυτά η απόδειξη αυτής της εικασίας παραμένει ως σήμερα ένα άλυτο πρόβλημα.

11.6 Η έννοια της αναγωγής - Το θεώρημα του Cook

11.6.1 Αναγωγή κατά Karp (Karp reduction)

Λέμε ότι ένα πρόβλημα A ανάγεται πολυωνυμικά κατά $Karp$ σε ένα πρόβλημα B και συμβολίζουμε, $A \leq_m^p B$, τότε και μόνο τότε, όταν υπάρχει μία συνάρτηση f υπολογίσιμη σε πολυωνυμικό χρόνο $f \in P_f$[1] τέτοια ώστε $\forall x(x \in A \iff f(x) \in B)$. Δηλαδή:

$$A \leq_m^p B : \exists f \in P_f, \forall x(x \in A \iff f(x) \in B)$$

Ισχύουν οι εξής ιδιότητες:

1. Ανακλαστική: $A \leq_m^p A$.
2. Μεταβατική: Αν $A \leq_m^p B$ και $B \leq_m^p C$ τότε $A \leq_m^p C$. Αυτό αποδεικνύεται εύκολα ως εξής:
   - $A \leq_m^p B : \exists f \in P_f, \forall x(x \in A \iff f(x) \in B)$
   - $B \leq_m^p C : \exists g \in P_f, \forall x(x \in B \iff g(x) \in C)$

[1] P_f είναι η κλάση των συναρτήσεων που υπολογίζονται σε πολυωνυμικό χρόνο.

Δηλαδή:$\exists f, g \in P_f, \forall x((x \in A \iff f(x) \in B) \iff g(f(x)) \in C)$. Άρα

$$\exists h \in P_f, \forall x(x \in Ax \iff h(x) \in C)$$

όπου h είναι η σύνθεση των f, g και είναι πολυωνυμική αφού η σύνθεση πολυωνύμων είναι πολυώνυμο. Συνεπώς $A \leq_m^p C$.

3. Αν $A \leq_m^p B$ και $B \leq_m^p A$ τότε $A \equiv_m^p B$, και λέμε ότι τα προβλήματα A, B είναι ισοδύναμα ως προς $\leq_m^p$, (π.χ. αν $A,\ B \in P \Rightarrow A \equiv_m^p B$).

4. Αν $A \leq_m^p B$ και $B \in P \Rightarrow A \in P$. Αυτό αποδεικνύεται εύκολα ως εξής: Αφού $A \leq_m^p B$ σημαίνει ότι υπάρχει συνάρτηση f υπολογίσιμη σε πολυωνυμικό χρόνο τέτοια ώστε:

$$\forall x(x \in A \iff f(x) \in B)$$

Κάθε στιγμιότυπο x του προβλήματος A λοιπόν, που μας δίνεται μπορούμε να το μετασχηματίζουμε (μέσω της f) σε ένα στιγμιότυπο $f(x)$, του προβλήματος B. Εκτός αυτού, το μήκος του $f(x)$ είναι πολυωνυμικό ως προς το μήκος του x. Όμως $B \in P$, δηλαδή υπάρχει ντετερμινιστικός αλγόριθμος ο οποίος λύνει σε πολυωνυμικό χρόνο το B. Τρέχουμε αυτόν τον αλγόριθμο και αν απαντά ότι $f(x) \in B$ τότε σημαίνει ότι $x \in A$, ειδάλλως, αν δηλαδή $f(x) \notin B$ τότε $x \notin A$. Άρα έχουμε έναν ντετερμινιστικό αλγόριθμο, ο οποίος σε πολυωνυμικό χρόνο αποφασίζει το A. Συνεπώς $A \in P$.

11.6.2 Hardness - Completeness

Αν έχουμε μία κλάση προβλημάτων C και ισχύει ότι: $\forall B \in C\colon B \leq A$ τότε το πρόβλημα A ονομάζεται C-δύσκολο (C-*hard*) ως προς $\leq$. Αν επιπλέον $A \in C$ τότε το A ονομάζεται C-**πλήρες** (C-*complete*), δηλαδή ανήκει στα πιο δύσκολα προβλήματα της κλάσης C (χαρακτηρίζει την δυσκολία της κλάσης C).

Ένα πρόβλημα L είναι **NP-complete** ως προς $\leq_m^p$ αν:

$$(L \in NP) \wedge (\forall L' \in NP :\ L' \leq_m^p L)$$

Τα **NP-complete** είναι τα πιο δύσκολα προβλήματα της κλάσης NP. Αν ένα **NP-complete** πρόβλημα αποδειχθεί ότι ανήκει στο P, τότε (αφού όλα τα προβλήματα που ανήκουν στο NP ανάγονται σ' αυτό), σύμφωνα με την ιδιότητα (4), όλα τα προβλήματα του NP θα ανήκουν στο P.

Λήμμα 11.6.1. Αν $L_1 \leq_m^p L_2$, το L_1 είναι **NP-complete** και $L_2 \in NP$ τότε το L_2 είναι **NP-complete**.

Proof. Αφού L_1 είναι NP-complete σημαίνει ότι για οποιοδήποτε $L_3 \in NP$ ισχύει $L_3 \leq_m^p L_1$. Όμως από την υπόθεση έχουμε: $L_1 \leq_m^p L_2$ και σύμφωνα με την ιδιότητα (2) της αναγωγής $\leq_m^p$, έχουμε $L_3 \leq_m^p L_2$. Συνεπώς το L_2 είναι NP-complete. □

Το παραπάνω λήμμα υποδεικνύει τον τρόπο με τον οποίο χρησιμοποιούμε την αναγωγή. Προσπαθούμε να ανάγουμε ένα γνωστό **NP-complete** πρόβλημα σε ένα πρόβλημα για το οποίο ξέρουμε ότι ανήκει στο NP, αποδεικνύοντας έτσι ότι και αυτό είναι NP-complete. Για να δείξουμε όμως ότι κάποια προβλήματα είναι NP-complete, χρειαζόμαστε ένα πρόβλημα που να ξέρουμε ότι είναι **NP-complete** προκειμένου να το ανάγουμε σε αυτά. Αυτό το «πρώτο» **NP-complete** πρόβλημα μας το έδωσε ο Cook με το θεώρημά του που θα διατυπώσουμε σε επόμενη παράγραφο.

11.6.3 Αναγωγή κατά Cook (Cook reduction)

Λέμε ότι ένα πρόβλημα A ανάγεται κατά Cook σε ένα πρόβλημα B και συμβολίζουμε, $A \leq_T^P B$, αν το A μπορεί να αποφασιστεί από μία πολυωνυμικού χρόνου ντετερμινιστική μηχανή Turing η οποία χρησιμοποιεί ένα **μαντείο** (*oracle*) για το B. Αυτό σημαίνει ότι η DTM μπορεί να κάνει οσεσδήποτε ερωτήσεις για οποιοδήποτε στιγμιότυπο του B και να πάρει στιγμιαία σωστές απαντήσεις.

Για να το επιτύχει αυτό η μηχανή Turing M έχει μια επιπλέον ταινία ονομαζόμενη *query -tape* στην οποία π.χ. τοποθετεί ένα string που κωδικοποιεί στιγμιότυπο x του προβλήματος B. Το μαντείο τότε αποφαίνεται $x \in B$ ή $x \notin B$ επάνω στην query -tape σε χρόνο που δεν μετράει για την πολυπλοκότητα της M. Η M μπορεί να χρησιμοποιήσει την απάντηση και αργότερα να τοποθετήσει άλλα ερωτήματα -strings στην query -tape.

Συμβολίζουμε με P^A την κλάση των προβλημάτων που μπορούν να λυθούν με ένα DTM σε πολυωνυμικό χρόνο χρησιμοποιώντας μαντείο για το A, ή αλλιώς:

$$P^A = \{L \mid L \leq_T^P A\}$$

Εντελώς ανάλογα με την αναγωγή κατά Karp, ισχύουν και εδώ οι ιδιότητες:

- ανακλαστική,
- μεταβατική,
- $\equiv_T^P$

Επίσης ορίζονται εντελώς ανάλογα οι έννοιες:

- NP-hard ως προς $\leq_T^P$

- NP-complete ως προς $\leq_T^P$

Επιπλέον ισχύει ότι:$A \leq_m^p B \Rightarrow A \leq_T^P B$

Συνεπώς αν ένα πρόβλημα είναι NP-complete ως προς $\leq_m^p$ τότε είναι NP-complete ως προς $\leq_T^P$. Το αντίστροφο όμως δεν ισχύει. Από εδώ και στο εξής (δηλαδή στις αναγωγές των προβλημάτων, αλλά και στο θεώρημα του Cook), θα χρησιμοποιήσουμε την αναγωγή κατά Karp ($\leq_m^p$) διότι είναι πιο εύχρηστη.

11.6.4 Το Θεώρημα του Cook

Πριν διατυπώσουμε το θεώρημα του Cook, κρίνεται σκόπιμη μια επανάληψη βασικών στοιχείων του προτασιακού λογισμού. Οι λογικές μεταβλητές που χρησιμοποιούμε στον προτασιακό λογισμό, παίρνουν τιμές στο σύνολο $\{T, F\}$. Οι τελεστές που συνδέουν λογικές μεταβλητές για το σχηματισμό λογικών εκφράσεων (boolean formulas) είναι: $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$.

Έτσι λοιπόν αν $p_1,\ p_2$ είναι λογικές μεταβλητές τότε αυτές μπορούν να δώσουν τις παρακάτω λογικές εκφράσεις:

- $\neg p$
- $p_1 \wedge p_2$
- $p_1 \vee p_2$
- $p_1 \rightarrow p_2$
- $p_1 \leftrightarrow p_2$

Δύο λογικές εκφράσεις λέγονται ισοδύναμες όταν έχουν την ίδια αληθοτιμή για όλες τις δυνατές απονομές αληθοτιμών στις λογικές μεταβλητές. Αυτό μπορεί να ελεγχθεί π.χ. με πίνακες αληθείας.

Η λογική έκφραση $p_1 \leftrightarrow p_2$ είναι ισοδύναμη με την $(p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_1)$. Συνεπώς δεν χρειαζόμαστε τον δυαδικό τελεστή $\leftrightarrow$. Επίσης η έκφραση $p_1 \rightarrow p_2$ είναι ισοδύναμη με την $\neg p_1 \vee p_2$. Άρα δεν χρειαζόμαστε ούτε τον τελεστή $\rightarrow$. Παρ' όλο που και από τους τρεις εναπομείναντες τελεστές, δεν είναι όλοι αναγκαίοι, θα χρησιμοποιήσουμε και τους τρεις για μεγαλύτερη ευκολία.

Ορισμός 11.6.2. Αν $x_1, x_2, \ldots$ είναι προτασιακές μεταβλητές, ονομάζουμε:

- literals: όρους όπως $x_1, x_3, \neg x_1, \neg x_5$
- clauses: διαζεύξεις (disjunctions) από litetals, π.χ. $(x_1 \vee \neg x_2 \vee \neg x_5)$

- Conjuctive Normal Form (CNF) την ακόλουθη μορφή που μπορεί να έχει η boolean formula:

$$(clause_1 \wedge clause_2 \wedge \ldots \wedge clause_m)$$

Θεώρημα 11.6.3. *Κάθε λογική έκφραση είναι ισοδύναμη με μία σε CNF.*

Ορισμός 11.6.4. Ικανοποιήσιμη (satisfiable) ονομάζεται μία boolean formula όταν υπάρχει απονομή αλήθειας (truth assignment) στις προτασιακές μεταβλητές που την αποτελούν, έτσι ώστε η έκφραση να παίρνει την τιμή True (T).

Το πρόβλημα SAT
Δεδομένα: Μία boolean formula σε CNF.
Ερώτηση: Είναι η boolean formula ικανοποιήσιμη;

Παράδειγμα 11.6.5. Η φόρμουλα $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ είναι ικανοποιήσιμη. Μία απονομή αλήθειας που την ικανοποιεί είναι η $(x_1, x_2) = (T, T)$. Η φόρμουλα, $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge \neg x_1$ δεν είναι ικανοποιήσιμη.

Θεώρημα 11.6.6. *(Cook) Το πρόβλημα SAT είναι NP-complete.*

Proof. Κατ' αρχάς πρέπει να αποδείξουμε ότι $SAT \in NP$. Αυτό όμως είναι εύκολο. Ένας μη-ντετερμινιστικός αλγόριθμος που λύνει το SAT είναι ο εξής:

1. μάντεψε μία απονομή αλήθειας
2. έλεγξε αν η έκφραση αποτιμάται σε True

Ο έλεγχος μπορεί να γίνει σε γραμμικό χρόνο ως προς το μέγεθος της φόρμουλας και συνεπώς $SAT \in NP$.

Το επόμενο που πρέπει να δείξουμε είναι ότι όλα τα προβλήματα που ανήκουν στο NP, ανάγονται (κατά Karp) στο SAT. Όλα αυτά τα προβλήματα όμως, μπορούν να επιλυθούν (εξ ορισμού αφού ανήκουν στο NP) από μία NDTM που τρέχει σε πολυωνυμικό χρόνο. Έτσι λοιπόν, αρκεί να αποδείξουμε το εξής: Ένας οποιοσδήποτε υπολογισμός που κάνει μία NDTM πολυωνυμικού χρόνου M μπορεί να αναπαρασταθεί από μία boolean formula Φ πολυωνυμικού μήκους έτσι ώστε, να υπάρχει υπολογισμός του M που αποδέχεται το x αν και μόνο αν υπάρχει απονομή αλήθειας που ικανοποιεί την boolean formula $\Phi(x)$.

Το NDTM πρόγραμμα M θα δέχεται λοιπόν, σαν είσοδο ένα string x με $|x| = n$. Αν υπάρχει υπολογισμός στο τέλος του οποίου το M αποδέχεται το x , αυτός ο υπολογισμός θα γίνεται σε πολυωνυμικό χρόνο $T_M(n)$. Δηλαδή θα υπάρχει ένα πολυώνυμο $p(n)$ έτσι ώστε:

$$T_M(n) \leq p(n), \forall n$$

Θεωρούμε ότι η κεφαλή βρίσκεται αρχικά στη θέση 0 και εφόσον ο αριθμός βημάτων είναι το πολύ $p(n)$, μπορούν να γραφτούν οι θέσεις της ταινίας από $-p(n)$ έως $p(n)$. Οι διαφορετικές χρονικές στιγμές είναι $p(n)+1$ γιατί μέσα σ' αυτό το χρόνο γίνονται $p(n)$ βήματα. Δηλαδή αν το πρόγραμμα M αποδέχεται το x, σημαίνει ότι μετά από $p(n)$ βήματα φτάνει σε κατάσταση q_Y. Φυσικά μπορεί να φτάσει και πιο νωρίς. Σ' αυτήν την τελευταία περίπτωση θεωρούμε ότι το configuration παραμένει σταθερό σε όλες τις μεταγενέστερες χρονικές στιγμές. θεωρούμε πως η χρονική στιγμή στην οποία αρχίζουν οι υπολογισμοί για την επαλήθευση είναι η στιγμή 0. Άρα η τελευταία χρονική στιγμή είναι η $p(n)$.

Το σύνολο των καταστάσεων στις οποίες μπορεί να βρεθεί η T.M. είναι,

$$Q = \{q_0, q_1, \ldots, q_r\}, \qquad r = |Q| - 1, \quad q_1 = q_Y, \quad q_2 = q_N$$

Το σύνολο των συμβόλων που μπορεί να γράφει η T.M. είναι,

$$\Gamma = \{s_0, s_1, \ldots, s_v\}, \qquad v = |\Gamma| - 1, \quad s_0 = \text{blank}$$

Οι λογικές μεταβλητές που θα χρησιμοποιήσουμε για να κατασκευάσουμε τη φόρμουλα $\Phi(x)$ είναι:

- $Q[i,k],\ 0 \le i \le p(n),\ 0 \le k \le r$ κωδικοποιούν την κατάσταση q_k στην οποία βρίσκεται η T.M. τη χρονική στιγμή i . Δηλαδή η μεταβλητή $Q[i,k]$ θα είναι True αν και μόνο αν τη χρονική στιγμή i η T.M. βρίσκεται στην κατάσταση q_k. Ο αριθμός αυτών των μεταβλητών είναι $(r+1)\cdot(p(n)+1)$.

- $H[i,j],\ 0 \le i \le p(n),\ -p(n) \le j \le p(n)$ κωδικοποιούν τη θέση j στην οποία βρίσκεται η κεφαλή τη χρονική στιγμή i . Δηλαδή η μεταβλητή $H[i,j]$ θα είναι True ανν τη χρονική στιγμή i η κεφαλή βρίσκεται στη θέση (κυψέλη) j. Ο αριθμός αυτών των μεταβλητών είναι $(p(n)+1)\cdot(2p(n)+1)$.

- $S[i,j,l],\ 0 \le i \le p(n),\ -p(n) \le j \le p(n),\ 0 \le l \le v$ κωδικοποιούν το σύμβολο s_l, το οποίο περιέχεται στη θέση j τη χρονική στιγμή i. Δηλαδή η μεταβλητή $S[i,j,l]$ θα είναι True ανν τη χρονική στιγμή i, στη θέση j περιέχεται το σύμβολο s_l. Ο αριθμός αυτών των μεταβλητών είναι $(p(n)+1)\cdot(2p(n)+1)\cdot(v+1)$.

Ο αριθμός όλων των μεταβλητών που χρησιμοποιήσαμε είναι $O(p^2(n))$ (r, v είναι σταθερές της μηχανής), δηλαδή πολυωνυμικό ως προς το μέγεθος του input n.

Τα clauses που θα περιέχει η φόρμουλα $\Phi(x)$ είναι χωρισμένα σε 6 groups:

- G_1: Το group αυτό θα εκφράζει το γεγονός, ότι σε μία δεδομένη χρονική στιγμή η T.M. θα βρίσκεται ακριβώς σε μία κατάσταση. Δηλαδή μία δεδομένη χρονική στιγμή i, θα είναι True ακριβώς μία από τις μεταβλητές $Q[i,k], 0 \leq k \leq r$. Αυτό θα πρέπει να ισχύει για κάθε χρονική στιγμή. Το G_1 λοιπόν, θα λέει ότι μία δεδομένη χρονική στιγμή i, κάποια από τις μεταβλητές $Q[i,k]$ είναι True ενώ η σύζευξη οποιωνδήποτε δύο μεταβλητών απ' αυτές είναι False. Αυτό κωδικοποιείται εύκολα, όπως μπορεί να επαληθεύσει κανείς ως εξής:

$$\wedge_{i=0}^{p(n)}((\vee_{j=0}^{r}Q[i,j]) \wedge (\wedge_{j=0}^{r} \wedge_{k=0}^{j-1} (\neg Q[i,j] \vee \neg Q[i,k]))) \qquad (11.1)$$

  Ο πρώτος όρος της παραπάνω έκφρασης εξασφαλίζει ότι σε μία δεδομένη χρονική στιγμή η μηχανή βρίσκεται τουλάχιστον σε μία κατάσταση και ο δεύτερος όρος ότι βρίσκεται το πολύ σε μία κατάσταση. Ο ολικός αριθμός των literals που περιέχονται σ' αυτά τα clauses είναι:

$$(p(n)+1)\cdot[(r+1)+\frac{r(r+1)}{2}\cdot 2] = (p(n)+1)\cdot(r+1)^2 = O(p(n))$$

- G_2: Το group αυτό θα εκφράζει το γεγονός ότι σε μία δεδομένη χρονική στιγμή, η κεφαλή θα βρίσκεται σε μία ακριβώς θέση. Δηλαδή μία δεδομένη χρονική στιγμή i, θα είναι True ακριβώς μία από τις μεταβλητές $H[i,j], -p(n) \leq j \leq p(n)$ και αυτό θα πρέπει να ισχύει για κάθε χρονική στιγμή. Εντελώς ανάλογα λοιπόν, με το group G_1, το G_2 κωδικοποιείται ως εξής:

$$\wedge_{i=0}^{p(n)}((\vee_{j=-p(n)}^{p(n)}H[i,j])\wedge(\wedge_{j=-p(n)}^{p(n)}\wedge_{k=-p(n)}^{j-1}(\neg H[i,j]\vee\neg H[i,k]))) \quad (11.2)$$

  Ο ολικός αριθμός των literals που περιέχονται σ' αυτά τα clauses είναι,

$$(p(n)+1)\cdot(2p(n)+1)^2) = O(p^3(n))$$

- G_3: Το group αυτό θα εκφράζει το γεγονός ότι σε μία δεδομένη χρονική στιγμή, σε κάθε θέση στην ταινία περιέχεται ακριβώς ένα σύμβολο. Δηλαδή μία δεδομένη χρονική στιγμή i, για μία συγκεκριμένη θέση j, θα είναι True ακριβώς μία από τις μεταβλητές $S[i,j,l], 0 \leq l \leq v$ και αυτό θα πρέπει να ισχύει για κάθε χρονική στιγμή και για κάθε θέση. Εντελώς ανάλογα λοιπόν, με τα groups G_1 και G_2, το G_3 θα αποτελείται από τα εξής clauses:

$$\wedge_{i=0}^{p(n)}\wedge_{j=-p(n)}^{p(n)}((\vee_{l=0}^{v}S[i,j,l])\wedge(\wedge_{l=0}^{v}\wedge_{k=0}^{l-1}(\neg S[i,j,l]\vee\neg S[i,j,k]))) \quad (11.3)$$

  Συνολικά ο αριθμός των literals που περιέχονται στα clauses του G_3 είναι:

$$(p(n)+1)\cdot(2p(n)+1)\cdot(v+1)^2 = O(p^2(n))$$

- G_4: Το group αυτό θα δηλώνει ότι τη χρονική στιγμή 0 η T.M. βρίσκεται στο αρχικό configuration. Δηλαδή: Η κατάσταση στην οποία βρίσκεται η T.M. είναι q_0: $Q[0,0]$, η κεφαλή βρίσκεται στη θέση 1: $H[0,1]$, ότι στις θέσεις 1 έως n είναι γραμμένο το input x: $S[0,1,l_1] \wedge S[0,2,l_2] \wedge \ldots \wedge S[0,n,l_n]$, αν υποθέσουμε ότι $x = S_{l_1}, S_{l_2}, \ldots S_{l_n}$ και από τη θέση $n+1$ έως τη θέση $p(n)$ έχουμε κενά (*blanks*):

  $$S[0,n+1,0] \wedge S[0,n+2,0] \wedge \ldots \wedge S[0,p(n),0]$$

  Ακόμα, οι θέσεις $-p(n)$ έως 0 τη χρονική στιγμή 0 περιέχουν τον κενό χαρακτήρα ($S[0,0,0]$, κ.τ.λ.).

  Η σύζευξη όλων αυτών των clauses αποτελεί το group G_4. Ο συνολικός αριθμός των literals είναι $2p(n)+3 = O(p(n))$.

- G_5: Το group αυτό αποτελείται μόνο από ένα clause το οποίο έχει ένα litetal που δηλώνει, ότι τη χρονική στιγμή $p(n)$, η κατάσταση στην οποία βρίσκεται η T.M. είναι η $q_1 = q_Y$. Δηλαδή:

  $$Q[p(n),1]$$

- G_6: Το τελευταίο αυτό group θα δηλώνει πως το configuration της T.M. τη χρονική στιγμή $i+1$ προκύπτει από την εφαρμογή της συνάρτησης μετάβασης (transition function) δ, στο configuration της χρονικής στιγμής i. Κατ' αρχάς το G_6 θα δηλώνει πως το περιεχόμενο της θέσης j, δεν μπορεί να αλλάξει τη χρονική στιγμή $i+1$, αν τη χρονική στιγμή i η κεφαλή δεν βρισκόταν στη θέση j. Δηλαδή:

  $$(\neg H[i,j] \wedge S[i,j,l]) \rightarrow S[i+1,j,l],$$

  ή αλλιώς:

  $$\begin{cases} (H[i,j] \vee \neg S[i,j,l]) \vee S[i+1,j,l] \\ 0 \leq i \leq p(n), -p(n) \leq j \leq p(n), 0 \leq l \leq v \end{cases}$$

  Έχουμε δηλαδή $3(2p(n)+1) \cdot p(n) \cdot (v+1)$ literals. Επιπλέον το G_6 θα δηλώνει πως οι αλλαγές που γίνονται στο configuration τη χρονική στιγμή $i+1$ προκύπτουν από την εφαρμογή της συνάρτησης μετάβασης δ, στο configuration της χρονικής στιγμής i. Δηλαδή:

  $$(H[i,j] \wedge Q[i,k] \wedge S[i,j,l]) \rightarrow \\ \vee_{m=1}^{|\delta(q_k,s_l)|} (H[i+1,j+\Delta_m] \wedge Q[i+1,k'_m] \wedge S[i+1,j,l'_m]) \quad (11.4)$$

Αυτό σημαίνει πως αν τη χρονική στιγμή i η T.M. βρίσκεται στην κατάσταση q_k με την κεφαλή στη θέση j να διαβάζει το σύμβολο s_l, τότε τη χρονική στιγμή $i+1$, το περιεχόμενο της θέσης j, η κατάσταση και η θέση της κεφαλής θα πρέπει να ικανοποιούν την μη ντετερμινιστική συνάρτηση μετάβασης. δηλαδή, αν $q_k \in Q\{q_Y, q_N\}$ τότε οι τιμές των Δ_m, k'_m, l'_m είναι τέτοιες ώστε να ισχύει:

$$(q_{k'_m}, s_{l'_m}, \Delta_m) \in \delta(q_k, s_l), \text{ όπου} \Delta_m \in \{-1, 0, 1\}.$$

Την παραπάνω έκφραση μπορούμε να την φέρουμε σε CNF κάνοντας μερικές πράξεις και τελικά ο ολικός αριθμός των literals για δεδομένα i, j, k, l είναι $(c+3)3^c$, όπου $c = |\delta(q_k, s_l)|$. Αν $q_k \in \{q_Y, q_N\}$ τότε $\Delta = 0, k' = k, l' = l$ (δηλαδή αν η T.M. έχει ήδη αποδεχθεί ή απορρίψει το x πριν τη στιγμή $p(n)$, διατηρεί το configuration όπως είναι μέχρι τη στιγμή $p(n)$). Ο ολικός αριθμός των literals που περιέχονται στα clauses του G_6 είναι: $O(p^2(n))$.

Η τελική φόρμουλα λοιπόν, είναι:

$$\Phi(x) = G_1 \wedge G_2 \wedge G_3 \wedge G_4 \wedge G_5 \wedge G_6$$

Το μήκος της είναι: $O(p^3(n))$ (καθοριστικό είναι το μήκος του G_2). Συνεπώς μπορούμε να την κατασκευάσουμε σε πολυωνυμικό χρόνο ως προς n.

Επιπλέον, αν υπάρχει υπολογισμός του NDTM προγράμματος M που κάνει αποδεχτό το x σε χρόνο $p(n)$, τότε σίγουρα η $\Phi(x)$ έχει απονομή αλήθειας που την ικανοποιεί, λόγω της κατασκευής της.

Αντίστροφα, αν η $\Phi(x)$ έχει απονομή αλήθειας που την ικανοποιεί, τότε λόγω της κατασκευής της, αυτή η απονομή αλήθειας αντιστοιχεί σε έναν υπολογισμό του M που κάνει αποδεχτό το x. **Άρα το SAT είναι NP-complete**. □

Κεφάλαιο 12

Μετασχηματισμοί προβλημάτων (transformations of problems)

12.1 Γενικά

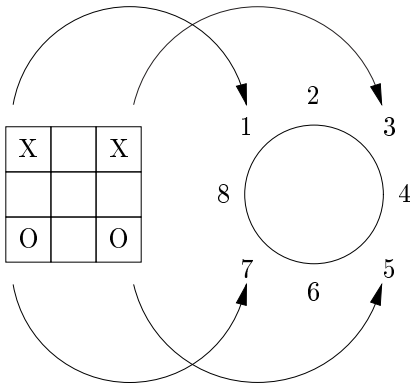
Πολλές φορές έχουμε να αντιμετωπίσουμε προβλήματα των οποίων η λύση, αν και απλή, δεν είναι προφανής. Τότε μετασχηματίζουμε το αρχικό πρόβλημα σε ένα άλλο πρόβλημα του οποίου μπορούμε εύκολα να βρούμε τη λύση. Έπειτα (με βάση το μετασχηματισμό) μεταφέρουμε τη λύση στο αρχικό μας πρόβλημα.

Παράδειγμα: Έστω το εξής πρόβλημα:
Δεδομένα: Μια σκακιέρα 3×3 με μαύρα (X) και λευκά (O) αλογάκια όπως φαίνεται στο σχήμα 12.1.
Ερώτηση: Πως μπορούμε να ανταλλάξουμε τις θέσεις των λευκών με τα μαύρα αλογάκια χρησιμοποιώντας νόμιμες κινήσεις;

X		X
O		O

Σχήμα 12.1: Πρόβλημα ίππων επί σκακιέρας

Λύση: Φτιάχνουμε έναν δακτύλιο όπως φαίνεται στο σχήμα 12.2. Τα μαύρα αλογάκια βρίσκονται στις θέσεις 1, 3 και τα λευκά στις θέσεις 5, 7. Η λύση του προβλήματος τώρα είναι προφανής. Με συνεχείς διαδοχικές κυκλικές μεταθέσεις, τα μαύρα αλογάκια θα ανταλλάξουν τελικά θέσεις με τα λευκά. Δηλαδή εκτελώντας μία περιστροφή κατά τη φορά των δεικτών του ρολογιού, το άλογο που βρίσκεται στη θέση 1 θα μετακινηθεί στη θέση 2, εκείνο που



Σχήμα 12.2: Μετασχηματισμός προβλήματος ίππων επί σκακιέρας

βρίσκεται στη θέση 3 θα πάει στη θέση 4, κ.ο.κ, έως ότου μετά από 4 μεταθέσεις καθενός αλόγου, τα μαύρα ανταλλάξουν θέσεις με τα λευκά.

Έτσι λοιπόν, αν ξαναγυρίσουμε στο αρχικό πρόβλημα, το μόνο που απομένει είναι να αριθμήσουμε τη σκακιέρα με βάση τις νόμιμες κινήσεις των αλόγων. Αυτό φαίνεται στο σχήμα 12.3. Δηλαδή η πορεία που θα ακολουθήσει το άλογο 1 θα είναι $1 \to 2 \to 3 \to 4 \to 5$ ενώ παράλληλα το άλογο 3 θα ακολουθεί την πορεία $3 \to 4 \to 5 \to 6 \to 7$, κ.ο.κ.

1	6	3
4		8
7	2	5

Σχήμα 12.3: Αρίθμηση σκακιέρας 3×3

Εδώ χρησιμοποιούμε τη μέθοδο της αναγωγής για να δείξουμε ότι ένα νέο πρόβλημα είναι δυσεπίλυτο: Ανάγουμε με εύκολο μετασχηματισμό ένα γνωστό δυσεπίλυτο πρόβλημα στο νέο μας πρόβλημα και έτσι δείχνουμε τη δυσεπιλυσιμότητά του (ειδάλλως αυτή η αναγωγή θα οδηγούσε σε εύκολη λύση του όμως γνωστού δυσεπίλυτου προβλήματος).

Για να δείξουμε ότι ένα πρόβλημα Π είναι NP-complete ακολουθούμε τα παρακάτω βήματα:

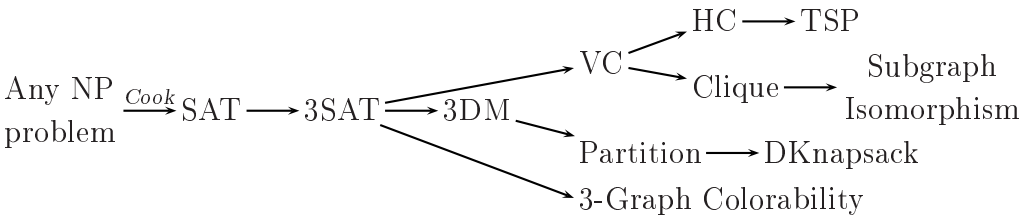
1. Δείχνουμε ότι $\Pi \in NP$.

2. Διαλέγουμε ένα γνωστό NP-complete πρόβλημα Π' και κατασκευάζοντας μια συνάρτηση f, το μετασχηματίζουμε στο πρόβλημα Π.

3. Δείχνουμε ότι ο μετασχηματισμός f γίνεται σε πολυωνυμικό χρόνο

4. Αποδεικνύουμε ότι $x \in \Pi' \iff f(x) \in \Pi$.

Οι αναγωγές προβλημάτων που θα δούμε στη συνέχεια, έγιναν με τη σειρά που φαίνεται στο σχήμα 12.4 και ιστορικά, οι περισσότερες από αυτές παρουσιάστηκαν από τον *Karp* (1972).



Σχήμα 12.4: Αναγωγές προβλημάτων στην κλάση NP

Πριν προχωρήσουμε στις αναγωγές, θα ορίσουμε όλα τα προβλήματα απόφασης που αναφέρονται στο διάγραμμα.

12.2 Ορισμοί Προβλημάτων Απόφασης

SAT (SATISFIABILITY)
Δεδομένα: Μία λογική έκφραση (boolean formula) σε κανονική συζευκτική μορφή (CNF).
Ερώτηση: Είναι η λογική έκφραση ικανοποιήσιμη; Δηλαδή, υπάρχει απονομή αλήθειας στις μεταβλητές τις τέτοια ώστε η boolean formula να αποτιμάται σε τιμή True.

3SAT
Δεδομένα: Μια boolean formula σε CNF, κάθε clause της οποίας έχει ακριβώς 3 literals.
Ερώτηση: Είναι η boolean formula ικανοποιήσιμη;

VC (VERTEX COVER)
Δεδομένα: Ένας γράφος $G(V, E)$ και ένας θετικός ακέραιος $k \leq |V|$.
Ερώτηση: Υπάρχει ένα vertex cover όλων των ακμών του E, μεγέθους $\leq k$; Δηλαδή, υπάρχει ένα σύνολο $V' \subseteq V$ τέτοιο ώστε $|V'| \leq k$ και $\forall \{u, v\} \in E : \ u \in V' \vee v \in V'$;

3DM (3-DIMENSIONAL MATCHING)
Δεδομένα: Ένα σύνολο $M \subseteq W \times X \times Y$, όπου W, X, Y είναι σύνολα ξένα μεταξύ τους (disjoint) με $|W| = |X| = |Y| = q$.
Ερώτηση: Περιέχει το M ένα ταίριασμα (matching); Δηλαδή, υπάρχει σύνολο $M' \subseteq M$ τέτοιο ώστε $|M'| = q$ και έτσι ώστε 2 οποιαδήποτε στοιχεία του M' να μην έχουν καμία κοινή συντεταγμένη;

GRAPH 3-COLORABILITY
Δεδομένα: Ένας γράφος $G(V, E)$.
Ερώτηση: Μπορούμε να βάψουμε τους κόμβους του γράφου G χρησιμοποιώντας **3 χρώματα** και έτσι ώστε 2 οποιοδήποτε γειτονικοί κόμβοι να έχουν διαφορετικό χρώμα; Δηλαδή, υπάρχει συνάρτηση $f : V \rightarrow \{1, 2, 3\}$ τέτοια ώστε, $\forall (u, v) \in E : f(u) \neq f(v)$;

HC (Hamilton Circuit)
Δεδομένα: Ένας γράφος $G(V, E)$.
Ερώτηση: Έχει ο γράφος κύκλο Hamilton; Δηλαδή, υπάρχει μία διάταξη των κόμβων του γράφου G, $< v_1, v_2, \ldots, v_n >, n = |V|$, τέτοια ώστε

$$(v_i, v_{i+1}) \in E, 1 \leq i \leq n-1, (v_n, v_1) \in E;$$

TSP (TRAVELING SALESMAN PROBLEM)
Δεδομένα: Δίνεται ένας πλήρης γράφος $G(V, E)$ με βάρη και ένας αριθμός B.
Ερώτηση: Υπάρχει μια κλειστή διαδρομή (tour) που να περνά απ' όλους τους κόμβους του G, $< v_{\pi_{(1)}}, v_{\pi_{(2)}}, \ldots, v_{\pi_{(m)}} >$ έτσι ώστε:

$$\sum w(v_{\pi_{(i)}}, v_{\pi_{(i+1)}}) \ + w(v_{\pi_{(m)}}, v_{\pi_{(1)}}) \leq B;$$

CLIQUE
Δεδομένα: Ένας γράφος $G(V, E)$ και ένας θετικός ακέραιος $j \leq |V|$.
Ερώτηση: Περιέχει ο γράφος G κλίκα μεγέθους $\geq j$; Δηλαδή υπάρχει $V' \subseteq V$, τέτοιο ώστε: $|V'| \geq j$ και $\forall u, v \in V' : (u, v) \in E$; Η ερώτηση μπορεί να γίνει ως εξής: Περιέχει ο γράφος G πλήρη υπογράφο με πλήθος κόμβων $\geq j$;

SUBGRAPH ISOMORPHISM
Δεδομένα: Δύο γράφοι $G(V_1, E_1)$ και $H(V_2, E_2)$.
Ερώτηση: Έχει ο γράφος G υπογράφο ισομορφικό με τον γράφο H; Δηλαδή, υπάρχουν $V \subseteq V_1, E \subseteq E_1$ τέτοια ώστε $|V| = |V_2|, |E| = |E_2|$ και συνάρτηση $f : V_2 \to V$, «1-1» και «επί» (bijection) ώστε να ισχύει, $(u, v) \in E_2 \iff (f(u), f(v)) \in E$;

PARTITION
Δεδομένα: Ένα πεπερασμένο σύνολο A με βάρη, $w(a) \in Z^+, \forall a \in A$.
Ερώτηση: Είναι δυνατόν το σύνολο A να μοιραστεί σε δύο ισοβαρή υποσύνολα; Δηλαδή, υπάρχει $A' \subseteq A$ τέτοιο ώστε,

$$\sum_{a \in A'} w(a) = \sum_{a \in (A - A')} w(a);$$

DKNAPSACK (DISCRETE KNAPSACK)
Δεδομένα: Ένα πεπερασμένο σύνολο U, μια συνάρτηση βάρους $w(u) \in Z^+, \forall u \in U$, μια συνάρτηση κόστους $p(u) \in Z^+, \forall u \in U$ και δύο θετικοί ακέραιοι W, P.
Ερώτηση: Μπορούμε να πάρουμε μερικά αντικείμενα από το σύνολο U και να τα βάλουμε μέσα σε ένα σακίδιο, έτσι ώστε το ολικό βάρος του σακιδίου να είναι $\leq W$ και η ολική του αξία $\geq P$; Δηλαδή υπάρχει $U' \subseteq U$ τέτοιο ώστε,

$$\sum_{u \in U'} w(u) \leq W \text{ και } \sum_{u \in U'} p(u) \geq P;$$

Από τα παραπάνω προβλήματα έχουμε ήδη αποδείξει πως το SAT είναι NP-complete.

12.3 Αναγωγή του SAT στο 3SAT

Θεώρημα 12.3.1. *Το 3SAT είναι* **NP-complete**.

Proof. Κατ' αρχάς είναι εύκολο να δούμε ότι το 3SAT $\in NP$. Πράγματι, ένας μη-ντετερμινιστικός αλγόριθμος, αφού μαντέψει μια απονομή αλήθειας, μπορεί πάντα να ελέγξει σε πολυωνυμικό χρόνο, αν αυτή ικανοποιεί τη boolean formula που δίνεται (το 3SAT είναι ένα υποπρόβλημα του SAT).

Για να αποδείξουμε ότι το 3SAT είναι NP-complete θα ανάγουμε το SAT σ' αυτό (SAT $\leq_m^p$ 3SAT). Έστω ότι μας δίνεται ένα οποιοδήποτε στιγμιότυπο του SAT δηλαδή ένα σύνολο C από m clauses, $C = \{c_1, c_2, \ldots, c_m\}$ που χρησιμοποιούν μεταβλητές από ένα σύνολο από n μεταβλητές, $U = \{z_1, z_2, \ldots, z_n\}$. Θα κατασκευάσουμε ένα καινούριο σύνολο από clauses C' και ένα καινούργιο σύνολο μεταβλητών V', έτσι ώστε κάθε clause που ανήκε στο C' να αποτελείται από 3 ακριβώς literals. Η κατασκευή γίνεται ως εξής:

- Για κάθε clause $c \in C$ της αρχικής φόρμουλας που αποτελείται από 1 literal $c = z$, κατασκευάζουμε τα εξής 4 clauses (φυσικά οι μεταβλητές y_1 και y_2 είναι νέες, δεν περιέχονται στην C):

  $$(z \vee y_1 \vee y_2) \wedge (z \vee y_1 \vee \neg y_2) \wedge (z \vee \neg y_1 \vee y_2) \wedge (z \vee \neg y_1 \vee \neg y_2)$$

  Είναι εύκολο να δούμε ότι η τιμή της παραπάνω έκφρασης, είναι πάντα ίδια με την τιμή του z. Ανεξάρτητη, δηλαδή, από τις τιμές των y_1, y_2 (*dummy variables*).

- Για κάθε clause $c \in C$ της αρχική φόρμουλας που αποτελείται από 2 literals $c = z_1 \vee z_2$, κατασκευάζουμε τα παρακάτω 2 clauses (νέα μεταβλητή y_1):

  $$(z_1 \vee z_2 \vee y_1) \wedge (z_1 \vee z_1 \vee \neg y_1)$$

  Και εδώ είναι εύκολο να δούμε ότι η τιμή της παραπάνω παράστασης είναι πάντα ίδια με την τιμή της έκφρασης $(z_1 \vee z_2)$.

- Κάθε clause της αρχικής φόρμουλας που αποτελείται από 3 literals το παίρνουμε όπως είναι στην καινούργια μας φόρμουλα.

- Τέλος, για κάθε clause της αρχικής φόρμουλας που έχει παραπάνω από 3 literals, έστω $c = (z_1 \vee z_2 \vee \ldots z_k)$, κατασκευάζουμε τα εξής clauses (y_i νέες μεταβλητές):

  $$(z_1 \vee z_2 \vee y_1) \wedge (\neg y_1 \vee z_3 \vee y_2) \wedge (\neg y_2 \vee z_4 \vee y_3) \wedge \ldots \\ \wedge (\neg y_{k-4} \vee z_{k-2} \vee y_{k-3}) \wedge (\neg y_{k-3} \vee z_{k-1} \vee z_k)$$

Οι y_i είναι dummy μεταβλητές, ενώ ο αριθμοί των καινούργιων clauses είναι $k-2$. Η κατασκευή της καινούργιας φόρμουλας Φ', η οποία είναι στιγμιότυπο του 3SAT έχει τελειώσει.

Αν το μήκος της αρχικής φόρμουλας Φ (μέγεθος του input) είναι $n \cdot m$ (m clauses με n το πολύ literals το καθένα), τότε το μήκος της φόρμουλας Φ', θα είναι $3m(n-2)$, δηλαδή $m(n-2)$ clauses με 3 literals το καθένα. Δηλαδή το μήκος Φ' θα είναι $O(m \cdot n)$, πολυωνυμικό ως προς το μήκος της Φ και

συνεπώς ο μετασχηματισμός γίνεται σε πολυωνυμικό χρόνο. Αυτό που μένει να αποδείξουμε λοιπόν, είναι ότι:«η αρχική φόρμουλα Φ έχει απονομή αλήθειας που την ικανοποιεί», ανν η νέα φόρμουλα Φ' έχει απονομή αληθείας που την ικανοποιεί.

Όπως είπαμε κατά την διάρκεια της κατασκευής, αν ένα clause της αρχικής φόρμουλας με 1, 2 ή 3 literals ικανοποιείται, τότε θα ικανοποιούνται και τα αντίστοιχα clauses της Φ' και αντίστροφα. Αν έχουμε ένα clause στην αρχική φόρμουλα , $c = (z_1 \vee z_2 \vee \ldots \vee z_k)$, με $k \geq 4$ τότε για να ικανοποιείται το c θα πρέπει να έχει τιμή True τουλάχιστον ένα από τα literals του. Έστω λοιπόν $t(z_l) = True$. Θα δώσουμε μια απονομή αλήθειας που ικανοποιεί τα αντίστοιχα clauses στη φόρμουλα Φ':

$$t(y_i) = \begin{cases} \text{True}, & 1 \leq i \leq l-2 \\ \text{False}, & l-1 \leq i \leq k-3 \end{cases}$$

Και εδώ τα αντίστοιχα clauses της Φ' ικανοποιούνται (αυτά που βρίσκονται πριν από το clause i, από τα literals y_i και εκείνα που βρίσκονται μετά τα clause i, από τα literals $\neg y_i$).

Η απόδειξη του αντιστρόφου, ότι δηλαδή αν ικανοποιούνται τα clauses της Φ' τότε ικανοποιείται το αντίστοιχο clause της Φ γίνεται ως εξής: Έστω ότι δεν ικανοποιείται η $\Phi(x)$, δηλαδή $\forall i \; z_i = false$. Τότε μπορούμε εύκολα να δείξουμε (επαγωγή) ότι για να ικανοποιούνται τα n πρώτα clauses της $\Phi'(x)$ πρέπει οι μεταβλητές $y_1, y_2, \ldots, y_n$ να παίρνουν την αληθοτιμή true. Λόγω του παραπάνω για $n = k-3$ το τελευταίο clause παίρνει την τιμή false. □

Παρατήρηση 12.3.2. Το πρόβλημα 2SAT ανήκει στο P. Θα μπορούσαμε να μετασχηματίσουμε το SAT, ή το ισοδύναμο του πλέον 3SAT στο 2SAT. Όμως ο αριθμός των clauses της καινούριας φόρμουλας που θα προκύψει, θα είναι εκθετικός ως προς την αρχική φόρμουλα.

Ένα άλλο υποπρόβλημα του SAT, το οποίο ανήκει στο P, είναι το *HORN-SAT*. Η δεδομένη boolean formula αποτελείται από *Horn clauses*. *Horn clause* ονομάζεται ένα clause το οποίο έχει το πολύ ένα θετικό literal. Δηλαδή, όλα τα literals εκτός πιθανόν από ένα, είναι αρνητικά, π.χ.:

$$(\neg x_2 \vee x_3), \quad (\neg x_1 \vee \neg x_2 \vee \neg x_3), \quad (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)$$

Τα clauses που έχουν ακριβώς ένα θετικό literal (όπως το πρώτο και το τελευταίο παράδειγμα), ονομάζονται *implications* διότι μπορούν να γραφτούν ως εξής: $(\neg x_2 \vee x_3) = (x_2 \rightarrow x_3)$, $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) = ((x_1 \wedge x_2 \wedge x_3) \rightarrow x_4)$. Ο αλγόριθμος που λύνει το **HORNSAT** βασίζεται σ' αυτήν ακριβώς την *implicational* μορφή των clauses.

12.4 Αναγωγή του 3SAT σε άλλα προβλήματα

Γενικά, όταν ανάγουμε το 3SAT σε κάποιο άλλο πρόβλημα, τα δεδομένα μας είναι ένα σύνολο μεταβλητών $u_1, \ldots, u_n$ και ένα σύνολο από clauses $c_1, \ldots, c_m$. Τα στοιχεία που συνθέτουν την αναγωγή μας είναι:

- *Truthsetting*: Εξασφαλίζουμε ότι κάθε μεταβλητή έχει μία και μοναδική αληθοτιμή (truth value) σε όλα τα clauses.
- *Satisfaction*: Εξασφαλίζουμε ότι κάθε clause περιέχει τουλάχιστον ένα literal που ικανοποιείται (έχει τιμή True).
- *Remaining (interconnections)-garbage collection*: Εξασφαλίζουμε ότι έχουμε ένα σωστό πρόβλημα του καινούργιου τύπου.

12.5 Η αναγωγή του 3SAT στο VERTEX COVER

Θεώρημα 12.5.1. *Το πρόβλημα VERTEX COVER (VC) είναι NP-complete.*

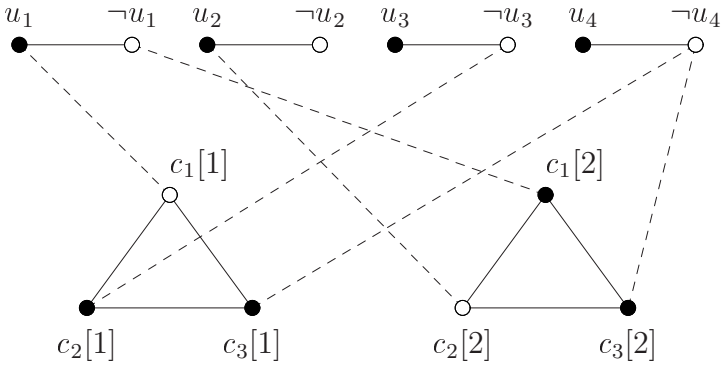
Proof. Ισχύει $VC \in NP$, διότι ένας μη-ντετερμινιστικός αλγόριθμος μαντεύει ένα σύνολο κόμβων V' και μετά ελέγχει αν το V' περιέχει τουλάχιστον ένα endpoind από κάθε πλευρά και αν $|V'| \leq k$ σε πολυωνυμικό χρόνο. Θα δείξουμε ότι το VC είναι NP-complete ανάγοντας το 3SAT σ' αυτό ($3SAT \leq_m^p VC$).

Μας δίνεται ένα σύνολο από clauses $C = \{c_1, \ldots, c_m\}$ που περιέχει μεταβλητές από ένα σύνολο μεταβλητών $U = \{u_1, \ldots, u_n\}$. Θα κατασκευάσουμε έναν γράφο $G(V, E)$ και έναν θετικό ακέραιο $k \leq |V|$ τέτοια ώστε ο G να έχει vertex cover μεγέθους $\leq k$ αν και μόνο αν η φόρμουλα $(c_1 \wedge c_2 \wedge \ldots \wedge c_m)$ είναι ικανοποιήσιμη. Η κατασκευή γίνεται ως εξής:

- Για κάθε μεταβλητή $u_i \in U$ εισάγουμε 2 κόμβους στο V, τους $u_i, \neg u_i$ και 1 πλευρά στο E, $(u_i, \neg u_i)$. Άρα συνολικά παίρνουμε $2n$ κόμβους και n πλευρές. Αυτό είναι το στάδιο truthsetting.
- Για κάθε clause $c_i \in C$ εισάγουμε 3 κόμβους στο V τους $c_1[i], c_2[i], c_3[i]$ και 3 πλευρές στο E, τις $(c_1[i], c_2[i]), (c_2[i], c_3[i])(c_3[i], c_1[i])$. Άρα συνολικά έχουμε $3m$ καινούργιους κόμβους και $3m$ πλευρές(αυτό είναι το στάδιο satisfaction).
- Τέλος προσθέτουμε τις πλευρές που χρειάζονται ώστε κάθε (satisfaction) τρίγωνο να συνδέεται με τρία αντίστοιχα (truthsetting) literals και το

καινούριο πρόβλημα να είναι το VERTEX COVER (remaining interconnections). Για κάθε κόμβο $c_k[i], 1 \leq k \leq 3$ προσθέτουμε την πλευρά $(c_k[i], u_j)$ ή $(c_k[i], \neg u_j)$ ανάλογα με το αν στην $k-$οστή θέση του clause c_i εμφανίζεται το literal u_j ή $\neg u_j$ αντίστοιχα. Οι καινούριες πλευρές λοιπόν, είναι $3m$ (3 για κάθε clause). Η κατασκευή του γράφου έχει τελειώσει.

Το πλήθος των κόμβων του γράφου είναι $|V| = 2n + 3m$, ενώ το πλήθος των πλευρών είναι $|E| = n + 6m$. Το μέγεθος του γράφου είναι πολυωνυμικό ως προς το μέγεθος της φόρμουλας $(3m)$ και η κατασκευή μπορεί να γίνει σε πολυωνυμικό χρόνο. Ορίζουμε $k = n + 2m$. Υποστηρίζουμε πως η φόρμουλα του προβλήματος 3SAT ικανοποιείται αν και μόνο αν υπάρχει vertex cover βαθμού $\leq k$ στο γράφο $G(V, E)$. Πριν προχωρήσουμε στην απόδειξη του ισχυρισμού, δίνουμε ένα παράδειγμα.



Σχήμα 12.5: Αναγωγή 3SAT στο VERTEX COVER

Παράδειγμα: Έστω ότι έχουμε τη φόρμουλα

$$\Phi\colon (u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$

Η Φ είναι ικανοποιήσιμη. Μια απονομή αλήθειας που την ικανοποιεί είναι η $t(u_1, u_2, u_3, u_4) = (T, T, T, T)$. Συνεπώς ο γράφος $G(V, E)$ που προκύπτει απ' αυτήν τη φόρμουλα και φαίνεται στο σχήμα 12.5, πρέπει να έχει vertex cover μεγέθους $\leq n + 2m = 4 + 2 \cdot 2 = 8$.

Οι 8 κόμβοι που αποτελούν ένα vertex cover του γράφου είναι οι κόμβοι που σημειώνονται με κουκκίδες.

Απόδειξη του ισχυρισμού: Έστω ότι η φόρμουλα Φ είναι ικανοποιήσιμη και έστω t μια απονομή αληθείας που ικανοποιεί την Φ, τότε βρίσκουμε ένα vertex cover ως εξής: Παίρνουμε εκείνους τους κόμβους που αντιστοιχούν σε literals οι οποίοι έχουν τιμή True. Αυτοί οι κόμβοι θα είναι ακριβώς n, αφού θα

ισχύει $t(u_i) = True$, είτε $t(\neg u_i) = True$ αλλά όχι συγχρόνως (truthsetting). Συνεπώς αυτοί οι κόμβοι θα καλύπτουν $n + m$ πλευρές τουλάχιστον, αφού κάθε clause της αρχικής φόρμουλας Φ, θα έχει τουλάχιστον ένα literal που το ικανοποιεί (satisfaction). Στη χειρότερη περίπτωση λοιπόν, δηλαδή όταν κάθε clause ικανοποιείται από ένα ακριβώς literal, έχουν μείνει $5m$ πλευρές ακάλυπτες. Τις καλύπτουμε παίρνοντας 2 κόμβους από κάθε τριγωνάκι (συνολικά $2m$ κόμβους), εκείνους που συνδέονται με literals που έχουν τιμή False. Γενικώς, επειδή κάθε clause ικανοποιείται από τουλάχιστον ένα literal, τουλάχιστον μια σύνδεση (satisfaction) τρίγωνο με truthsetting είναι ήδη καλυμμένη. Οπότε παίρνοντας τις 2 άλλες κορυφές του τριγώνου καλύπτουμε σίγουρα όλες τις υπόλοιπες πλευρές. Έτσι καλύπτονται όλες οι πλευρές του γράφου με $n+2m$ κόμβους.

Αντίστροφα, αν ο G έχει vertex cover $V' \sqsubseteq V$ μεγέθους $|V'| \leq k = n+2m$, το V' περιλαμβάνει τουλάχιστον έναν κόμβο από κάθε πλευρά (truthsetting) της μορφής $u_i \ \neg u_i$ (ειδάλλως θα υπάρχει πλευρά αυτής της μορφής που δεν θα καλύπτεται) και τουλάχιστον δύο κόμβους από κάθε τρίγωνο (ειδάλλως δεν θα καλύπτονται όλες οι πλευρές του τριγώνου). Δηλαδή το V' θα περιέχει τουλάχιστον $n+2m$ κόμβους. Συνεπώς (αφού από την υπόθεση $V' \leq n+2m$, το V' θα περιέχει ακριβώς 1 κόμβο από κάθε πλευρά της μορφής $u_i \ \neg u_i$ και ακριβώς 2 κόμβους από κάθε τρίγωνο. Θα δώσουμε μια απονομή αλήθειας που ικανοποιεί την Φ: $t(u_i) = \begin{cases} \text{True}, & u_i \in V' \\ \text{False}, & \neg u_i \in V'. \end{cases}$

Οι δύο κόμβοι που έχουμε πάρει από κάθε τρίγωνο μπορούν να καλύπτουν μόνο δύο από τις πλευρές της μορφής $(c_k[i], u_j)$. Η τρίτη πλευρά πρέπει υποχρεωτικά να καλύπτεται από κάποιο u_i ή $\neg u_i$ που ανήκει στο V'. Δηλαδή, κάθε τριγωνάκι συνδέεται με κάποιο από τα u_i ή $\neg u_i$ που ανήκουν στο V'. Άρα κάθε clause της φόρμουλας Φ, έχει ένα literal u_i ή $\neg u_i$ με $t(u_i)$ ή $t(\neg u_i)$ True. Συνεπώς η Φ ικανοποιείται. □

12.6 Η αναγωγή του 3SAT στο 3-DIMENSIONAL MATCHING

Θεώρημα 12.6.1. *Το πρόβλημα 3-DIMENSIONAL MATCHING είναι NP-complete.*

Proof. Ισχύει 3DM $\in NP$, διότι ένας μη-ντετερμινιστικός αλγόριθμος μαντεύει ένα σύνολο $M' \subseteq M = W \times X \times Y$ από q τριάδες και μετά ελέγχει σε πολυωνυμικό χρόνο ότι ανά 2 δεν έχουν κοινές συντεταγμένες. Θα δείξουμε ότι το 3DM είναι NP-complete ανάγοντας το 3SAT σ' αυτό (3SAT $\leq_m^p$ 3DM).

Μας δίνεται ένα σύνολο μεταβλητών $U = \{u_1, \ldots, u_n\}$ και ένα σύνολο από clauses $C = \{c_1, \ldots, c_m\}$. Θα κατασκευάσουμε 3 ξένα μεταξύ τους σύνολα W, X, Y με $|W| = |X| = |Y|$ και ένα σύνολο $M \subseteq W \times X \times Y$ έτσι ώστε το M να έχει matching αν και μόνο αν η φόρμουλα Φ: $c_1 \wedge c_2 \wedge \ldots \wedge c_m$ είναι ικανοποιήσιμη. Η κατασκευή γίνεται ως εξής:

Βάζουμε στο W όλα τα στοιχεία $u_i[j], \neg u_i[j], 1 \leq j \leq m, 1 \leq i \leq n$ ($2mn$ το πλήθος). Στο σύνολο X βάζουμε τα στοιχεία:

- $a_i[j], 1 \leq j \leq m,\ 1 \leq i \leq n$, ($nm$ το πλήθος, τα οποία θα χρησιμοποιηθούν στο truthsetting)
- $S_1[j], 1 \leq j \leq m$, (m το πλήθος, τα οποία θα χρησιμοποιηθούν στο satisfaction)
- $g_1[k], 1 \leq k \leq (n-1)m$ ($(n-1)m$ το πλήθος, τα οποία θα χρησιμοποιηθούν στο garbage collection).

Όμοια με το σύνολο X φτιάχνουμε και το σύνολο Y βάζοντας τα στοιχεία:

- $b_i[j], 1 \leq j \leq m,\ 1 \leq i \leq n$.
- $S_2[j], 1 \leq j \leq m$.
- $g_2[k], 1 \leq k \leq (n-1)m$.

Το σύνολο $M \subseteq W \times X \times Y$ το κατασκευάζουμε ως εξής: Για κάθε μεταβλητή u_i βάζουμε στο M τα σύνολα τριάδων T_i^t και T_i^f όπου:

$$T_i^t = \{(\neg u_i[j], a_i[j], b_i[j]), 1 \leq j \leq m\}$$
$$T_i^f = \{(u_i[j], a_i[j+1], b_i[j]), 1 \leq j \leq m\} \cup \{(u_i[m], a_i[1], b_i[m])\}$$

Όλες αυτές οι τριάδες είναι $2nm$ στο πλήθος και χρησιμοποιούνται για το truthsetting. Για κάθε clause c_j βάζουμε στο M το σύνολο τριάδων C_j όπου:

$$C_j = \{(u_i[j], s_1[j], s_2[j]) \mid u_i \in c_j \text{ clause}\} \cup \{(\neg u_i[j], s_1[j], s_2[j]) \mid \neg u_i \in c_j \text{ clause}\}, 1 \leq j \leq m\}$$

Συνολικά είναι $3m$ τριάδες και χρησιμοποιούνται για το satisfaction.

Τέλος βάζουμε στο M τα στοιχεία που λείπουν για να ανήκει το στιγμιότυπό μας στο πρόβλημα 3DM (garbage collection). Βάζουμε δηλαδή το σύνολο τριάδων G όπου:

$$G = \{(u_i[j], g_1[k], g_2[k]),\ (\neg u_i[j], g_1[k], g_2[k])\},\ 1 \leq k \leq m(n-1),\ 1 \leq i \leq n, 1 \leq j \leq m\}$$

Δηλαδή το G αποτελείται συνολικά από $2nm^2(n-1)$ στοιχεία.

Ένα οποιοδήποτε matching $M' \subseteq M$ θα πρέπει να περιλαμβάνει τουλάχιστον nm τριάδες από τα T_i^f, T_i^t ώστε να καλύψουμε όλα τα $a_i[j]$ και $b_i[j]$, τα οποία εμφανίζονται μόνο σε τριάδες στα T_i^f, T_i^t. Επίσης, επειδή υπάρχουν ακριβώς mn διαφορετικά $a_i[j]$ προκύπτει ότι το M' περιλαμβάνει ακριβώς mn τριάδες από τα T_i^f, T_i^t. Μάλιστα το M' θα περιλαμβάνει για κάθε u_i, είτε ολόκληρο το T_i^f είτε ολόκληρο το T_i^t. Επίσης το matching M' προτείνει μια απονομή αλήθειας που ικανοποιεί τη φόρμουλα Φ:

$$t(u_i) = \begin{cases} \text{True}, & T_i^t \subseteq M' \\ \text{False}, & T_i^f \subseteq M' \end{cases}$$

Ένα οποιοδήποτε matching M' θα πρέπει να περιέχει για κάθε clause μία ακριβώς τριάδα (σύνολο m τριάδες). Το literal $u_i[j]$ (ή $\neg u_i[j]$) που θα εμφανίζεται στην τριάδα C_j στο matching M' θα είναι ακριβώς εκείνο που ικανοποιεί το clause c_j, αφού για να υπάρχει στο M', σημαίνει ότι $T_i^t \subseteq M' (T_i^f \subseteq M')$.

Το matching M' έχει ως τώρα $nm + m$ τριάδες και συνεπώς του λείπουν $2mn - nm - m = m(n-1)$ τριάδες. Αυτές θα μπορεί πάντα να τις πάρει από το G, αφού θα υπάρχουν $m(n-1)$ literals που δεν έχει ως τώρα το M'. Η κατασκευή έχει τελειώσει και συνολικά το πλήθος των τριάδων που περιέχει το M είναι $|M| = 2nm + 3m + 2m^2n(n-1)$. Συνεπώς και εφόσον ο τρόπος κατασκευής του από το 3SAT είναι σχεδόν άμεσος, το M μπορεί να κατασκευαστεί σε πολυωνυμικό χρόνο.

Ήδη αποδείξαμε πως αν το M περιέχει ένα matching, τότε η φόρμουλα Φ είναι ικανοποιήσιμη. Αν η φόρμουλα Φ είναι ικανοποιήσιμη, τότε θα κατασκευάσουμε ένα matching M' ως εξής:

- Διαλέγουμε όλα τα σύνολα T_i^t για τα οποία $t(u_i) = True$ και όλα τα σύνολα T_i^f για τα οποία $t(u_i) = False$. Δηλαδή, όλα τα literals που θα ανήκουν σ' αυτές τις τριάδες θα έχουν τιμή False και θα είναι ακριβώς mn.

- Επίσης διαλέγουμε m τριάδες της μορφής

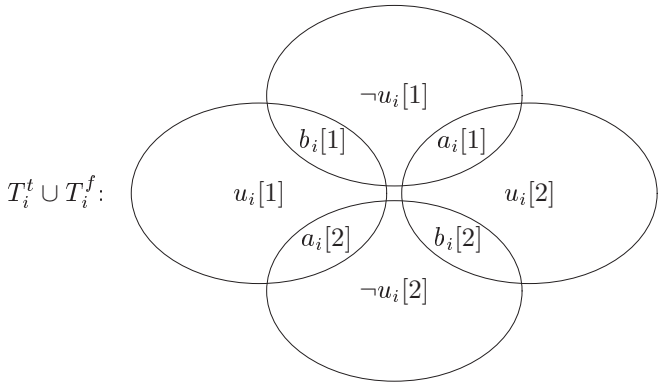
  $$(u_i[j], s_1[j], s_2[j]) \text{ ή } (\neg u_i[j], s_1[j], s_2[j])$$

  οι οποίες προφανώς θα περιέχουν literals με τιμή True (ειδάλλως θα καταστρέφεται το matching). Θα υπάρχουν πάντα m τέτοιες τριάδες, αφού κάθε clause θα έχει ένα literal που το ικανοποιεί.

- Τέλος παίρνουμε ένα σύνολο $G' \subseteq G$ με $m(n-1)$ τριάδες οι οποίες θα περιέχουν literals με τιμή True. Ξέρουμε ότι υπάρχουν $m(n-1)$ τέτοιες

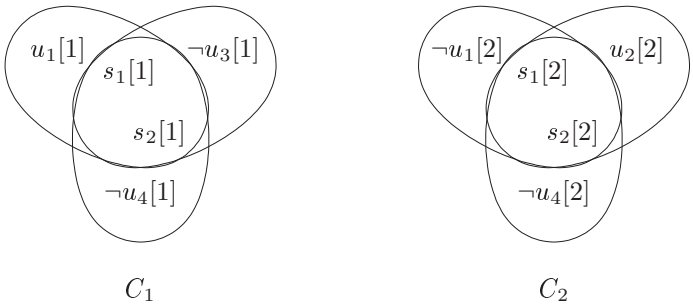
τριάδες αφού έχουμε πάρει μόνο m με literals που έχουν τιμή True ως τώρα. Συνεπώς το M' περιέχει $2nm$ διαφορετικές τριάδες και άρα είναι ένα ταίριασμα του M.

□



Σχήμα 12.6: Αναγωγή του 3SAT στο 3DM: Truthsetting

Παράδειγμα 12.6.2. Έχουμε τη φόρμουλα Φ: $(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$ η οποία είναι ικανοποιήσιμη. Μία απονομή αλήθειας που την ικανοποιεί είναι η $t(u_1, u_2, u_3, u_4) = (T, T, F, T)$. Για κάθε μεταβλητή u_i, το M θα περιλαμβάνει το σύνολο που φαίνεται στο σχήμα 12.6. Επίσης το M θα περιλαμβάνει τα σύνολα C_1 και C_2 που φαίνονται στο σχήμα 12.7. Τέλος το M θα περιλαμβάνει το σύνολο G με τα στοιχεία που απομένουν. Ένα



Σχήμα 12.7: Αναγωγή του 3SAT στο 3DM: Satisfaction

matching $M' \subseteq M$ που προκύπτει από την απονομή (T, T, F, T) είναι το εξής:

$$\begin{aligned} M' = \; & \{\neg u_1[1], a_1[1], b_1[1]), \\ & (\neg u_1[2], a_1[2], b_1[2]), \\ & (\neg u_2[1], a_2[1], b_2[1]), \\ & (\neg u_2[2], a_2[2], b_2[2]), \\ & (u_3[1], a_3[1], b_3[1]), \\ & (u_3[2], a_3[2], b_3[2]), \\ & (\neg u_4[1], a_4[1], b_4[1]), \\ & (\neg u_4[2], a_4[2], b_4[2])\} \bigcup T' \bigcup G' \end{aligned}$$

όπου

$$T' = \; \{(u_1[1], s_1[1], s_2[1]), (u_2[2], s_1[2], s_2[2])\}$$

και

$$\begin{aligned} G' = \; & \{(u_1[2], g_1[1], g_2[1]), (u_2[1], g_1[2], g_2[2]), \\ & (\neg u_3[1], g_1[3], g_2[3]), (\neg u_3[2], g_1[4], g_2[4]), \\ & (u_4[1], g_1[5], g_2[5]), (u_4[2], g_1[6], g_2[6])\} \end{aligned}$$

Παρατήρηση 12.6.3. Το πρόβλημα 2-DIMENSIONAL MATCHING ανήκει στο P. Δηλαδή, όταν έχουμε 2 σύνολα n στοιχείων και θέλουμε να βρούμε ένα ταίριασμα, μπορούμε να το κάνουμε σε πολυωνυμικό χρόνο με έναν ντετερμινιστικό αλγόριθμο. Το πρόβλημα $2DM$ αναφέρεται και σαν πρόβλημα γάμου (*marriage problem*) λόγω των προφανών προεκτάσεων που μπορεί να έχει στη ζωή (το ένα σύνολο περιέχει n άντρες και το άλλο n γυναίκες). Επίσης, άλλα παρόμοια προβλήματα είναι τα εξής:

- Μπορούν να τοποθετηθούν 8 βασίλισσες (queens) σε μια σκακιέρα (8×8) έτσι ώστε καμιά βασίλισσα να μην απειλεί κάποια άλλη;
- Μπορούν να τοποθετηθούν 8 πύργοι σε μια σκακιέρα (8×8) έτσι ώστε κανένας πύργος να μην απειλεί κάποιον άλλο; (εύκολο)
- Μπορούν να τοποθετηθούν 8 πύργοι όπως και πριν, σε ένα δεδομένο υποσύνολο της σκακιέρας; (δύσκολο, αλλά στο P: είναι το πρόβλημα γάμου)

12.7 Η αναγωγή του 3SAT στο GRAPH 3-COLORABILITY

Θεώρημα 12.7.1. *Το πρόβλημα GRAPH 3-COLORABILITY είναι NP-complete.*

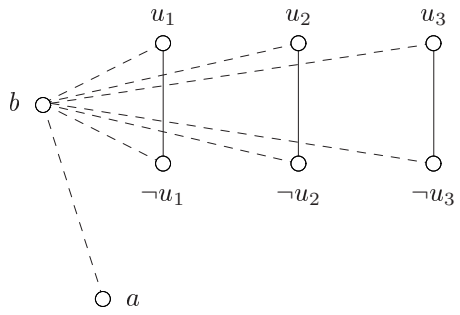
Proof. Το GRAPH 3-COLORABILITY ανήκει στο NP, αφού ένας μη-ντετερμινιστικός αλγόριθμος μπορεί να ελέγξει σε πολυωνυμικό χρόνο, αν η f που μάντεψε (η απονομή των χρωμάτων στους κόμβους δηλαδή), ικανοποιεί τη συνθήκη $f(u) \neq f(v), \forall (u,v) \in E$.

Θα δείξουμε ότι το GRAPH 3-COLORABILITY είναι NP-complete ανάγοντας το 3SAT σ' αυτό ($3SAT \leq_m^p GRAPH3-COLORABILITY$).

Μας δίνεται ένα σύνολο μεταβλητών $U = \{u_1, \ldots, u_n\}$ και ένα σύνολο από clauses $C = \{c_1, \ldots, c_m\}$.

Θα κατασκευάσουμε έναν γράφο $G(V,E)$ έτσι ώστε η φόρμουλα Φ: $(c_1 \wedge c_2 \wedge \ldots \wedge c_m)$ να είναι ικανοποιήσιμη αν και μόνο αν ο γράφος G βάφεται με 3 χρώματα. Η κατασκευή του G γίνεται ως εξής:

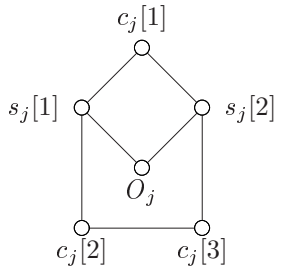
- Για κάθε μεταβλητή $u_i \in U$ βάζουμε στο V τους κόμβους $u_i, \neg u_i$ και στο E τις πλευρές $(u_i, \neg u_i)$. Επίσης βάζουμε στο V τους κόμβους a, b και στο E τις πλευρές $(a,b), (u_i,b), (\neg u_i, b)$. Δηλαδή ως τώρα $|V| = 2n+2, |E| = 3n+1$. Ο γράφος G παίρνει τη μορφή που φαίνεται στο σχήμα 12.8. Αυτό είναι το στάδιο truthsetting.



Σχήμα 12.8: Αναγωγή του 3SAT στο 3-Colorability: Truthsetting

- Για κάθε clause c_j προσθέτουμε στο γράφο G, έναν υπογράφο σαν αυτόν που φαίνεται στο σχήμα 12.9. Επίσης βάζουμε στο E τις πλευρές $(c_j[k], u_i)$ ή $(c_j[k], \neg u_i)$ ανάλογα με το αν το k-οστό literal του c_j είναι το u_i ή $\neg u_i$ αντίστοιχα. Δηλαδή έχουμε $6m$ καινούργιους κόμβους για το V και $10m$ καινούριες πλευρές για το E. Αυτό είναι το στάδιο satisfaction.

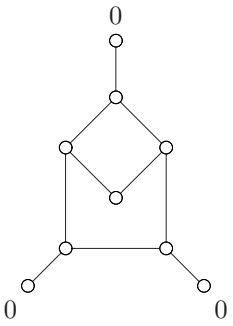
- Τέλος κάνουμε τις συνδέσεις που απομένουν (remaining interconnections) για να είναι το πρόβλημά μας, του σωστού τύπου. Προσθέτουμε, δηλαδή, τις πλευρές (O_j, b) και τις πλευρές (O_j, a) ($2m$ στο πλήθος). Η κατασκευή μας έχει τελειώσει.



Σχήμα 12.9: Αναγωγή του 3SAT στο 3 Colorability: Satisfaction

Έχουμε συνολικά $|V| = 2n + 6m + 2, |E| = 3n + 12m + 1$ συνεπώς ο μετασχηματισμός γίνεται σε πολυωνυμικό χρόνο ως προς το μέγεθος της φόρμουλας Φ. Μένει, λοιπόν να δείξουμε ότι η Φ είναι ικανοποιήσιμη τότε και μόνο τότε, όταν ο $G(V, E)$ βάφεται με 3 χρώματα. Έστω ότι η Φ ικανοποιείται. Βάφουμε τον κόμβο b με το χρώμα 2 και τον κόμβο a με το χρώμα 1. Συνεπώς, όλες οι κορυφές O_j πρέπει να βαφούν με το χρώμα 0. Επίσης βάφουμε τους κόμβους $u_i, \neg u_i$ ως εξής:

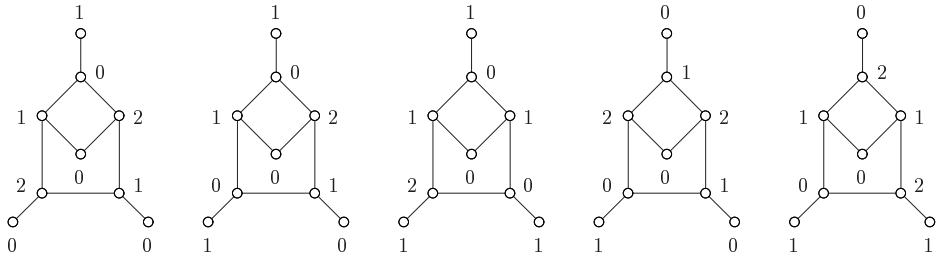
$$f(u_i) = \begin{cases} 1, & t(u_i) = True \\ 0, & t(u_i) = False \end{cases} \quad \text{και} \quad f(\neg u_i) = \begin{cases} 0, & t(u_i) = True \\ 1, & t(u_i) = False \end{cases}$$



Σχήμα 12.10: Υπογράφος μη ικανοποιήσιμης clause

Δηλαδή, όλα τα literals που γίνονται True με την απονομή που ικανοποιεί την Φ τα βάφουμε με το χρώμα 1 και όλα τα literals που γίνονται False τα βάφουμε με το χρώμα 0. Απομένει να βάψουμε τους υπογράφους που αντιστοιχούν στα clauses. Επειδή η φόρμουλα Φ ικανοποιείται σημαίνει ότι υπάρχει τουλάχιστον ένας κόμβος u_i ή $\neg u_i$ γειτονικός ενός κόμβου $c_j[k]$, ο οποίος έχει βαφεί με το χρώμα 1. Είναι ο κόμβος που αντιστοιχεί στο literal το οποίο ικανοποιεί το clause c_j. Συνεπώς δε μπορεί να υπάρχει υπογράφος που έχει χρωματιστεί όπως στο σχήμα 12.10. Όλες οι δυνατές περιπτώσεις,

είναι αυτές που φαίνονται στο σχήμα 12.11 και οι συμμετρικές τους. Άρα σε οποιαδήποτε περίπτωση το G βάφεται με 3 χρώματα.

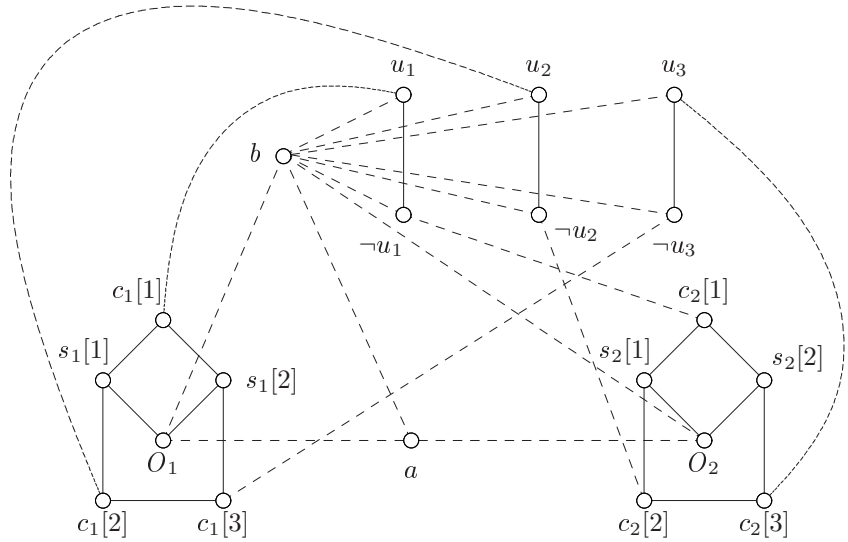


Σχήμα 12.11: Δυνατές περιπτώσεις χρωματισμών

Παράδειγμα: Έστω ότι έχουμε τη φόρμουλα

$$\Phi\colon (u_1 \vee u_2 \vee \neg u_3) \wedge (\neg u_1 \vee \neg u_2 \vee u_3)$$

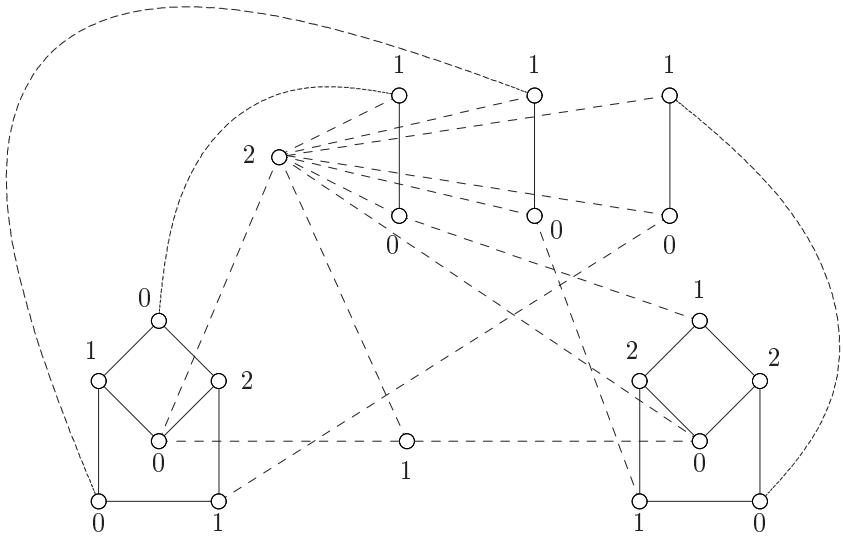
Μία απονομή που την ικανοποιεί είναι η $t(u_1, u_2, u_3) = (T, T, T)$. Ο γράφος $G(V, E)$ φαίνεται στο σχήμα 12.12. Ο χρωματισμός του γράφου $G(V, E)$ με 3 χρώματα φαίνεται στο σχήμα 12.13.



Σχήμα 12.12: Γράφος προς χρωματισμό

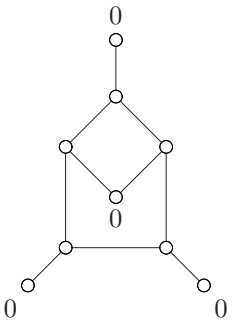
Αντίστροφα, έστω ότι ο γράφος G βάφεται με 3 χρώματα.

Έστω ότι το b έχει βαφεί με το χρώμα 2, το a με το χρώμα 1 και το O_j με το χρώμα 0. Τότε οι κόμβοι $u_i, \neg u_i$ πρέπει να βαφούν με τα χρώματα 0, 1. Θεωρούμε την απονομή $t(u_i) = True$ όταν το u_i έχει χρώμα 1 και



Σχήμα 12.13: Χρωματισμός γράφου με τρία χρώματα

$t(u_i) = False$ όταν το u_i έχει χρώμα 0. Αυτή η απονομή θα ικανοποιεί τη φόρμουλα Φ, διότι αν δεν την ικανοποιεί, σημαίνει ότι θα υπάρχει clause του οποίου όλα τα literals θα είναι False. Δηλαδή στον γράφο G θα υπάρχει υπογράφος σαν αυτόν του σχήματος 12.14. Αυτός ο υπογράφος, όπως εύκολα μπορεί να διαπιστώσει κανείς, δεν βάφεται με 3 χρώματα. ΑΤΟΠΟ. □



Σχήμα 12.14: Υπογράφος που δεν χρωματίζεται με 3 χρώματα

12.8 Η αναγωγή του VERTEX COVER στο HAMILTON CIRCUIT

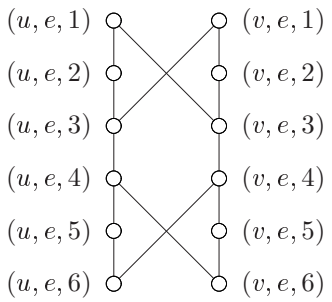
Θεώρημα 12.8.1. *Το πρόβλημα HAMILTON CIRCUIT (HC) είναι NP-complete.*

Proof. Το HC ανήκει στο NP, διότι ένας μη-ντετερμινιστικός αλγόριθμος μπορεί να ελέγξει αν η διατεταγμένη n-άδα κόμβων που μάντεψε είναι λύση του HC, σε πολυωνυμικό χρόνο. Θα δείξουμε ότι το HC είναι NP-complete ανάγοντας το VERTEX COVER (VC) σ' αυτό ($VC \leq_m^p HC$).

Μας δίνεται ένας γράφος $G(V, E)$ και ένας θετικός ακέραιος $k \leq |V|$. Θα κατασκευάσουμε έναν γράφο $G'(V', E')$ έτσι ώστε ο γράφος $G(V, E)$ να έχει vertex cover μεγέθους $k \leq |V|$ αν και μόνο αν ο γράφος $G'(V', E')$ έχει κύκλο Hamilton. Η κατασκευή γίνεται ως εξής:

- Βάζουμε στο V', k κόμβους $a_1, a_2, \ldots, a_k$ οι οποίοι ονομάζονται και selector vertices διότι θα χρησιμοποιηθούν για να επιλέξουν k κόμβους από το σύνολο V του G.

- Για κάθε πλευρά (u, v), βάζουμε στο γράφο G' μια «γέφυρα», δηλαδή το component που φαίνεται στο σχήμα 12.15. Δηλαδή, προσθέτουμε στο V' $12|E|$ κόμβους και στο E' $14|E|$ πλευρές. Αυτές οι γέφυρες αποτελούν τα cover testing components, διότι θα χρησιμοποιηθούν για να εξασφαλίζουν ότι τουλάχιστον ένας από τους κόμβους u, v ανήκει στους k κόμβους που αποτελούν vertex cover για το G.



Σχήμα 12.15: Γέφυρα για την αναγωγή του VC στο HC

- Ας θεωρήσουμε, $\forall v \in V$, μία μετάθεση όλων των πλευρών που ξεκινούν απ' το v:

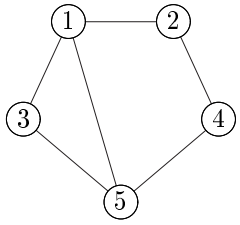
$$\left\langle e_{v_{(1)}}, e_{v_{(2)}}, \ldots, e_{v_{(d(v))}} \right\rangle$$

όπου $d(v)$ είναι ο βαθμός της κορυφής v. Θεωρούμε, δηλαδή, n τέτοιες μεταθέσεις ($|V| = n$), μία για κάθε κόμβο του V. Η κάθε μία απ' αυτές

θα έχει μήκος ίσο με το βαθμό της αντίστοιχης κορυφής ($n-1$ το πολύ). Για κάθε κόμβο v, ενώνουμε όλες τις γέφυρες που αντιστοιχούν σε μια τέτοια ακολουθία. Προσθέτουμε, δηλαδή, στο E' το σύνολο των πλευρών E'_v, όπου

$$E'_v = \{((v, e_{v_{(i)}}, 6), (v, e_{v_{(i-1)}}, 1))\}, 1 \leq i \leq d(v), \forall v \in V$$

Στη χειρότερη περίπτωση ο αριθμός πλευρών που προσθέτουμε είναι $O(n^2)$ ($\sum_{v \in V}(d(v)-1)$). Έτσι λοιπόν, τώρα έχουν δημιουργηθεί στο G', n μονοπάτια που αντιστοιχούν στις n μεταθέσεις.



Σχήμα 12.16: Γράφος παραδείγματος αναγωγής VC στο HC

Παράδειγμα: Έστω ότι έχουμε το γράφο που φαίνεται στο σχήμα 12.16. Επιλέγουμε τις μεταθέσεις:

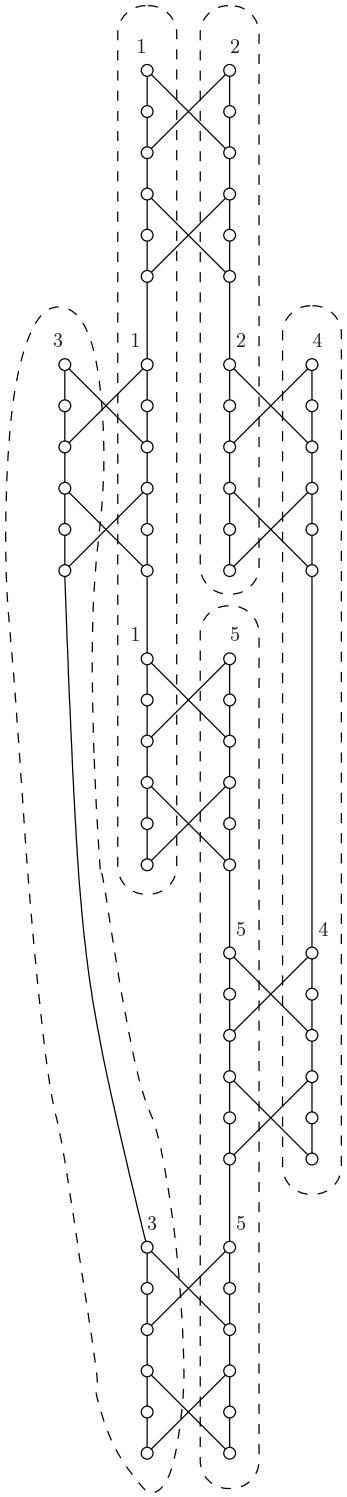
- Για τον κόμβο 1: $< (1,2), (1,3), (1,5) >$
- Για τον κόμβο 2: $< (2,1), (2,4) >$
- Για τον κόμβο 3: $< (3,1), (3,5) >$
- Για τον κόμβο 4: $< (4,2), (4,5) >$
- Για τον κόμβο 5: $< (5,1), (5,4), (5,3) >$

Τα n μονοπάτια που δημιουργούνται φαίνονται στο σχήμα 12.17.

Τέλος, συνδέουμε τα δύο άκρα του κάθε μονοπατιού με όλα τα σημεία $a_1, a_2, \ldots, a_k$. Δηλαδή, προσθέτουμε στο E' το σύνολο $E >>$ το οποίο αποτελείται από τις εξής $2kn$ πλευρές:

$$E >>= \{((v, e_{v_{(1)}}, 1), a_i), ((v, e_{v_{(d(v))}}, 6), a_i)\}, 1 \leq i \leq k, \forall v \in V$$

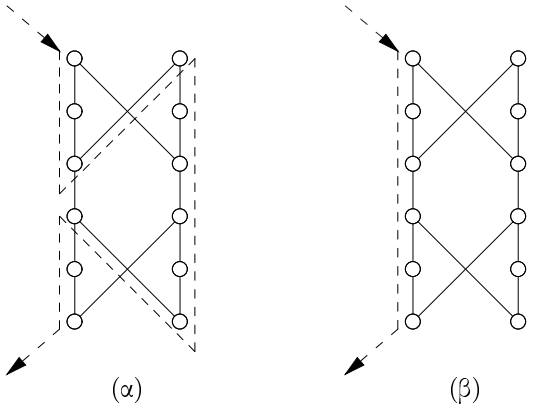
Η κατασκευή του G' έχει τελειώσει και εφόσον τα V', E' έχουν πολυωνυμικό μέγεθος ως προς τα V, E αντίστοιχα, το G' μπορεί να κατασκευαστεί σε πολυωνυμικό χρόνο.



Σχήμα 12.17: Γέφυρες που προκύπτουν από γράφο παραδείγματος

Έτσι λοιπόν, το μόνο που απομένει να δείξουμε είναι ότι το $G(V, E)$ έχει vertex cover μεγέθους $\leq k$ αν και μόνο αν το $G'(V', E')$ έχει κύκλο Hamilton. Έστω ότι ο γράφος G έχει vertex cover μεγέθους $\leq k$. Τότε σίγουρα υπάρχουν και ακριβώς k κόμβοι που καλύπτουν όλες τις πλευρές του G. Συμβολίζουμε το σύνολο αυτών των k κόμβων με V_c. Θα περιγράψουμε έναν κύκλο Hamilton στο γράφο G'.

Ξεκινάμε από έναν κόμβο a_i, έστω χωρίς βλάβη της γενικότητας, τον a_1 και από εκεί πηγαίνουμε στον κόμβο $(v, e_{v_{(1)}}, 1)$ όπου $v \in V_c$. Για να διατρέξουμε την γέφυρα που ξεκινά με τον κόμβο $(v, e_{v_{(1)}}, 1)$, περνώντας μία ακριβώς φορά από κάθε σημείο της, πρέπει να επιλέξουμε έναν από τους δύο τρόπους που φαίνονται στο σχήμα 12.18.



Σχήμα 12.18: Οι δύο τρόποι διάσχισης της κάθε γέφυρας

Αν $u \notin V_c$ περνάμε την γέφυρα με τον τρόπο (α). Αν $u \in V_c$ περνάμε την γέφυρα με τον τρόπο (β). Μετά πηγαίνουμε στη γέφυρα που ξεκινά με τον κόμβο $(v, e_{v_{(2)}}, 1)$ και αντιστοιχεί στην πλευρά (v, u') του G. Την περνάμε κι αυτή με τον ίδιο τρόπο (ανάλογα με το αν ($u' \in V_c$ ή $u' \notin V_c$). Έτσι περνάμε και την τελευταία γέφυρα του κόμβου v καταλήγοντας σε έναν άλλο κόμβο από τους k selectors, έστω a_2.

Από το a_2 περνάμε στα testing components ενός άλλου κόμβου που ανήκει στο V_c, κ.ο.κ., έως ότου περάσουμε k τέτοια μονοπάτια καταλήγοντας τελικά στο σημείο a_1 από το οποίο ξεκινήσαμε. Αυτή η διαδρομή είναι κύκλος Hamilton, διότι αν υπήρχε γέφυρα από την οποία δεν περνούσαμε, θα σήμαινε ότι η αντίστοιχη πλευρά στο G δεν είχε καλυφθεί. ΑΤΟΠΟ.

Επίσης, αν δεν έφταναν τα k selectors για να πάρουμε όλα τα testing components, τότε ο G δεν θα είχε vertex cover μεγέθους k. ΑΤΟΠΟ.

Αντίστροφα, αν το G' έχει κύκλο Hamilton, τότε επειδή διατρέχουμε k μονοπάτια, σημαίνει ότι κάθε γέφυρα πρέπει να ανήκει σε ένα τουλάχιστον απ' αυτά τα μονοπάτια. Συνεπώς κάθε πλευρά του G (που αντιστοιχεί σε κάποια

γέφυρα του G') καλύπτεται τουλάχιστον από 1 κόμβο του G (που αντιστοιχεί σε κάποιο από τα k μονοπάτια). □

12.9 Η αναγωγή του HAMILTON CIRCUIT στο TRAVELING SALESMAN PROBLEM

Θεώρημα 12.9.1. *Το TRAVELING SALESMAN PROBLEM (TSP) είναι NP-complete.*

Proof. Το TSP ανήκει στο NP, διότι ένας μη-ντετερμινιστικός αλγόριθμος μαντεύει μια διάταξη των κόμβων και ελέγχει σε πολυωνυμικό χρόνο αν το άθροισμα των βαρών των αντίστοιχων πλευρών είναι $\leq B$. Θα δείξουμε ότι το TSP είναι NP-complete ανάγοντας το HC σ' αυτό ($HC \leq_m^p TSP$).

Μας δίνεται ένας γράφος $G(V, E)$. Θέλουμε να κατασκευάσουμε έναν πλήρη γράφο $G'(V', E')$ με βάρη $d(u, v), \forall (u, v) \in E'$ και έναν θετικό ακέραιο B, έτσι ώστε ο γράφος $G(V, E)$ να έχει κύκλο Hamilton αν και μόνο αν ο $G'(V', E')$ έχει tour με βάρος $\leq B$. Η κατασκευή γίνεται ως εξής:

- Σαν γράφο $G'(V', E')$, παίρνουμε τον γράφο $G(V, E)$, προσθέτοντας όλες τις ακμές που υπολείπονται για να γίνει πλήρης. Βάζουμε βάρη στις ακμές του G' ως εξής:

$$d(u, v) = \begin{cases} 1, & (u, v) \in G \\ 2, & (u, v) \notin G \end{cases}$$

- Τέλος, παίρνουμε $B = |V'| = |V|$. Η κατασκευή έχει τελειώσει και μπορεί, προφανώς να γίνει σε πολυωνυμικό χρόνο.

Έστω ότι ο $G(V, E)$ έχει κύκλο Hamilton. Τότε παίρνοντας αυτόν τον κύκλο σαν tour στον γράφο G', προφανώς περνά μία ακριβώς φορά από κάθε κόμβο και έχει συνολικό βάρος $B = |V'| = |V|$ εφόσον κάθε πλευρά έχει βάρος 1 (αφού ανήκε στον G).

Αντίστροφα, έστω ότι ο γράφος G' έχει κάποιο tour με συνολικό βάρος $\leq B = |V'| = |V|$. Αφού όμως το G' έχει $|V'|$ κόμβους, το tour θα περνά από $|V'|$ πλευρές και συνεπώς το συνολικό βάρος θα είναι ακριβώς $B = |V'|$. Αυτό όμως μπορεί να συμβεί μόνο όταν κάθε μία από τις $|V'|$ πλευρές έχει βάρος 1. Άρα όλες αυτές οι πλευρές ανήκουν στον G και συνεπώς ο G έχει κύκλο Hamilton (είναι το tour του G'). □

12.10 Η αναγωγή του VERTEX COVER στο CLIQUE

Θεώρημα 12.10.1. *Το πρόβλημα CLIQUE είναι NP-complete.*

Proof. Το CLIQUE ανήκει στο NP, διότι ένας μη-ντετερμινιστικός αλγόριθμος, αφού μαντέψει ένα σύνολο κόμβων $V' \subseteq V$ με $|V'| \geq j$ μπορεί να ελέγξει σε πολυωνυμικό χρόνο αν κάθε δύο κόμβοι που ανήκουν στο V' ενώνονται με μία πλευρά που ανήκει στο E. Θα δείξουμε ότι το CLIQUE είναι NP-complete ανάγοντας το VERTEX COVER (VC) σ' αυτό (VC $\leq_m^p$ Clique).

Μας δίνεται ένας γράφος $G(V, E)$ και ένας θετικός ακέραιος $k \leq |V|$. Θα κατασκευάσουμε έναν γράφο $G'(V', E')$ και έναν θετικό ακέραιο $j \leq |V'|$ έτσι ώστε ο γράφος G να έχει vertex cover μεγέθους $\leq k$ αν και μόνο αν ο γράφος G' έχει κλίκα μεγέθους $\geq j$. Η κατασκευή γίνεται ως εξής:

- Παίρνουμε σαν G' το συμπληρωματικό γράφο του G. Δηλαδή $V' = V$ και $E' = \{(u, v) \mid (u, v) \notin E\}$. Επίσης παίρνουμε $j = |V| - k = |V'| - k$. Η κατασκευή γίνεται προφανώς σε πολυωνυμικό χρόνο.

Έστω ότι ο γράφος $G(V, E)$ έχει ένα σύνολο κόμβων $V_c \subseteq V$ με $|V_c| \leq k$ το οποίο καλύπτει όλες τις πλευρές του G. Αυτό σημαίνει ότι όλοι οι κόμβοι που ανήκουν στο $V - V_c$ είναι ανά δύο μη-γειτονικοί (διότι αν υπήρχε πλευρά που ένωνε δύο τέτοιους κόμβους, τότε αυτή η πλευρά δεν θα είχε καλυφθεί από το V_c). Άρα λοιπόν, όλοι αυτοί οι κόμβοι συνδέονται ανά δύο μεταξύ τους στον γράφο G'. Συνεπώς ο G' έχει κλίκα μεγέθους $|V| - |V_c| \geq |V| - k = j$. Αντίστροφα, αν ο γράφος $G'(V', E')$ έχει κλίκα μεγέθους $\geq j = |V'| - k$ τότε στον γράφο $G(V, E)$ θα υπάρχουν τουλάχιστον $|V'| - k = |V| - k$ κόμβοι που δεν θα συνδέονται μεταξύ τους. Συνεπώς οι υπόλοιποι (το πολύ k) κόμβοι θα φρουρούν υποχρεωτικά όλες τις ακμές του G. Άρα ο γράφος G έχει vertex cover μεγέθους $\leq k$. □

12.11 Η αναγωγή του CLIQUE στο SUBGRAPH ISOMORPHISM

Θεώρημα 12.11.1. *Το πρόβλημα SUBGRAPH ISOMORPHISM είναι NP-complete.*

Proof. Το SUBGRAPH ISOMORPHISM ανήκει στο NP, διότι ένας μη-ντετερμινιστικός αλγόριθμος μπορεί να ελέγξει σε πολυωνυμικό χρόνο ότι η f που έχει μαντέψει, απεικονίζει το V_2 (βλ. ορισμό του προβλήματος) σε ένα υποσύνολο του V τέτοιο ώστε,$\forall u, v[(u, v) \in E_2 \iff (f(u), f(v)) \in E]$. Θα δείξουμε ότι το

SUBGRAPH ISOMORPHISM είναι NP-complete ανάγοντας το CLIQUE σ' αυτό ($CLIQUE \leq_m^p SUBGRAPH\ ISOMORPHISM$).

Μας δίνεται ένας γράφος $G(V, E)$ και ένας θετικός ακέραιος $k \leq |V|$. Θα κατασκευάσουμε δύο γράφους $G_1(V_1, E_1)$ και $H(V_2, E_2)$ έτσι ώστε ο γράφος $G(V, E)$ να έχει κλίκα μεγέθους $\geq k$ αν και μόνο αν ο γράφος $G_1(V_1, E_1)$ έχει υπογράφο ισομορφικό με τον $H(V_2, E_2)$. Παίρνουμε σαν γράφο G_1 τον γράφο G και σαν H, ένα πλήρη γράφο με k κόμβους.

Έστω ότι ο γράφος G έχει κλίκα με $V_c = k$. Αυτό σημαίνει ότι ο G_1 έχει υπογράφο ισομορφικό με τον H (εξ ορισμού). Αντίστροφα, έστω ότι ο G_1 έχει υπογράφο ισομορφικό με τον H που είναι πλήρης γράφος με k κόμβους. Συνεπώς ο G έχει κλίκα μεγέθους k τουλάχιστον. □

12.12 Η αναγωγή του 3DM στο PARTITION

Θεώρημα 12.12.1. *Το πρόβλημα PARTITION είναι NP-complete.*

Proof. Το PARTITION ανήκει στο NP, διότι ένας μη-ντετερμινιστικός αλγόριθμος μαντεύει ένα $A' \subseteq A$ (βλέπε ορισμό του προβλήματος) και μπορεί να ελέγξει σε πολυωνυμικό χρόνο ότι $\sum_{a \in A'} w(a) = \sum_{a \in (A - A')} w(a)$. Θα δείξουμε ότι το PARTITION είναι NP-complete ανάγοντας το 3-DIMENSIONAL MATCHING σ' αυτό (3DM $\leq_m^p PARTITION$).

Μας δίνονται τρία ξένα μεταξύ τους σύνολα W, X, Y με $|W| = |X| = |Y|$ και ένα σύνολο $M \subseteq W \times X \times Y$. Θα κατασκευάσουμε ένα σύνολο A και κάποια συνάρτηση βάρους $w(a) \in Z^+, \forall a \in A$, έτσι ώστε το σύνολο M να έχει ένα matching $M' \subseteq M$ αν και μόνο αν το σύνολο A μπορεί να χωριστεί σε δύο ισοβαρή υποσύνολα. Η κατασκευή γίνεται ως εξής:

- Έστω

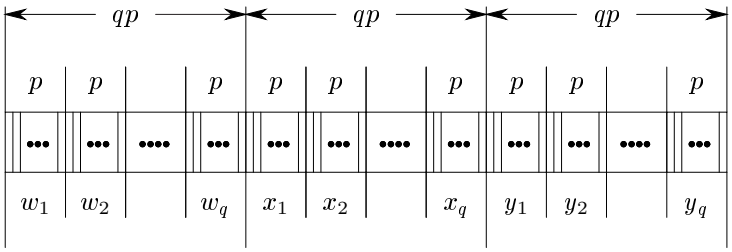
$$\begin{aligned} W &= \{w_1, w_2, \ldots, w_q\} \\ X &= \{x_1, x_2, \ldots, x_q\} \\ Y &= \{y_1, y_2, \ldots, y_q\} \\ M &= \{m_1, m_2, \ldots, m_q\}, \quad \text{όπου } m_i = (w_r, x_l, y_n) \end{aligned}$$

  Βάζουμε στο σύνολο A, k στοιχεία $a_1, a_2, \ldots, a_k$ καθένα από τα οποία αντιστοιχεί σε κάποιο m_i. Το βάρος $w(a_i)$ θα οριστεί ως εξής:

  – Αναπαριστούμε το κάθε $w(a_i)$ με μία ακολουθία δυαδικών ψηφίων. Η ακολουθία αυτή αποτελείται από $3q$ ζώνες και η κάθε ζώνη από $p = \lceil \log_2(k+1) \rceil$ δυαδικά ψηφία (σχήμα 12.19). Αν βάλουμε κάτω από τις ζώνες σαν labels τα

$$w_1, w_2, \ldots, w_q \qquad x_1, x_2, \ldots, x_q \qquad y_1, y_2, \ldots, y_q$$

όπως φαίνεται στο σχήμα 12.19, έχουμε μια αντιστοίχηση μεταξύ a_i και m_i.



Σχήμα 12.19: Ακολουθία από bits στην αναγωγή του 3DM στο PARTITION

Αν $m_i = (w_r, x_l, y_n)$, τότε στις αντίστοιχες θέσεις w_r, x_l, y_n του $w(a_i)$ κάνουμε το δεξιότερο απ' όλα τα bits 1 και όλα τ' άλλα bits 0. Δηλαδή κάθε αναπαράσταση των $w(a_i)$ θα έχει τρεις άσσους, ένα σε κάθε ζώνη μήκους q, και όλα τ' άλλα ψηφία 0. Συνεπώς αν μετατρέψουμε αυτόν τον δυαδικό αριθμό σε δεκαδικό έχουμε:

$$w(a_i) = 2^{p(3q-r)} + 2^{p(2q-l)} + 2^{p(q-n)}$$

Θέλουμε λοιπόν, k τέτοιες ακολουθίες των $3qp$ bits. Συνεπώς μπορούμε να το κάνουμε σε πολυωνυμικό χρόνο.

– Πριν προχωρήσουμε θα εξηγήσουμε γιατί επιλέχθηκε το p ίσο με $\lceil \log_2(k+1) \rceil$. Αν λύσουμε ως προς k έχουμε $2^p \geq k+1 \Rightarrow k \leq 2^p - 1$. Δηλαδή, ακόμα και αν όλα τα στοιχεία m_i είχαν μία κοινή συντεταγμένη, αν προσθέταμε όλα τα $w(a_i)$, η άθροιση στην ζώνη που αντιστοιχεί σ' αυτήν την κοινή τους συντεταγμένη θα είναι:

$$\begin{array}{cccccc} & & p & & & \\ 0 & 0 & \cdots & 0 & 1 & \\ & & \vdots & & & k \\ 0 & 0 & \cdots & 0 & 1 & \end{array}$$

Το άθροισμα, δηλαδή, σ' αυτή τη ζώνη θα είναι ο αριθμός $k \leq 2^p - 1$, ένας αριθμός ο οποίος μπορεί πάντα να παρασταθεί με p bits. Αυτό σημαίνει πως δεν είναι δυνατόν να έχουμε μεταφορές κρατουμένων από κάποια ζώνη στην επόμενη. Δηλαδή αν για παράδειγμα στις m_k τριάδες δεν περιλαμβάνεται καθόλου κάποια συντεταγμένη, έστω η w_r, τότε η ζώνη με label w_r θα έχει μόνο μηδενικά και δε θα μπορούσε να αποκτήσει ποτέ 1 αν αθροίζαμε όλα τα $w(a_i)$.

– Ορίζουμε τον αριθμό B ως εξής:

$$B = 2^{3qp-p} + 2^{3qpp-2p} + \ldots + 2^0 = \sum_{j=0}^{3q-1} 2^{pj}$$

Δηλαδή, η δυαδική αναπαράσταση του B έχει έναν άσσο στη δεξιότερη θέση κάθε ζώνης p. Παρατηρούμε ότι σε ένα matching, κάθε συντεταγμένη θα λαμβάνεται ακριβώς μία φορά. Συνεπώς, όπως εύκολα μπορεί να διαπιστώσει κανείς,

$\exists A' \subseteq A\colon \sum_{a_i \in A'} w(a_i) = B \qquad \Longleftrightarrow$
$\exists M' \subseteq M\colon M'$ είναι matching του M με $M' = \{m_i \mid a_i \in A'\}$ $(*)$

- Τέλος, προσθέτουμε δύο ακόμα στοιχεία στο A. Το στοιχείο b_1 με βάρος $w(b_1) = 2\sum_{i=1}^{k} w(a_i) - B$ και το στοιχείο b_2 με βάρος $w(b_2) = \sum_{i=1}^{k} w(a_i) + B$. Αυτά τα δύο αθροίσματα μπορούν να υπολογιστούν επίσης σε πολυωνυμικό χρόνο.

Αυτό που μένει να αποδείξουμε είναι ότι το M έχει matching αν και μόνο αν το A μπορεί να χωριστεί σε δύο ισοβαρή σύνολα. Έστω ότι το M έχει ένα matching M'. Τότε λόγω της $(*)$, $\exists A' \subseteq A\colon \sum_{a_i \in A} w(a_i) = B$.

Όλο το βάρος του A είναι:

$$W_{tot} = \sum_{i=1}^{k} w(a_i) + w(b_1) + w(b_2) = 4\sum_{i=1}^{k} w(a_i).$$

Αν πάρουμε στο ένα σύνολο τα $a_i \in A'$ και το b_1 θα έχουμε βάρος

$$2\sum_{i=1}^{k} w(a_i) - B + B = 2\sum_{i=1}^{k} w(a_i).$$

Συνεπώς το άλλο σύνολο που περιέχει τα $a_i \in (A - A')$ και το b_2 θα έχει βάρος

$$W_{tot} - 2\sum_{i=1}^{k} w(a_i) = 2\sum_{i=1}^{k} w(a_i).$$

Αντίστροφα, έστω ότι το A χωρίζεται σε δύο ισοβαρή σύνολα. Ο καθένας θα έχει βάρος $2\sum_{i=1}^{k} w(a_i)$. Συνεπώς δεν μπορεί στο ίδιο σύνολο να υπάρχουν

τα b_1, b_2. Το σύνολο στο οποίο θα ανήκει το b_1 θα περιέχει και κάποια $a_i \in A'$ έτσι ώστε:

$$w(b_1) + \sum_{a_i \in A'} w(a_i) = 2\sum_{i=1}^{k} w(a_i) \Rightarrow \sum_{a_i \in A'} w(a_i) = B$$

Όμως, λόγω της $(*)$, αυτό σημαίνει ότι $\exists M' \subseteq M : M' = \{m_i \mid a_i \in A'\}$ και M' είναι matching. □

12.13 Η αναγωγή του PARTITION στο DISCRETE KNAPSACK

Θεώρημα 12.13.1. *Το πρόβλημα DISCRETE KNAPSACK είναι NP-complete.*

Proof. Το DKNAPSACK ανήκει στο NP, διότι ένας μη ντετερμινιστικός αλγόριθμος μαντεύει ένα $U' \subseteq U$ (βλ. ορισμό του προβλήματος) και μετά ελέγχει σε πολυωνυμικό χρόνο αν $\sum_{u \in U'} w(u) \leq W$ και $\sum_{u \in U'} p(u) \geq p$. Θα δείξουμε ότι το DKNAPSACK είναι NP-complete ανάγοντας το PARTITION σ' αυτό ($PARTITION \leq_m^p DKNAPSACK$).

Μας δίνεται ένα σύνολο A και μία συνάρτηση βάρους $w(a_i), \forall a_i \in A$. Θα κατασκευάσουμε ένα σύνολο U, μια συνάρτηση βάρους $w'(u_i), \forall u_i \in U$, μια συνάρτηση κόστους $p(u_i), \forall u_i \in U$ και δύο θετικούς ακέραιους W, P έτσι ώστε το σύνολο A να χωρίζεται σε δύο ισοβαρή σύνολα αν και μόνο αν υπάρχει, $U' \subseteq U : \sum_{u \in U'} w'(u) \leq W$, $\sum_{u \in U'} p(u) \geq p$. Η κατασκευή γίνεται ως εξής:

- Σαν σύνολο U παίρνουμε το σύνολο A
- Σαν $w'(u), \forall u \in U$ παίρνουμε το $w(a), \forall a \in A$
- Σαν $p(u), \forall u \in U$ παίρνουμε το $w(a), \forall a \in A$
- Δηλαδή, παίρνουμε το ίδιο σύνολο όπως έχει με τα ίδια βάρη και το κόστος κάθε αντικειμένου το παίρνουμε ίσο με το βάρος του. Τέλος, τους αριθμούς W, P τους παίρνουμε ως εξής:

$$W = P = \frac{1}{2}\sum_{a \in A} w(a) = \frac{1}{2}\sum_{u \in U} w'(u)$$

Η κατασκευή έχει τελειώσει και είναι προφανές ότι μπορεί να γίνει σε πολυωνυμικό χρόνο.

Έστω λοιπόν, ότι το σύνολο A χωρίζεται σε δύο ισοβαρή υποσύνολα, στον A' και στον $A - A'$. Επειδή το ολικό βάρος του A είναι $\sum_{a \in A} w(a)$, σημαίνει ότι το βάρος του υποσυνόλου A', για παράδειγμα, είναι $\frac{1}{2} \sum_{a \in A} w(a)$. Άρα αν πάρουμε σαν U' το σύνολο A' έχουμε, $\sum_{u \in U'} w'(u) = W \leq W$ και $\sum_{u \in U'} p(u) = P \geq P$

Αντίστροφα, έστω ότι $\exists U' \subseteq U$: $\sum_{u \in U'} w'(u) \leq W$ και $\sum_{u \in U'} p(u) \geq p$. Όμως $\sum_{u \in U'} w'(u) = \sum_{u \in U'} p(u) = B = P = \frac{1}{2} \sum_{u \in U} w'(u)$. Συνεπώς, αν πάρουμε σαν A' το U', τότε το A χωρίζεται σε δύο ισοβαρή υποσύνολα, A' και $A - A'$. □

Κεφάλαιο 13

Κλάσεις πολυπλοκότητας

Όπως είναι ήδη κατανοητό, πολλά από τα προβλήματα για τα οποία έχει αποδειχθεί ότι είναι NP-complete έχουν μεγάλη πρακτική αξία. Έτσι λοιπόν, η προσπάθεια για κατάταξη κάποιου προβλήματος σε κάποια κλάση, δείχνοντας ότι είναι complete σ' αυτήν την κλάση, τις περισσότερες φορές εκτός από θεωρητικό, έχει και τεράστιο πρακτικό ενδιαφέρον.

Παρά τις συνεχείς και μακροχρόνιες προσπάθειες πολλών επιστημόνων, υπάρχουν αρκετά ανοιχτά προβλήματα. Προβλήματα δηλαδή, τα οποία αν και ανήκουν στο NP, δεν έχει βρεθεί πολυωνυμικός αλγόριθμος γι' αυτά αλλά ούτε απόδειξη ότι είναι NP-complete.

Ένα πρόβλημα που ανήκει σ' αυτήν την κατηγορία είναι το GRAPH ISOMORPHISM. Δηλαδή, δεδομένων δύο γράφων είναι ισομορφικοί; (πρβλ με το SUBGRAPH ISOMORPHISM.)

Ένα πρόβλημα που ανήκε μέχρι πρόσφατα στην κατηγορία αυτή είναι το PRIMALITY. Δηλαδή, δεδομένου ενός ακέραιου αριθμού k, υπάρχουν ακέραιοι $m, n > 1$ έτσι ώστε $k = m \cdot n$; Με άλλα λόγια είναι ο k πρώτος αριθμός ή όχι; Το 2002 απεδείχθη από τους Agrawal, Kayal και Saxena (αλγόριθμος AKS) ότι το PRIMALITY ανήκει στην κλάση P.

Ένα άλλο πρόβλημα που παρέμενε για χρόνια ανοιχτό είναι το LINEAR PROGRAMMING. Δηλαδή, δεδομένου ενός συστήματος γραμμικών εξισώσεων και ανισοτήτων και μιας γραμμικής αντικειμενικής συνάρτησης (μεγιστοποίησης ή ελαχιστοποίησης) να ευρεθεί μια βέλτιστη εφικτή λύση. Για αυτό το πρόβλημα υπήρχε από την δεκαετία του 1950 η μέθοδος *Simplex* του Dantzig η οποία όμως στη χειρότερη περίπτωση χρειαζόταν εκθετικό χρόνο. Το 1979 ο Khachiyan ανακάλυψε τον πρώτο πολυωνυμικό αλγόριθμο για τον γραμμικό προγραμματισμό, την ελλειψοειδή μέθοδο, η οποία όμως δεν είχε μεγάλο πρακτικό ενδιαφέρον γιατί στις περισσότερες πρακτικές εφαρμογές έδινε χειρότερα αποτελέσματα από τη μέθοδο *Simplex*. Τελικά το 1984 ο Karmarkar ανακάλυψε έναν άλλο

πολυωνυμικό αλγόριθμο που είχε και πρακτικά αποτελέσματα καλύτερα από τη μέθοδο *Simplex*.

Η DSPACE($f(n)$) (NSPACE($f(n)$)) είναι η κλάση προβλημάτων που λύνονται με ντετερμινιστικό τρόπο (μη ντετερμινιστικό τρόπο, αντίστοιχα) σε χώρο $f(n)$ (μόνο ο επιπλέον χώρος εργασίας μετριέται όχι ο χώρος της εισόδου και της εξόδου)

Στη συνέχεια αναφέρουμε μερικές γνωστές σχέσεις μεταξύ διαφόρων κλάσεων πολυπλοκότητας.

Ορίζουμε,

- $P = PTIME = \bigcup_{i\geq 1} DTIME(n^i)$
- $NP = NPTIME = \bigcup_{i\geq 1} NTIME(n^i)$
- $PSPACE = \bigcup_{i\geq 1} DSPACE(n^i)$
- $NPSPACE = \bigcup_{i\geq 1} NSPACE(n^i)$
- $L = DSPACE(\log n)$
- $NL = NSPACE(\log n)$

Αν η f είναι μία συνάρτηση πολυπλοκότητας[1] τότε ισχύουν:

- $DSPACE(f(n)) \subseteq NSPACE(f(n))$
- $DTIME(f(n)) \subseteq NTIME(f(n))$

διότι κάθε ντετερμινιστική μηχανή Turing μπορεί να θεωρηθεί ως μη ντετερμινιστική με μία μόνο επιλογή σε κάθε βήμα.

- $DTIME(f(n)) \subseteq DSPACE(f(n))$
- $NTIME(f(n)) \subseteq DSPACE(f(n))$

διότι σε χρόνο $f(n)$ δεν μπορεί να εξεταστεί χώρος (αριθμός θέσεων στην ταινία της $T.M.$) παραπάνω από $f(n)$.

- $NSPACE(f(n)) \subseteq DTIME(k^{\log n + f(n)})$

Αν $f(n) > \log n$ τότε:

- $DSPACE(f(n)) \subseteq DTIME(c^{f(n)})$

[1] Η f πρέπει να είναι constructible, δηλαδή πρέπει να υπάρχει μία TM τέτοια ώστε: $\forall$ input x με $|x| = n$, αποδέχεται το input σε χρόνο $O(n+f(n))$ και working space $O(f(n))$

- $NTIME(f(n)) \subseteq DTIME(c^{f(n)})$
- $NSPACE(f(n)) \subseteq DSPACE(f^2(n))$ (από το θεώρημα του *Savitch*) συνεπώς $PSPACE = NPSPACE$
- $L \subset PSPACE$

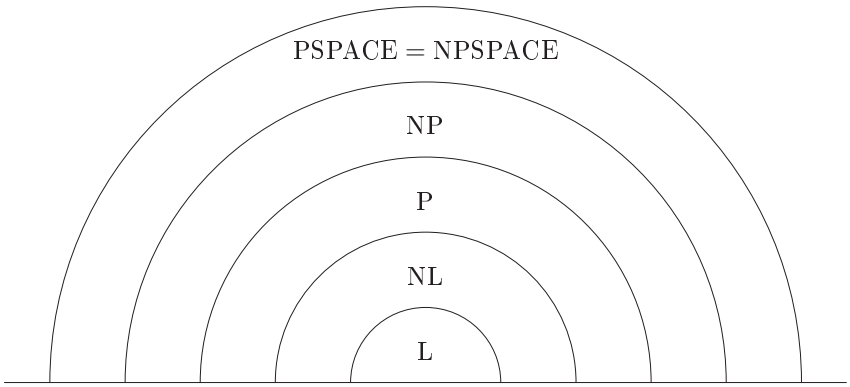
Από τις παραπάνω σχέσεις προκύπτει η εξής ιεραρχία:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE$$

Γνωρίζουμε ότι $L \neq PSPACE$ και $NL \neq PSPACE$
Ανοιχτά παραμένουν τα προβλήματα:

$$L \supseteq NL \supseteq P \supseteq NP \supseteq PSPACE$$

Ο κόσμος μοιάζει, ως τώρα, να είναι όπως στο σχήμα 13.1.



Σχήμα 13.1: Κλάσεις πολυπλοκότητας

Κεφάλαιο 14

Ψευδοπολυωνυμικοί και Προσεγγιστικοί Αλγόριθμοι

14.1 Προβλήματα Βελτιστοποίησης

Σε ένα πρόβλημα βελτιστοποίησης σε κάθε στιγμιότυπο του προβλήματος αντιστοιχούν κάποιες εφικτές (feasible) -δηλαδή επιτρεπτές- λύσεις, που κάθε μια τους έχει μία τιμή ως προς μια αντικειμενική συνάρτηση. Ζητάμε μια βέλτιστη λύση, δηλαδή μια **εφικτή** λύση που έχει βέλτιστη τιμή.

Τα προβλήματα βελτιστοποίησης είναι ή μεγιστοποίησης (οπότε ζητάμε τη λύση που μεγιστοποιεί την αντικειμενική συνάρτηση ανάμεσα σε όλες τις εφικτές λύσεις) ή ελαχιστοποίησης (οπότε ζητάμε την εφικτή λύση που ελαχιστοποιεί την αντικειμενική συνάρτηση).

Παράδειγμα 14.1.1. Το πρόβλημα Vertex Cover. Θυμηθείτε ότι στο πρόβλημα αυτό, αναζητούμε ένα σύνολο από κόμβους του γράφου με ελάχιστο πληθάριθμο, έτσι ώστε κάθε ακμή του γράφου να καλύπτεται από τουλάχιστον έναν κόμβο. Είναι αντίστοιχο με το να θεωρήσουμε το σύνολο των ακμών του γράφου σαν ένα σύνολο από διαδρόμους σε ένα μουσείο και να προσπαθήσουμε να τοποθετήσουμε φύλακες στις διασταυρώσεις των διαδρόμων (κορυφές του γράφου) κατά τέτοιο τρόπο, ώστε να φρουρούνται όλοι οι διάδρομοι, ενώ συγχρόνως προσπαθούμε να ελαχιστοποιήσουμε τον αριθμό των φυλάκων. Για το πρόβλημα αυτό έχουμε:

- Είναι πρόβλημα ελαχιστοποίησης
- Στιγμιότυπο είναι ένας γράφος
- Κάθε Vertex Cover είναι εφικτή λύση (π.χ. $V(G)$)
- Αντικειμενική Συνάρτηση: Ο πληθάριθμος του Vertex Cover

- Βέλτιστη λύση: Ένα Vertex Cover με ελάχιστο πληθάριθμο

14.2 Κλάσεις προσεγγιστικών προβλημάτων

14.2.1 Η κλάση PO

Η κλάση PO περιέχει όλα τα προβλήματα βελτιστοποίησης των οποίων το αντίστοιχο πρόβλημα απόφασης είναι στο P. Αυτό σημαίνει ότι περιέχει όλα τα προβλήματα βελτιστοποίησης τα οποία μπορούν να επιλυθούν σε πολυωνυμικό χρόνο.

Ας δούμε όμως τι εννοούμε με τη φράση «αντίστοιχο πρόβλημα απόφασης»: εννοούμε ένα πρόβλημα στο οποίο μπορούμε να αναγάγουμε το πρόβλημα βελτιστοποίησης -σε πολυωνυμικό πάντα χρόνο-, έτσι ώστε λύνοντας το πρόβλημα απόφασης (για κατάλληλα επιλεγμένες εισόδους που θα δούμε παρακάτω) να μπορούμε να βρούμε την τιμή μίας βέλτιστης λύσης στο πρόβλημα βελτιστοποίησης. Ο λόγος που αναφερόμαστε σε αντίστοιχο πρόβλημα απόφασης, είναι ότι ως τώρα η πολυπλοκότητα και τα διάφορα υπολογιστικά μοντέλα που έχουμε, αναφέρονται αποκλειστικά σε προβλήματα απόφασης. Ως εκ τούτου, για να μελετήσουμε τα προβλήματα βελτιστοποίησης με πιο τυπικό τρόπο, προσπαθούμε να τα αντιστοιχίσουμε σε προβλήματα απόφασης.

Στην προκείμενη περίπτωση, τα αντίστοιχα προβλήματα είναι:

Πρόβλημα Βελτιστοποίησης (έστω ελαχιστοποίησης): Δοθέντος ενός στιγμιοτύπου μεγέθους n, βρες μία εφικτή λύση με ελάχιστη τιμή της αντικειμενικής συνάρτησης.

Πρόβλημα απόφασης: Δοθέντος ενός στιγμιοτύπου μεγέθους n και ενός αριθμού k, υπάρχει εφικτή λύση με τιμή $\leq k$;

Αν μπορούμε να λύσουμε το πρόβλημα βελτιστοποίησης τότε τετριμμένα μπορούμε να απαντήσουμε και στο πρόβλημα απόφασης.

Ισχύει όμως και το αντίστροφο: αν το πρόβλημα απόφασης επιλύεται σε χρόνο πολυωνυμικό ως προς n και $\log k$ (δηλαδή ως προς το μέγεθος αναπαράστασης του αριθμού k), τότε μπορούμε να λύσουμε και το αντίστοιχο πρόβλημα βελτιστοποίησης σε πολυωνυμικό ως προς n χρόνο.

Μία πρώτη ιδέα είναι να τρέξουμε τον αλγόριθμο για το πρόβλημα απόφασης (έστω A) K φορές, για $k = 1, \ldots, K$ (όπου K ένας κατάλληλα επιλεγμένος μεγάλος αριθμός, τέτοιος ώστε να υπάρχει εφικτή λύση με κόστος λιγότερο από K) και να επιστρέψουμε την πρώτη λύση για την οποία ο A επιστρέφει yes.

Δυστυχώς μία τέτοια προσέγγιση, θα απαιτούσε να τρέξουμε τον αλγόριθμο A στη χειρότερη περίπτωση K φορές, δηλαδή θα απαιτούσε χρόνο εκθετικό ως προς το μέγεθος αναπαράστασης του K.

Αντ' αυτού επιλέγουμε την εξής πιο γρήγορη λύση: ψάχνουμε να βρούμε κάποιο k, τέτοιο ώστε να υπάρχει μία εφικτή λύση με κόστος $\leq k$ αλλά όχι με κόστος $\leq k/2$. Το k αυτό, μπορεί να βρεθεί εύκολα σε $\log k$ βήματα (δοκιμάζοντας διαδοχικά τις τιμές $2^0, 2^1, \ldots$). Αφού βρούμε αυτό το k εφαρμόζουμε δυαδική αναζήτηση στο διάστημα $[k, k/2]$, ώστε να βρούμε τη μικρότερη τιμή k για την οποία ο A επιστρέφει yes. Η δυαδική αναζήτηση κοστίζει επίσης $\log \frac{k}{2}$ χρόνο, οπότε συνολικά, τρέχοντας $O(\log k)$ φορές τον αλγόριθμο A (δηλαδή πολυωνυμικό αριθμό φορών ως προς το μέγεθος της εισόδου) μπορούμε να λύσουμε και το αντίστοιχο πρόβλημα βελτιστοποίησης.

Παρατήρηση. Προσέξτε ότι με την παραπάνω μέθοδο, μπορούμε να χρησιμοποιήσουμε το πρόβλημα απόφασης για να βρούμε την τιμή μίας βέλτιστης λύσης του προβλήματος βελτιστοποίησης· όχι για να βρούμε μία βέλτιστη λύση αυτή καθεαυτή (πχ αν το πρόβλημα είναι το Vertex Cover, με βάση την παραπάνω μέθοδο θα μπορέσουμε να βρούμε το μέγεθος ενός ελάχιστου Vertex Cover, αλλά όχι από ποιους κόμβους αποτελείται ένα ελάχιστο VC). Συνήθως όμως, όταν μπορούμε να βρούμε την τιμή μίας βέλτιστης λύσης, μπορούμε να βρούμε και μία βέλτιστη λύση καθεαυτή, εξετάζοντας πιθανά στοιχεία της λύσης ένα-ένα (Άσκηση: σκεφτείτε πώς μπορεί να γίνει αυτό για το Vertex Cover).

14.3 Η κλάση NPO και οι προσεγγιστικοί αλγόριθμοι

Προβλήματα στην PO μπορούν να επιλυθούν σε πολυωνυμικό χρόνο. Τώρα θα συζητήσουμε για προβλήματα βελτιστοποίησης στο NP, δηλαδή τέτοια που το αντίστοιχο πρόβλημα απόφασης είναι στο NP και μάλιστα είναι NP-complete. Εκτός αν P=NP δεν μπορούμε να υπολογίσουμε σε πολυωνυμικό χρόνο τη βέλτιστη τιμή ενός προβλήματος βελτιστοποίησης που αντιστοιχεί σε ένα NP-hard πρόβλημα απόφασης, άρα

- είτε επιλύουμε **σε πολυωνυμικό χρόνο** το πρόβλημα **ακριβώς**, **όχι** όμως **για όλα τα στιγμιότυπα**,
- είτε επιλύουμε το πρόβλημα **ακριβώς**, **για όλα τα στιγμιότυπα**, με χρήση κάποιου **εκθετικού αλγορίθμου** με χαμηλή exponentiality (βάση της δύναμης) όμως (π.χ. με πολυπλοκότητα $O(1.01^n)$),
- είτε βρίσκουμε **προσεγγιστικούς** αλγορίθμους **πολυωνυμικού χρόνου**, για **όλα τα στιγμιότυπα**.

Στο υπόλοιπο μέρος του κεφαλαίου θα παρουσιάσουμε κάποιες βασικές έννοιες των προσεγγιστικών αλγορίθμων, καθώς και συγκεκριμένους αλγόριθμους, για μερικά πολύ γνωστά NP-hard προβλήματα.

Συμβολισμοί[1]:

- Π: το πρόβλημα
- I: στιγμιότυπο
- $SOL_A(\Pi, I)$: η τιμή της λύσης για το στιγμιότυπο I του προβλήματος Π που επιστρέφει ο αλγόριθμος A
- $OPT(\Pi, I)$: η βέλτιστη τιμή λύσης του προβλήματος Π για το στιγμιότυπο I

Με βάση τα παραπάνω μπορούμε να ορίσουμε το λόγο προσέγγισης (approximation ratio ή approximation factor) ενός προσεγγιστικού αλγορίθμου ως εξής:

Ορισμός 14.3.1. Ένας αλγόριθμος A για ένα πρόβλημα ελαχιστοποίησης Π, έχει λόγο προσέγγισης $\rho_A(n)$, αν για κάθε στιγμιότυπο του προβλήματος I (με $|I| = n$) ισχύει:

$$\frac{SOL_A(I)}{OPT(I)} \leq \rho_A(n)$$

Ο παραπάνω ορισμός μπορεί επίσης να διατυπωθεί ως εξής:

$$OPT(I) \leq SOL_A(I) \leq \rho_A(n) \cdot OPT(I)$$

Η παραπάνω μορφή καθιστά πιο σαφές το νόημα του παράγοντα προσέγγισης σε έναν προσεγγιστικό αλγόριθμο A: δοθέντος ενός στιγμιοτύπου του προβλήματος I μεγέθους n, ο αλγόριθμος επιστρέφει μία λύση $SOL_A(I)$ η οποία είναι χειρότερη (μεγαλύτερη) από τη βέλτιστη (ελάχιστη) λύση, αλλά το πολύ $\rho_A(n)$ φορές χειρότερη. Με άλλα λόγια έχουμε ένα είδος «εγγύησης», ότι για οποιοδήποτε input ο αλγόριθμος αυτός δεν θα μας δώσει μία λύση που θα απέχει πολύ από τη βέλτιστη.

Αντίστοιχα μπορούμε να διατυπώσουμε και τον ορισμό για προβλήματα μεγιστοποίησης:

[1] συνήθως παραλείπουμε το Π, I και A από τους παρακάτω συμβολισμούς

Ορισμός 14.3.2. Ένας αλγόριθμος Α για ένα πρόβλημα μεγιστοποίησης Π, έχει λόγο προσέγγισης $\rho_A(n)$, αν για κάθε στιγμιότυπο του προβλήματος I (με $|I| = n$) ισχύει:

$$\frac{SOL_A(I)}{OPT(I)} \geq \rho_A(n)$$

Τον αλγόριθμο αυτό τον λέμε $\rho_A(n)$ - προσεγγιστικό αλγόριθμο.

Το επόμενο φυσικό ερώτημα είναι, τι ακριβώς κρύβεται πίσω από το $\rho_A(n)$; Έχουμε λοιπόν τις εξής περιπτώσεις:

- $\rho_A(n) = \cup_i O(n^i)$, δηλαδή ο παράγοντας προσέγγισης είναι μία πολυωνυμική συνάρτηση του μεγέθους της εισόδου. Τα προβλήματα που έχουν τέτοιο αλγόριθμο προσέγγισης ανήκουν στην κλάση poly-APX.
- $\rho_A(n) = O(\log n)$, δηλαδή ο παράγοντας προσέγγισης είναι μία λογαριθμική συνάρτηση του μεγέθους της εισόδου. Τα προβλήματα που έχουν τέτοιο αλγόριθμο προσέγγισης ανήκουν στην κλάση log-APX.
- $\rho_A(n) = \rho$, οπότε ο παράγοντας προσέγγισης είναι σταθερός (ανεξάρτητος του μεγέθους της εισόδου). Τα προβλήματα που έχουν τέτοιο αλγόριθμο προσέγγισης ανήκουν στην κλάση APX.

Από τις τρεις παραπάνω περιπτώσεις, η APX είναι η πιο επιθυμητή, αφού μας εξασφαλίζει ότι για οποιαδήποτε (οσοδήποτε μεγάλη) είσοδο, μπορούμε να προσεγγίσουμε μία λύση που διαφέρει κατά ένα συγκεκριμένο παράγοντα από τη βέλτιστη. Αντίστοιχα η log-APX περίπτωση είναι καλύτερη από την poly-APX, αφού -για δεδομένο μέγεθος εισόδου- η λογαριθμική συνάρτηση μας δίνει καλύτερο παράγοντα (μικρότερο) προσέγγισης από μία πολυωνυμική.

Προσέξτε τώρα, ότι η κλάση APX περιέχει προβλήματα βελτιστοποίησης που επιδέχονται σταθερό παράγοντα προσέγγισης. Έτσι, λέγοντας για παράδειγμα ότι το Vertex Cover επιδέχεται έναν 2-προσεγγιστικό αλγόριθμο, εννοούμε ότι επιδέχεται έναν αλγόριθμο που επιστρέφει ένα VC με το πολύ το διπλάσιο αριθμό κόμβων από ένα ελάχιστο VC. Ιδανικά θα θέλαμε για κάθε παράγοντα προσέγγισης ρ να μπορούμε να βρούμε έναν ρ-προσεγγιστικό αλγόριθμο. Με άλλα λόγια θα ήταν πολύ χρήσιμο αν μπορούσαμε να κατασκευάσουμε μία οικογένεια προσεγγιστικών αλγορίθμων, έναν για κάθε παράγοντα προσέγγισης ρ, για το πρόβλημα.

Πιο τυπικά θα ορίσουμε την οικογένεια αυτή των αλγορίθμων σαν ένα προσεγγιστικό σχήμα (approximation scheme) ως εξής:

Ορισμός 14.3.3. Ο A είναι ένα προσεγγιστικό σχήμα (approximation scheme) για το πρόβλημα Π, αν για είσοδο (I, ε), όπου I είναι ένα στιγμιότυπο και $\varepsilon > 0$ η παράμετρος λάθους έχουμε:

- $OPT(I) \leq SOL_A(I, \varepsilon) \leq (1+\varepsilon) \cdot OPT(I)$, για πρόβλημα ελαχιστοποίησης
- $OPT(I) \geq SOL_A(I, \varepsilon) \geq (1-\varepsilon) \cdot OPT(I)$, για πρόβλημα μεγιστοποίησης

Επίσης λέμε ότι ο A είναι PTAS (Polynomial Time Approximation Scheme) αν για κάθε $\varepsilon > 0$ χρειάζεται πολυωνυμικό χρόνο ως προς το μέγεθος του I. Ο A είναι FPTAS (Fully PTAS) αν για κάθε $\varepsilon > 0$ χρειάζεται πολυωνυμικό χρόνο ως προς το μέγεθος του I και ως προς την τιμή $1/\varepsilon$.

Τα προσεγγιστικά σχήματα PTAS και FPTAS λοιπόν είναι αλγόριθμοι που δέχονται ως είσοδο, πέρα από το στιγμιότυπο του προβλήματος, και έναν φυσικό αριθμό ε που είναι η παράμετρος λάθους και μπορείτε να δείτε πώς σχετίζεται με τον παράγοντα προσέγγισης ρ σε κάθε είδος προβλήματος βελτιστοποίησης. Αν ο αλγόριθμος αυτός τρέχει σε πολυωνυμικό χρόνο ως προς το $|I|$ έχουμε την περίπτωση του PTAS. Παρόλα αυτά, αυτό μπορεί να μην είναι αρκετό: για παράδειγμα μπορεί να έχουμε βρει έναν PTAS που επιλύει το πρόβλημα με πολύ μικρό σφάλμα ε, αλλά ο χρόνος εκτέλεσής του αυξάνει εκθετικά όσο μειώνεται το ε (δηλαδή όσο αυξάνεται το $1/\varepsilon$). Για αυτό το λόγο θεωρούμε την οικογένεια FPTAS όπου προσθέτουμε την απαίτηση ο αλγόριθμος να είναι πολυωνυμικός και ως προς $1/\varepsilon$. Έτσι, ναι μεν χρειαζόμαστε παραπάνω χρόνο για περισσότερη ακρίβεια προσέγγισης, αλλά μόνο πολυωνυμικά περισσότερο.

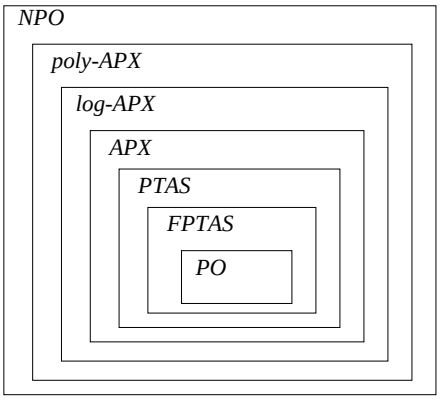
Από την παραπάνω συζήτηση πρέπει να έχει γίνει κατανοητό ότι τα FPTAS συνιστούν (μάλλον) την καλύτερη κατηγορία προσεγγιστικών αλγορίθμων. Δυστυχώς αρκετά προβλήματα δεν επιδέχονται FPTAS (όπως για παράδειγμα το Traveling Salesperson Problem), ενώ για αρκέτα άλλα δε γνωρίζουμε αν επιδέχονται ή όχι (όπως για παράδειγμα το Vertex Cover).

Παρατήρηση. Προσέξτε ότι στα FPTAS πρέπει ο αλγόριθμος να είναι πολυωνυμικός ως προς το $1/\varepsilon$. Αυτό είναι πολύ σημαντικό, διότι αν ο αλγόριθμος ήταν πολυωνυμικός ως προς το μήκος της αναπαράστασης του $1/\varepsilon$, θα μπορούσαμε να επιλέξουμε κατάλληλα το ε ώστε η προσεγγιστική λύση να είναι μεταξύ του OPT και του OPT+1. Αυτό όμως πρακτικά θα σήμαινε (για προβλήματα με output ακέραιες τιμές) ότι μπορούμε να λύσουμε ακριβώς το πρόβλημα (δηλαδή ότι το πρόβλημα είναι στην PO). Περισσότερα για αυτό το θέμα στην ενότητα που μελετά το πρόβλημα Discrete Knapsack.

Οι σχέσεις αυτές μεταξύ των διαφόρων κατηγοριών προσεγγιστικών αλγορίθμων απεικονίζονται υπό τη μορφή ιεραρχίας στο Σχήμα 14.1.

Τέλος, μερικά χαρακτηριστικά προβλήματα για κάθε κλάση είναι τα εξής:

- NPO: Traveling Salesman Problem
- poly-APX: Clique



Σχήμα 14.1: Κλάσεις προβλημάτων βελτιστοποίησης.

- log-APX: Set Cover
- APX: Vertex Cover
- PTAS: Bin Packing
- FPTAS: Discrete Knapsack
- PO: Matching

Τα παραπάνω προβλήματα έχουν τοποθετηθεί στη μικρότερη από τις κλάσεις που είναι μέχρι στιγμής γνωστό ότι ανήκουν. Για αρκετά από αυτά έχουμε ενδείξεις (ή και αποδείξεις με την προϋπόθεση φυσικά ότι P $\neq$ NP) ότι δεν είναι πιο κάτω στην ιεραρχία.

14.4 Αντιπροσωπευτικοί προσεγγιστικοί αλγόριθμοι

Προχωράμε τώρα στη μελέτη τριών γνωστών NP-hard προβλημάτων βελτιστοποίησης και εξετάζουμε σε ποιο βαθμό επιδέχονται προσεγγιστικό αλγόριθμο.

14.4.1 Vertex Cover Problem

Όπως αναφέραμε και πιο πριν το πρόβλημα του Vertex Cover επιδέχεται έναν 2-προσεγγιστικό αλγόριθμο.

Πριν προχωρήσουμε στην παρουσίαση του προσεγγιστικού αλγορίθμου, θα παρουσιάσουμε μία γενική στρατηγική για τη σχεδίαση προσεγγιστικών αλγορίθμων. Η στρατηγική αυτή έχει δύο βασικά βήματα:

1. Βρες ένα κάτω φράγμα L για τη βέλτιστη λύση ($L \leq OPT$)

2. Σχεδίασε έναν αλγόριθμο που εκμεταλλεύεται το φράγμα αυτό, επιστρέφοντας μία λύση κόστους $SOL \leq \rho \cdot L \leq \rho \cdot OPT$

Στην περίπτωση του VC το κάτω φράγμα αυτό θα μας προκύψει από ένα οποιοδήποτε maximal matching. Ας θυμηθούμε λοιπόν κάποια πράγματα σχετικά με matchings:

Ορισμός 14.4.1. Δίνεται ένας γράφος $G = (V, E)$. Ένα ταίριασμα (matching) είναι ένα υποσύνολο των ακμών του $M \subseteq E$ τέτοιο ώστε ποτέ δύο ακμές να μην έχουν κοινό άκρο.

- Maximal matching είναι ένα matching στο οποίο δεν μπορούμε να προσθέσουμε καμία επιπλέον ακμή. Δηλαδή αν προσθέσουμε οποιαδήποτε άλλη ακμή, παύει να είναι matching. Ένα τέτοιο matching είναι πολύ εύκολο να βρεθεί σε πολυωνυμικό χρόνο με έναν greedy αλγόριθμο ο οποίος προσθέτει ακμές στο M, ώσπου να μην μπορούν να προστεθούν άλλες.

- Maximum matching είναι ένα mathcing το οποίο έχει μέγιστη πληθικότητα, δηλαδή περιέχει τις περισσότερες δυνατές ακμές. Ένα maximum matching είναι και maximal, ενώ το αντίστροφο δεν ισχύει αναγκαστικά. Ένα maximum matching μπορεί επίσης να βρεθεί σε πολυωνυμικό χρόνο, μέσω αναγωγής στο πρόβλημα max-flow, που μελετήσαμε σε προηγούμενο κεφάλαιο.

Ο προσεγγιστικός αλγόριθμος για το VC είναι πολύ απλός και έχει ως εξής:

Αλγόριθμος 14.1 2-προσεγγιστικός αλγόριθμος για VC

Βρες ένα maximal matching M και επίστρεψε το σύνολο V' των κορυφών όλων των ακμών του.

Θεώρημα 14.4.2. *Το σύνολο V' καλύπτει όλες τις ακμές του γράφου, είναι δηλαδή όντως VC.*

Απόδειξη. Προφανώς οι κορυφές του V' καλύπτουν όλες τις ακμές του matching M. Άρα χρειάζεται να εξετάσουμε μόνο τις ακμές εκτός του M. Έστω λοιπόν ότι υπάρχει κάποια ακμή (που δεν ανήκει στο matching M) που δεν καλύπτεται από κανέναν κόμβο του V'. Αυτό όμως σημαίνει ότι η ακμή αυτή μπορεί να προστεθεί στο M, μιας και δε θα έχει κοινό άκρο με καμία από τις ακμές του matching, δημιουργώντας ένα νέο matching με μεγαλύτερο μέγεθος. Αυτό μας οδηγεί σε άτοπο, αφού το M είναι maximal matching. □

Είδαμε λοιπόν ότι όντως ο παραπάνω αλγόριθμος μας επιστρέφει ένα VC. Ας επαληθεύσουμε τώρα ότι το VC αυτό έχει μέγεθος το πολύ 2 φορές μεγαλύτερο από το βέλτιστο (ελάχιστο).

Θεώρημα 14.4.3. *Ο παραπάνω αλγόριθμος είναι 2-προσεγγιστικός.*

Απόδειξη. Όπως αναφέραμε και πιο πριν, σκοπός μας είναι να χρησιμοποιήσουμε το μέγεθος του maximal matching $|M|$ ως ένα κάτω φράγμα για το OPT του VC. Όντως ισχύει $|M| \leq OPT$, αφού κάθε VC πρέπει να διαλέξει τουλάχιστον ένα άκρο από κάθε ακμή του M: αν για κάποια ακμή δε διαλέξει κανένα άκρο της, τότε αυτή η ακμή δε θα καλύπτεται[2]. Παρατηρήστε τώρα ότι $|V'| = 2 \cdot |M|$, οπότε εύκολα προκύπτει:

$$SOL = |V'| = 2 \cdot |M| \leq 2 \cdot OPT \Rightarrow SOL \leq 2 \cdot OPT$$

□

Τέλος είναι εύκολο να δούμε ότι ο παραπάνω αλγόριθμος εκτελεί μία φορά τον αλγόριθμο για το maximal matching και είναι επομένως πολυωνυμικού χρόνου.

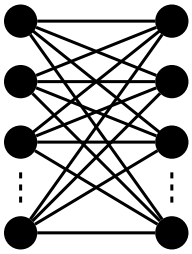
Γίνεται καλύτερα;

Αφού σχεδιάσουμε έναν οποιοδήποτε αλγόριθμο και μελετήσουμε την ορθότητα και την αποδοτικότητά του (και στην περίπτωση των προσεγγιστικών αλγορίθμων και το λόγο προσέγγισής τους), το επόμενο βήμα είναι να αναρωτηθούμε κατά πόσον μπορούμε να πετύχουμε κάτι καλύτερο. Στην περίπτωση των προσεγγιστικών αλγορίθμων αυτό μεταφράζεται στο «κατά πόσον μπορούμε να πετύχουμε καλύτερη προσέγγιση», που με τη σειρά του αντιστοιχεί σε τρία διακριτά ερωτήματα:

1. Είναι όντως ο λόγος προσέγγισης του αλγορίθμου τόσο μεγάλος, ή μήπως η ανάλυσή μας (το Θεώρημα 14.4.3 δηλαδή) δεν είναι τόσο ακριβής (tight); Με άλλα λόγια, υπάρχει είσοδος για την οποία ο αλγόριθμος επιστρέφει μία λύση που είναι 2 φορές χειρότερη από τη βέλτιστη; (κρατάμε το κάτω φράγμα του OPT και τον αλγόριθμο και επανεξετάζουμε την ανάλυσή του)

2. Είναι καλός ο αλγόριθμός μας, ή μήπως, χρησιμοποιώντας το ίδιο κάτω φράγμα για το OPT, θα μπορούσαμε να σχεδιάσουμε έναν πιο έξυπνο

[2] αυτό ισχύει για κάθε matching, άρα και για το maximal, και για κάθε VC άρα και για το ελάχιστο



Σχήμα 14.2: Ένας πλήρης διμερής γράφος της οικογένειας $K_{n,n}$

αλγόριθμο. Στο παράδειγμά μας, μήπως, ας πούμε, θα μπορούσαμε να μην επιλέξουμε όλες τις κορυφές του matching αλλά κάποιες μόνο από αυτές; (κρατάμε το κάτω φράγμα του OPT και επανεξετάζουμε τον αλγόριθμο)

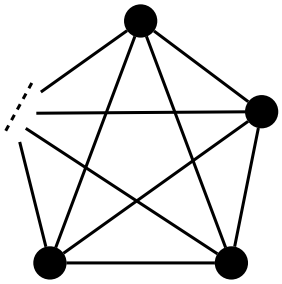
3. Μήπως τελικά το κάτω φράγμα που χρησιμοποιήσαμε δεν είναι και τόσο καλό και πρέπει να επανεξετάσουμε την όλη προσέγγισή μας στο πρόβλημα;

Από τα δύο παραπάνω ερωτήματα τα δύο πρώτα είναι συνήθως εύκολα να απαντηθούν, ενώ το τρίτο όχι και τόσο.

Ας εστιάσουμε όμως στο πρόβλημα του VC και ας προσπαθήσουμε να απαντήσουμε στο πρώτο ερώτημα: είναι δυνατόν με μία καλύτερη ανάλυση να έχουμε καλύτερο λόγο προσέγγισης;

Η απάντηση είναι όχι και αυτό το δείχνουμε ως εξής: βρίσκουμε μία άπειρη οικογένεια στιγμιοτύπων του προβλήματος (εδώ μία οικογένεια γράφων) για την οποία η λύση που επιστρέφει ο αλγόριθμος είναι όντως 2 φορές χειρότερη από τη βέλτιστη. Μία τέτοια οικογένεια παραδειγμάτων ονομάζεται tight και η σημασία τους για την ανάλυση των προσεγγιστικών αλγορίθμων είναι τεράστια. Είναι δε αναγκαίο να είναι άπειρη η οικογένεια (ή τουλάχιστον να είναι δύσκολο να εντοπίσουμε κάθε μέλος της), αλλιώς ο προσεγγιστικός αλγόριθμος θα μπορούσε ανάλογα με την είσοδο να επιλέγει τη συμπεριφορά του.

Παράδειγμα 14.4.4. Θεωρήστε την άπειρη οικογένεια από στιγμιότυπα της μορφής του σχήματος (14.2), δηλαδή από πλήρεις διμερείς γράφους $K_{n,n}$. Μιας και οποιοδήποτε maximal matching στο γράφο αυτό έχει μέγεθος n, ο αλγόριθμος θα επιστρέψει πάντα ένα VC μεγέθους $2n$. Όμως το βέλτιστο VC έχει μέγεθος n (επιλέγοντας για παράδειγμα όλες τις κορυφές του ενός μέρους του γράφου) και έτσι η λύση που επιστρέφει ο αλγόριθμος είναι όντως 2 φορές χειρότερη από τη βέλτιστη.



Σχήμα 14.3: Ένας πλήρης γράφος της οικογένειας K_{2n+1}

Ας προχωρήσουμε τώρα στο δεύτερο ερώτημα: μήπως είναι υπερβολικό να βάλουμε τον αλγόριθμό μας να επιστρέψει όλους τους κόμβους των ακμών ενός maximal matching; Και πάλι η απάντηση είναι όχι, όπως προκύπτει συνήθως με χρήση ενός ακόμα κατάλληλα επιλεγμένου παραδείγματος.

Παράδειγμα 14.4.5. Θεωρήστε την άπειρη οικογένεια K_{2n+1} πλήρων γράφων με περιττό αριθμό κόμβων, όπως αυτός του Σχήματος (14.3). Στην περίπτωση αυτή ένα maximal matching έχει μέγεθος n και έτσι το μέγεθος του επιστρεφόμενου VC θα είναι $2n$. Όμως είναι εύκολο να επαληθεύσουμε ότι και το ελάχιστο VC έχει μέγεθος $2n$. Έτσι ο αλγόριθμός μας θα επιστρέψει μία βέλτιστη λύση. Το παράδειγμα αυτό καθιστά σαφές ότι δεν μπορούμε να βελτιώσουμε τον αλγόριθμο κάνοντας μία πιο «έξυπνη» επιλογή κορυφών (για παράδειγμα μόνο τις μισές κορυφές του M), αφού υπάρχουν περιπτώσεις εισόδων (όπως ο K_{2n+1}) όπου χρειάζεται να πάρουμε όλες τις κορυφές. Με άλλα λόγια, εφόσον ο αλγόριθμος αυτός για κάποιες εισόδους επιστρέφει το ελάχιστο VC, αν προσπαθούσαμε να ελαττώσουμε τον αριθμό των επιστρεφόμενων κορυφών, τότε για τις συγκεκριμένες αυτές εισόδους η επιστρεφόμενη λύση δε θα ήταν καν VC. Αν π.χ. προσπαθούσαμε να βρούμε αλγόριθμο που να επιστρέφει λύση κόστους $SOL \leq 3/2|M|$ αυτό δε θα ήταν δυνατό.

Το τελευταίο ερώτημα, αν και πολύ σημαντικό, συνήθως είναι δύσκολο να απαντηθεί. Για παράδειγμα για το VC, ξέρουμε ότι υπάρχει αλγόριθμος με λόγο προσέγγισης $2-\Theta(\frac{1}{\sqrt{\log n}})$ (χάρη σε μία πρόσφατη εργασία του Καρακώστα (George Karakostas) που βασίζεται σε τεχνικές γραμμικού προγραμματισμού), αλλά δεν ξέρουμε αν το πρόβλημα επιδέχεται προσεγγιστικό αλγόριθμο με σταθερό λόγο προσέγγισης < 2 ή PTAS.

Παρατήρηση. Παρόλο που τα προβλήματα VC, Clique και Independent Set μπορούν να αναχθούν πολυωνυμικά το ένα στο άλλο, αυτό δε σημαίνει ότι ένας πολυωνυμικός προσεγγιστικός αλγόριθμος για το VC μας δίνει έναν αντίστοιχο αλγόριθμο για κάποιο από τα άλλα δύο (θυμηθείτε για παράδειγμα ότι το Clique είναι στην κλάση poly-APX).

14.4.2 Discrete Knapsack Problem

Στο κεφάλαιο του Δυναμικού Προγραμματισμού είχαμε αναφερθεί στο πρόβλημα Discrete Knapsack, στο οποίο πρακτικά θέλουμε να επιλέξουμε από ένα σύνολο από αντικείμενα αυτά που χωράνε σε ένα σακίδιο πεπερασμένης χωρητικότητας, έτσι ώστε να μεγιστοποιήσουμε τη συνολική αξία του περιεχομένου του σακιδίου (κάθε αντικείμενο μπορεί να έχει διαφορετική αξία). Είχαμε επίσης αναφέρει ότι το πρόβλημα αυτό επιδέχεται έναν ψευδοπολυωνυμικό αλγόριθμο και είχαμε πει ότι τέτοια προβλήματα συνήθως επιδέχονται και καλό προσεγγιστικό αλγόριθμο. Στο μέρος αυτό θα αναφερθούμε με περισσότερη λεπτομέρεια στη σχέση αυτή των προσεγγιστικών αλγορίθμων με τους ψευδοπολυωνυμικούς, χρησιμοποιώντας ως αντιπροσωπευτικό πρόβλημα το Discrete Knapsack.

Ψευδοπολυωνυμικοί αλγόριθμοι

Για να καταλάβουμε καλύτερα τι είναι ένας ψευδοπολυωνυμικός αλγόριθμος θα χρειαστεί να κάνουμε μία διάκριση ανάμεσα στα συστατικά στοιχεία ενός στιγμιοτύπου εισόδου σε έναν αλγόριθμο. Η είσοδος μπορεί να αποτελείται από αντικείμενα (όπως για παράδειγμα σύνολα, γράφους) πλήθους n και από αριθμούς, π.χ. m.

Στην πρώτη περίπτωση είναι σαφές τι εννοούμε «μέγεθος εισόδου n»: τον πληθάριθμο του συνόλου, ή το μέγεθος του συνόλου κορυφών ή ακμών του γράφου, αντίστοιχα, που φυσικά είναι πολυωνυμικά συσχετισμένο με το μήκος της αναπαράστασης του συνόλου.

Τι γίνεται όμως όταν η επιπλέον είσοδος περιέχει και αριθμούς; Σε αυτή την περίπτωση θεωρούμε ως μέγεθος της εισόδου το μέγεθος της δυαδικής αναπαράστασης των αριθμών. Για παράδειγμα, αν δοθεί ως είσοδος ένας αριθμός m, τότε λέμε ότι ο αλγόριθμος είναι πολυωνυμικός αν χρειάζεται χρόνο $O(p(|m|)) = O(p(\log m))$, όπου p ένα πολυώνυμο. Αυτό σημαίνει ότι αν γράψουμε ένα πρόγραμμα το οποίο διαβάζει (δέχεται ως είσοδο) έναν αριθμό m και τρέχει ένα for loop (με σώμα πολυωνυμικού χρόνου ως προς n) m φορές, δεν μπορούμε να πούμε ότι ο χρόνος εκτέλεσής του είναι πολυωνυμικός: λέμε ότι είναι ψευδοπολυωνυμικός, δηλαδή τρέχει σε χρόνο $p(n \cdot m)$. Αν εκφράσουμε το χρόνο εκτέλεσης ως συνάρτηση του μεγέθους της εισόδου **σε unary αναπαράσταση του** m (ο αριθμός 42 για παράδειγμα αναπαρίσταται με 43 άσσους[3]) τότε λέμε ότι ένας ψευδοπολυωνυμικός αλγόριθμος είναι πολυωνυμικός ως προς το μέγεθος της unary αναπαράστασης του m (που είναι το ίδιο με πολυωνυμικός ως προς την τιμή του m). Ένας ψευδοπολυωνυμικός αλγόριθμος έχει ικανοποιητική απόδοση για «μικρά» m. Συγκεκριμένα, για στιγμιότυπα

[3]λόγω της σύμβασης ότι 1 σε unary είναι το 0 σε δεκαδικό

όπου $m = O(p(n))$, ο αλγόριθμος είναι πραγματικά πολυωνυμικός ως προς το μήκος της εισόδου.

Ο λόγος που για πολλά προβλήματα Δυναμικού Προγραμματισμού έχουμε ψευδοπολυωνυμικούς αλγορίθμους είναι ότι η επίλυσή τους βασίζεται συνήθως στη μέθοδο του πίνακα (που γεμίζει με for loops) του οποίου το μέγεθος συχνά καθορίζεται από την τιμή των αριθμών που δίνονται στην είσοδο.

Strong NP-hardness

Είδαμε στα προηγούμενα κεφάλαια πως στη Θεωρητική Πληροφορική θεωρούμε εν γένει εύκολα τα προβλήματα που λύνονται σε πολυωνυμικό χρόνο και δύσκολα τα NP-hard προβλήματα.

Οι ψευδοπολυωνυμικοί αλγόριθμοι είναι πρακτικά αποδοτικοί για μικρές τιμές των εισόδων, αλλά γενικά συνιστούν μία σαφώς ασθενέστερη (ως προς την αποδοτικότητα) κλάση αλγορίθμων σε σχέση με τους πολυωνυμικούς. Σε πλήρη αντιστοιχία λοιπόν με αυτή την ασθενέστερη έννοια αποδοτικότητας, θα ορίσουμε και μία πιο ισχυρή έννοια «δυσκολίας» (intractability), την strong NP-hardness.

Ορισμός 14.4.6. Ένα πρόβλημα καλείται strongly NP-hard αν κάθε πρόβλημα του NP μπορεί να αναχθεί σε αυτό σε πολυωνυμικό χρόνο, με τους αριθμούς στην είσοδό του να είναι γραμμένα σε unary αναπαράσταση.

Διαισθητικά ο παραπάνω ορισμός μας λέει ότι τα strongly NP-hard προβλήματα παραμένουν δύσκολα να λυθούν ακόμα και αν αφήσουμε τον αλγόριθμο να εκτελεστεί για πολυωνυμικό ως προς m χρόνο, δηλαδή αρκετά (εκθετικά) περισσότερο από πολυωνυμικό ως προς $\log m$ χρόνο.

Παρατηρήστε τώρα ότι αν μπορούσαμε να λύσουμε κάποιο strongly NP-hard πρόβλημα σε ψευδοπολυωνυμικό χρόνο, τότε θα μπορούσαμε να λύσουμε κάθε πρόβλημα του NP σε πολυωνυμικό χρόνο. Με άλλα λόγια, εκτός και αν P=NP, κανένα strongly NP-hard πρόβλημα δεν επιδέχεται ψευδοπολυωνυμικό αλγόριθμο.

Πριν προχωρήσουμε στη μελέτη του προσεγγιστικού αλγορίθμου για το Discrete Knapsack θα διατυπώσουμε ένα θεώρημα που καταδεικνύει τη σχέση μεταξύ των FPTAS και των ψευδοπολυωνυμικών αλγορίθμων.

Θεώρημα 14.4.7. *Για ένα πρόβλημα ελαχιστοποίησης Π, αν για κάθε στιγμιότυπο εισόδου I,*

- *$OPT < p(|I_u|)$ για κάποιο πολυώνυμο p, όπου I_u είναι η unary αναπαράσταση του I και*

- *η αντικειμενική συνάρτηση παίρνει ακέραιες τιμές*

τότε, αν το Π επιδέχεται FPTAS, τότε το Π επιδέχεται και ψευδοπολυωνυμικό αλγόριθμο.

Πόρισμα 14.4.8. Ένα strongly NP-hard πρόβλημα βελτιστοποίησης με τις παραπάνω ιδιότητες δεν επιδέχεται FPTAS, εκτός αν P=NP.

Το πόρισμα προκύπτει άμεσα από το Θεώρημα 14.4.7 και από την παραπάνω παρατήρηση ότι τα strongly NP-hard προβλήματα δεν επιδέχονται ψευδοπολυωνυμικό αλγόριθμο.

Ας δούμε τώρα τι συμβαίνει με το πρόβλημα Discrete Knapsack. Για τους τυπικούς ορισμούς μπορείτε να ανατρέξετε στο κεφάλαιο του Δυναμικού Προγραμματισμού. Εδώ θα παρουσιάσουμε έναν άλλο αλγόριθμο Δυναμικού Προγραμματισμού που χρησιμοποιεί τη μέθοδο του πίνακα, αλλά διαφέρει από τον αλγόριθμο που παρουσιάσαμε στο προηγούμενο κεφάλαιο στο εξής: αντί να διατηρούμε έναν πίνακα $K : n \times W$ όπου το στοιχείο $A[i, w]$ περιέχει το υποσύνολο του του $\{x_1, \ldots, x_i\}$ βάρους w με τη μέγιστη αξία, διατηρούμε τον πίνακα $W : n \times n \cdot P$, όπου $P = \max_i p_i$ και η θέση $W[i, p]$ περιέχει το υποσύνολο του $\{x_1, \ldots, x_i\}$ αξίας p με το ελάχιστο βάρος[4].

Ο νέος αλγόριθμος είναι ο 14.2. Παρατηρούμε ότι ο χρόνος εκτέλεσής του είναι $O(n^2P)$, όπου P είναι ο μεγαλύτερος από τους αριθμούς που δέχεται ο αλγόριθμος ως είσοδο, δηλαδή ο αλγόριθμος είναι ψευδοπολυωνυμικός.

[4]προφανώς η αξία μίας οποιαδήποτε λύσης δεν μπορεί να υπερβαίνει το $n \cdot P$

Αλγόριθμος 14.2 Δυναμικός Προγραμματισμός για Discrete Knapsack

function DKnap(n:**integer**; profit: **array**; weight: **array**) :**integer**;

```
var P: integer; (*P = max_i{profits[i]}*)
    W: array[1..n][1..n * P];
    i, j: integer;

begin
    for i := 1 to P do
        if i = profit[i] then W[1, i] = weight[i];
        else W[1, i] := ∞;

    for i := 1 to n do
        for j := 1 to n * P do
            W[i + 1, j] := min{W[i, j], weight[i + 1] + W[i, j − profit[i + 1]]};

    DKnap := max_j{W[n, j] ≤ Σ_i weight[i]};
end.
```

Ας δούμε τώρα πώς μπορούμε να αλλάξουμε λίγο τον παραπάνω αλγόριθμο προκειμένου να πάρουμε ένα FPTAS. Η ιδέα είναι να αγνοήσουμε μερικά λιγότερο σημαντικά ψηφία των p_i, έτσι ώστε τελικά τα p_i, άρα και το P, να είναι πολυωνυμικά ως προς το μέγεθος της εισόδου (n) και ως προς την παράμετρο σφάλματος $1/\varepsilon$. Οπότε εργαζόμαστε όπως φαίνεται στον αλγόριθμο 14.3. Μπορεί να αποδειχθεί ότι για τη λύση SOL που επιστρέφει ο αλγόριθμος αυτός ισχύει:

$$SOL \geq (1 - \varepsilon)OPT$$

Η πολυπλοκότητα του αλγορίθμου είναι $O(n^2 \cdot P') = O\left(\left\lceil \frac{n^3}{\varepsilon} \right\rceil\right)$, καθώς $P' = \left\lceil \frac{P}{K} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$. Επομένως ο αλγόριθμος είναι FPTAS.

Αλγόριθμος 14.3 FPTAS για Discrete Knapsack

1. Δοθέντος του ε όρισε το $K = \frac{\varepsilon \cdot P}{n}$
2. Θέσε τα νέα κέρδη στις τιμές $p'_i = \left\lceil \frac{p_i}{K} \right\rceil$
3. Εκτέλεσε τον αλγόριθμο 14.2 με κέρδη τα p'_i

14.4.3 Traveling Salesman Problem

Στο τελευταίο αυτό μέρος θα εστιάσουμε στο πρόβλημα του πλανόδιου πωλητή, το οποίο θα δείξουμε ότι δεν επιδέχεται κανένα χρήσιμο παράγοντα προσέγγισης. Για να δείξουμε ότι ένα πρόβλημα βελτιστοποίησης Π είναι δύσκολο να προσεγγιστεί γενικά υπάρχουν δύο μέθοδοι:

- Οι gap-introducing reductions, στις οποίες ανάγουμε ένα NP-complete πρόβλημα απόφασης Π' στο πρόβλημα Π.
- Οι gap-preserving reductions, στις οποίες ανάγουμε ένα άλλο δύσκολα προσεγγίσιμο πρόβλημα Π' στο πρόβλημα Π.

Ας δούμε τώρα την εφαρμογή μίας gap-introducing reduction από το πρόβλημα Hamilton Cycle στο TSP (θυμηθείτε ότι με αντίστοιχη αναγωγή δείξαμε ότι το TSP είναι NP-complete).

Θεώρημα 14.4.9. *Για κάθε πολυωνυμικά υπολογίσιμη συνάρτηση $\rho(n)$, το TSP δεν μπορεί να προσεγγιστεί με παράγοντα προσέγγισης $\rho(n)$, εκτός αν P=NP (όπου n το πλήθος των κόμβων του γράφου).*

Απόδειξη. Ας υποθέσουμε ότι υπάρχει ένας $\rho(n)$-προσεγγιστικός αλγόριθμος για το TSP. Θα δείξουμε πώς ο αλγόριθμος αυτός μπορεί να χρησιμοποιηθεί για να επιλύσουμε το πρόβλημα Hamilton Cycle. Η κεντρική ιδέα είναι μία αναγωγή από ένα Hamilton Cycle στιγμιότυπο σε ένα TSP στιγμιότυπο, που μετασχηματίζει έναν γράφο G με n κορυφές, σε έναν πλήρη γράφο G' με n κορυφές, έτσι ώστε:

- αν ο G έχει κύκλο Hamilton, τότε το κόστος της βέλτιστης λύσης στον TSP είναι n και
- αν ο G δεν έχει κύκλο Hamilton, τότε το κόστος της βέλτιστης λύσης στον TSP είναι $> \rho(n) \cdot n$

Παρατηρήστε τώρα ότι μπορούμε να αποφανθούμε για το αν ο G έχει κύκλο ή όχι, εκτελώντας τον $\rho(n)$-προσεγγιστικό αλγόριθμο για το TSP: στην πρώτη περίπτωση, ο προσεγγιστικός αλγόριθμος θα επιστρέψει μία λύση που θα είναι $\leq \rho(n) \cdot OPT = \rho(n) \cdot n$, ενώ στη δεύτερη θα επιστρέψει μία λύση που θα είναι $\geq OPT > \rho(n) \cdot n$. Ανάλογα λοιπόν με την τιμή που επιστρέφεται για τον G' μπορούμε να καταλάβουμε τι ισχύει για τον G.

Η αναγωγή τώρα προκειμένου να πετύχουμε τα ως άνω είναι πολύ απλή: αναθέτουμε βάρος 1 σε κάθε ακμή του αρχικού γράφου G και βάρος $\rho(n) \cdot n$ σε όλες τις υπόλοιπες ακμές, ώστε να αποκτήσουμε τον πλήρη γράφο G'. Η αναγωγή γίνεται προφανώς σε πολυωνυμικό χρόνο. Παρατηρήστε τώρα ότι,

αν ο G έχει κύκλο Hamilton, τότε η βέλτιστη λύση του TSP είναι προφανώς αυτός ο κύκλος, κόστους n. Αντίθετα, σε περίπτωση που ο G δεν έχει κύκλο Hamilton, η βέλτιστη λύση στο TSP θα περνά από τουλάχιστον μία ακμή βάρους $\rho(n) \cdot n$ και έτσι το συνολικό της κόστος θα είναι $OPT > \rho(n) \cdot n$.

Κεφάλαιο 15

Παράρτημα

15.1 Εισαγωγή-Ιστορική Αναδρομή

15.2 Υπολογιστικά Προβλήματα και Τυπικές Γλώσσες

newpage1

15.3 Ντετερμινιστικές Μηχανές Turing

newpage2

15.4 Υπολογιστότητα

15.4.1 Μη-Υπολογιστότητα: Το πρόβλημα Τερματισμού

15.5 Μη Ντετερμινιστικές Μηχανές Turing

15.6 Υπολογιστική Πολυπλοκότητα

15.7 Χρονική Πολυπλοκότητα

15.7.1 Ντετερμινιστική Χρονική Πολυπλοκότητα

newpage4

15.7.2 Μη Ντετερμινιστική Χρονική Πολυπλοκότητα

newpage6

15.8 Αναγωγή και Πληρότητα

15.9 NP-Πληρότητα

15.9.1 NP-Πλήρη Προβλήματα

newpage7

15.10 Άλλες Κλάσεις Πολυπλοκότητας

15.10.1 Χωρική Πολυπλοκότητα

15.10.2 Πολυπλοκότητα Συναρτήσεων

15.10.3 Συμπληρωματικές Κλάσεις Πολυπλοκότητας

15.10.4 Κλάσεις Πολυπλοκότητας για Πιθανοτικούς Αλγόριθμους

newpage9

15.10.5 Πλειοψηφικοί ποσοδείκτες

15.10.6 Διαλογικά Συστήματα Απόδειξης

15.11 Βιβλιογραφία

15.12 Ανοικτά Ερευνητικά Προβλήματα

15.13 Ασκήσεις