Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών

Αλγόριθμοι & Πολυπλοκότητα

$2^{η}$ Γραπτή Σειρά Ασκήσεων

Άπληστοι Αλγόριθμοι - Δυναμικός Προγραμματισμός

Δήμος Δημήτρης (031 17 165)
7ο Εξάμηνο - Ροή Λ
Φθινόπωρο 2020

1 Άσκηση 1: Δίσκοι και Σημεία

Για την διευκόλυνση της περιγραφής του προβλήματος, θεωρούμε πως η γραμμή l είναι οριζόντια και τα n σημεία βρίσκονται στο επίπεδο εκατέρωθέν της ή πάνω της.

1.1 Συμβολισμοί:

- p_{edge} : στοιχείο που αν καλυφθει από την αριστερή περιφέρεια δίσκου, δεν μένει ακάλυπτο κανένα στοιχείο αριστερότερα του δίσκου.
- d_{edge} : δίσκος που καλύπτει με την αριστερή περιφέρειά του το p_{edge}.
- Αν X αλγόριθμος, τότε X(A) η λύση του αλγορίθμου για είσοδο το σύνολο σημείων A.

1.2 Διαίσθηση - Ιδέα Αλγορίθμου

Παρατηρούμε πως το p_{edge} θα καλυφθεί από έναν δίσκο, ο οποίος ενδέχεται να μπορεί να καλύψει και επιπλεόν στοιχεία. Αν αυτός ο δίσκος είναι d_{edge} (καλύπτει δηλαδή το p_{edge} με την αριστερή περιφέρειά του) τότε θα καλύπτει τα περισσότερα δυνατά σημεία που μπορεί. Υποπτευόμαστε, λοιπόν, ότι αν επιλέξουμε αυτόν τον δίσκο να είναι d_{edge} τότε αυτός είναι μέρος μιας βέλτιστης λύσης

Επιλέγοντας τον d_{edge} για να καλύψουμε το p_{edge} καλύπτονται, ενδεχομένως, και κάποια ακόμη σημεία (≥ 0). Όσα σημεία καλύφθηκαν από τον d_{edge} τα «αφαιρούμε» από το επίπεδο, υπό την έννοια ότι παύουν να μας απασχολούν, αφού έχουν καλυφθεί.

Το ίδιο σκεπτικό μπορούμε να εφαρμόσουμε και στα εναπομείναντα σημεία. Να βρούμε το νέο αριστερότερο p_{edge}, να επιλέξουμε τον νέο d_{edge} κ.ο.κ.

Έτσι, επιλέγουμε κάθε φορά έναν δίσκο d_{edge} που είναι μεν αναγκαίος, ώστε να καλύψει το κάθε αριστερότερο p_{edge} στοιχείο, αλλά με τρόπο τέτοιο ώστε να καλύπτει ταυτόχρονα και όσο το δυνατό περισσότερα ακόμα στοιχεία.

1.3 Διατύπωση Αλγορίθμου - Αλγόριθμος A

Παραθέτουμε τον άπληστο αλγόριθμο, χωρίς τις «τεχνικές λεπτομέρειες». Στην ανάλυση υπολογιστικής πολυπλοκότητας, εξηγούμε λεπτομερώς τις κινήσεις που γίνονται.

Αλγόριθμος A:

1. Βρες το p_{edge} κι επίλεξε δίσκο d_{edge} για να το καλύψεις.
2. Αφαίρεσε όσα στοιχεία καλύπτονται από τον δίσκο.
3. Επανάλαβε τα βήματα (1) και (2) έως ότου να μην μείνει κανένα σημείο.

1.4 Ανάλυση Ορθότητας

Θα αποδείξουμε με μαθηματική επαγωγή ότι ∀ σύνολο σημείων S, $A(S) \leq Opt(S)$, όπου Opt αλγόριθμος που επιστρέφει βέλτιστη λύση.

Βάση Επαγωγής: Για S = ∅, τετριμμένα η βέλτιστη λύση είναι $Opt(\emptyset) = A(\emptyset) = 0$ δίσκοι.

Επαγωγική Υπόθεση: Έστω ότι ο αλγόριθμος A επιστρέφει μια βέλτιστη λύση για το πρόβλημα του συνόλου σημείων $F \subset S$. Δηλαδή ∀ $F \subset S$, $A(F) \leq Opt(F)$.

Επαγωγικό Βήμα: Συμβολίζουμε με F_d το σύνολο των σημείων που απομένουν στο πρόβλημα αν αφαιρέσουμε όσα καλύπτονται από τον δίσκο d.

Η λύση που παράγει ο A είναι η: $\mathrm{A(S)} = 1 + \mathrm{A}(F_{d_{edge}})$
Η λύση που παράγει ο Opt είναι η: $\mathrm{Opt(S)} = 1 + \min_d\{Opt(\mathrm{F}_d)\}$, όπου d ο δίσκος που χρησιμοποιεί για την κάλυψη του p_{edge}.

Ακόμη, ο d καλύπτει **το πολύ** τόσα σημεία όσα ο d_{edge}, άρα οι δίσκοι που θα χρησιμοποιήσει ο Opt για να καλύψει τα υπόλοιπα σημεία θα είναι όσοι θα χρησιμοποιούσε αν επέλεγε τον d_{edge} ή περισσότεροι. Δηλαδή, $\mathrm{Opt}(F_d) \geq Opt(\mathrm{F}_{d_{edge}}) \implies \mathrm{Opt}(F_{d_{edge}}) \leq \min_d\{Opt(\mathrm{F}_d)\}$. Επομένως:

$$\begin{aligned} A(S) &= 1 + A(F_{d_{edge}}) \\ &\leq 1 + Opt(F_{d_{edge}}) \\ &\leq 1 + \min_d\{Opt(F_d)\} = Opt(S) \end{aligned}$$

Τελικά, ο αλγόριθμος A επιστρέφει μια λύση για το πρόβλημα του συνόλου σημείων S που είναι τουλάχιστον τόσο καλή όσο του αλγορίθμου Opt. Άρα **είναι βέλτιστος**.

1.5 Ανάλυση Υπολογιστικής Πολυπλοκότητας

Θεωρούμε ότι η ευθεία l είναι οριζόντια και ταυτίζεται με τον άξονα $x'x$. Σύμφωνα με τον αλγόριθμο Α ακολουθούμε τα εξής βήματα:

1. Ταξινομούμε τα σημεία κατά αύξουσα τετμημένη, με κόστος O(nlogn).

2. Επιλέγουμε το σημείο, έστω (x, y), με τη μικρότερη τετμημένη.

3. Υποθέτουμε ότι (x, y) = p_{edge}.

   Έστω k η τετμημένη του σημείου τομής της περιφέρειας του d_{edge} με την ευθεία l. Θα είναι k = x + $\sqrt{r^2 - y^2}$ - r.

   Ελέγχουμε κάθε σημείο (α, β) με α ∈ [x, k + r], από αυτά μικρότερης τετμημένης προς αυτά μεγαλύτερης.

   Αν $\sqrt{(2 \cdot k - \alpha + r)^2 + \beta^2} \leq r$, αφαιρούμε το στοιχείο από το σύνολο.

   Αν για κάποιο (α, β) ισχύει $\sqrt{(2 \cdot k - \alpha + r)^2 + \beta^2} > r$, τότε επανάλαμβάνουμε το βήμα (3) με (x, y) = (α, β).

4. Αφού ολοκληρώσαμε το βήμα (3), αυτό σημαίνει ότι βρήκαμε δίσκο d_{edge} και τα στοιχεία που αφαιρέσαμε έχουν καλυφθεί από την αριστερή του μεριά.

5. Διασχίζουμε τα στοιχεία που βρίσκονται στη δεξιά μεριά του d_{edge}, δηλαδή (από αριστερά προς τα δεξιά) όλα τα στοιχεία (γ, δ) ∈ (k + r, k + 2 · r]. Αν $\sqrt{(\gamma - (k + r))^2 + \delta^2} \leq r$, τότε αφαιρούμε το στοιχείο (γιατί καλύπτεται απο τη δεξιά μεριά του d_{edge}). Συνεχίζουμε μέχρι να βρούμε στοιχείο με $\sqrt{(\gamma - (k + r))^2 + \delta^2} > r$.

   - αν βρούμε ένα τέτοιο στοιχείο, έστω (μ, ν), σημαίνει ότι ο d_{edge} δεν το καλύπτει και για την κάλυψή του χρειάζεται νέος d'_{edge}. Αυξάνουμε τον μετρητή δίσκων κατά ένα και επαναλαμβάνουμε τον αλγόριθμο από το βήμα (3) με (x, y) = (μ, ν) [1].
   - αν δεν βρούμε κανένα τέτοιο στοιχείο, τότε προχωράμε στο βήμα (6).

6. Αυξάνουμε τον μετρητή δίσκων κατά ένα.

7. Αν υπάρχουν υπόλοιπα στοιχεία, επαναλαμβάνουμε τον αλγόριθμο από το βήμα 2. Αλλιώς, ολοκληρώνουμε.

Από τα παραπάνω, βλέπουμε πως ο αλγόριθμος (μετά την ταξινόμηση) διατρέχει όλα τα n στοιχεία του αρχικού συνόλου ακριβώς μία φορά. Επομένως, από το βήμα 2 μέχρι το τέλος του αλγορίθμου έχουμε κόστος το πολύ μίας διάσχισης των στοιχείων = $O(n)$.

Τελική υπολογιστική πολυπλοκότητα: $O(nlogn) + O(n) = O(nlogn)$.

[1] Δεν υπάρχουν σημεία αριστερότερα του (μ,ν) που να μην καλύπτει ο d_{edge} και όσα σημεία δεξιότερά του καλύπτει ο d_{edge} δεν χρειάζεται να αφαιρεθούν ακόμα, γιατί θα τα καλύψει και ο d'_{edge}

2 Άσκηση 2: Μεταφορά Δεμάτων

2.1 Ερώτημα (α)

2.1.1 Στοίβαξη με βάση το βάρος

Αντιπαράδειγμα: Έστω τα πακέτα $n_1 = (1, 3), n_2 = (4, 2), n_3 = (2, 10)$

Ο αλγόριθμος στοίβαξης με βάση το βάρος παράγει τη στοίβαξη (n_2, n_3, n_1), που δεν είναι ασφαλής, αφού $w_1 + w_3 > d_2$ και ο αλγόριθμος τοποθετεί τα n_1 και n_3 πάνω από το n_2. Παρόλαυτά, υπάρχει η σωστή στοίβαξη (n_3, n_2, n_1).

2.1.2 Στοίβαξη με βάση την αντοχή

Αντιπαράδειγμα: Έστω τα πακέτα $n_1 = (1, 3), n_2 = (4, 2), n_3 = (2, 10)$

Ο αλγόριθμος στοίβαξης με βάση την αντοχή παράγει τη στοίβαξη (n_3, n_1, n_2), που δεν είναι ασφαλής, αφού $w_2 > d_1$ και ο αλγόριθμος τοποθετεί το n_2 πάνω από το n_1. Παρόλαυτά, υπάρχει η σωστή στοίβαξη (n_3, n_2, n_1).

2.1.3 Στοίβαξη με βάση το άθροισμα βάρους και αντοχής

Ισχυριζόμαστε πως ο αλγόριθμος στοίβαξης με κριτήριο το άθροισμα βάρους και αντοχής οδηγεί σε ασφαλή στοίβαξη των πακέτων, εαν κι εφόσον αυτή υπάρχει. Για να το αποδείξουμε αυτό, αρκεί να αποδείξουμε την παρακάτω πρόταση:

$\exists$ ασφαλής στοίβαξη $\implies$ $\exists$ ασφαλής στοίβαξη με κριτήριο το άθροισμα βάρους και αντοχής

Απόδειξη:

- Αν δεν υπάρχει ασφαλής στοίβαξη, ισχύει το ζητούμενο.
- Έστω ασφαλής στοίβαξη Α. Αν αυτή πληροί το κριτήριο αθροίσματος βάρους αντοχής, τότε ισχύει το ζητούμενο.
- Αν δεν το πληροί, τότε θα δείξουμε ότι υπάρχει μια άλλη στοίβαξη Β που το πληροί.

  Εφόσον η Α δεν είναι στοιβαγμένη με το εν λόγω κριτήριο, θα πρέπει να υπάρχει τουλάχιστον ένα ζεύγος διαδοχικών πακέτων $n_i = (w_i, d_i)$ και $n_{i+1} = (w_{i+1}, d_{i+1})$, με: $w_{i+1} + d_{i+1} > w_i + d_i$, ενώ η στοίβαξη έχει μορφή (..., n_i, n_{i+1}, ...).

  Τότε αν ανταλλάξουμε αυτά τα δύο πακέτα, η στοίβαξη που προκύπτει είναι ακόμα ασφαλής. Πράγματι:

  $w_{i+1} + \sum_{j>i+1} w_j < d_i \implies \sum_{j>i+1} w_j < d_i - w_{i+1}$ (1) και
  $w_{i+1} + d_{i+1} > w_i + d_i \implies \mathrm{d}_i - w_{i+1} < d_{i+1} - w_i$ (2)
  Από (1) και (2) $\implies \sum_{j>i+1} w_j < d_{i+1} - w_i \implies \mathrm{d}_{i+1} > \sum_{j>i+1} w_j + w_i$, άρα το n_{i+1} μπορεί να σηκώσει το βάρος όσων σηκώνει το n_i μαζί με το w_i.

Ανταλλάζοντάς αυτά τα δύο πακέτα, για όσα πακέτα βρισκόντουσαν κάτω από το n_i δεν αλλαζει κάτι, για όσα πακέτα βρισκόντουσαν πάνω από το n_{i+1} δεν αλλάζει κάτι, για το n_i μειώνεται το βάρος που σηκώνει και για το n_{i+1} αυξάνεται σε βαθμό που παραμένει ασφαλής η στοίβαξη. Επομένως, κάνοντας αλλεπάλληλες ανταλλαγές των διαδοχικών στοιχείων που δεν είναι στοιβαγμένα με το κριτήριο του αθροίσματος βάρους και αντοχής [2], μπορούμε να κατασκευάσουμε τελικά μια στοίβαξη B που να πληροί το εν λόγω κριτήριο.

Αποδείξαμε, έτσι, την παραπάνω πρόταση και άρα ο αλγοριθμος στοίβαξης με κριτήριο το άθροισμα βάρους και αντοχής οδηγεί σε ασφαλή στοίβαξη των πακέτων, εαν κι εφόσον αυτή υπάρχει.

2.2 Ερώτημα (β)

2.2.1 Συμβολισμοί

- S_{wd}: άθροισμα βάρους και αντοχής πακέτου
- w_i: βάρος i-οστού πακέτου
- d_i: αντοχή i-οστού πακέτου
- p_i: αξία i-οστού πακέτου

2.2.2 Αξιοποίηση ερωτήματος (α)

Έστω ένα σύνολο πακέτων Π κι ένα $A \subseteq \Pi$. Στο ερώτημα (α) αποδείξαμε ότι οι ασφαλείς στοιβάξεις που περιέχουν **όλα** τα πακέτα του A (αν υπάρχουν) μπορούν να ταξινομηθούν κατά φθίνον S_{wd} (από κάτω προς τα πάνω) και να παραμείνουν ασφαλείς. Άρα, το σύνολο των ασφαλών στοιβάξεων που περιέχουν όλα τα πακέτα του A, περιέχει (τουλάχιστον) μία ταξινομημένη ασφαλή στοίβαξη. Όλες οι στοιβάξεις του εν λόγω συνόλου είναι ισοδύναμες μεταξύ τους ως προς την αξία των πακέτων που περιέχουν, αφού όλες περιέχουν τα ίδια πακέτα.

Άρα, για κάθε $A \subseteq \Pi$ υπάρχει ένα σύνολο S - που μπορεί να είναι κενό - το οποίο περιέχει όλες τις ασφαλείς στοιβάξεις των πακέτων του A και (αν το S είναι μη κενό, τουλάχιστον) μία ταξινομημένη. Συνεπώς, για κάθε ασφαλή στοίβαξη που περιέχει κάποια από τα πακέτα του Π υπάρχει (τουλάχιστον) μία **ισάξια, αλλά ταξινομημένη** ασφαλής στοιβαξη.

Συμπεραίνουμε, λοιπόν, πως αν επιθυμούμε να υπολογίσουμε τη μέγιστη αξία που μπορεί να προκύψει στοιβάζοντας ασφαλώς κάποια πακέτα του Π, αρκεί να υπολογίσουμε τη μέγιστη αξία που μπορεί να προκύψει από κάποια ταξινομημένη ασφαλή στοίβαξη.

2.2.3 Αποσαφήνιση των ελαττωμάτων του προβλήματος

Στο πρόβλημά μας θέλουμε να βρούμε την ασφαλή στοίβαξη με τη μεγαλύτερη δυνατή αξία. Αν με κάποιο τροπο είχαμε όλες τις ασφαλείς στοιβάξεις (με τις αξίες τους), τότε θα λύναμε το πρόβλημα

[2] ανταλλαγές διαδοχικών πακέτων με ίσα αθροίσματα βάρους και αντοχής οδηγούν, επίσης, σε ασφαλείς στοιβάξεις, όπως εύκολα φαίνεται αν επαναλάβουμε την απόδειξη του (2.1.3), αλλά με "=" στη σχέση (2)

αν από αυτές τις στοιβάξεις διαλέγαμε αυτή με τη μεγαλύτερη αξία.

Αν δοκιμάσουμε να κατασκευάσουμε όλες τις ασφαλείς στοιβάξεις χωρίς κάποιο σύστημα, τότε θα χρειαστεί να κατασκευάσουμε $\Omega(2^n)$ στοίβες (όσο είναι και το πλήθος των υποσυνόλων των πακέτων).

2.2.4 Δομή της βέλτιστης λύσης

Αξιοποιώντας τα συμπεράσματα που προέκυψαν στην παράγραφο (2.2.2), μπορούμε να αναζητήσουμε τη λύση μόνο στο σύνολο των **ταξινομημένων** ασφαλών στοιβάξεων των υποσυνόλων των πακέτων, καθώς εκεί υπάρχουν όλες οι πιθανές συνολικές αξίες, άρα και η μέγιστη δυνατή. Από εδώ και πέρα θα προσπαθούμε να βρούμε τη λύση σε ταξινομημένες στοιβάξεις (δεν θα αναφέρεται η λέξη "ταξινομημένη").

Έστω τώρα ότι μια βέλτιστη στοίβαξη (1,2,...,n) (δεν περιέχει απαραίτητα όλα τα πακέτα) έχει μια υποστοίβα Υ = (i, i+1, ... ,n) με βάρος w_{sub}. Αν υπήρχε άλλη στοίβαξη, έστω Υ^*, με βάρος το πολύ w_{sub} (ώστε να μπορεί να σηκωθεί από τα κατώτερα πακέτα) και αξία μεγαλύτερη της Υ, τότε αν αντικαταστήσουμε την Υ^* με την Υ στην (1,2,...,n), η στοίβαξη που προκύπτει έχει μεγαλύτερη αξία από την βέλτιστη στοίβαξη $\implies$ Άτοπο.

Άρα, κάθε υποστοίβα Υ της βέλτιστης λύσης, από ένα πακέτο i και μέχρι την κορυφή της στοίβαξης, είναι βέλτιστη, δηλαδή δεν υπάρχει άλλη στοίβα με βάρος το πολύ ίσο με της Υ που μπορεί να την αντικαταστήσει και να πετύχει καλύτερη αξία.

Επομένως, η βέλτιστη στοίβαξη αποτελείται από μικρότερες βέλτιστες υποστοιβάξεις.

Αναλυτικότερα, ακόμα, αν η βέλτιστη στοίβαξη (1,2,...,n) έχει βάρος το πολύ W τότε η (2,...,n) είναι η στοίβαξη μεγαλύτερης αξίας με βάρος το πολύ $W - w_i$, η (3,...,n) είναι η στοίβαξη μεγαλύτερης αξίας με βάρος το πολύ $W - w_1 - w_2$ κοκ.

Ισχύει, λοιπόν, η Αρχή της Βελτιστότητας στο πρόβλημα που πετυχαίνει τη βέλτιστη στοίβαξη με βάρος το πολύ ίσο με $\sum w_i$.

2.2.5 Διατύπωση Αναδρομικής Εξίσωσης

Ταξινομούμε τα πακέτα του συνόλου Π σε φθίσουσα σειρά S_{wd} και προκύπτει το σύνολο $\{1, 2, ..., n\}$. Ορίζουμε ως $P(i, B)$ την μέγιστη συνολική αξία μιας ασφαλούς (ταξινομημένης) στοίβαξης που περιέχει (ή όχι) πακέτα από το σύνολο $\{i, i+1, i+2, ..., n\}$ (με τη σειρά) και με συνολικό βάρος το πολύ B.

$$P(i,B) = \begin{cases} 0 & B \leq 0 \vee i > n \\ max\{P(i+1,B), p_i + P(i+1, min(B - w_i, d_i))\} & i = \{0,...,n\} \wedge w_i = \{1,...,B\} \\ P(i+1,B) & w_i > B \end{cases}$$

Η μέγιστη αξία, λοιπόν, που μπορεί να προκύψει από ασφαλή στοίβαξη κάποιων πακέτων του Π

είναι η $P(1, \sum w_i)$.

Εφόσον ισχύουν οι απαραίτητες προϋποθέσεις για την αποδοτική επίλυση της αναδρομικής σχέσης θα σχεδιάσουμε αλγόριθμο που χρησιμοποιεί **δυναμικό προγραμματισμό**.

2.2.6 Η επίλυση της αναδρομικής εξίσωσης

Για να υπολογίσουμε το $P(i, B)$ θα θέλαμε να γνωρίζουμε τα $P(i+1, B)$ και $P(i+1, min(B-w_i, d_i))$ εκ των προτέρων. Διαφορετικά, πολλά υποπροβλήματα θα λύνονται πολλαπλές φορές. Γι'αυτό, ξεκινάμε από μικρότερα υποπροβλήματα. Εξετάζοντας τα (ταξινομημένα) πακέτα από το n-οστό προς τα πίσω, πρώτα βρίσκουμε τη βέλτιστη στοίβαξη με βάρος το πολύ 0, μετά τη βέλτιστη στοίβαξη με βάρος το πολύ 1, μετά τη βέλτιστη στοίβαξη με βάρος το πολύ 2 κοκ. Αποθηκεύουμε κάθε αποτέλεσμα σε έναν πίνακα $(n \times \sum w_i)$.

Ο αλγόριθμος υπολογίζει τη μέγιστη δυνατή αξία που μπορεί να προκύψει με ταξινομημένη ασφαλή στοίβαξη. Όχι την ίδια τη στοίβαξη. Γι' αυτό κατά την συμπλήρωση του πίνακα, κάθε φορά που η βέλτιστη επιλογή είναι να τοποθετηθεί κάποιο στοιχείο αποθηκεύουμε και το βάρος που απομένει μετά την τοποθέτησή του. Μόλις ολοκληρωθεί η συμπλήρωση του πίνακα μπορούμε, σε $O(n)$, να ανακατασκευάσουμε την βέλτιστη στοίβαξη ως εξής.

1. Έστω j η στήλη που εξετάζουμε
2. Ελέγχουμε την τελευταία στήλη ($j = \sum w_i$)
3. Βρίσκουμε τη γραμμή, έστω i όπου η αξία της στοίβαξης αυξήθηκε (δηλαδή επιλέχθηκε να μπει κάποιο πακέτο). Προσθέτουμε στην κορυφή της στοίβαξης το πακέτο της γραμμής i.
4. Πηγαίνουμε στην γραμμή $i-1$ και στη στήλη που αντιστοιχεί στο βάρος που απομένει μετά την προσθήκη του πακέτου της i γραμμής (έχουμε αποθηκευμένο τον αριθμό αυτής της στήλης).
5. Αν βρισκόμαστε στη θέση (0,0) του πίνακα, τερματίζουμε. Αλλιώς, επαναλαμβάνουμε από το βήμα (3).

Παράδειγμα
(βάρος, αντοχή, αξία): (1,1,5), (1,3,5), (4,1,20), (2,4,5), (2,5,5), (3,8,10)

i/B	0	1	2	3	4	5	6	7	8	9	10	11	12	13
∅	(0)	0	0	0	0	0	0	0	0	0	0	0	0	0
(1,1,5)	0	(5^0)	5^1	5^1	5^1	5^1	5^1	5^1	5^1	5^1	5^1	5^1	5^1	5^1
(1,3,5)	0	5	10^1	10^2	10^3	10^3	10^3	10^3	10^3	10^3	10^3	10^3	10^3	10^3
(4,1,20)	0	5	10	10	20^0	(25^1)	25^1	25^1	25^1	25^1	25^1	25^1	25^1	25^1
(2,4,5)	0	5	10	10	20	25	25	25	25	25	25	25	25	25
(2,5,5)	0	5	10	10	20	25	25	25	(30^5)	30^5	30^5	30^5	30^5	30^5
(3,8,10)	0	5	10	10	20	25	25	30^4	35^5	35^6	35^7	40^8	40^8	(40^8)

Άρα μια βέλτιστη στοίβαξη για το παράδειγμα είναι η:

$w_1 = 1, d_1 = 1, p_1 = 5$
$w_3 = 4, d_3 = 1, p_3 = 20$
$w_5 = 2, d_5 = 5, p_5 = 5$
$w_6 = 3, d_6 = 8, p_6 = 10$

2.2.7 Ορθότητα Αλγορίθμου

Δείξαμε ότι όλες οι αξίες στοιβάξεων μπορούν να προκύψουν αναζητώντας μόνο σε ταξινομημένες στοιβάξεις. Άρα εφόσον ο αλγόριθμος υπολογίζει μια ταξινομημένη στοίβαξη με βάρος το πολύ ίσο με $\sum w$ και με μέγιστη αξία, τότε βρίσκει μια βέλτιστη στοίβα.

2.2.8 Υπολογιστική Πολυπλοκότητα

- Η χρονική πολυπλοκότητα του αλγορίθμου είναι το χρονικό κόστος της ταξινόμησης του συνόλου πακέτων και της συμπλήρωσης του πίνακα: $O(nlogn) + O(n \times \sum w_i) = O(n \times \sum w_i + nlogn)$
- Η χωρική πολυπλοκότητα οφείλεται στον χώρο που καταλαμβάνει ο πίνακας $(n \times \sum w_i)$.

3 Άσκηση 3: Τριγωνοποίηση Πολυγώνου

3.1 Παρατήρηση επί του προβλήματος

Έστω ότι διαθέτουμε ένα πολύγωνο $P = (u_0, u_1, ..., u_{n-1})$, μια τυχαία τριγωνοποίηση $T(P)$ κι ένα σχηματιζόμενο τρίγωνο $u_i u_j u_k$. Παρατηρούμε ότι:

- αν το τρίγωνο $u_i u_j u_k$ έχει μία μόνο πλευρά, έστω $\overline{u_i u_j}$, που είναι ακμή του P, τότε η αφαίρεση αυτής από το πολύγωνο, οδηγεί σε δύο υποπολύγωνα $(u_i, u_{i+1}, ..., u_j)$ και $(u_j, u_{j+1}, ..., u_i)$.
- αν το τρίγωνο $u_i u_j u_k$ έχει δύο πλευρές, έστω $\overline{u_i u_j}$ και $\overline{u_j u_k}$, που είναι ακμές του P, τότε η αφαίρεση αυτών από το πολύγωνο, οδηγεί σε δύο υποπολύγωνα $(u_i, u_{i+1}, ..., u_k)$ και $\emptyset$.

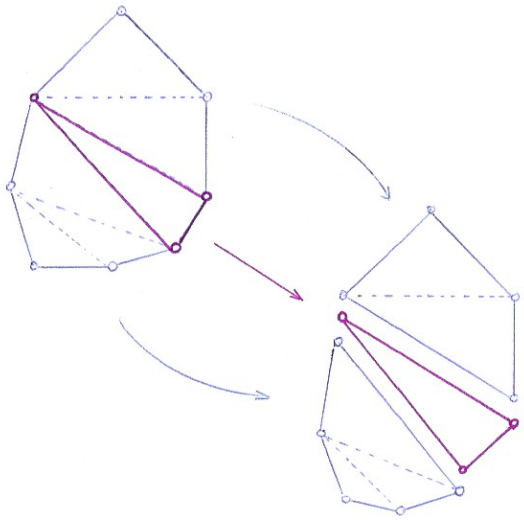

Figure 1: "Σπάσιμο" πολυγώνου

Αν, λοιπόν, "αφαιρέσουμε" ένα τρίγωνο από ένα τριγωνοποιημένο πολύγωνο, τότε προκύπτουν 2 μικρότερα τριγωνοποιημένα υποπολύγωνα.

3.2 Αρχή βελτιστότητας

Έστω ότι διαθέτουμε μια τριγωνοποίηση $T(P)$ που περιέχει χορδές με το ελάχιστο δυνατό συνολικό μήκος, δηλαδη είναι βέλτιστη. Αν "σπάσουμε" το πολύγωνο P, όπως στην εικονα 1, σε δύο υποπολύγωνα P_1 και P_2, τότε θα έχουμε αντίστοιχα και δύο τριγωνοποιήσεις $T(P_1) \in T(P)$ και $T(P_2) \in T(P)$. Αν θεωρήσουμε πως για το υποπολύγωνο P_1 η τριγωνοποίηση $T(P_1)$ δεν είναι βέλτιστη, τότε σημαίνει πως υπάρχει μια άλλη τριγωνοποίηση $T'(P_1)$ με μικρότερο συνολικό μήκος χορδών από την $T(P_1)$. Τότε όμως, αν τριγωνοποιήσουμε το P_1 με βάση την $T'(P_1)$ κι ενώσουμε τα P_1 και P_2 με χρήση του τριγώνου που αφαιρέσαμε για να σπάσουμε το P, τότε ξαναπροκύπτει το P τριγωνοποιημένο διαφορετικά από πριν και με συνολικό μήκος χορδών μικρότερο από το αρχικό. Συνεπώς, η αρχική τριγωνοποίηση $T(P)$ δεν ήταν βέλτιστη $\implies$ Άτοπο.

Άρα, τα (οποιαδήποτε δύο) υποπολύγωνα προκύπτουν αν "σπάσουμε" ένα βέλτιστα τριγωνοποιημένο πολύγωνο είναι, επίσης, βέλτιστα τριγωνοποιημένα. Βλέπουμε πως στο πρόβλημα της τριγωνοποίησης, η βέλτιστη λύση μπορεί να συντεθεί από βέλτιστες υπολύσεις, δηλαδή ισχύει η αρχή της βελτιστότητας.

3.3 Ιδέα Αλγορίθμου

Λόγω της αρχής της βελτιστότητας, η βέλτιστη λύση αποτελείται από δυο βέλτιστες τριγωνοποιήσεις υποπολυγώνων κι ένα τρίγωνο που τα ενώνει στο αρχικό πολύγωνο (κυριολεκτικά, μόνο οι πλευρές του τριγώνου που είναι χορδές θα ανήκουν στην τριγωνοποίηση). Όμως, δεν γνωρίζουμε ποιό είναι το καταλληλότερο τρίγωνο. Εξαρχής, εφόσον δεν έχουμε αναπτύξει κάποιο σύστημα, δεν έχουμε παρά να δοκιμάσουμε όλα τα τρίγωνα.

Ένας αλγόριθμος θα ήταν, λοιπόν:

1. Επιλέγουμε δυο διαδοχικά σημεία του πολυγώνου (τυχαία), τα οποία γνωρίζουμε ότι αντιστοιχούν (μαζί) σε μοναδικό τρίγωνο στη βέλτιστη λύση.
2. Προκειμένου να βρούμε το κατάλληλο τρίγωνο τα δοκιμάζουμε όλα. Για κάθε τρίγωνο που δοκιμάζουμε, το πολύγωνο σπάει σε δύο.
3. Επαναλαμβάνουμε αναδρομικά το ίδιο για κάθε υποπολύγωνο που προκύπτει, μέχρι να φτάσουμε σε πολύγωνα με τρεις πλευρές.
4. Ύστερα επιστρέφουμε από κάθε υποπρόβλημα, επιλέγοντας την βέλτιστη τριγωνοποίηση για το καθένα.

Ωστόσο, έτσι το πρόβλημα επιλύεται αμιγώς αναδρομικά και πολλά από τα υποπροβλήματα που επιλύονται για να φτάσουμε σε απάντηση για το μεγαλύτερο πρόβλημα, ανέρχονται πολλαπλές φορές,

όπως φαίνεται στο παρακάτω παράδειγμα. Στη συνέχεια, για να αποφύγουμε αυτό το πρόβλημα θα χρησιμοποιήσουμε δυναμικό προγραμματισμό.

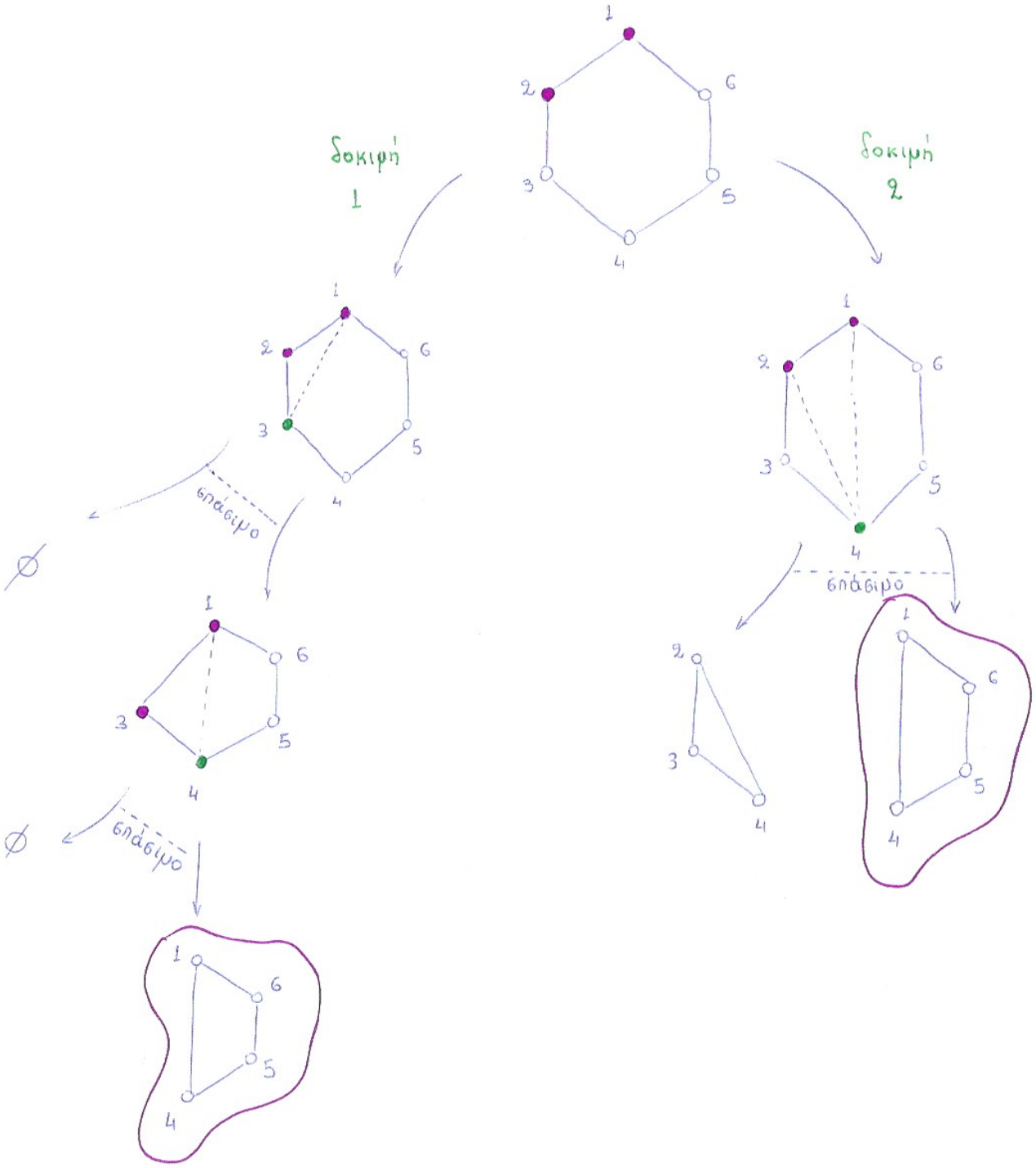

Figure 2: Παράδειγμα Επικάλυψης Υποπροβλημάτων

3.4 Βελτίωση της Υπολογιστικής Πολυπλοκότητας

3.4.1 Συμβολισμοί

- $P(i,j)$: υποπολύγωνο με κορυφές τις $(u_i, u_{i+1}, ..., u_j)$
- $T(P(i,j))$: βέλτιστο σύνολο χορδών που τριγωνοποιούν το υποπολύγωνο $P(i,j)$ (βέλτιστη τριγωνοποίηση του $P(i,j)$)
- $L(T(P(i,j)))$: συνολικό μήκος χορδών της τριγωνοποίησης $T(P(i,j))$

3.4.2 Αναδρομική Σχέση

Η αναδρομική σχέση που εκφράζει **το συνολικό μήκος**[3] της βέλτιστης λύσης για το υποπολύγωνο $P(i,j)$ είναι:

$$L(T(P(i,j))) = \begin{cases} 0 & , j = i+1 \\ \min\limits_{i<x<j}\{\Delta(u_i u_x u_j) + L(T(P(i,x))) + L(T(P(x,j)))\} & , otherwise \end{cases}$$

Η άνω αναδρομική σχέση δίνει **το συνολικό μήκος** της βέλτιστης λύσης, όχι την ίδια τη βέλτιστη λύση, δηλαδή το βέλτιστο σύνολο τριγωνοποίησης. Επιλύοντας, ωστόσο, την εξίσωση με δυναμικό προγραμματισμό, μπορούμε να παράγουμε ταυτόχρονα και το βέλτιστο σύνολο, όπως θα δείξουμε παρακάτω.

3.4.3 Επίλυση με Δυναμικό Προγραμματισμό

Μέχρι στιγμής το πρόβλημά μας διέπει η Αρχή της Βελτιστότητας, ενώ το συνολικό μήκος της βέλτιστης λύσης δίνεται από την αναδρομική σχέση της παραγράφου 3.4.2.

Το μήκος της λύσης για το πολύγωνο εισόδου είναι η $L(T(P(0, n-1)))$.

Προκειμένου να επιλύσουμε την αναδρομική εξίσωση με τρόπο που αποφεύγει την επίλυση προβλήματος που έχει συναντηθεί προηγουμένως θα χρησιμοποιήσουμε δυναμικό προγραμματισμό. Αντί να ξεκινήσουμε από το μεγάλο πολύγωνο και να υπολογίσουμε τα μικρότερα, θα υπολογίσουμε πρώτα όλα τα μικροτερα και ύστερα θα πηγαίνουμε σε μεγαλύτερα που χρησιμοποιούν τα μικρότερα.

[3] Στην πραγματικότητα, από τον τρόπο που υπολογίζεται, το μήκος οποιασδήποτε τριγωνοποίησης, περιλαμβάνει το μήκος κάθε χορδής δύο φορές και το μήκος κάθε αμκής του αρχικού πολυγώνου μία φορά. Αυτό δεν μας ενοχλεί, γιατί το συνολικό μήκος των ακμών του πολυγώνου είναι ίδιο για όλες τις τριγωνοποιήσεις και το υπόλοιπο άθροισμα είναι το διπλάσιο του πραγματικού αθροίσματος. Έτσι η διάταξη των τριγωνοποιήσεων ως προς το συνολικό μήκος παραμένει η ίδια. Γι' αυτο και καταχρηστικά λέμε ότι η L είναι το μήκος της βέλτιστης λύσης.

Αρχικά, δημιουργούμε δύο πίνακες L και T διαστάσεων $n \times n$.
Κάθε κελί (i, j) του L:

- αν $j > i + 1$, τοτε θα γεμίσει με το μήκος της βέλτιστης τριγωνοποίησης του υποπολυγώνου P(i,j)
- αν $j = i + 1$, τότε αρχικοποιείται στο 0. Πρόκειται για το μήκος τριγωνοποίησης ενός "2-γώνου", που ναι μεν δεν υφίσταται, αλλά αποτελεί το μικρότερο υποπρόβλημα (πρώτος κλάδος της αναδρομικής εξίσωσης).
- δεν μας ενδιαφέρουν τα υπόλοιπα κελιά

Κάθε κελί (i, j) του T:

- αν $j > i+1$, τοτε θα γεμίσει με την κορυφή x, του τριγώνου $u_i u_x u_j$ που είναι το καταλληλότερο για το "σπάσιμο" του P(i,j) σε υποπολύγωνα βέλτιστης τριγωνοποίησης.
- δεν μας ενδιαφέρουν τα υπόλοιπα κελιά.

Επιλέγουμε να περιγράψουμε τον αλγόριθμο με χρήση ψευδοκώδικα.

Αλγόριθμος Polygon

```
for i from 0 to n-2
    L[i, i+1] <- 0
for k from 2 to n-2
    for i = 0 to n-k-1
        j <- i+k
        L[i,j] <- INF
        for x from i+1 to j-1
            temp <- L[i,k] + L[k,j] + delta(u_i, u_j, u_k)
            if temp < L[i,j] then
                L[i,j] <- temp
                T[i,j] <- x
```

Στην παρακάτω εικόνα φαίνεται ένα οπτικοποιημένο παράδειγμα εκτέλεσης του αλγορίθμου. Ο αλγόριθμος εκτελείται από πάνω προς τα κάτω και από αριστερά προς τα δεξιά.

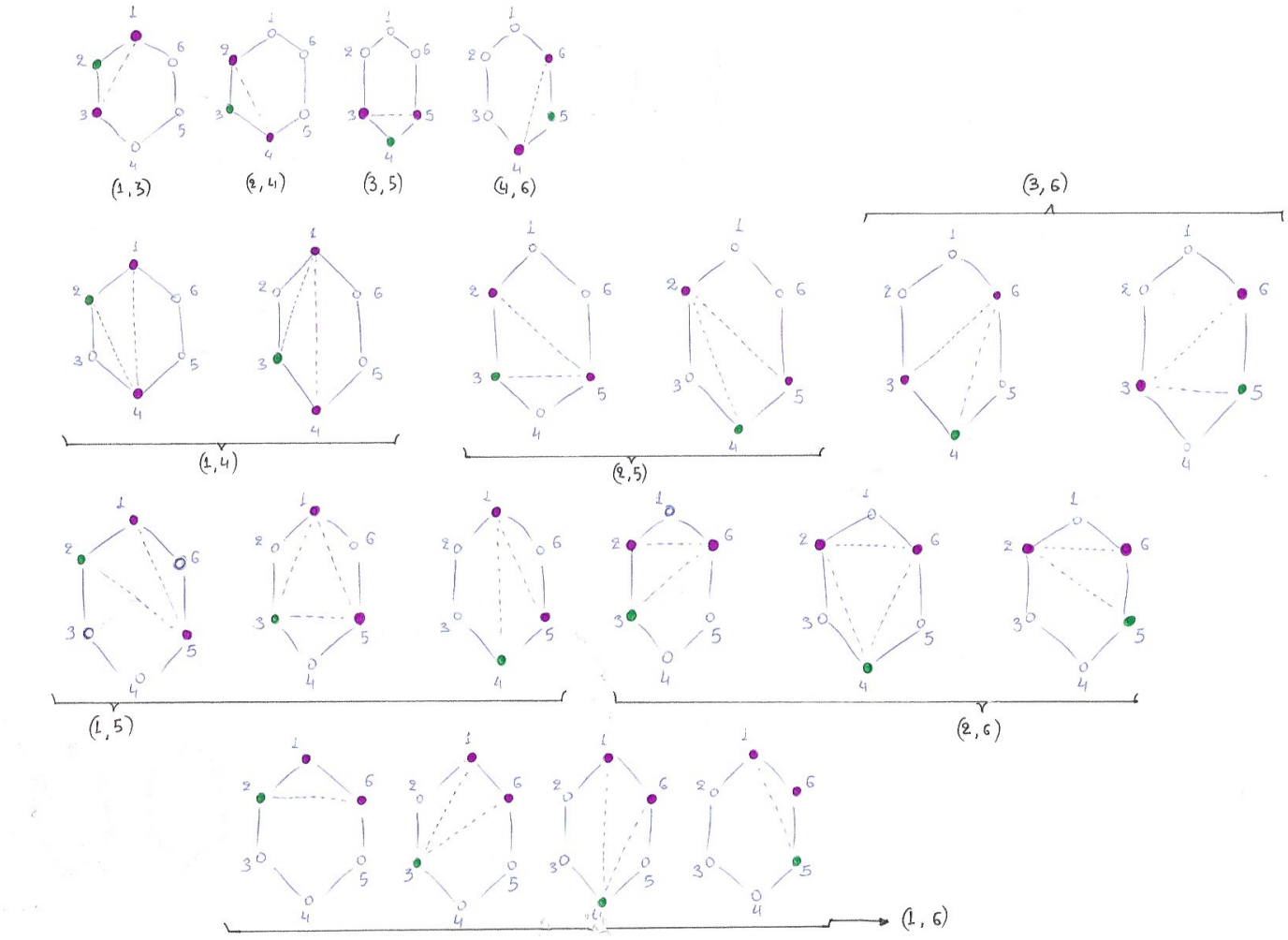

Figure 3: Εποπτικό παράδειγμα εκτέλεσης του αλγορίθμου Polygon. Έχουμε θεωρήσει (για ευκολία) ότι οι κορυφές είναι αριθμημένες από το 1 έως το n.

Με άγκιστρο ομαδοποιούνται οι εκδοχές που εξετάζονται σε κάθε υποπρόβλημα. Το υποπρόβλημα (2,6), για παράδειγμα, για να εξετάσει ποιό είναι το κατάλληλο τρίγωνο πρέπει να συγκρίνει τα αποτελέσματα 3 εκδοχών.

- Στην πρώτη εκδοχή επιλέγεται το $u_2u_3u_6$, και για να υπολογιστεί το L χρησιμοποιείται το αποτέλεσμα του υποπροβλήματος (3,6).
- Στην δεύτερη εκδοχή επιλέγεται το $u_2u_4u_6$, και για να υπολογιστεί το L χρησιμοποιούνται τα αποτελέσματα των υποπροβλημάτων (2,4) και (4,6).
- Στην τρίτη εκδοχή επιλέγεται το $u_2u_5u_6$, και για να υπολογιστεί το L χρησιμοποιείται το αποτέλεσμα του υποπροβλήματος (2,5).

Στο τέλος του αλγορίθμου ο πίνακας T περιέχει τις κορυφές που δίνουν τις κατάλληλες διασπάσεις, δηλαδή τα κατάλληλα τρίγωνα, άρα τις χορδές της βέλτιστης τριγωνοποίησης. Ο τρόπος με τον οποίο

μπορούμε να ανακτήσουμε τώρα τη βέλτιστη τριγωνοποίηση, δίνεται από τον παρακάτω ψευδοκώδικα:

Αλγόριθμος Triangulation_Recovery

```
t <- empty_set      // set to contain optimal solution
q <- empty_queue
q.enqueue((0,n))

while q is not empty
    pair <- dequeue(q)  // check polygon (i,...,j)
    i <- pair[0]
    j <- pair[1]
    x = T[i,j]

    // add the triangular sides that are not edges to solution
    if x != i+1              // if the (i,x) is not an edge, then
        t.add((i, x))        // add to solution
        q.enqueue((i, x))    // add to queue
    if x != j-1          // same for (x,j)
        t.add((x,j))
        q.enqueue((x, j))
```

Μετά το πέρας των παραπάνω αλγορίθμων, το σύνολο t θα περιέχει τις χορδές που τριγωνοποιούν το αρχικό πολύγωνο κι έχουν το ελάχιστο δυνατό μήκος.

3.5 Υπολογιστική Πολυπλοκότητα

Η πολυπλοκότητα του αλγορίθμου που γεμίζει τους πίνακες L και T είναι:

$$O(n) + O(n) \cdot O(n) \cdot O(n) = O(n^3)$$

Για την πολυπλοκότητα του αλγορίθμου που ανασυνθέτει από τον πίνακα T την βέλτιστη τριγωνοποίηση $T(P)$, βλέπουμε ότι σε κάθε επανάληψη "αφαιρείται" ένα τρίγωνο από το τριγωνοποιημένο πολύγωνο κι εξετάζονται τα επιμέρους υποπολύγωνα που προκύπτουν. Ο αλγοριθμος τελειώνει όταν πρακτικά "αφαιρεθούν" όλα τα τρίγωνα που απαρτίζουν το αρχικό πολύγωνο. Άρα, θα κάνει το πολύ n-2 επαναλήψεις κι έχει πολυπλοκότητα: $O(n-2) = O(n)$.

Τελικά, η χρονική πολυπλοκότητα της επίλυσης του προβληματος είναι $O(n^3)$, ενώ η χωρική πολυπλοκότητα είναι συνάρτηση των πινάκων που χρησιμοποιούμε, δηλαδή $O(n^2) + O(n^2) = O(n^2)$

3.6 Ορθότητα

Ο αλγόριθμός μας είναι σχεδιασμένος, ώστε να παράγει ελάχιστο L, σύμφωνα με την αναδρομική σχέση της παραγράφου 3.4.2. Όπως αναφέρουμε και στην υποσημείωση 3, στην πραγματικότητα η

συνάρτηση L που φτιάξαμε υπολογίζει το:

$$L(T(P(0,n-1))) = \sum_{i=0}^{n-2} \|u_i u_{i+i}\| + 2 \cdot \sum chords$$

όπου το $\sum_{i=0}^{n-2} \|u_i u_{i+i}\|$ είναι ίδιο για όλες τις τριγωνοποιήσεις.

Άρα, ο αλγόριθμος παράγει ελάχιστο $\sum chords$, δηλαδή ελάχιστο συνολικό μήκος χορδών. Συνεπώς, παράγει το ελάχιστο $L(T(P(0,n-1)))$, κατά τον υπολογισμό του οποίου παράγεται και ο πίνακας από τον οποίο θα προκύψει το βέλτιστο σύνολο T.

4 Άσκηση 4: Τοποθέτηση Στεγάστρων (και Κυρτό Κάλυμμα)

4.1 Ερώτημα (α)

4.1.1 Μια πρώτη (αφελής) προσέγγιση

Εκ πρώτης όψεως προσπαθούμε να βρούμε κάποια προφανή λύση. Μήπως έχοντας βρει μια λύση για τα πρώτα i σημεία, μπορούμε απλά να εξετάσουμε αν μας συμφέρει να σκεπάσουμε το $(i+1)$-οστό επεκτείνοντας το τελευταίο σκέπαστρο των i σημείων ή να φτιάξουμε καιρνούργιο; Δυστυχώς, το παρακάτω αντιπαράδειγμα φανερώνει πως μια γραμμική λύση, δεν θα είναι τόσο εύκολο να προκύψει.

Αντιπαράδειγμα

$x_1 = 1$
$x_2 = 2$
$x_3 = 5$
$x_4 = 6$
$C = 20$

Με αυτές τις παραμέτρους, η γραμμική προσέγγισή μας θα εξέταζε τις περιπτώσεις των Figure 4 και Figure 5, διαλέγοντας την καλύτερη, δηλαδή αυτή της Figure 5. Ωστόσο, το μικρότερο δυνατό κόστος πετυχαίνεται για την επιλογή στεγάστρων της Figure 6.

Επομένως, εγκαταλείπουμε αυτή την προσπάθεια.

Figure 4: cost = $(1\text{-}5)^2 + 2 \cdot 20 = 56$

Figure 5: cost = $(1\text{-}6)^2 + 20 = 45$

Figure 6: cost = $(1\text{-}2)^2 + (5-6)^2 + 2 \cdot 20 = 42$

4.1.2 Χαρακτηριστικά Προβλήματος

Πριν προχωρήσουμε στη διατύπωση κάποιου αλγορίθμου, προσπαθούμε να κατανοήσουμε τα χαρακτηριστικά του προβλήματος. Παρατηρούμε, λοιπόν, τα εξής:

- Το πρόβλημα πρόκειται (εμφανώς) για την αναζήτηση μιας λύσης που ελαχιστοποιεί ένα μέγεθος: το τελικό κόστος.
- Έστω ότι γνωρίζουμε μια βέλτιστη λύση S για ένα δεδομένο σύνολο n σημείων. Ακόμη, έστω i το αριστερότερο σημείο που καλύπτει το στέγαστρο που σκεπάζει το x_n. Τότε, βλέπουμε τα εξής:
  - Έστω S^1_{i-1} ο συνδυασμός στεγάστρων που σκεπάζει τα σημεία από το x_1 μέχρι και το x_{i-1} και S^i_n ο συνδυασμός στεγάστρων που σκεπάζει τα σημεία από το x_i μέχρι και το x_n. Ο S^1_{i-1} είναι ο πιο οικονομικός τρόπος να στεγάσουμε τα $i-1$ πρώτα σημεία. Άν υπήρχε πιο οικονομικός, έστω S'^1_{i-1}, τότε η λύση S' που θα προέκυπτε αν στεγάζαμε τα $i-1$ πρώτα σημεία με τον S'^1_{i-1} και τα υπόλοιπα $n-i+1$ με τον S^i_n θα είχε μικρότερο κόστος από την S. Άρα η S δε θα είναι βέλτιστη λύση $\implies$ Άτοπο. Ομοίως και για την αντικατάσταση του S^i_n από κάποιον S'^i_n. Επομένως, η βέλτιστη λύση του προβλήματος n σημείων συντίθεται από τις βέλτιστες λύσεις των υποπροβλημάτων στα οποία το πρόβλημα "σπάει" $\implies$ Ισχύει η Αρχή της Βελτιστότητας.

Συμπεραίνουμε, λοιπόν, πως το κόστος μιας λύση (όχι απαραίτητα βέλτιστης) για την στέγαση n σημείων μπορεί να γραφεί με τη μορφή:

$$total\_cost = cost[i-1] + (x_i - x_n)^2 + C$$

,

όπου $cost[i-1]$: κόστος στέγασης των $i-1$ πρώτων σημείων (άγνωστο i).

4.1.3 Αναδρομική Σχέση

Γενικά το κόστος της βέλτιστης λύσης για k σημεία θα είναι:

$$cost[k] = \begin{cases} 0 & , k = 0 \\ C & , k = 1 \\ \min\limits_{0<i\leq k}\{cost[i-1] + (x_i - x_k)^2 + C\} & , otherwise \end{cases}$$

και η βέλτιστη λύση για το αρχικό πρόβλημα n σημείων θα έχει κόστος $cost[n]$.

Όμως, δεν γνωρίζουμε ποιό είναι το καταλληλότερο i που θα δώσει το ελάχιστο δυνατό $total\_cost$. Επομένως, θα πρέπει να δοκιμάσουμε όλα τα (ακέραια) $i \in [1, k]$. Παρατηρούμε, ωστόσο, πως για να βρούμε το $cost[k]$ θα πρέπει να βρούμε τα $cost[i-1], \forall i \in [1, k]$ και για να βρούμε το $cost[k+1]$ θα

πρέπει να βρούμε τα $cost[i-1], \forall i \in [1, k+1]$, διάστημα που περιλαμβάνει όλα τα $cost$ που υπολογίσαμε για να βρούμε το $cost[k]$. Τα υποπροβλήματα, λοιπόν, που απαρτίζουν το αρχικό πρόβλημα n σημείων, παρουσιάζουν επικάλυψη. Γι'αυτό τόν λόγο θα πρέπει να υπολογίσουμε πρώτα τα μικρότερα υποπροβλήματα κι ύστερα τα μεγαλύτερα, φροντίζοντας τα αποθηκεύουμε τις λύσεις τους, ώστε να μην τις ξαναυπολογίζουμε όταν τις χρειαστούμε αργότερα.

Είμαστε πλεόν έτοιμοι, με όλες τις απαραίτητες προϋποθέσεις να ισχύουν, να σχεδιάσουμε έναν αλγόριθμο που χρησιμοποιεί δυναμικό προγραμματισμό.

4.1.4 Αλγορίθμος σε Python

Παρακάτω, παραθέτουμε τον αλγόριθμο σε γλώσσα Python:

```
import numpy as np

cost = np.ones(n+1) * np.inf

cost[0] = 0
cost[1] = C

for i in range(2,len(cost)):
    for j in range(1,i+1):
        temp = cost[j-1] + (A[j] - A[i])**2 + C
        if cost[i] > temp:
            cost[i] = temp
```

4.1.5 Υπολογιστική Πολυπλοκότητα

Είναι αρκετά εμφανές ότι η λύση μας έχει χρονική πολυπλοκότητα τάξης $O(n^2)$ που οφείλεται στο εμφωλευμένο *for loop*. Η χωρική πολυπλοκότητα οφείλεται στους πίνακες που γεμίζουμε και είναι $O(n)$.

4.2 Ερώτημα (β) - bonus

4.2.1 Σκοπός του ερωτήματος

Όντας προϊδεασμένοι από την εκφώνηση του ερωτήματος, καταλαβαίνουμε πως ο σκοπός μας είναι να δημιουργήσουμε ένα σύστημα που ικανοποιεί το ζητούμενο πρόβλημα των ευθειών $(a_j, b_j) \in Q^2$ και των σημείων $x_i \in Q$, ώστε να το χρησιμοποιήσουμε σαν "μαύρο κουτί" στη συνέχεια και να λύσουμε το ερώτημα (α) με μικρότερη υπολογιστική πολυπλοκότητα.

4.2.2 Δημιουργία Συστήματος

Αρχικός μας σκοπός είναι να βρούμε τα διαστήματα του Q στα οποία η κάθε ευθεία βρίσκεται χαμηλότερα από όλες τις άλλες. Επισημαίνουμε, προτού ξεκινήσουμε, ότι η τομή δύο ευθειών μπορεί να βρεθεί με μια απλή μαθηματική πράξη σε $O(1)$.

Δημιουργούμε μια δομή δεδομένων, ονόματι *Lines*, στην οποία υλοποιούμε την λειτουγία $add\_line(l)$. Η λειτουργία αυτή αποθηκεύει (πχ σε μια απλά συνδεδεμένη λίστα ή σε ένα vector) τις ευθείες που δίνονται ως είσοδος την μία μετά την άλλη. Εφόσον οι ευθείες εισόδου είναι (σύμφωνα με την εκφώνηση) ήδη διατεταγμένες κατά φθίνουσα κλίση (πάντα αρνητική), όταν καλείται η $add\_line(l)$ μπορεί να συμβεί μία από τις εξής τρεις εκδοχές:

- η ευθεία l τέμνεται με την τελευταία ευθεία της δομής *Lines* σε σημείο που βρίσκεται δεξιότερα από το σημείο τομής της τελευταίας ευθείας με την προτελευταία (όπως η l_4 στο Figure 7). Τότε η l προστίθεται και η διαδικασία ολοκληρώνεται.
- η ευθεία l τέμνεται με την τελευταία ευθεία της δομής *Lines* σε σημείο που βρίσκεται αριστερότερα από το (ή πάνω στο) σημείο τομής της τελευταίας ευθείας με την προτελευταία (όπως η l_5 στο Figure 8). Τότε η l προστίθεται, αλλά η ευθεία που μέχρι πριν την l ήταν τελευταία (όπως η l_4 του Figure) πλέον δεν είναι μικρότερη από όλες τις υπόλοιπες σε κανένα διάστημα του Q. Άρα πρέπει να αφαιρεθεί. Επομένως, όταν λαμβάνει χώρα αυτή η εκδοχή αφαιρούνται με τη σειρά (από το τέλος προς την αρχή) όλες οι ευθείες για τις οποίες ισχύει ό,τι ίσχυε και για την τελευταία.
- δεν υπάρχει ευθεία εντός της δομής αποθήκευσης. Τότε η l προστίθεται και η διαδικασία ολοκληρώνεται.

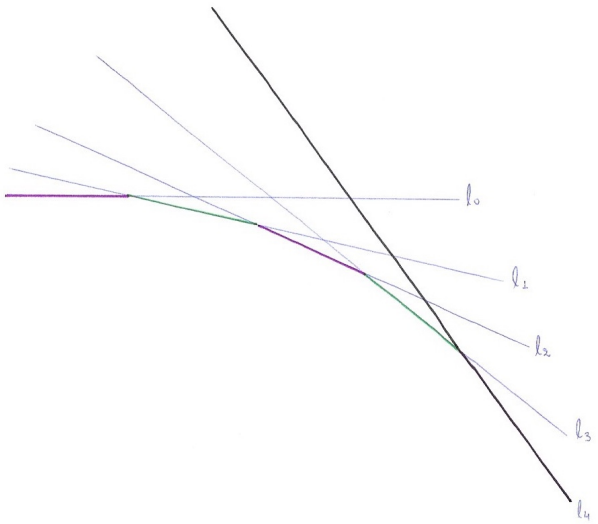

Figure 7: Εκδοχή 1

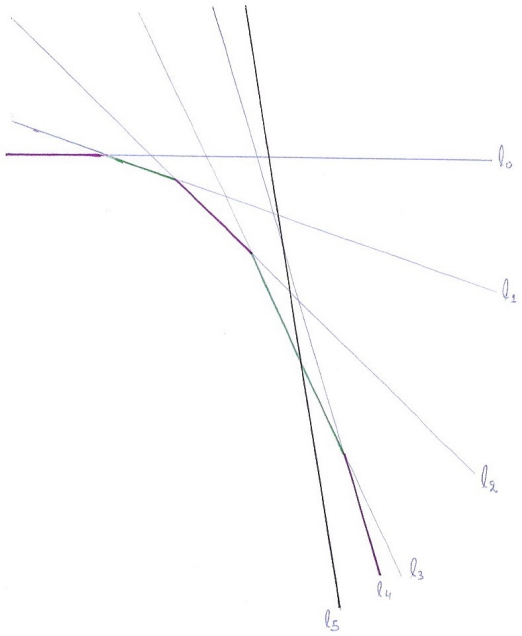

Figure 8: Εκδοχή 2

Με χρώμα (εναλλάξ μωβ και πράσινο) εμφανίζονται τα τμήματα των ευθειών όταν αυτές βρίσκονται χαμηλότερα από όλες τις άλλες.

Αν προσθέσουμε n ευθείες στην $Lines$ (καλέσουμε δηλαδή n φορές την $add\_line(l)$, τότε θα γίνουν το πολύ n προσθέσεις και το πολύ $n-1$ αφαιρέσεις ευθειών, άρα για εισαγωγή n ευθειών έχουμε πολυπλοκότητα $O(n)$.

Στη συνέχεια, θα πρέπει να υλοποιήσουμε μια λειτουργία που να απαντά στο ερώτημα "ποιά ευθεία είναι η μικρότερη από όλες στο σημείο x;" επιστρέφοντας τον δείκτη j της. Ονομάζουμε αυτήν την λειτουργία $query(x)$. Από την εκφώνηση γνωρίζουμε ότι τα σημεία x_i που θα δοθούν ως είσοδος στην $query(x)$ είναι διατεταγμένα σε αύξουσα σειρά. Αυτό σημαίνει ότι αν για κάποιο x_i η $query(x_i)$ επιστρέψει την ευθεία l_x (στο εξής η φράση "επιστρέφει ευθεία" σημαίνει "επιστρέφει τον δείκτη j της ευθείας"), τότε για κάθε ένα από τα επόμενα $x_j, j > i$ η $query(x_j)$ θα επιστρέψει κάποια ευθεία που βρίσκεται δεξιότερα της l_x στην δομή αποθήκευσης ή την ίδια την l_x. Εκμεταλλευόμαστε αυτήν την ιδιότητα και υλοποιούμε την $query(x)$ ως εξής:

Ορίζουμε έναν δείκτη $pointer$ που αρχικά δείχνει στην l_1. Μόλις δοθεί είσοδος x_i τότε:

- αν η ευθεία στην οποία δείχνει ο $pointer$ είναι η τελευταία της δομής, τότε η $query(x_i)$ επιστρέφει αυτή την ευθεία. Διαφορετικά,
- βρίσκουμε το σημείο τομής της ευθείας στην οποία δείχνει ο $pointer$, με την επόμενή της. Αν το x_i βρίσκεται πριν από αυτό το σημείο τομής (ή πάνω του), τότε η $query(x_i)$ επιστρέφει την ευθεία στην οποία δείχνει ο $pointer$. Αλλιώς,
- μετακινούμε τον $pointer$ στην επόμενη ευθεία κι επαναλαμβάνουμε.

Για k σημεία, η $query(x)$ καλείται k φορές. Επιπλέον, στη χειρότερη περίπτωση θα διασχιθεί όλο το σύνολο των αποθηκευμένων ευθειών. Άρα, έχουμε τελική πολυπλοκότητα $O(n+k)$ για k κλήσεις της $query(x)$ και σύνολο n ευθειών.

Επιπλέον, θέλουμε να μπορούμε να καλούμε την $query(x)$ και ύστερα να μπορούμε να προσθέτουμε κι άλλες ευθείες. Τότε, υπάρχει ο κίνδυνος, ενώ ο $pointer$ δείχνει στην τελευταία ευθεία, να λάβει χώρα η εκδοχή 2 της $add\_line(l)$. Χωρίς κάμια αλλαγή, ο $pointer$ μένει "ξεκρέμαστος". Γι' αυτό φροντίζουμε σε κάθε κλήση της $add\_line(l)$, αν ο $pointer$ έδειχνε στην τελευταία ευθεία, να τον βάζουμε να δείχνει στην νέα τελευταία μετά την προσθήκη της l. Η τελική πολυπλοκότητα για εισαγωγή n ευθειών και k κλήσεων της $query(x)$ δεν επηρεάζεται από αυτή την διόρθωση κι εν τέλει είναι $\Theta(n+k)$.

Με χρήση της παραπάνω δομής, μπορούμε πλέον να εκπληρώσουμε το ζητούμενο του ερωτήματος (β) σε γραμμικό χρόνο.

4.2.3 Βελτίωση πολυπλοκότητας ερωτήματος (α)

Αν ξαναγράψουμε την αναδρομική σχέση μπορούμε να κάνουμε μια σημαντική παρατήρηση.

$$cost[k] = \begin{cases} 0 & , k = 0 \\ C & , k = 1 \\ \min\limits_{0<i\leq k}\{cost[i-1] + (x_i - x_k)^2 + C\} & , otherwise \end{cases}$$

$$\implies cost[k] = \begin{cases} 0 & , k = 0 \\ C & , k = 1 \\ \min\limits_{0<i\leq k}\{cost[i-1] + (x_i^2 - 2 \cdot x_i x_k + x_k^2) + C\} & , otherwise \end{cases}$$

$$\implies cost[k] = \begin{cases} 0 & , k = 0 \\ C & , k = 1 \\ \min\limits_{0<i\leq k}\{\underbrace{-2 \cdot x_i}_{a}\underbrace{x_k}_{x} + \underbrace{x_i^2 + cost[i-1]}_{b}\} + x_k^2 + C & , otherwise \end{cases}$$

Το ελάχιστο κόστος για την κάλυψη των k πρώτων σημείων είναι το άθροισμα μιας ποσότητα που εξαρτάται από το k-οστό σημείο $(x_k^2 + C)$ και μιας ποσότητας που ταυτίζεται με την τιμή της ευθείας $l_i = (-2 \cdot x_i, x_i^2 + cost[i-1])$ που είναι η χαμηλότερη από όλες τις ευθείες l_0 έως και l_k στο σημείο x_k. Παρατηρούμε ότι οι ευθείες αυτές έχουν αρνητική και φθίνουσα κλίση. Ακόμη, τα σημεία x_k είναι διατεταγμένα σε αύξουσα σειρά. Επομένως, υποπτευόμαστε πως μπορούμε να λύσουμε το πρόβλημα με χρήση της δομής που κατασκευάσαμε προηγουμένως. Ακολουθούμε, λοιπόν, τα εξής βήματα:

- cost[0] = 0
- cost[1] = C, $l_1 = (-2x_1, x_1^2 + cost[0])$ και $add\_line(l_1)$
- $\forall i \in [2, n]$:
  - $j = query(x_i)$
  - $cost[i] = l_j(x_i) + x_i^2 + C$
  - $l_i = (-2x_i, x_i^2 + cost[i-1])$ και $add\_line(l_i)$

Μετά το πέρας της διαδικασίας, η λύση του προβλήματος, δηλαδή το ελάχιστο κόστος στέγασης και των n σημείων δίνεται από το $cost[n]$. Για να ολοκληρωθεί η διαδικασία, χρειάστηκαν n προσθήκες ευθειών και n κλήσεις της query(x).

Η τελική πολυπλοκότητα είναι σημαντικά βελτιωμένη σε σχέση με την αρχική $O(n^2)$, καθώς είναι $\Theta(n) + \Theta(n) = \Theta(n)$, δηλαδή γραμμική ως προς το μήκος εισόδου.

5 Άσκηση 5: Καλύπτοντας ένα Δέντρο

5.1 Ερώτημα (α)

5.1.1 Σκιαγράφηση ζητούμενου

Για αρχή ας αναλύουμε το πρόβλημα, ώστε να καταλάβουμε σε βάθος τι είναι αυτό που θα πρέπει ο αλγόριθμος να πετυχαίνει.

Μια σημαντική παρατήρηση είναι η εξής: όταν η ρίζα *sub_root* ενός υποδέντρου *sub_T* μπαίνει στο σύνόλο K, τότε οι αποστάσεις των κορυφών του από το σύνολο K είναι πλέον οι αποστάσεις των κορυφών του από τη ρίζα. Άρα, όταν μια κορυφή *sub_root* εισάγεται στο σύνολο K, τότε οι αποστάσεις των απογόνων της *sub_root* εξαρτώνται από το αν η *sub_root* μπήκε στο K ή όχι.

Θεωρούμε ένα υπόδεντρο *sub_T* με ρίζα *sub_root*. Εφόσον πρόκειται για υπόδεντρο, οι κορυφές του *sub_T* που μπορούμε να εισάγουμε στο K, είναι ενδεχομένως λιγότερες από k, αφού μπορεί να έχουν ήδη μπει κορυφές $\notin$ *sub_T* στο σύνολο K. Έστω, λοιπόν, ότι μπορούμε να εισάγουμε i ακόμα κορυφές του *sub_T* στο K. Για να οπτικοποιήσουμε το πρόβλημα, δείχνουμε μια τυπική μορφή υποδέντρου όπως αυτή του Figure 9.

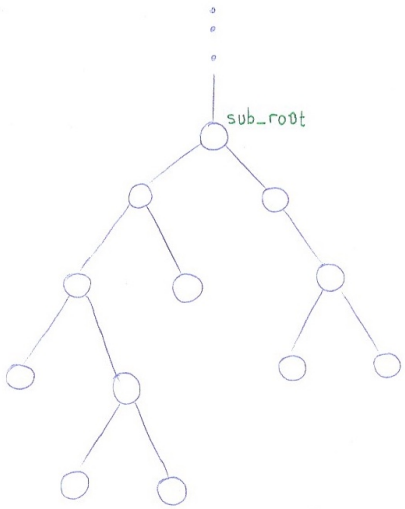

Figure 9: Υπόδεντρο *sub_T* με ρίζα *sub_root*

Τώρα υποθέτουμε πως η *sub_root* απέχει απόσταση *dist* από το σύνολο K. Οι αποστάσεις των απογόνων της *sub_root* από το σύνολο K εξαρτώνται από το αν η *sub_root* μπαίνει ή όχι το K. Για το υπόδεντρο του προηγούμενου παραδείγματος έχουμε τις δύο αυτές εκδοχές οπτικοποιημένες στα Figure 10 και Figure 11.

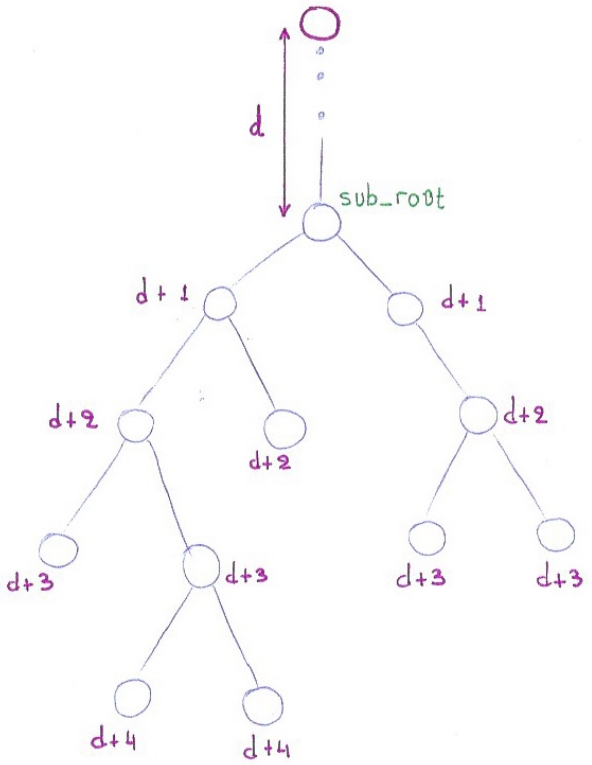

Figure 10: Η *sub_root* δεν μπαίνει στο Κ

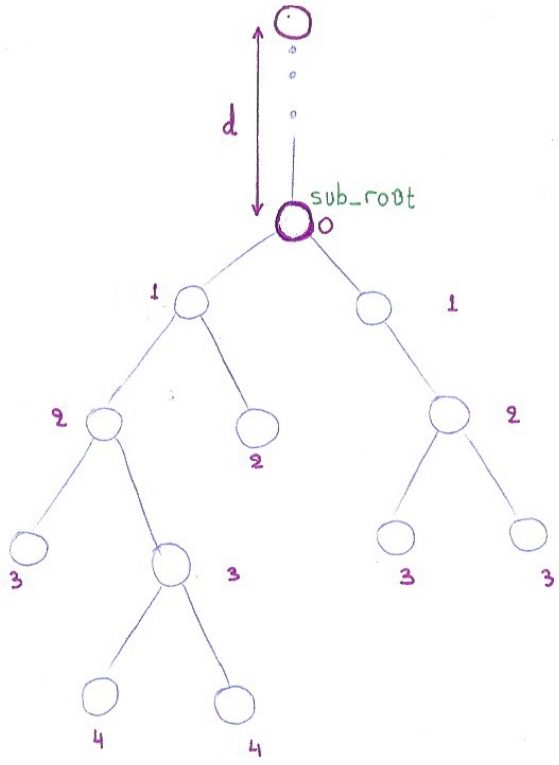

Figure 11: Η *sub_root* μπαίνει στο Κ

Επομένως, σε κάθε απόφαση εισαγωγής ή μη της *sub_root* στο Κ, πρέπει να ελέγξουμε τους πιθανούς τρόπους διαμοιρασμού των εναπομεινάντων θέσεων στο Κ.

- Αν η *sub_root* μπει στο Κ, τότε μένουν ακόμα $i-1$ κορυφές για να μπουν στο Κ. Αυτές θα είναι j από το αριστερό υπόδεντρο του sub_T και $i-1-j$ από το δεξί.
- Αν η *sub_root* δεν μπει στο Κ, τότε μένουν ακόμα i κορυφές για να μπουν στο Κ. Αυτές θα είναι j από το αριστερό υπόδεντρο του sub_T και $i-j$ από το δεξί.

Και στις δύο παραπάνω περιπτώσεις το j θα πρέπει να είναι το καλύτερο δυνατό, δηλαδή εκείνο που ελαχιστοποιεί το κόστος κάλυψης των δύο υποδέντρων του sub_T.

Επομένως, σε κάθε υποπρόβλημα πρέπει να επιλέξουμε μία από δύο εκδοχές (θα μπει η ρίζα ή όχι) και για κάθε εκδοχή θα πρέπει να επιλέξουμε τον καταλληλότερο διαμοιρασμό των εναπομεινάντων θέσεων στο Κ (το καταλληλότερο j).

Στην γενική περίπτωση που εξετάζουμε ένα υποδέντρο με ρίζα u που απέχει d από το σύνολο Κ και οι υπόλοιπες θέσεις που χωράει το Κ είναι i, το κόστος κάλυψης του υποδέντρου είναι:

$$cost[u,d,i] = \min\{(u \to K), (u \not\to K)\}$$

5.1.2 Αναδρομική Σχέση

Η συγκεντρωτική αναδρομική σχέση που μπορεί να δώσει και το τελικό ζητούμενο είναι:

$$cost[u,d,i] = \begin{cases} d & ,u : leaf \wedge i = 0 \\ 0 & ,u : leaf \wedge i > 0 \\ \min\{(u \to K), (u \not\to K)\} & ,otherwise \end{cases}$$

όπου:

$$(u \to K) = \min_{j \le i-1} \Big\{ max\Big\{ cost[u \to left, 1, j], cost[u \to right, 1, i-1-j] \Big\} \Big\}$$

$$(u \not\to K) = \min_{j \le i} \Big\{ max\Big\{ cost[u \to left, d+1, j], cost[u \to right, d+1, i-j], d \Big\} \Big\}$$

Ο δυναμικός προγραμματισμός που χρησιμοποιούμε στην λύση μας βασίζεται στον τρόπο με τον οποίο θα εκμεταλλευτούμε το αποτέλεσμα των μικρότερων υποδέντρων. Σκεφτόμαστε ότι αν γνωρίζουμε το ελάχιστο κόστος για τα δύο παιδιά μιας ρίζας *sub_root* και για κάθε συνδυασμό (d,i), τότε μπορούμε να αποφασίσουμε για κάθε έναν συνδυασμό από αυτούς αν είναι συμφέρον να βάλουμε την *sub_root* στο K. Έτσι καταλήγουμε στον παρακάτω αλγόριθμο.

5.1.3 Αλγόριθμος

Αλγόριθμος Tree_Cover

1. Για όλες τις κορυφές του χαμηλότερου επιπέδου υπολογίζουμε το cost[u,d,i], για όλους του δυνατούς συνδυασμούς (d,i).

2. Για κάθε δυνατό συνδυασμό πρέπει να βρούμε το κατάλληλο j (για κάθε μια από τις εκδοχές $(u \to K)$ και $(u \not\to K)$). Οπότε για κάθε συνδυασμό (d,i), βρίσκουμε το cost[u,d,i] δοκιμάζοντας όλα τα δυνατά j και κρατώντας το καλύτερο. Για κάθε έναν εκ των άνω συνδυασμών αποθηκεύουμε επιπλέον αν η κορυφή συμφέρει να μπει στο K ή όχι, μαζί με το βέλτιστο μοίρασμα των j θέσεων στα υποδέντρα. (Αυτό το βήμα είναι απαραίτητο για να ανακατασκευάσουμε στο τέλος του αλγορίθμου το σύνολο K)

3. Επαναλαμβάνουμε το ίδιο για κάθε ένα από τα ανώτερα επίπεδα, ανεβαίνοντας μέχρι να φτάσουμε στην ρίζα του αρχικού δέντρου.

4. Για τον υπολογισμό του cost[u,d,i] της αρχικής ρίζας, θα πρέπει αναγκαστικά να επιλέξουμε απευθείας την επιλογή $(u \to K)$.

5.1.4 Υπολογιστική Πολυπλοκότητα

Η υπολογιστική πολυπλοκότητα του αλγορίθμου είναι $\underbrace{O(n)}_{A} \times \underbrace{O(n \cdot k)}_{B} \times \underbrace{O(k)}_{\Gamma} = O(n^2k^2)$.

- A: Διασχίζουμε τις κορυφές κάθε επιπέδου, άρα εν τέλει όλες τις κορυφές
- B: Για κάθε κορυφή, δοκιμάζουμε όλους τους συνδυασμούς (d,i)
- Γ: Για κάθε συνδυασμό (d,i), πρέπει να βρούμε το κατάλληλο j μέσα από το πολύ k επιλογές (για κάθε μια από τις εκδοχές $(u \to K)$ και $(u \not\to K)$).

5.1.5 Ορθότητα Αλγορίθμου

Με το τέλος του αλγορίθμου έχε υπολογιστεί το ελάχιστο κόστος για την κάλυψη του δέντρου με χρήση του σύνολου K κορυφών. Παράλληλα με την εκτέλεση φροντίζαμε να αποθηκεύουμε επιπλέον αν η κορυφή συμφέρει να μπει στο K ή όχι, μαζί με το βέλτιστο μοίρασμα των i θέσεων στα υποδέντρα. Έτσι μπορούμε με μια αναδρομική διάσχιση του δέντρου να ανακατασκευάσουμε το σύνολο K.

1. Ξεκινάμε από την ρίζα με $i = k$ εναπομείναντα στοιχεία και $d = 0$.
2. Προσθέτουμε την ρίζα στο K.
3. Από εδώ και πέρα επαναλαμβάνουμε για τις άλλες κορυφές, εκτός της αρχικής ρίζας.
4. Για το αριστερό παιδί, κάνουμε αναδρομή με j (γνωστό-αποθηκευμένο j) εναπομείναντα στοιχεία και $d+1$ απόσταση αν η ρίζα δεν μπηκε στο K, αλλιώς 1. Για το δεξί κάνουμε αναδρομή με $i-j$ εναπομείναντα στοιχεία και $d+1$ απόσταση αν η ρίζα δεν μπηκε στο K, αλλιώς 1.
5. Ανάλογα με το (i, d), προσθέτουμε ή όχι στο K τη ρίζα.
6. Επαναλαμβάνουμε αναδρομικά (από το βήμα 3) μέχρι να φτάσουμε σε φύλλο, το οποίο μπαίνει ή όχι στο K ανάλογα με το αν έχουν περισσέψει θέσεις στο K.

Έτσι, προκύπτει τελικά το ζητούμενο σύνολο K σε συνολικό χρόνο $O(n^2k^2)$.

5.2 Ερώτημα (β)

5.2.1 Ιδέα Αλγορίθμου

Με δεδομένο ένα ανώτατο κόστος κάλυψης δέντρου, έστω $z \leq n$, γνωρίζουμε ότι υπάρχει τουλάχιστον μία κάλυψη με κόστος $\leq z$. Από αυτές, εμείς θέλουμε αυτή με τις λιγότερες κορυφές.

Διαισθητικά, θα θέλαμε μια κάλυψη με αραιά τοποθετημένες κορυφές, ώστε λίγες κορυφές να καλύπτουν ολόκληρο το δέντρο. Πόσο αραιά τοποθετημένες μπορεί να είναι; Κάθε κορυφή u της κάλυψης μπορεί να είναι το πολύ σε απόσταση $z+1$ από την προηγούμενή της v (η οποία είναι πρόγονος της u), διαφορετικά υπάρχει κορυφή στο μεσοδιάστημα $v \to u$ (τουλάχιστον ο γονέας της v) που απέχει από το σύνολο K περισσότερο από z.

Προτείνουμε, λοιπόν, τον παρακάτω άπληστο αλγόριθμο.

5.2.2 Αλγόριθμος

Άπληστος Αλγόριθμος

1. Πηγαίνουμε σε ένα φύλλο του κατώτερου επιπέδου
2. Ανεβαίνουμε σε απόσταση z
3. Αν φτάσουμε στη ρίζα του δέντρου την βάζουμε στο K και τερματίζουμε. Αλλιώς:
4. Εισάγουμε την κορυφή r στην οποία φτάσαμε στο σύνολο K
5. "Κόβουμε" το υποδέντρο με ρίζα την r
6. Επαναλαμβάνουμε από το βήμα 1, με το εναπομείνον δέντρο.

Η πολυπλοκότητα του αλγορίθμου οφείλεται στο ότι θα διασχίσουμε κάθε φύλλο το πολύ μια φορά. Είναι δηλαδή $O(n)$*.

5.2.3 Ορθότητα

Ο αλγόριθμος παράγει μια κάλυψη K με κόστος $\leq z$. Το ζητούμενο είναι τώρα να αποδείξουμε ότι αυτή η εξασφαλίζει και ελάχιστο αριθμό κορυφών. Για να το πετύχουμε αυτό, θα δείξουμε ότι με δεδομένη μια βέλτιστη λύση, μπορούμε να την μετατρέψουμε σε λύση που παράγει ο δικός μας αλγόριθμος διατηρώντας την ελαχιστότητα των κορυφών. Για αρχή θα χρειαστεί να αποδείξουμε ότι ισχύει η Αρχή της Βελτιστότητας.

Αρχή Βελτιστότητας

Θεωρούμε ένα δέντρο και μια βέλτιστη κάλυψη του δέντρου. Εντοπίζουμε μια κορυφή του δέντρου που ανήκει στην κάλυψη. Τότε αν αφαιρέσουμε αυτή την κορυφή και όλο το υπόδεντρο που την έχει ως ρίζα, τότε το δέντρο που περισσεύει καλύπτεται βέλτιστα από το εναπομείνον K.

Αυτό αν δεν συνέβαινε, τότε επανατοποθετώντας το κομμένο υπόδεντρο στη θέση του (καμία κορυφή αυτού του υποδέντρου δεν καλύπτει κάποιον πρόγονο, καθώς πρόγονοι δεν μπορεί να καλύπτονται από απογόνους), το προκύπτον δέντρο δε θα μπορούσε να είναι βέλτιστα καλυμμένο.

Ορθότητα Αλγορίθμου

Θεωρούμε μια βέλτιστη λύση $K*$. Έστω i η κορυφή που επιλέγει πρώτα ο greedy αλγόριθμος. Τότε αν η $K*$ δεν περιέχει την i, θα δείξουμε ότι υπάρχει μια άλλη λύση που είναι βέλτιστη και περιέχει την i. Ο αλγόριθμός μας έχει επιλέξει την i για να καλύψει το κατώτερο φύλλο, από το οποίο έχει και τη μέγιστη δυνατή απόσταση. Αφού η $K*$ δεν επιλέγει την i θα πρέπει για την κάλυψη του κατώτερου φύλλου να έχει επιλέξει κάποια κορυφή-απόγονο της i, έστω j. Τώρα η j καλύπτει ένα υποσύνολο από τις κορυφές που κάλυπτε η i. Επομένως, αν στη λύση $K*$ αντικαταστήσουμε την

j από την i, το κόστος κάλυψης παραμένει ίδιο, ενώ οι κορυφές του K* είναι ίσες σε πλήθος με πριν. Άρα η νέα $K*$ παραμένει βέλτιστη (επιχείρημα ανταλλαγής).

Κόβουμε τώρα το υπόδεντρο με ρίζα την i. Λόγω της Αρχής της Βελτιστότητας, το δέντρο που απομένει και είναι καλυμμένο από το εναπομείνον $K*$ είναι βέλτιστα καλυμμένο. Αν συνεχίσουμε την παραπάνω διαδικασία, σε κάθε βήμα αντικαθιστούμε μια κορυφή της βέλτιστης λύσης με μια κορυφή που θα επέλεγε ο greedy αλγόριθμος και η βέλτιστη λύση $K*$ μετατρέπεται σε μια (επίσης βέλτιστη) λύση που θα παρήγαγε ο greedy αλγόριθμος.

Επομένως, ο greedy αλγόριθμος που σχεδιάσαμε είναι βέλτιστος.

5.2.4 Βελτίωση Πολυπλοκότητας (α) ερωτήματος

Σκοπός του ερωτήματος (β) είναι να χρησιμοποιηθεί για να βελτιώσει την πολυπλοκότητα του αλγορίθμου του (α). Αν γνωρίζαμε στο (α) ερώτημα το ελάχιστο κόστος z που θα έχει η κάλυψη με k κορυφές, τότε θα εκτελούσαμε τον αλγόριθμο του (β) ερωτήματος για να βρούμε ένα σύνολο K σε $O(n)$.

Μια πρώτη προσέγγιση θα ήταν να δοκιμάσουμε όλα τα $z \leq n$ και να κρατήσουμε εκείνη τη λύση του greedy αλγορίθμου με μέγεθος το πολύ k. Αυτό θα μας κόστιζε $O(n^2)$ σε χρονική πολυπλοκότητα.

Παρατηρούμε, ωστόσο, το εξής χαρακτηριστικό του προβλήματος. Έστω z το κόστος για κάλυψη k κορυφών. Αν αυξήσουμε το z, τότε οι κορυφές που χρειάζονται είναι λιγότερες ή ίσες από k. Άρα όσο αυξάνεται το z, τόσο μειώνεται το k. Θα λέγαμε, λοιπόν, πως το k είναι συνάρτηση του z και, μάλιστα, φθίνουσα. Αν, λοιπόν, το z θεωρηθεί δείκτης σε πίνακα με στοιχεία $A[z] = k$, τότε αυτός ο πίνακας είναι ταξινομημένος σε φθίνουσα σειρά.

Έτσι, αντί να δοκιμάσουμε όλα τα z για να βρούμε εκείνο που δίνει το κατάλληλο k (που είναι το πολύ όσο το ζητούμενο πλήθος συνόλου K), κάνουμε $binary\_search()$ στα z.

Μέχρι να βρούμε το κατάλληλο z θα έχουμε χρονικό κόστος $O(logn)$ αναζητήσεων, ενώ σε καθε αναζήτηση θα εκτελούμε τον $O(n)$ greedy αλγόριθμο.

Τελική χρονική πολυπλοκότητα $O(nlogn)$.

---

*Στην διατύπωση του αλγορίθμου, δεν διευκρυνίζεται το πως μπορούμε να βρίσκουμε το κατώτερο φύλλο του εναπομείναντος δέντρου χωρίς να αυξάνεται η τάξη μεγέθους της χρονικής πολυπλοκότητας. Αυτό μπορεί να επιτευχθεί ως εξής:

- Δημιουργούμε μια δομή (πχ πίνακα με απλά συνδεδεμένες λίστες) στην οποία ομαδοποιούμε τις κορυφές ίδιου επιπέδου ($A[i]$ = απλά συνδεδεμένη λίστα με στοιχεία τις κορυφές επιπέδου i), με μια διάσχιση του δέντρου σε $O(n)$.

- Ύστερα ενώνουμε τις απλά συνδεδεμένες λίστες, τοποθετώντας πιο μπροστά τις λίστες με τις κορυφές κατώτερου επιπέδου, σε $O(n)$. Έτσι, έχουμε δημιουργήσει μια ουρά που περιέχει τις κορυφές ταξινομημένες ως προς το επίπεδό τους σε $O(n)$.

- Στο βήμα (1) του αλγορίθμου, "πηγαίνουμε σε ένα φύλλο του κατώτερου επιπέδου". Αυτό το φύλλο είναι κάθε φορά το πρώτο στοιχείο της ουράς μας. Καθώς ανεβαίνουμε προς τα πάνω φύλλα, κάθε φύλλο που προσπερνάμε, το μαρκάρουμε. Έτσι για να "πάμε σε ένα φύλλο του κατώτερου επιπέδου" ελέγχουμε πρώτα αν είναι μαρκαρισμένο. Αν είναι, τότε αυτό σημαίνει πως έχει κοπεί, άρα το προσπερνάμε και πάμε στο επόμενο.

- Έτσι εξασφαλίζουμε, ότι στην χειρότερη περίπτωση θα "επενεργήσουμε" πάνω σε κάθε φύλλο του δέντρου το πολύ δύο φορές (μία όταν μεταβαίνουμε σε αυτό καθώς ανεβαίνουμε και μία όταν το συναντάμε στην ουρά) $\implies 2 \cdot n$.

- Έτσι η τάξη μεγέθους της πολυπλοκότητας παραμένει $O(n)$.