034 - Ανάλυση και Σχεδιασμός Αλγορίθμων
Προαιρετική Εργασία

Διδάσκων: Επίκ. Καθ. Παναγιώτης Πετραντωνάκης

Ομάδα 54
Ιωάννης Δημουλιός 10641
Χριστόφορος Μαρινόπουλος 10522

Εαρινό εξάμηνο 2024

Πρόβλημα 1

Ερώτημα 1

Θέλουμε να δούμε αν είναι εφικτό ένα δρομολόγιο από την πόλη s στην πόλη t δίχως να χρησιμοποιήσουμε ακμές e με μήκος $l_e > L$.
Επομένως, φτιάχνουμε ένα νέο γράφο $G' = (V, E')$, ο οποίος διαφέρει από τον αρχικό μόνο στις ακμές. Το νέο σύνολο ακμών E' δεν έχει τις ακμές που αναφέρονται παραπάνω, δηλαδή:

$$E' = \{e \in E \mid l_e \leq L\}$$

. Στον γράφο G' τρέχουμε τον αλγόριθμο DFS ξεκινόντας από την κορυφή s και ελέγχουμε έτσι αν υπάρχει μονοπάτι μέχρι την κορυφή t, που είναι και το ζητούμενο. Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(|V| + |E|)$, αφού προσπελαύνουμε κάθε κορυφή και κάθε ακμή του γράφου G μια φορά. ■

Ερώτημα 2

Θέλουμε να βρούμε το ελάχιστο μήκος του μακρύτερου δρόμου που πρέπει να διασχίσουμε από την πόλη s στην πόλη t. Αυτό το μήκος θα είναι και το ζητούμενο L.
Επομένως θα χρησιμοποιήσουμε τον αλγόριθμο Dijkstra ελαφρώς τροποποιημένο, ώστε το βάρος της κάθε κορυφής v να μην είναι πλέον το ελάχιστο άθροισμα των μηκών των ακμών ενός μονοπατιού από την κορυφή s έως την v, αλλά το ελάχιστο των μέγιστων μηκών των ακμών αυτών των μονοπατιών.
Για να γίνει πιο ξεκάθαρο, αν P το σύνολο των μονοπατιών από την κορυφή s μέχρι την v και, $p \in P$ ένα τέτοιο μονοπάτι, τότε ορίζουμε

$$f(p) = \max_{e \in p} l_e$$

και αν $w(\cdot)$ η συνάρτηση βάρους μιας κορυφής, τότε

$$w(v) = \min_{p \in P} f(p)$$

. Ο αλγόριθμος, λοιπόν, βρίσκει και το ζητούμενο $w(t)$.
Το μόνο που χρειάζεται να αλλάξουμε στην υλοποίηση σε σύγκριση με τον κλασικό Dijkstra είναι η συνάρτηση "χαλάρωσης", ώστε να αντικατοπτριστεί η παραπάνω αλλαγή (βλ. σχετικό παράρτημα).
Η χρονική πολυπλοκότητα του Dijkstra, εφόσον η ουρά προτεραιότητας που απαιτείται υλοποιηθεί με δυαδικό σωρό (binary heap), είναι $O((|V| + |E|) \log |V|)$. ■

Πρόβλημα 2

Έστω ότι η ουρά εξυπηρέτησης αποτελείται απο n πολίτες και ο χρόνος αναμονής του κάθε πολίτη i είναι A_i για $1 \leq i \leq n$. Ο χρόνος αναμονής του i-οστού πολίτη (A_i) για είναι το άθροισμα των χρόνων εξυπηρέτησης των πολιτών που εξυπηρετούνται πριν απο αυτόν. Εάν ο χρόνος εξυπηρέτησης του i-οστού πολίτη είναι ίσος με S_i και ορίσουμε κατά σύμβαση $S_0 = 0$, τότε

$$A_i = S_0 + S_1 + S_2 + ... + S_{i-1}$$

. Ο συνολικός χρόνος αναμονής είναι το άθροισμα των χρόνων αναμονής κάθε πολίτη που ανήκει στην ουρά.
Είναι λογικό, λοιπόν, πως για αυτή την ουρά ο συνολικός χρόνος αναμονής είναι ίσος με:

$$\begin{aligned} t(n) &= A_1 + A_2 + ... + A_n \\ &= (n-1)S_1 + (n-2)S_2 + ... + S_{n-1} \end{aligned}$$

Έστω μία ουρά αναμονής πλήθους n πολιτών με συνολικό χρόνο εξυπηρέτησης $t_1(n)$ η οποία δεν είναι ταξινομημένη κατά αύξουσα σειρά χρόνου εξυπηρέτησης κάθε πολίτη.
Τότε, υπάρχει σε αυτή τουλάχιστον ένα ζευγάρι πολιτών (x, y) με $x < y$ τέτοιο ώστε $S_x > S_y$ (συνθήκη **Σ**), οπότε

$$t_1(n) = (n-1)S_1 + (n-2)S_2 + ... + (n-x)S_x + ... + (n-y)S_y + ... + S_{n-1}$$

. Εάν αλλάξουμε την θέση του x με τον y, ο νέος συνολικός χρόνος εξυπηρέτησης $t_2(n)$ αυτής της ουράς θα είναι:

$$\begin{aligned} t_2(n) &= (n-1)S_1 + (n-2)S_2 + ... + (n-x)S_y + ... + (n-y)S_x + ... + S_{n-1} \\ &= t_1(n) - (y-x)(S_x - S_y) \end{aligned}$$

Αφού $x < y$ και $S_x > S_y$, προκύπτει ότι

$$-(y-x)(S_x - S_y) < 0 \implies t_2(n) < t_1(n)$$

. Κάνουμε "άπληστα" εναλλαγές ζευγαριών πολιτών (x, y) που ικανοποιούν τη συνθήκη **Σ**, έως ότου να μην υπάρχει άλλο τέτοιο ζεύγος, δηλαδή η ουρά να έχει ταξινομηθεί. Μετά από κάθε εναλλαγή ο συνολικός χρόνος αναμονής με βάση τα παραπάνω μειώνεται και άρα ελαχιστοποιείται ακριβώς όταν η ουρά ταξινομηθεί.
Αποδείξαμε, λοιπόν, ότι αρκεί να ταξινομήσουμε τους πολιτές κατά αύξοντα χρόνο εξυπηρέτησης.
Ένας αποδοτικός αλγόριθμος για την ταξινόμηση της ουράς εξυπηρέτησης είναι heap sort, αφού έχει χρονική πολυπλοκότητα $O(n \log n)$. ■

Πρόβλημα 3

Ορίζουμε ότι οι δείκτες πινάκων και συμβολοσειρών ξεκινούν από το 0 και υποθέτουμε ότι $i+1$-οστό στοιχείο του πίνακα X είναι το x_i.

Δεν μας νοιάζει η συμβολοσειρά που θα χωρίσουμε αυτούσια, όσο το πλήθος των χαρακτήρων της, έστω n.
Υποθέτουμε ότι οι τομές θα γίνουν στις θέσεις της συμβολοσειράς b_i για $1 \leq i \leq m$ και προσθέτουμε στον πίνακα των τομών το $b_0 = -1$ στην αρχή και το $b_{m+1} = n-1$ στο τέλος, ώστε η αρχή και το τέλος της συμβολοσειράς να εμφανίζονται ως τομές.
Οπότε ο πίνακας των τομών γράφεται

$$B = (-1, b_1, \ldots, b_m, n-1)$$

. Έστω

$$S = (s_0, \ldots, s_{n-1})$$

η αρχική συμβολοσειρά.
Ορίζουμε

$$S_{i,j} = (s_{b_i+1}, \ldots s_{b_j})$$

με $0 \leq i < j \leq m+1$. Τότε $S = S_{0,m+1}$.
Με το πέρας της διαδικασίας θέλουμε να έχουμε τις $m+1$ συμβολοσειρές

$$S_{0,1}, S_{1,2}, \ldots S_{m,m+1}$$

. Ορίζουμε τον $(m+2) \times (m+2)$ πίνακα D του οποίου το στοιχείο $d_{i,j}$ αναπαριστά τις ελάχιστες μονάδες χρόνου που απαιτούνται, ώστε με χρήση των τομών να λάβουμε από την $S_{i,j}$ τις

$$S_{i,i+1}, \ldots, S_{j-1,j}$$

Σκοπός του προβλήματος, λοιπόν είναι να υπολογίσουμε το $d_{0,m+1}$.
Εφόσον, στην $S_{i,i+1}$ δεν μπορούμε να κάνουμε άλλη τομή έχουμε

$$d_{i,i+1} = 0 \quad \text{για} \quad 0 \leq i \leq m$$

. Τώρα όμως για διαφορά δεικτών $j - i > 1$ προκύπτει

$$\begin{aligned} d_{i,j} &= \min_{i<k<j} \left((d_{i,k} + b_k - b_i) + (d_{k,j} + b_j - b_k)\right) \\ &= \min_{i<k<j} \left(d_{i,k} + d_{k,j} + b_j - b_i\right) \end{aligned}$$

. Υπολογίζουμε, έτσι, με τη χρήση δυναμικού προγραμματισμού τα στοιχεία του πίνακα D πάνω από την κύρια διαγώνιό του, προσπελαύνοντας τις διαγωνίους σταθερής διαφοράς δεικτών, έως ότου φτάσουμε στο στοιχείο $d_{0,m+1}$.

Η χρονική πολυπλοκότητα του αλγορίθμου προκύπτει ως εξής:

- Προσπελαύνουμε τα μισά (όλα πάνω από την κύρια διαγώνιο) στοιχεία του πίνακα D διαστάσεων $(m+2) \times (m+2)$, άρα $O(m^2)$.
- Για τον υπολογισμό κάθε $d_{i,j}$ έχουμε έναν βρόχο με το πολύ m επαναλήψεις, άρα $O(m)$.

Επομένως, τελικά είναι $O(m^3)$. ■

Παράρτημα

Παραθέτουμε ενδεικτικές υλοποιήσεις σε Python για κάθε πρόβλημα.

Πρόβλημα 1

Ερώτημα 1

Ο αλγόριθμος `dfs` υλοποείται αναδρομικά. Τα ορίσματα της συνάρτησης είναι τα εξής:

- `graph`, δομή dictionary και αναπαριστά την λίστα γειτνίασης του γράφου G,
- `s`, κόμβος έναρξης του αλγορίθμου,
- `t`, κόμβος προορισμού,
- `L`, μέγιστη απόσταση μεταξύ δύο διαδοχικών πόλεων,
- `visited`, βοηθητικός πίνακας που αποθηκεύει τις πόλεις που έχουμε επισκεφτεί.

Επιστρέφει `True` αν είναι δυνατή η μετάβαση από την πόλη s στην πόλη t, αλλιώς `False`.

```python
def dfs(graph, s, t, L, visited = set()):
    if s == t:
        return True
    visited.add(s)

    for adj, length in graph[s]:
        if length <= L and adj not in visited:
            if dfs(graph, adj, t, L, visited):
                return True

    return False
```

Ερώτημα 2

Τα ορίσματα της συνάρτησης είναι τα εξής:

- graph, δομή dictionary και αναπαριστά την λίστα γειτνίασης του γράφου G,
- s, κόμβος έναρξης του αλγορίθμου,
- t, κόμβος προορισμού.

Ο αλγόριθμος επιστρέφει το ελάχιστο δυνατό L, εφόσον είναι δυνατή η μετάβαση από την πόλη s στην πόλη πόλη t, αλλιώς επιστρέφει άπειρο.
Η νέα ρουτίνα "χαλάρωσης" φαίνεται στη σειρά 16.

```
1  import heapq
2  
3  def dijkstra(graph, s, t):
4      priority_queue = [(0, s)]
5  
6      weights = {v: float('inf') for v in graph}
7      weights[s] = 0
8  
9      while priority_queue:
10         weight, v = heapq.heappop(priority_queue)
11 
12         if v == t:
13             return weight
14 
15         for adj, length in graph[v]:
16             weight_updated = max(weight, length)
17 
18             if weight_updated < weights[adj]:
19                 weights[adj] = weight_updated
20                 heapq.heappush(priority_queue, (weight_updated, adj))
21 
22     return float('inf')
```

Πρόβλημα 2

Η συνάρτηση heapSort παίρνει ως όρισμα τον πίνακα arr με τους χρόνους εξυπηρέτησης και τον ταξινομεί χρησιμοποιώντας τη βοηθητική συνάρτηση heapify.

```
def heapify(arr, n, i):
        largest = i # Initialize largest as root
        l = 2 * i + 1 # left = 2*i + 1
        r = 2 * i + 2 # right = 2*i + 2
        if l < n and arr[i] < arr[l]:
                largest = l
        if r < n and arr[largest] < arr[r]:
                largest = r
        if largest != i:
                (arr[i], arr[largest]) = (arr[largest], arr[i])
                heapify(arr, n, largest)

def heapSort(arr):
        n = len(arr)
        for i in range(n // 2, -1, -1):
                heapify(arr, n, i)
        for i in range(n - 1, 0, -1):
                (arr[i], arr[0]) = (arr[0], arr[i]) # swap
                heapify(arr, i, 0)
```

Πρόβλημα 3

Ο αλγόριθμος `breakString` παίρνει ως όρισμα το μήκος της συμβολοσειράς `n` και τον πίνακα των θέσεων των τομών `B` και επιστρέφει το ελάχιστο υπολογιστικό κόστος σε μονάδες χρόνου για να γίνουν οι ζητούμενες τομές στη συμβολοσειρά.

```python
import numpy as np

def breakString(n, B):
    m = len(B)
    B = [-1] + B + [n-1]

    time_units = np.full(shape=(m+2, m+2), fill_value=np.inf)

    for i in range(m+1):
        time_units[i, i+1] = 0

    for delta in range(2, m + 2):
        for i in range(0, m + 2 - delta):
            j = i + delta

            minimum = np.inf
            for k in range(i+1, j):
                time_curr = time_units[i, k] + time_units[k, j] + B[j] - B[i]
                if time_curr < minimum:
                    minimum = time_curr

            time_units[i, j] = minimum

    return time_units[0, m+1]
```