Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Προαιρετική Εργασία για το μάθημα Ανάλυση και Σχεδιασμός Αλγορίθμων

Διδάσκων: Επικ. Καθ. Παναγιώτης Πετραντωνάκης (ppetrant@ece.auth.gr)
5 Απριλίου 2024

Εαρινό εξάμηνο 2023/24

Ομάδα 21

Δεϊρμεντζόγλου Ιωάννης
Α.Ε.Μ.: 10015
Email: deirmentz@ece.auth.gr

Τριανταφυλλίδης Βασίλειος
Α.Ε.Μ.: 10057
Email: vastridam@ece.auth.gr

Περιεχόμενα

Πρόβλημα 1 3

Πρόβλημα 2 4

Πρόβλημα 3 5

Παράρτημα 7

Πρόβλημα 1

Ζητούμενο για το 1ο σκέλος του προβλήματος είναι να βρούμε αν υπάρχει εφικτή διαδρομή από μία αρχική πόλη s σε μία τελική πόλη t, χρησιμοποιώντας ένα ρεζερβουάρ με μέγιστη απόσταση κάλυψης L χιλιομέτρων. Για αυτό , χρειάζεται η ανάπτυξη ενός αλγόριθμου που θα υπολογίζει την απόσταση μεταξύ των πόλεων, και ανάλογα με το L (απόσταση που μπορεί να καλυφθεί με γεμάτο ρεζερβουάρ) θα καθορίζει αν μπορεί να μετακινηθεί από την πόλη του στον τελικό προορισμό.

Για να βρούμε αν υπάρχει εφικτή διαδρομή από την πόλη s στην πόλη t με τον περιορισμό της απόστασης, μπορούμε να ακολουθήσουμε τα παρακάτω βήματα:

1. Φιλτράρισμα των Ακμών: Δημιουργούμε έναν νέο γράφο G ’=(V,E’) όπου E’ περιλαμβάνει μόνο τις ακμές $e \in E$ με μήκος $l_e \leq L$. Αυτό το βήμα αφαιρεί όλους τους δρόμους που δεν μπορεί να διανύσει το αυτοκίνητο με γεμάτο ρεζερβουάρ.

2. Διάσχιση Γράφου: Χρησιμοποιούμε έναν τυπικό αλγόριθμο διάσχισης γράφου (όπως BFS ή DFS) στον τροποποιημένο γράφο G’ για να ελέγξουμε αν υπάρχει μονοπάτι από την πόλη s στην πόλη t. Για αυτήν την άσκηση υλοποιήθηκε ο αλγόριθμος BFS.

Ο χρόνος εκτέλεσης εκτιμήθηκε $O(|V|+|E|)$.

Η υλοποίηση σε python καθώς περισσότερες επεξηγήσεις και παραδείγματα μαζί με την ανάλυση χρονικής πολυπλοκότητας βρίσκονται στο παράρτημα 1.

Στο 2ο σκέλος του προβλήματος ζητείται η υλοποίηση ενός αλγορίθμου για την εύρεση της ελάχιστης χωρητικότητας καυσίμων που αρκούν για την μετακίνηση από την πόλη s στην πόλη t. Ο αλγόριθμος που χρησιμοποιήσαμε λειτουργεί ως εξής:

1. Μελέτη Γράφου: Καταγράφουμε τις αποστάσεις μεταξύ όλων των ακμών με τις γειτονικές τους και τις ταξινομούμε σε μία λίστα.

2. Φιλτράρισμα Ακμών: Δημιουργούμε έναν νέο γράφο όπως στο πρώτο σκέλος.

3. Διάσχιση Γράφου: Χρησιμοποιούμε BFS στον νέο γράφο όπως στο πρώτο σκέλος.

4. Υπολογισμός Ντεπόζιτου: Εντοπίζουμε μέσω binary search την μεγαλύτερη υποδιαδρομή από την πόλη s στην πόλη t και επιστρέφουμε το μήκος της, καθώς αυτό ορίζει την ελάχιστη χωρητικότητα του ντεπόζιτου.

Πρόβλημα 2

Σε αυτό το πρόβλημα ζητείται η δημιουργία ενός αποδοτικού αλγορίθμου για τον υπολογισμό της βέλτιστης σειράς εξυπηρέτησης σε μία δημόσια υπηρεσία για δεδομένο χρόνο εξυπηρέτησης του κάθε πελάτη και η απόδειξη της ορθότητάς του. Ο τρόπος λειτουργίας της δημόσιας υπηρεσίας για την εξυπηρέτηση, είναι πως οποιοσδήποτε πελάτης έρθει πρώτος εξυπηρετείται πρώτος, έχουμε δηλαδή μια ουρά προτεραιότητας First In First Out (FIFO), χωρίς να λαμβάνεται υπόψη ο χρόνος που απαιτείται για να εξυπηρετηθεί και ο συνολικός χρόνος αναμονής.

Επειδή ο σκοπός του αλγορίθμου είναι η ελαχιστοποίηση του χρόνου αναμονής των πελατών, θέτει σε προτεραιότητα τους πελάτες με τον μικρότερο χρόνο εξυπηρέτησης. Επομένως, η βασική ιδέα είναι μια ουρά προτεραιότητας (**priority queue**), η οποία θα ταξινομείται βάσει του χρόνου εξυπηρέτησης κάθε πελάτη, και όχι με αποκλειστικά με βάση την ώρα άφιξης του κάθε πελάτη. Αυτό επιτυγχάνεται με τα εξής βήματα:

Αρχικά, δημιουργείται μια κενή ουρά με σκοπό να περιέχει τους πελάτες που είναι να εξυπηρετηθούν. Οι πελάτες καταχωρούνται στην ουρά την στιγμή που φτάνουν στην δημόσια υπηρεσία, ταξινομούνται με βάση τον χρόνο εξυπηρέτησης και περιμένουν το τέλος της εξυπηρέτησης που λαμβάνει χώρα εκείνη την στιγμή. Όταν αυτή τελειώσει, τότε ξεκινάει η εξυπηρέτηση του πρώτου πελάτη σύμφωνα με τον πίνακα, δηλαδή αυτού με τον ελάχιστο χρόνο εξυπηρέτησης.

Λειτουργία ουράς προτεραιότητας:

Ο πρώτος πελάτης τοποθετείται κατευθείαν στην ουρά προτεραιότητας. Από τον δεύτερο πελάτη και μετά, γίνεται έλεγχος για τον χρόνο εξυπηρέτησής τους και έπειτα τοποθέτηση στην ουρά. Στην συνέχεια, καθώς έρχονται πελάτες και η ουρά αρχίζει να μην είναι άδεια, την χρονική στιγμή που έρθει ένας πελάτης, γίνεται έλεγχος στον χρόνο εξυπηρέτησής του, εφόσον είναι μικρότερος του χρόνου εξυπηρέτησης των μέχρι τώρα πελατών, θα μπει στο πρώτο μέρος της ουράς, ταξινομημένος. Αν, ο χρόνος εξυπηρέτησής του είναι μεγαλύτερος από τον χρόνο εξυπηρέτησης των πελατών που έχουν ήδη φτάσει θα τοποθετηθεί στο τέλος της ουράς.

Με αυτόν τον τρόπο είναι δυνατόν να ελαχιστοποιηθεί ο συνολικός χρόνος αναμονής. Ωστόσο, ένα πρόβλημα που προκύπτει είναι αυτό με πελάτες με μεγάλο χρόνο εξυπηρέτησης, οι οποίοι αν θέλουμε να ελαχιστοποιήσουμε πλήρως τον συνολικό χρόνο αναμονής όλων των πελατών είναι πιθανό να περιμένουν πάρα πολύ ώρα μέχρι να εξυπηρετηθούν (εάν εμφανίζονται συνεχώς πελάτες με

μικρό/μικρότερο χρόνο εξυπηρέτησης). Έτσι, μπορούμε να επιλέξουμε ένα συγκεκριμένο χρονικό οροί μέγιστης αναμονής, προκειμένου να εξυπηρετηθούν και αυτοί σε ένα εύλογο χρονικό διάστημα. Το κόστος αυτής της επιλογής θα είναι η μη βελτιστοποίηση του συνολικού χρόνου αναμονής.

Πρόβλημα 3

Στην παρούσα άσκηση, εξετάζουμε ένα πρόβλημα δυναμικού προγραμματισμού που αφορά τον υπολογισμό του ελάχιστου κόστους διαίρεσης μιας συμβολοσειράς σε πολλαπλά τμήματα με γνωστά σημεία διακοπής. Στόχος μας είναι η ανάπτυξη ενός αλγορίθμου που υλοποιεί αυτή τη λειτουργία και ο προσδιορισμός της χρονικής πολυπλοκότητάς του. Ακολουθώντας τη μέθοδο επίλυσης προβλημάτων με δυναμικό προγραμματισμό καλούμαστε να βρούμε μια βέλτιστη λύση, η οποία προκύπτει από βέλτιστες λύσεις σε υποπροβλήματα του αρχικού. Κάθε υποπρόβλημα μπορεί να προκύψει ως το τμήμα της συμβολοσειράς μεταξύ δύο διαδοχικών σημείων διαχωρισμού. Επομένως, για κάθε υποπρόβλημα μπορεί να υπολογιστεί το ελάχιστο κόστος σε time units για την διάσπαση του σε 2 υποσυμβολοσειρές. Ακολουθώντας επαναληπτικά αυτήν την διαδικασία είναι δυνατόν να υπολογιστεί το ελάχιστο κόστος για την ολοκλήρωση της διάσπασης της συμβολοσειράς από την αρχή μέχρι το τέλος, λαμβάνοντας υπόψη τα κόστη των υποπροβλημάτων.

Ο αλγόριθμος που αναπτύξαμε χρησιμοποιεί έναν πίνακα δυναμικού προγραμματισμού για να υπολογίσει το ελάχιστο κόστος διαίρεσης της συμβολοσειράς. Ο πίνακας costs είναι ένας πίνακας δυναμικού προγραμματισμού που χρησιμοποιείται για να αποθηκεύσει το κόστος του διαχωρισμού της συμβολοσειράς σε υπο-συμβολοσειρές μεταξύ διαφορετικών σημείων διαχωρισμού. Ο πίνακας αυτός έχει διαστάσεις (m+2) x (m+2), όπου m είναι ο αριθμός των σημείων διαχωρισμού. Κάθε στοιχείο του πίνακα costs[i][j] αντιπροσωπεύει το ελάχιστο κόστος διαχωρισμού της υπο-συμβολοσειράς που ξεκινά από τη θέση i και τελειώνει στη θέση j.

Καθορισμός του Υποπροβλήματος

Ας υποθέσουμε ότι έχουμε συμβολοσειρά εισόδου **S** μήκους **n**. Στη συνέχεια, ας υποδηλώσει επίσης το **i** την θέση στην οποία θέλουμε να διαχωρίσουμε την συμβολοσειρά. Ετσι, i = 1, … ,m .

Άρα, αν γίνουν m τομές, αυτό σημαίνει ότι έχουμε m+1 υποσυμβολοσειρές.

Ο $1^{ος}$ διαχωρισμός στην αρχική συμβολοσειρά :

Θα έχουμε δύο υποσυμβολοσειρές : S[1….i] και S[i+1….n]

Έτσι, με βάση τον επόμενο δείκτη που προκύπτει από τα σημεία διαχωρισμού χωρίζεται ένα από τα 2 substrings.

Η αναδρομική σχέση για τον υπολογισμό του ελάχιστου κόστους που χρησιμοποιείται είναι :

- **cost(i,j) = min {len(substring) + cost(i,k) + cost(k,j)**

  where i < k < j}

Στο παράρτημα παρατίθεται ο κώδικας σε python για την υλοποίηση του αλγορίθμου μέσω της συνάρτησης optimalStringSeparator που δέχεται ως είσοδο την συμβολοσειρά και τα σημεία διαχωρισμού.

Ο αλγόριθμος έχει κυβική χρονική πολυπλοκότητα **O (n^3) (3 recursive loops) .** Αυτό σημαίνει ότι η απόδοσή του μειώνεται δραματικά με την αύξηση του μεγέθους της εισόδου. Ως εκ τούτου, για μεγάλες εισόδους, θα μπορούσε να είναι απαραίτητος ο εντοπισμός μιας πιο αποδοτικής λύσης.

Παράρτημα

Πρόβλημα 1

Ο αλγόριθμος BFS είναι κατάλληλος για την εύρεση εφικτής διαδρομής μεταξύ δύο κόμβων σε έναν μη κατευθυνόμενο γράφο. Χρησιμοποιώντας BFS σε έναν τροποποιημένο γράφο όπου οι ακμές έχουν φιλτραριστεί βάσει του περιορισμού απόστασης, μπορούμε να ελέγξουμε αν υπάρχει εφικτή διαδρομή από μια πόλη s σε μια πόλη t με το δεδομένο όριο απόστασης L. Παρακάτω περιγράφεται εν συντομία η λογική του αλγορίθμου πάνω, στην οποία υλοποιήθηκε και ο κώδικας που παρουσιάζεται παρακάτω.

Επιλέγουμε έναν αρχικό κόμβο, τη ρίζα. Από αυτό το σημείο, ξεκινάμε να εξερευνούμε τους γειτονικούς κόμβους του. Αρχικά εξερευνούμε τους γείτονες της ρίζας, στη συνέχεια τους γείτονες των γειτόνων, και έτσι συνεχίζουμε, επισκέπτοντας τους κόμβους σε αυξανόμενη απόσταση από τη ρίζα. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να έχουμε εξερευνήσει όλους τους κόμβους του γράφου ή μέχρι να βρεθεί ο κόμβος προορισμού, αν αυτός υπάρχει. Χρησιμοποιούμε μια ουρά για να κρατάμε τους κόμβους που πρέπει να εξερευνήσουμε, ενώ σημειώνουμε τους επισκεπτόμενους κόμβους για να μην επαναλαμβάνουμε τις επισκέψεις.

Ο κώδικας, ο οποίος υλοποιήθηκε για τα παραδείγματα φαίνεται παρακάτω. Η συνάρτηση feasible_route() επιστρέφει True αν υπάρχει εφικτή διαδρομή από την πόλη s στην πόλη t με τον περιορισμό απόστασης L, διαφορετικά επιστρέφει False.

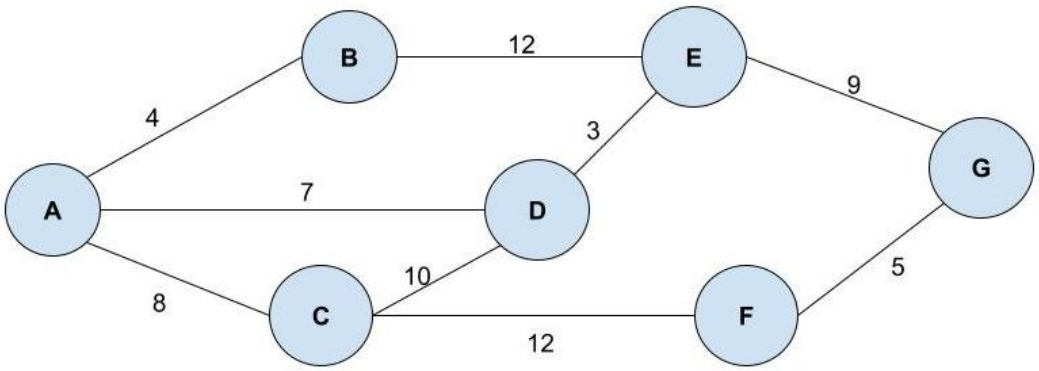
Οι είσοδοι της συνάρτησης feasible_route είναι οι ακόλουθες

- **V**: Μια λίστα που περιέχει τους κόμβους του γράφου.
- **E**: Μια λίστα που περιέχει τις ακμές του γράφου, κάθε ακμή αναπαρίσταται ως τριάδα (u, v, l) όπου u και v είναι οι κόμβοι που συνδέονται από την ακμή και l είναι το μήκος της ακμής.
- **s**: Ο κόμβος εκκίνησης της διαδρομής.
- **t**: Ο κόμβος προορισμού της διαδρομής.
- **L**: Το μέγιστο μήκος που μπορεί να καλυφθεί με μια πλήρης δεξαμενή καυσίμων.

```python
from collections import deque

def feasible_route(V, E, s, t, L):
    # Step 1: Filter the edges
    E_prime = [(u, v) for (u, v, l) in E if l <= L]

    # Create adjacency list for the modified graph G'
    adj = {v: [] for v in V}
    for (u, v) in E_prime:
        adj[u].append(v)
        adj[v].append(u)

    # Step 2: Use BFS to check connectivity from s to t
    def bfs(start, goal):
        queue = deque([start])
        visited = set()
        visited.add(start)

        while queue:
            node = queue.popleft()
            if node == goal:
                return True
            for neighbor in adj[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        return False

    return bfs(s, t)
```

Παραδείγματα

Έστω το παρακάτω δίκτυο πόλεων και δρόμων μαζί με τις αποστάσεις που απεικονίζεται στην εικόνα:



Για L = 9 παίρνουμε τα παρακάτω αποτελέσματα :

```
There is a feasible route from C to G with mileage constraint 9.
```

Από τον γράφο φαίνεται ότι κάτι τέτοιο ισχύει -> Διαδρομή : C - A - D - E - G

```
There is a feasible route from C to F with mileage constraint 9.
```

Από τον γράφο φαίνεται ότι κάτι τέτοιο ισχύει -> Διαδρομή : C - A - D - E - G - F

```
There is a feasible route from A to G with mileage constraint 9.
```

Από τον γράφο φαίνεται ότι κάτι τέτοιο ισχύει -> Διαδρομή : A - D - E - G

Για L = 5 παίρνουμε τα παρακάτω αποτελέσματα :

```
There is no feasible route from A to G with mileage constraint 5.
```

Από τον γράφο φαίνεται ότι κάτι τέτοιο ισχύει καθώς δεν υπάρχει κάποια εφικτή διαδρομή.

πρόσθετα σχόλια – Εύρεση ελάχιστης/βέλτιστης διαδρομής

Σημειώνεται ότι ο αλγόριθμος που προτάθηκε παραπάνω απλά βρίσκει εάν υπάρχει εφικτή διαδρομή από την πόλη s στην πόλη t. Αυτό που δεν μπορεί να προσδιορίσει είναι η διαδρομή με το ελάχιστο δυνατό κόστος για ένα συγκεκριμένο ντεπόζιτο L χιλιομέτρων, καθώς είναι δυνατόν να υπάρχουν πολλαπλές διαδρομές μεταξύ δύο πόλεων. Μία μέθοδος που θα μπορούσε να λειτουργήσει για την αναζήτηση του ελάχιστου, λαμβάνοντας υπόψη το μήκος κάθε δρόμου είναι ο αλγόριθμος Dijkstra.

Έστω ότι ξεκινάμε από την πόλη s. Προφανώς, δεν υπάρχει κόστος για την πόλη s (0) γιατί βρισκόμαστε ήδη εκεί. Το πρώτο βήμα του αλγορίθμου είναι η αναζήτηση των γειτόνων, δηλαδή των πόλεων που συνδέονται με δρόμο με την πόλη s και αποθηκεύεται το κόστος μετακίνησης σε αυτές. Ανανεώνεται το κόστος των πόλεων αυτών ανάλογα, κρατώντας πάντα το ελάχιστο κόστος που έχει βρεθεί μέχρι στιγμής. Στη συνέχεια, επιλέγεται η πόλη με το ελάχιστο κόστος από τους γείτονες και ακολουθείται παρόμοια διαδικασία για αυτήν. Η πόλη με το ελάχιστο κόστος σε κάθε βήμα συνεχίζει να αποθηκεύεται. Ο αλγόριθμος τερματίζει όταν φτάσει στην τελική πόλη t ή μέχρι να εξερευνηθούν όλες οι πόλεις που μπορεί να φτάσει (περιορισμός ντεπόζιτου L). Κάθε φορά που ερευνάται μια πόλη, ανανεώνεται το κόστος των γειτονικών πόλεων, επιλέγοντας το μικρότερο δυνατό κόστος. ΄Όταν φτάσει στην τελική πόλη t, ελέγχεται αν το κόστος για να φτάσει εκεί είναι μικρότερο από τη μέγιστη απόσταση L που μπορεί να διανύσει το αυτοκίνητο με γεμάτο το ντεπόζιτο και έτσι προκύπτει η ελάχιστη διαδρομή. Είναι φανερό ότι αυτός ο αλγόριθμος έχει μεγαλύτερο χρόνο εκτέλεσης σε σχέση με αυτόν που αναφέρθηκε παραπάνω αλλά έχει την πρόσθετη δυνατότητα εύρεσης της ελάχιστης διαδρομής (εάν υπάρχει).

Ανάλυση Χρονικής Πολυπλοκότητας του 1^{ου} σκέλους

Αναλύουμε την χρονική πολυπλοκότητα του αλγορίθμου βήμα-βήμα:

Φιλτράρισμα των Ακμών: Περνάμε από όλες τις ακμές του γράφου για να φιλτράρουμε αυτές που έχουν μήκος μεγαλύτερο από L. Η πολυπλοκότητα αυτού του βήματος είναι O(|E|).

Δημιουργία της adjacency list: Η πολυπλοκότητα για τη δημιουργία της λίστας είναι O(|E’|), όπου |E’|≤|E|.

Ο αλγόριθμος BFS διασχίζει κάθε κορυφή και κάθε ακμή του τροποποιημένου γράφου G ’.Η πολυπλοκότητα του BFS είναι O(|V|+|E’|), όπου |V| είναι ο αριθμός των κορυφών και |E’| είναι ο αριθμός των ακμών στον φιλτραρισμένο γράφο.

Συνολική Χρονική Πολυπλοκότητα

Η συνολική χρονική πολυπλοκότητα του αλγορίθμου είναι το άθροισμα των χρονικών πολυπλοκοτήτων όλων των βημάτων: O(|E|)+O(|E'|)+O(|V|+|E'|)

=O(|E|)+O(|E|)+O(|V|+|E|)

=O(|E|+|V|+|E|)

=O(|V|+2|E|)

=O(|V|+|E|)

Έτσι, η συνολική χρονική πολυπλοκότητα του αλγορίθμου είναι O(|V|+|E|).

2ο Σκέλος του Προβλήματος

Ο αλγόριθμος που υλοποιήσαμε καθορίζει την ελάχιστη χωρητικότητα του ντεπόζιτου ενός οχήματος για την μετακίνησή του από έναν κόμβο του γράφου σε κάποιον άλλο. Παράλληλα εξασφαλίζει ότι κάθε ακμή της διαδρομής δεν ξεπερνάει ένα συγκεκριμένο μήκος. Αρχικά χρησιμοποιείται η get_unique_edge_lengths(graph). Η συνάρτηση αυτή δέχεται στην είσοδό της τον γράφο και βρίσκει τα μήκη των ακμών και τα ταξινομεί σε αύξουσα σειρά. Αυτό το βήμα εξασφαλίζει την ορθή εκτέλεση του binary search που αναφέρεται παρακάτω. Έπειτα, η συνάρτηση is_feasible_route(graph, s, t, L) έχει ως είσοδο τον γράφο, τον αρχικό κόμβο s, τον κόμβο προορισμού t και την απόσταση L, την οποία μπορεί να διανύσει το όχημα με ένα γεμάτο ντεπόζιτο. Καθορίζει αν υπάρχει διαδρομή από τον κόμβο s στον κόμβο t όπου δεν υπάρχει ακμή με μήκος μεγαλύτερο από το μήκος. Αυτό επιτυγχάνεται μέσω του BFS αλγορίθμου. Ο αλγόριθμος αυτός βοηθάει στην διάσχιση του γράφου ένα επίπεδο την φορά και βρίσκει την μικρότερη ακμή. Αν κατά την εκτέλεσή του βρεθεί κόμβος με ακμή μικρότερη του L, τότε αυτός προστίθεται στην ουρά για εξέταση του επόμενου επιπέδου. Αυτή η διαδικασία συνεχίζεται μέχρι να φτάσει στον προορισμό t ή εξερευνηθούν όλοι οι κόμβοι. Στην συνέχεια, η συνάρτηση minimum_fuel_capacity_with_route(graph, s, t), με είσοδο παραμέτρους που εξηγήθηκαν παραπάνω, υπολογίζει την ελάχιστη χωρητικότητα ντεπόζιτου που απαιτείται για την διαδρομή από τον κόμβο s στον κόμβο t. Διαβάζει την ταξινομημένη λίστα που δημιουργήθηκε από την συνάρτηση get_unique_edge_lengths() με την μέθοδο του binary search, η οποία μειώνει σημαντικά τον αριθμό των ακμών που πρέπει να εξεταστούν. Για κάθε υποψήφια ακμή ελέγχεται εάν υπάρχει διαδρομή από τον κόμβο s στον κόμβο t. Αν αυτή υπάρχει, τότε ανανεώνεται το κάτω όριο του πεδίου αναζήτησης. Αυτή η διαδικασία

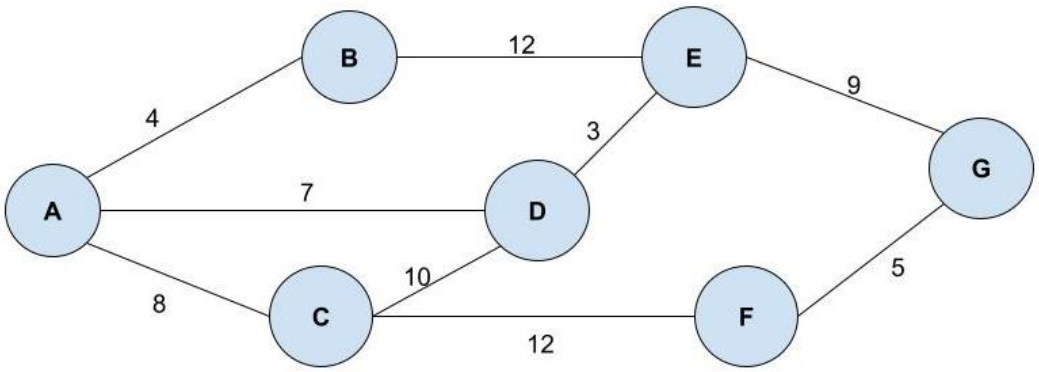
συνεχίζεται μέχρι να καθοριστεί η ελάχιστη χωρητικότητα του ντεπόζιτου. Όταν αυτή υπολογιστεί μαζί με την αντίστοιχη διαδρομή που ακολούθησε το όχημα, τότε ο κώδικας επιστρέφει αυτά τα δεδομένα.

```
from collections import deque

def get_unique_edge_lengths(graph):
    edge_lengths = set()
    for u in graph:
        for v, length in graph[u]:
            edge_lengths.add(length)
    return sorted(edge_lengths)

def is_feasible_route(graph, s, t, L):
    # Create subgraph G_L with edges <= L
    G_L = {v: [] for v in graph}
    for u in graph:
        for v, length in graph[u]:
            if length <= L:
                G_L[u].append((v, length))
                G_L[v].append((u, length))

    # BFS to check reachability and track the path
    queue = deque([(s, [s])])
    visited = set()
    while queue:
        current, path = queue.popleft()
        if current == t:
            return True, path
        if current not in visited:
            visited.add(current)
            for neighbor, _ in G_L[current]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))
    return False, []

def minimum_fuel_capacity_with_route(graph, s, t):
    # Step 1: Extract and sort unique edge lengths
    edge_lengths = get_unique_edge_lengths(graph)

    # Step 2: Binary search over the sorted edge lengths
    low, high = 0, len(edge_lengths) - 1
    final_path = []
    while low < high:
        mid = (low + high) // 2
        feasible, path = is_feasible_route(graph, s, t,
edge_lengths[mid])
        if feasible:
            high = mid
            final_path = path
        else:
            low = mid + 1
```

```
    # Final check to confirm the route for the last value of low
    feasible, path = is_feasible_route(graph, s, t,
edge_lengths[low])
    if feasible:
        final_path = path

    return edge_lengths[low], final_path

# Example usage
if __name__ == "__main__":
    # Define the graph as an adjacency list where each edge has a
length
    graph = {
        'A': [('B', 14), ('C', 12)],
        'B': [('A', 14), ('C', 11), ('D', 15)],
        'C': [('A', 12), ('B', 11), ('D', 18), ('E', 10)],
        'D': [('B', 15), ('C', 18), ('E', 12)],
        'E': [('C', 10), ('D', 12)]
    }
    s = 'D'
    t = 'A'

    # Determine the minimum fuel capacity required to travel from
s to t
    min_capacity, route = minimum_fuel_capacity_with_route(graph,
s, t)
    print(f"The minimum fuel tank capacity needed to travel from
{s} to {t} is {min_capacity} miles.")
    print(f"The route taken is: {' -> '.join(route)}")
```

Παραδείγματα

Έστω το ίδιο παράδειγμα με το πρώτο σκέλος του προβλήματος:



Για s = A και t = G και αντίστοιχα για την αντίστροφη διαδρομή (s = G και t = A) παίρνουμε τα παρακάτω αποτελέσματα :

```
The minimum fuel tank capacity needed to travel from A to G is 9 miles.
The route taken is: A -> D -> E -> G
```

```
The minimum fuel tank capacity needed to travel from G to A is 9 miles.
The route taken is: G -> E -> D -> A
```

Για s = B και t = F και αντίστοιχα για την αντίστροφη διαδρομή (s = F και t = B) παίρνουμε τα παρακάτω αποτελέσματα :

```
The minimum fuel tank capacity needed to travel from B to F is 9 miles.
The route taken is: B -> A -> D -> E -> G -> F
```

```
The minimum fuel tank capacity needed to travel from F to B is 9 miles.
The route taken is: F -> G -> E -> D -> A -> B
```

Ανάλυση Χρονικής Πολυπλοκότητας του 2ου σκέλους

Αναλύουμε την χρονική πολυπλοκότητα του αλγορίθμου βήμα-βήμα:

get_unique_edge_lengths: Περνάμε από όλες τις ακμές του γράφου για να φιλτράρουμε αυτές που έχουν μήκος μεγαλύτερο από L. Η πολυπλοκότητα αυτού του βήματος είναι O(|E|).

is_feasible_route: Η δημιουργία του γράφου G_L, ο οποίος περιέχει τους κόμβους V_L, έχει και αυτή πολυπλοκότητα O(|E|). Όμως, στην εκτέλεση του BFS, η χειρότερη περίπτωση είναι να εξεταστούν όλοι οι κόμβοι. Τότε θα έχουμε πολυπλοκότητα O(|V|+ |E|).

minimum_fuel_capacity_with_route: Στην συνάρτηση αυτή έχουμε binary search, όπου κάθε φορά καλείται η συνάρτηση is_feasible_route. Η binary search έχει πολυπλοκότητα log(E). Αυτό μας δίνει πολυπλοκότητα O((|V| +|E|) * log |E|).

Η συνολική χρονική πολυπλοκότητα του αλγορίθμου είναι το άθροισμα των χρονικών πολυπλοκοτήτων όλων των βημάτων:

= O(|E|) + O(|E|) + O((|V| + |E|) * log |E|)

= O(2|E|) + O((|V| + |E|) * log |E|)

= O((|V| + |E|) * log |E| + 2|E|)

= O((|V| + |E|) * log |E|)

(E =< V οπότε είναι εφικτή η λύση της εκφώνησης)

Πρόβλημα 2

Για την υλοποίηση χρησιμοποιήσαμε μία λίστα available, η οποία περιέχει τους πελάτες που είναι σε αναμονή στην υπηρεσία ταξινομημένους κατά αύξοντα χρόνο αναμονής. Κάθε φορά εξυπηρετούμε τον πρώτο πελάτη από την ταξινομημένη λίστα available, ο οποίος έχει τον ελάχιστο χρόνο εξυπηρέτησης, και ενημερώνουμε τον χρόνο t κατά τον χρόνο εξυπηρέτησής του. Έπειτα, υπολογίζουμε τους χρόνους ολοκλήρωσης και αναμονής για τον εκάστοτε πελάτη, αφαιρούμε τον εξυπηρετούμενο πελάτη από τη λίστα process_list και καταγράφουμε τα αποτελέσματα. Μετά την εξυπηρέτηση όλων των πελατών, επιστρέφουμε τη σειρά εξυπηρέτησης, καθώς και τους χρόνους ολοκλήρωσης και αναμονής κάθε πελάτη. Τέλος, υπολογίζουμε και επιστρέφουμε τον μέσο χρόνο αναμονής για όλους τους πελάτες που εξυπηρετήθηκαν.

```
# Function to implement Shortest Job First (SJF) scheduling algorithm
#Process: [service_time, arrival_time, processid]

def shortestJobFirst(process_list):
    t = 0
    order of service = []
    completed service = {}
    total waiting time = 0  # Variable to store the total waiting time

    # Until all processes are serviced
    while process_list != []:
        available = []
        # Gather processes who have arrived
        for p in process_list:
            if p[1] <= t:
                available.append(p)

        # Sorting and serving based on lowest service time
        available.sort()
        process = available[0]
        service time = process[0]
        processid = process[2]
        arrival_time = process[1]
        t += service_time
        order_of_service.append(processid)

        # Calculating completion and wait times for each process
        completion_time = t
        waiting_time = completion_time - arrival_time - service_time
        total_waiting_time += waiting_time
        process list.remove(process)
        completed service[processid] = [completion time, waiting time]

    #Display Results
    print("Order of Service:", order_of_service)
    print("Completed Service:")
    for processid, info in completed_service.items():
        completion_time, waiting_time = info
        print(f"Process ID: {processid}, Completion Time:
{completion_time}, Waiting Time: {waiting_time}")

    # Calculate and print the average waiting time
    if len(completed_service) > 0:
        avg waiting time = total waiting time / len(completed service)
        print(f"Average Waiting Time: {avg waiting time}")

process list = [[4, 5, "p1"], [2, 8, "p2"], [6, 0, "p3"], [5, 2, "p4"]]
shortestJobFirst(process_list)
```

Παραδείγματα υλοποίησης του αλγορίθμου

Process: [Χρόνος εξυπηρέτησης, Χρόνος άφιξης, Id Πελάτη]

Παράδειγμα 1ο

Id Πελάτη	Χρόνος εξυπηρέτησης	Χρόνος άφιξης
1	4	5
2	2	8
3	6	0
4	5	2

Στην έξοδο θα επιστραφεί το εξής αποτέλεσμα:

```
Order of Service: ['p3', 'p1', 'p2', 'p4']
Completed Service:
Process ID: p3, Completion Time: 6, Waiting Time: 0
Process ID: p1, Completion Time: 10, Waiting Time: 1
Process ID: p2, Completion Time: 12, Waiting Time: 2
Process ID: p4, Completion Time: 17, Waiting Time: 10
Average Waiting Time: 3.25
```

Παράδειγμα 2ο

Id Πελάτη	Χρόνος εξυπηρέτησης	Χρόνος άφιξης
1	5	13
2	11	0
3	6	7
4	7	12
5	1	9
6	10	16

Στην έξοδο θα επιστραφεί το εξής αποτέλεσμα:

```
Order of Service: ['p2', 'p5', 'p3', 'p1', 'p4', 'p6']
Completed Service:
Process ID: p2, Completion Time: 11, Waiting Time: 0
Process ID: p5, Completion Time: 12, Waiting Time: 2
Process ID: p3, Completion Time: 18, Waiting Time: 5
Process ID: p1, Completion Time: 23, Waiting Time: 5
Process ID: p4, Completion Time: 30, Waiting Time: 11
Process ID: p6, Completion Time: 40, Waiting Time: 14
Average Waiting Time: 6.166666666666667
```

Πρόβλημα 3

Για την υλοποίηση της συνάρτησης χρησιμοποιήθηκε η βιβλιοθήκη NumPy.

```
def optimalStringSeparator(string, breakpoints):
    breakpoints = sorted(breakpoints)
    lenString = len(string)
    numBreakPoints = len(breakpoints)

    # Add to the breakpoints the first and the last indices of the string
    breakpoints = [0] + breakpoints + [lenString]
    numBreakPoints = len(breakpoints)
    breakpoints = [0] + breakpoints

    # Initialize the dynamic programming table
    costs = np.array([[float('inf')] * (numBreakPoints + 1) for _ in
range(numBreakPoints + 1)])

    # Base case: cost of splitting a substring of length 0 or 1  is 0
    for i in range(1, numBreakPoints + 1):
       costs[i, i:i+2] = 0

    # Iterate over all possible substring lengths
    for s in range(2, numBreakPoints):
       for i in range(1, numBreakPoints - s + 1):
          j = i + s
          for k in range(i, j + 1):
             costs[i, j] = min(costs[i, j], breakpoints[j] - breakpoints[i]
                               + costs[i, k] + costs[k, j])

    # Minimum cost of splitting the entire string into numBreakPoints + 1
sections
    return costs[1, numBreakPoints]
```

Ανάλυση Χρονικής Πολυπλοκότητας

Ο αλγόριθμος πραγματοποιεί τις ακόλουθες ενέργειες:

1.Αρχικοποίηση του πίνακα κόστους.

2.Εκτέλεση του πρώτου βρόχου.

3.Εκτέλεση του 1ου εμφωλευμένου βρόχου.

4.Εκτέλεση του 2ου εμφωλευμένου βρόχου.

Η πολυπλοκότητα κάθε βήματος είναι η εξής:

Βήμα 1: O(1)

Βήμα 2: O(m)

Βήμα 3: $O(m^2)$

Βήμα 4: $O(m^3)$

Συνεπώς, η συνολική χρονική πολυπλοκότητα του αλγορίθμου είναι $O(m^3)$.

Παραδείγματα υλοποίησης του αλγορίθμου

- Για string = "This_is_a_test" με breakpoints = [8 , 11 , 4] θα έχουμε στην κονσόλα το εξής αποτέλεσμα:

  Length of string: 14

  Minimum splitting cost: 28.0

- Για string = "Algorithm_Analysis_and_Design" με breakpoints = [2, 18 , 9, 14, 25] θα έχουμε στην κονσόλα το εξής αποτέλεσμα:

  Length of string: 29

  Minimum splitting cost: 76.0