ΕΡΓΑΣΙΑ 2024-ΑΝΑΛΥΣΗ ΚΑΙ ΣΧΕΔΙΑΣΜΟΣ ΑΛΓΟΡΙΘΜΩΝ

Ομάδα 12

Μέλη: Γαλαζούλας Αλέξανδρος, ΑΕΜ: 10629

Πούλιος Αστέριος, ΑΕΜ: 10887

Πρόβλημα 1:

1) Ο αλγόριθμος που προτείνουμε για να βρει αν υπάρχει εφικτό δρομολόγιο από την πόλη s στην πόλη t, είναι ο DFS με αρχικό κόμβο τον s, στον οποίο θα προσθέσουμε στο τέλος μία σύγκριση των πινάκων f από τους παραπάνω κόμβους. Αν ft<fs, σημαίνει ότι ο κόμβος t είναι προσβάσιμος από τον κόμβο s και άρα η διεργασία θα επιστρέφει True, αλλιώς θα επιστρέφει False.
Η ιδέα χρήσης αυτού του αλγορίθμου προήλθε από τον ορισμό του, αφού απαντάει στην ερώτηση: Ποιοι κόμβοι είναι προσπελάσιμοι από έναν αρχικό κόμβο;
Ως γνωστόν, ο χρόνος εκτέλεσης του DFS αλγορίθμου είναι O(|V|+|E|) και η σύγκριση που θα προστεθεί στο τέλος του θεωρείται πράξη σταθερού χρόνου, η οποία σε big O-notation αγνοείται εντελώς. Συνεπώς, ο συνολικός χρόνος του αλγορίθμου μας είναι: O(|V|+|E|).

2) Σε αυτή την περίπτωση, ο αλγόριθμος που θα ισχυριστούμε ότι βρίσκει την ελάχιστη χωρητικότητα καυσίμων(σε χιλιόμετρα απόστασης) ώστε να ταξιδέψουμε από την πόλη s στην πόλη t, είναι ο Dijkstra με αρχικό κόμβο τον s(αρχική πόλη αναφοράς) και υλοποιημένος με Binary Heap. Αυτός θα βρίσκει από τις γειτονικές πόλεις της εκάστοτε πόλης αναφοράς ποια απέχει την ελάχιστη απόσταση από αυτήν και θα τη θεωρεί ως νέα πόλη αναφοράς, έως ότου, αναδρομικά, να φτάσει στην t[*1]. Έτσι, παίρνοντας τη διαδρομή των ελαχίστων αποστάσεων θα έχει προκύψει το ζητούμενο.

Ωστόσο, πρέπει να προστεθεί ένας έλεγχος σε κάθε επανάληψη του αλγορίθμου, για το αν είναι επαρκής η βενζίνη(σε χιλιόμετρα), ώστε να διανύσουμε την ελάχιστη απόσταση από την πόλη αναφοράς στη γειτονική της πόλη. Αν δεν είναι θα εμφανίζει μήνυμα τερματισμού*2. Η συγκεκριμένη εντολή είναι σταθερού χρόνου και δε θα επηρεάσει τη πολυπλοκότητα του Dijkstra. Όπως και στο προηγούμενο ερώτημα, η ιδέα προήλθε από τον ορισμό του εν λόγω αλγορίθμου.
Ο χρόνος εκτέλεσης θα είναι O((|V|+|E|)log|V|), το οποίο απαντάει και στο τελευταίο ερώτημα του προβλήματος 1.

Πρόβλημα 2

Σε αυτό το πρόβλημα, ψάχνω έναν αποδοτικό αλγόριθμο για τη βέλτιστη σειρά εξυπηρέτησης σε μία ουρά, η οποία θα ελαχιστοποιεί το συνολικό χρόνο αναμονής (δηλαδή το άθροισμα των χρόνων αναμονής του κάθε πολίτη), ενώ γνωρίζω χρόνο που χρειάζεται για να εξυπηρετηθεί κάθε πολίτης ξεχωριστά. Έστω ότι οι πολίτες είναι n και έχουν τη μορφή $S=\{a_1,a_2,\ldots,a_n\}$, όπου ο χρόνος εξυπηρέτησης του i-οστού πολίτη st_i και wt_i ο χρόνος αναμονής του i-οστού πολίτη. Επίσης, έστω $ST=\{st_1,st_2,\ldots,st_n\}$.

Επιλέγω προσέγγιση άπληστων αλγορίθμων.

Ξεκινάω, χρησιμοποιώντας μία sort στον ST, ώστε να έχω τους πολίτες σε αύξοντα αριθμό st_i . Επιλέγω τη βέλτιστη αρχική λύση, που είναι ο a_1 να μπει πρώτος αφού το wt θα ελαχιστοποιηθεί για τον επόμενο. Άρα, πρέπει να λύσω το υποπρόβλημα της ελαχιστοποίησης του χρόνου αναμονής του $a_3,a_4,\ldots$

Πρέπει να αποδειχθεί, ότι το a_1 είναι πάντα μέρος της βέλτιστης λύσης. Έστω ότι A_k είναι μία βέλτιστη λύση του προβλήματος με τον a_j να έχει το μικρότερο st σε αυτή, γνωρίζοντας ότι το a_m έχει το μικρότερο st για όλο το πρόβλημα. Αν $a_j=a_m$, αποδείχθηκε. Ειδάλλως, θεωρώ το A_k' από το οποίο αφαιρώ το a_j και προσθέτω το a_m. Τότε, αφού $wt_m \leq wt_j$, θα περιέχεται το a_m σε οποιαδήποτε βέλτιστη λύση.

Ο αλγόριθμος σε ψευδογλώσσα επισυνάπτεται στο παράρτημα.

Επιστρέφοντας τον πίνακα WT και το άθροισμα των χρόνων αναμονής, διακρίνω είναι το ελάχιστο.

Ο παραπάνω αλγόριθμος έχει πολυπλοκότητα O(nlogn), λόγω της διαδικασίας ταξινόμησης. Η διαδικασία υπολογισμού του συνολικού χρόνου αναμονής είναι O(n). Επομένως, η συνολική πολυπλοκότητα του αλγορίθμου είναι O(nlogn).

Πρόβλημα 3

1) Έχουμε μια συμβολοσειρά με n στοιχεία και θέλουμε να την διασπάσουμε σε m+1 τμήματα με m γνωστές τομές. Κάθε διάσπαση απαιτεί n μονάδες χρόνου λόγω της αντιγραφής ολόκληρης της αρχικής συμβολοσειράς. Το ζητούμενο είναι να βρούμε το ελάχιστο υπολογιστικό κόστος για να πραγματοποιηθούν οι m τομές.

   Έστω C(i,j) το ελάχιστο κόστος για να διασπάσουμε το υποδιάστημα της συμβολοσειράς από το σημείο i έως το σημείο j, $T= [0, t_1, t_2, \dots , t_m, n]$, ο πίνακας των τομών (στον οποίο έχουμε προσθέσει το 0 και το n ώστε να αποφύγουμε τις περιπτώσεις των ακραίων προβλημάτων) και D ο πίνακας στον οποίο θα αποθηκεύουμε τα ελάχιστα κόστη. Το D[i][j] θα περιέχει το ελάχιστο κόστος για να διασπάσουμε την υποσυμβολοσειρά από το T[i] έως το T[j]. Επίσης, όταν έχω δύο διαδοχικές τομές i,j, τότε δεν υπάρχει κόστος διάσπασης γιατί δεν υπάρχει χώρος για τομή. Για κάθε υποδιάστημα [i,j] υπολογίζω το κόστος διάσπασης για κάθε πιθανή ενδιάμεση τομή k μεταξύ των i και j, το οποίο θα είναι το άθροισμα του κόστους για να διασπάσουμε τα υποδιαστήματα [i,k] και [k,j] συν το κόστος της διάσπασης της υποσυμβολοσειράς από T[i] έως T[j], δηλαδή T[j]−T[i].
   Ουσιαστικά, ξεκινάω από τις υποσυμβολοσειρές της μεγάλης συμβολοσειράς, ώστε να διασπάσω το πρόβλημά μου και να δουλέψω με δυναμικό προγραμματισμό.
   Επισυνάπτεται ο κώδικας σε ψευδογλώσσα στο παράρτημα.

2) Η πολυπλοκότητα του αλγορίθμου είναι $O(m^3)$, όπου m είναι ο αριθμός των τομών. Αυτό προκύπτει από το γεγονός ότι έχουμε δύο βρόχους για τα διαστήματα (i,j) και έναν εσωτερικό βρόχο για την επιλογή της ενδιάμεσης τομής k.

Παράρτημα

1)β)*1Προφανώς αν έχω μία γειτονική πόλη της πόλης αναφοράς, την ελάχιστη απόσταση θα την έχει η ίδια. Οι αποστάσεις θα αναγράφονται ως βάρη πάνω στο γράφο.

*2 Για να τρέξει ο αλγόριθμος σωστά, προϋποθέτει να είναι προσπελάσιμος ο t από τον s, άρα θα μπορούσε αρχικά να τρέξει ο DFS του πρώτου ερωτήματος.

2) Ο κώδικας του προβλήματος 2 σε ψευδογλώσσα:

//ST: Πίνακας που περιέχει όλους τους χρόνους st.

Greedy_queue(ST)

n=st.length

sum_of_waiting_times=0

sort(ST)//βάζω τους πολίτες σε αύξουσα σειρά st

WT={0} //αρχικοποίηση πίνακα που θα αποθηκεύσω όλα τα wt των πολιτών

for m=2 to n

$wt_m = wt_{m-1} + st_{m-1}$

sum_of_waiting_times+=wt_m

store wt_m in WT

return WT and sum_of_waiting_times

3) Ο κώδικας του προβλήματος 3 σε ψευδογλώσσα:

min_cost_to_split_string(n, T)

//n: μήκος της συμβολοσειράς

//T: πίνακας με τις θέσεις των τομών (με μέγεθος m)

// Προσθήκη αρχικών και τελικών τομών

T = [0] + T+ [n]

```
 m = length T


 // Δημιουργία πίνακα D για αποθήκευση των ελάχιστων κόστων
 Δημιουργία πίνακα D[m][m] και αρχικοποίηση όλων των τιμών σε 0


 // Υπολογισμός του κόστους για κάθε υποδιάστημα
  for length= 2 : m
    for i= 0 : m - length
     j = i + length
     D[i][j] = ∞
       for k= i + 1 : j - 1
       cost = T[j] - T[i] + D[i][k] + D[k][j]
       D[i][j] = min(D[i][j], cost)
       end for
     end for
   end for
// Επιστροφή του ελάχιστου κόστους για τη διάσπαση όλης της συμβολοσειράς
  return D[0][m-1]
end
```

Όπως διακρίνουμε ισχύουν οι λειτουργίες του δυναμικού προγραμματισμού:

- **Βασίζεται στη βέλτιστη υποδομή**: Χρησιμοποιεί τις ήδη βέλτιστες λύσεις των μικρότερων υποπροβλημάτων για να λύσει μεγαλύτερα προβλήματα.
- **Ελαχιστοποίηση κόστους**: Για κάθε διάστημα, εξετάζει όλες τις δυνατές τομές και επιλέγει αυτή που δίνει το ελάχιστο συνολικό κόστος.

- **Αποθήκευση αποτελεσμάτων και επαναχρησιμοποίησή τους**: Με την αποθήκευση των αποτελεσμάτων στον πίνακα D, ο αλγόριθμος αποφεύγει επαναλαμβανόμενους υπολογισμούς.