ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧ/ΚΩΝ & ΜΗΧ/ΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Μάθημα: "Αλγόριθμοι και πολυπλοκότητα"

Λουκάς Άγγελος 03119877

Σειρά Ασκήσεων 2

Άσκηση 1

Θα εφαρμόσουμε δυναμικό προγραμματισμό για την λύση του προβλήματος. Το πρόγραμμά μας θα υπολογίζει τις ελάχιστες μέρες D(i,j) που χρειάζεται για να βαφτεί ο πεζόδρομος από την πλάκα i μέχρι την πλάκα j με την εξής συνάρτηση:

$$D(i,j)=\begin{cases} min\{D(i+1,j),D(i,j-1),D(i+1,j-1),min_{0\le k\le j-1-1}[D(i,i+k)+D(i+k+1,j)]\}, \alpha\nu\, c_i=c_j \\ min_{0\le k\le j-1-1}[D(i,i+k)+D(i+k+1,j)], \alpha\nu\, c_i\neq c_j \end{cases}$$

Με αρχικές συνθήκες:

$D(i,j)=1, i=j$ καθώς κάθε πλάκα από μόνη της χρειάζεται μία μέρα για να βαφτεί.

Ορθότητα:

- Αν $c_i\neq c_j$ τότε οι ελάχιστες μέρες που θα χρειαστούμε για το βάψιμο του διαστήματος [i,j] είναι το άθροισμα $D(i,k)+D(k+1,j)$ για κάποιο k με $i\le k\le j-i-1$. Άρα υπολογίζουμε το άθροισμα αυτό για όλες τις πιθανές τιμές του k και διαλέγουμε το ελάχιστο αυτών.
- Αν $c_i=c_j$ τότε θα χρειαστεί και πάλι να υπολογίσουμε το ελάχιστο του αθροίσματος $D(i,k)+D(k+1,j)$ για κάθε k με $i\le k\le j-i-1$. Τώρα όμως υπάρχουν τα σημεία m,n με $i\le m\le n$ και $m\le n\le j$ για τα οποία $c_w=c_i=c_j$ για κάθε $w\in[i,m]\cup[n,j]$. Οι πλάκες αυτές δεν χρειάζεται να ξαναχρωματιστούν άρα μπορούμε να τις παραλείψουμε από την επόμενη αναζήτηση μη επικαλυπτόμενων διαστημάτων. Αυτό γίνεται με τα $D(i+1,j), D(i,j-1), D(i+1,j-1)$. Θα αφαιρεθούν δηλαδή όλα τα πλακάκια ίδιου χρώματος στα άκρα του διαστήματος [i,j] πριν αρχίσει να βάφει με άλλο χρώμα τα υποδιαστήματα του διαστήματος αυτού.

Πολυπλοκότητα. Το δυναμικό πρόγραμμα έχει $\frac{n^2}{2}=O(n^2)$ states. Το κάθε state υπολογίζει το ελάχιστο από $O(n)$ τιμές. Συνεπώς συνολικά η χρονική πολυπλοκότητα είναι $O(n^3)$

Άσκηση 2

2.1)

Θέλουμε να αποδείξουμε πως μπορούμε να ταιριάξουμε το μοτίβο μας αν βρούμε τους πυρήνες των προθεμάτων του κειμένου **pt**. Για να ταιριάξουμε το p με το t πρέπει να βρούμε έναν πυρήνα ενός προθέματος του pt που ισούται με το μοτίβο p δηλαδή πρέπει να βρούμε ένα πρόθεμα του pt που ισούται με pAp όπου A ένα τυχαίο κείμενο. Υποθέτουμε πως t=ApB όπου A και B είναι πρόθεμα και επίθεμα του t αντίστοιχα. Ξεκινάμε την αναζήτηση των πυρήνων των προθεμάτων του pt = pApB.

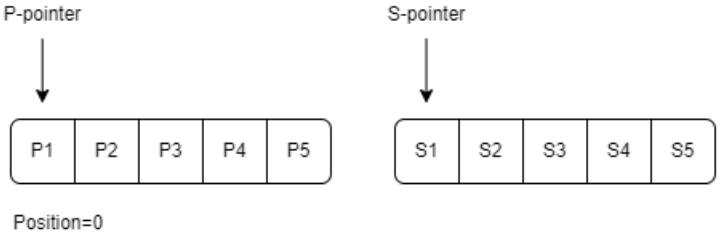
Κατά την αναζήτηση όλων των προθεμάτων φτάνουμε σε κάποια φάση στο κείμενο pAp το οποίο υποδεικνύει πως μπορέσαμε και ταιριάξαμε το p με το κείμενο t.

2.2)

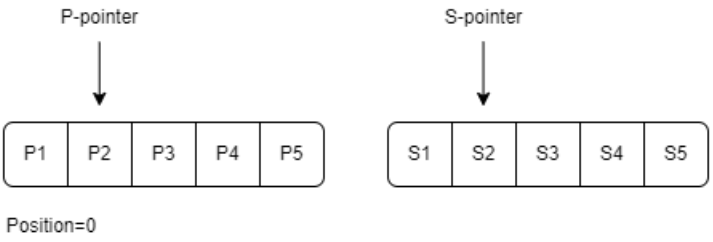
Έστω πως ο κ-πυρήνας της συμβολοσειράς υ = $C_κAC_κ$ όπου $\mathbf{C_κ}$ ο κ-πυρήνας της υ και **A** το κομμάτι της συμβολοσειράς υ που δεν ανήκει στον κ-πυρήνα.
Για να βρούμε το $C_κ$ θα πάρουμε τα $\mathbf{P_k}$ που είναι το πρόθεμα της υ μήκους κ και $\mathbf{S_k}$ που είναι το επίθεμα της υ μήκους κ. Θα ορίσουμε επίσης μία μεταβλητή: **Position=0** η οποία θα χρησιμοποιηθεί για να γνωρίζουμε στο τέλος το συνολικό μήκος **m** του πυρήνα μας που θα ισούται με m=k-Position.
Για τον έλεγχο του προθέματος και το επιθέματος ορίζουμε δύο pointers, τον P-pointer και τον S-pointer που δείχνουν αρχικά στην αρχή των δύο συβολοσειρών $\mathbf{P_k}$ και $\mathbf{S_k}$.

- Αν ο χαρακτήρας του $\mathbf{P_k}$ στον οποίο δείχνει ο P-pointer είναι ίδιος με τον χαρακτήρα του $\mathbf{S_k}$ στον οποίο δείχνει ο S-pointer, τότε:
  1) Αν ο S-pointer βρίσκεται στον τελευταίο χαρακτήρα του S_k τότε βρήκαμε τον κ-πυρήνα μας C_k που ισούται με τους τελευταίους k-Position χαρακτήρες του S_k.
  2) Αλλιώς μετακινούμε την θέση και των 2 pointers κατά 1 δεξιά.
- Αν ο χαρακτήρας του $\mathbf{P_k}$ στον οποίο δείχνει ο P-pointer **ΔΕΝ** είναι ίδιος με τον χαρακτήρα του $\mathbf{S_k}$ στον οποίο δείχνει ο S-pointer, τότε:
  1) Μετακινούμε τον P-pointer στο στοιχείο P_1
  2) Αυξάνουμε την τιμή της μεταβλητής Position κατά 1
  3) Μετακινούμε τον S-pointer στο στοιχείο $P_{1+Position}$ (Αυτά τα βήματα γίνονται επειδή βρήκαμε λάθος στον έλεγχο του προθέματος με το επίθεμα. Δεν υπάρχει δηλαδή πυρήνας με ακριβές μήκος κ και μεταφέροντας την θέση του S-pointer κατά 1 δεξιά αρχίζουμε να ψάχνουμε για πυρήνα μήκους κ-1)
- Αν η τιμή του Position μας υπερβεί την τιμή k-1 τότε ο πυρήνας μας είναι κενή συμβολοσειρά.

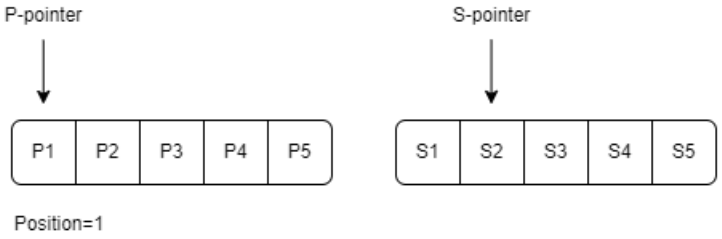
Παράδειγμα :
Ψάχνουμε τον πυρήνα για k=5. Παίρνουμε τα P_5, S_5 και αρχικοποιούμε τους pointers και την μεταβλητή position.



Ελέγχουμε τις τιμές των P-pointer, S-pointer και παρατηρούμε πως $P_5[1]=S_5[1]$ άρα μετακινούμε τους pointers.



Ελέγχουμε τις τιμές των P-pointer, S-pointer και παρατηρούμε πως $P_5[2] \neq S_5[2]$ άρα αυξάνουμε την τιμή του Position και μεταφέρουμε τον P-pointer στο $P_5[1]$ και τον S-pointer στο $S_5[1+Position]=S_5[2]$.



Συνεχίζεται αυτή η διαδικασία μέχρι να φτάσει ο S-pointer στο τελευταίο χαρακτήρα του S_5 στο οποίο $P_5[3]=S_5[5]$ και άρα έχουμε πως ο πυρήνας μας είναι $C_5= S_5[3]\ S_5[4]\ S_5[5]$ και μήκους m=k-position=3

Άσκηση 3

3.1)

Ο τρόπος λύσης του προβλήματος μας θα είναι για κάθε ακμή (v_1,v_2) να βρούμε το άθροισμα των αποστάσεων $d(s,v_1) + d(v_2,t)$ (όπου d(a,b) η ελάχιστη απόσταση από τον κόμβο a στον b). Θα υπολογίσουμε δηλαδή για κάθε ακμή v ποιο είναι το ελάχιστο μονοπάτι από τον κόμβο s που περνάει από την ακμή v, χωρίς να προσμετρήσουμε την απόσταση της v και να καταλήγει στον κόμβο t.

Ο αλγόριθμος μας θα είναι ο εξής:

(i) Αρχικοποιούμε δυο πίνακες $d_s[1..n]$ και $d_t[1..n]$. Τον πρώτο θα τον γεμίσουμε με τις ελάχιστες αποστάσεις όλων των κόμβων από την κορυφή s και τον δεύτερο με τις ελάχιστες αποστάσεις όλων των κορυφών από την κορυφή t.

(ii) Το παραπάνω το επιτυγχάνουμε τρέχοντας δυο φορές τον αλγόριθμο Dijkstra. Την πρώτη υπολογίζουμε τις ελάχιστες αποστάσεις όλων των κορυφών από την κορυφή s. Για να υπολογίσουμε τις ελάχιστες αποστάσεις από την κορυφή t πρώτα αντιστρέφουμε το γράφημα (

$O(n+m)$ για υλοποίηση του γραφήματος με λίστα γειτνίασης) και μετά εφαρμόζουμε Dijkstra στο γράφημα αυτό.

(iii) Για κάθε ακμή στο γράφημα που ενώνει τις κορυφές (v_1,v_2) προσθέτουμε τα $d_s[v_1]$και $d_t[v_2]$ και κρατάμε το ελάχιστο άθροισμα. Το ελάχιστο άθροισμα αυτό είναι η ελάχιστη απόσταση που μπορεί να επιτευχθεί από το *s* στο *t* και είναι η απόσταση του μονοπατιού έχοντας αφαιρέσει την ακμή (v_1,v_2).

Η χρονική πολυπλοκότητα του αλγορίθμου είναι δύο φορές ο χρόνος που χρειάζεται ο αλγόριθμος Dijkstra, ο οποίος εξαρτάται από τον τρόπο που είναι υλοποιημένο το γράφημα. Συνεπώς, συνολικά θέλουμε $O(Dijkstra)$.

3.2)

Θεωρούμε G το αρχικό μας γράφημα με ακμές (v_i, v_j, w_{ij}) όπου v_i μία κορυφή του γράφου και w_{ij} το βάρος της κάθε ακμής.

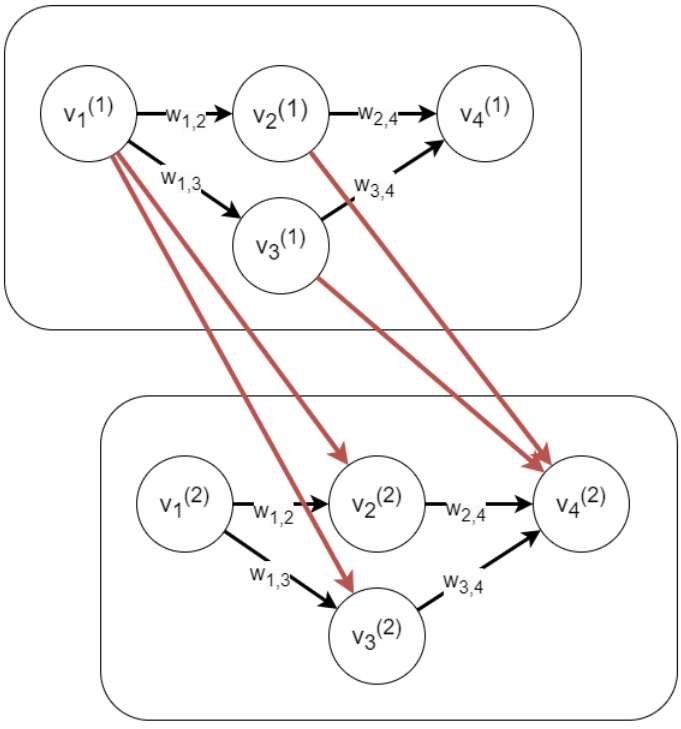
Θα ακολουθήσουμε τα παρακάτω βήματα για την λύση του προβλήματος:

1. Κατασκευάζουμε ένα πολυεπίπεδο γράφημα G’ με k+1 επίπεδα με το επίπεδο a να ορίζεται ως $G^{(a)}$. Το κάθε επίπεδο είναι αντίγραφο του G.
2. Για κάθε ακμή (v_i^a, v_j^a, w_{ij}) όλων των επιπέδων δημιουργούμε μία ακμή $(v_i^a, v_j^{a+1}, 0)$. Η ακμή αυτή ενώνει τα επίπεδα $G^{(a)}$,$G^{(a+1)}$ με μηδενικό κόστος. Όταν είμαστε δηλαδή στο επίπεδο a έχουμε δύο επιλογές, να συνεχίσουμε στο επίπεδο που είμαστε ή να μετακινηθούμε με μηδενικό κόστος στο επίπεδο a+1. Και αφού υπάρχουν k+1 επίπεδα, μπορούμε να χρησιμοποιήσουμε το πολύ k ακμές με μηδενικό βάρος.
3. Εφαρμόζοντας τον αλγόριθμο Dijkstra στον γράφο G’ με αρχικό κόμβο τον s και τελικό οποιονδήποτε $t^{(a)}$ κόμβο του επιπέδου a, θα μπορέσουμε να βρούμε το ελάχιστο μονοπάτι με το πολύ k μηδενικές ακμές.

Αν n οι ακμές του G και m οι κορυφές του τότε το γράφημα G’ έχει:

- (k+1)m+km = O(km) ακμές
- (k+1)n = O(kn) κορυφές

Στο διπλανό σχήμα φαίνεται η υλοποίηση της κατασκευής του G’ με k=1 και συνολικά 2 επίπεδα με τα κόκκινα βελάκια να είναι οι ακμές μηδενικού βάρους που ενώνουν τα διαδοχικά επίπεδα μεταξύ τους.



Σημειώνουμε ότι k=O(m) καθώς δεν έχει νόημα να δοκιμάσουμε να αφαιρέσουμε παραπάνω απο τον αριθμό των ακμών του αρχικού γραφήματος G.

Η πολυπλοκότητα κατασκευής του G' είναι O(k(m+n)) και για την αναζήτηση στον γράφο θα χρησιμοποιήσουμε το Fibonacci Heap δηλαδή
$O(|E|+|V|\log|V|)=O(km+km\log km)=O(m^2+mn\log mn)$.

Άσκηση 4

4.1)

Υποθέτουμε πως δεν υπάρχει b(e) στο s-t μονοπάτι μας για το οποίο ισχύει b(e) > B.
Ο τρόπος υλοποίησης του προβλήματος είναι: Όταν θα βρισκόμαστε σε έναν κόμβο θα γεμίζουμε το ντεπόζιτο μας με ακριβώς αρκετή βενζίνη ώστε να φτάσουμε στον επόμενο κόμβο που έχει φθηνότερη βενζίνη.
Ο αλγόριθμος μας θα είναι ο εξής:

a) Εισάγουμε το μονοπάτι $(s,u_2,\dots,u_{n-1},t)$ στην λίστα A με $A[0]=s$ και $A[n-1]=t$.
b) Δημιουργούμε την λίστα H με n στοιχεία (H[0,…,n-1]). Η λίστα αυτή θα κρατάει για κάθε θέση i την θέση του κοντινότερου κόμβου j για τον οποίον ισχύει i<j και c[i]>c[j].

Ο αλγόριθμος υπολογισμού του H:

```
H[ n-1 ] = n
For i from n-2 to 0:
        s.push(i+1)
        while s.not_epty():
                if A[ i ] ≤ A[ s.top() ] :
                        s.pop()
                else:
                        break
        if s.not_empty():
                H[ i ] = s.top()
        else:
                H[ i ] = n
```

Στην περίπτωση που δεν υπάρχει θέση j στην λίστα μας για έναν κόμβο i για την οποία να ισχύει j>i και A[j] ≤ A[i] τότε η τιμή του H[i] θα ισούται με το n για να γνωρίζουμε στο τέλος αν υπάρχουν πιο οικονομικές στάσεις πέρα της θέσης i.

c) Αφού υπολογίσουμε τον H μπορούμε να αρχίσουμε την αναζήτηση του φθηνότερου τρόπου ανεφοδίασης του αυτοκινήτου μας. Έχουμε για κάθε στάση μας i την θέση j της αμέσως επόμενης φθηνότερης στάσης μας.
   - Αν η στάση j θέλει παραπάνω βενζίνη απο όση μπορούμε να βάλουμε στο αυτοκίνητό μας τότε γεμίζουμε το αμάξι μέχρι το όριο B και μεταφερόμαστε στην επόμενη στάση i+1 (κάνουμε το ίδιο έλεγχο για την καινούρια στάση).
   - Αν η στάση j θέλει συνολική βενζίνη b(a)≤B τότε γεμίζουμε το ντεπόζιτο με ακριβώς αρκετή βενζίνη b(a) ώστε όταν φτάσουμε στην στάση j να μην έχουμε άλλα καύσιμα. (κάνουμε το ίδιο έλεγχο για την καινούρια στάση)

Αλγόριθμος υπολογισμού επόμενης στάσης:

Η λίστα fuel_Buy[] έχει μήκος n και κρατάει το ποσό της βενζίνης που αγοράζουμε σε κάθε στάση. (Στην τελευταία στάση δεν αγοράζουμε καύσιμα). Η συνάρτηση distance_fuel(i , j) είναι γραμμική συνάρτηση και επιστρέφει την συνολική βενζίνη που χρειαζόμαστε για να μεταφερθούμε από τον κόμβο i στον j.

```
fuel_Buy[0 ] = 0
For i from 1 to n-1:
        _Fuel_To_Next_Stop = distance_fuel( i , H[ i ] )
        if _Fuel_To_Next_Stop ≤ B:
                if _Fuel_To_Next_Stop – fuel_Buy[ i-1 ] < 0:
                        fuel_Buy[ i ] = 0
                else:
                        fuel_Buy[ i ] = _Fuel_To_Next_Stop – fuel_Buy[ i-1 ]
        else:
                fuel_Buy[ i ] = B
```

Η συνάρτησή μας είναι γραμμική με πολυπλοκότητα O(n)

Η ορθότητα του βασίζεται στα εξής επιχειρήματα:

- Αν τα καύσιμα που έχουν απομείνει στο ρεζερβουάρ είναι αρκετά για να φτάσουμε στον προορισμό μας, δεν χρειάζεται να σταματήσουμε πουθενά.
- Αν βρισκόμαστε σε μια πόλη όπου το επόμενο φθηνότερο πρατήριο καυσίμων είναι στην θέση j, τότε γεμίζουμε μόνο όσο χρειάζεται για να φτάσουμε εκεί.
- Αν τα υπόλοιπα καύσιμα δεν φτάνουν μέχρι την επόμενη πόλη ή δεν υπάρχουν φθηνότερα πρατήρια καυσίμων μπροστά, γεμίζουμε το ρεζερβουάρ και επαναλαμβάνουμε τη διαδικασία στην επόμενη πόλη.

Θα αποδείξουμε ότι αυτός ο αλγόριθμος, καθοδηγούμενος από μια άπληστη προσέγγιση, δίνει τη βέλτιστη λύση. Οι κατευθυντήριες αρχές για αυτόν τον "άπληστο" αλγόριθμο είναι τα τρία βήματα που αναφέρθηκαν παραπάνω. Είναι προφανές ότι αν μια λύση δεν τηρεί το πρώτο βήμα, δεν μπορεί να θεωρηθεί βέλτιστη. Επομένως, προχωράμε στο δεύτερο βήμα. Εάν ισχύει η συνθήκη του δεύτερου

βήματος, μια λύση που δεν ακολουθεί την λογική αυτή πρέπει να αγοράσει καύσιμα σε κάποια ενδιάμεση πόλη (πριν φτάσει στην στάση j και μετά την αρχική στάση i) για να εξασφαλίσει την επίτευξη του προορισμού. Εάν, σε αυτή τη λύση, "μεταφέρουμε" κάποια ποσότητα αγορασμένου καυσίμου από την ενδιάμεση πόλη και την προμηθευτούμε στην αρχική πόλη, επιτυγχάνουμε χαμηλότερο κόστος. Για να δειχθεί πως το τρίτο βήμα είναι σωστό, μπορεί να χρησιμοποιηθεί ο ίδιος μηχανισμός ανταλλαγής. Κατά συνέπεια, ο αλγόριθμός μας παράγει τη βέλτιστη λύση.

4.2)

Για την λύση του προβλήματος αυτού θα χρησιμοποιήσουμε τις σημειώσεις του To Fill or not to Fill: The Gas Station Problem. Θα θεωρήσουμε $C(u,b)$ το ελάχιστο κόστος για να μεταβούμε από τον κόμβο u στον τελικό κόμβο t έχοντας αρχική ποσότητα καυσίμων b.
Ο αλγόριθμος λύσης του προβλήματος είναι :

```
Function C( u , b ):
        if u==t:
                return ( 0 , [0 for i in range(n)] )
        R ← {v ϵ V | d_uv ≤ B }
        MinCost= ∞
        return_list=[]
        For v ϵ R do:
                if c(v) ≤ c(u) :
                        cost_to_t ,fuel_list= C( v , 0  )
                        cost = cost_to_t  + d_uv c(u)
                else:
                        cost_to_t ,fuel_list= C( v , B-d_uv  )
                        fuel_list[v] += B-b
                        cost = cost_to_t + B c(u)
                if cost < MinCost:
                        MinCost = cost
                        return_list = fuel_list
        return MinCost , return_list
```

Ξεκινάμε τον αλγόριθμο μας με το κάλεσμα του C(s , 0). Η συνάρτηση $C(u,b)$ περνάει όλους τους κόμβους v ϵ V όπου duv ≤ B δηλαδή προσδιορίζονται οι προσβάσιμες πόλεις από την τρέχουσα πόλη με βάση την εναπομένουσα χωρητικότητα καυσίμου .
Εάν το κόστος καυσίμου στην τρέχουσα πόλη είναι μικρότερο ή ίσο με την πόλη προορισμού, ο αλγόριθμος προσομοιώνει το γέμισμα του ρεζερβουάρ για να φτάσει στην επόμενη πόλη. Αντίθετα, εάν το κόστος καυσίμων είναι υψηλότερο στην τρέχουσα πόλη, αγοράζει μόνο το ακριβές αναγκαίο ποσό καυσίμου. Καθ' όλη τη διάρκεια αυτής της διαδικασίας, ο αλγόριθμος διατηρεί αρχείο με το ελάχιστο κόστος και τον αντίστοιχο κατάλογο κατανάλωσης καυσίμων για ολόκληρη τη διαδρομή ώστε να έχουμε εν τέλη το ελάχιστο κόστος και το αντίστοιχο πλάνο ανεφοδιασμού. Η στρατηγική αυτή υπαγορεύει ότι όταν οι τιμές των καυσίμων στην τρέχουσα πόλη είναι υψηλότερες από εκείνες της επόμενης πόλης, το ρεζερβουάρ γεμίζει μέχρι το τέρμα- διαφορετικά, αγοράζεται μόνο τόσο καύσιμο όσο χρειάζεται για να φτάσει στην επόμενη πόλη.

Πολυπλοκότητα:

Με βάση τον αλγόριθμο θα για κάθε κόμβο u (συνολικά |V| κόμβοι) θα ψάξουμε για το πολύ |V| άλλους κόμβους με είσοδο το πολύ |V| διαφορετικές τιμές για το d. Άρα η πολυπλοκότητα είναι $O(|V|^3)$

Άσκηση 5

5.α)

Για την επίλυση του προβλήματος αυτού θα χρησιμοποιήσουμε τον αλγόριθμο Hopcroft Karp που χρησιμοποιείται για την εύρεση των Maximum *cardinality matches* σε διμερή γραφήματα. Ο αλγόριθμος hopcroft karp προσπαθεί να ενώσει κάθε κόμβο στο αριστερό σετ S_L (στην περίπτωσή μας τα κάστρα) με έναν κόμβο στο δεξί σετ S_R (στην περίπτωσή μας οι ιππότες). Γνωρίζουμε όμως πως ένας ιππότης μπορεί να έχει το πολύ c_i κάστρα υπό την εποπτεία του. Για αυτόν τον λόγο ο ιππότης i εισάγεται c_i φορές στο σετ S_R των ιπποτών. Οι ιππότες στο σετ S_R δεν είναι απαραίτητο να ενωθούν με κάστρο.

Αλγόριθμος :

1. Αρχικά θα περάσουμε από κάθε κάστρο j του S_L και θα το ενώσουμε με τον πρώτο ιππότη i που βρίσκονται στο S_R και ο οποίος δεν έχει ενωθεί ακόμη με κάστρο.
2. Αν όλα τα j του S_L έχουν ενωθεί τότε σταματάμε τον αλγόριθμο μας.
3. Αλλιώς για κάθε j του S_L το οποίο δεν έχει ενωθεί με κάποιον ιππότη:
   i. Θα κάνουμε μια BFS αναζήτηση με αρχικό κόμβο τον j με κάθε επίπεδο να είναι κόμβοι ενός διαφορετικού από τα σετ S_L , S_R. Η αναζήτηση ολοκληρώνεται στο πρώτο στρώμα k όπου φτάνει σε έναν ή περισσότερους ελεύθερους κόμβους του S_R. (Στην περίπτωση που δεν υπάρχει ανάθεση που ικανοποιεί όλους τους περιορισμούς , ο BFS μας θα το διαπιστώσει)
   ii. Για την αναζήτηση των νέων ζευγαριών κόμβων χρησιμοποιείται μια αναζήτηση DFS για να βρεθεί το μονοπάτι απο το τελευταίο επίπεδο του προηγούμενου BFS δέντρου μας προς το πρώτο κόμβο στο δέντρο. Σε κάθε δεδομένο επίπεδο, η αναζήτηση πρώτα σε βάθος θα ακολουθήσει τις ακμές που οδηγούν σε έναν αχρησιμοποίητο κόμβο από το προηγούμενο επίπεδο.
   iii. Μόλις ο αλγόριθμος βρει έναν αρχικό αχρησιμοποίητο κόμβο του S_L σταματάει την DFS αναζήτηση και ξεκινώντας απο το επίπεδο 0 ενώνει κάθε κόμβο (του S_L) στο επίπεδο k με τον κόμβο (του S_R) του επιπέδου k+1

Πολυπλοκότητα:

Με βάση τον ορισμό του αλγορίθμου Hopcroft Karp, η πολυπλοκότητα μας θα είναι $O(|E|\sqrt{|V|})$ χρόνο στη χειρότερη περίπτωση, όπου E είναι το σύνολο των ακμών του γραφήματος και V είναι το σύνολο των κορυφών του γραφήματος.

Άσκηση 6

Κάνουμε μετασχηματισμό των δοθέντων προσφορών σε ένα κατευθυνόμενο γράφημα με τον εξής τρόπο:

1. Για κάθε προσφορά δημιουργούμε μία κορυφή.
2. Από κάθε κορυφή i εξέρχονται ακμές προς τους κόμβους j για τους οποίους ισχύει $t_i < s_j$. Όπου s_j χρονική στιγμή δέσμευσης του αυτοκινήτου και t_i η χρονική στιγμή αποδέσμευσης του αυτοκινήτου. Με αυτόν τον τρόπο δημιουργούμε όλα τα δυνατά μονοπάτια ενοικίασης που μπορούν να ικανοποιηθούν με ένα αυτοκίνητο. Κάθε ακμή έχει κόστος $-p_i$.
3. Δημιουργούμε την τελική κορυφή t. Από όλες τις κορυφές i που δημιουργήθηκαν στο βήμα 1 εξέρχονται ακμές κόστους $-p_i$ και μεταφορικής ικανότητας 1 προς την κορυφή t'.
4. Δημιουργούμε την αρχική κορυφή s. Από την κορυφή αυτή εξέρχονται ακμές προς όλες τις κορυφές που δημιουργήθηκαν στο βήμα 1. Οι κορυφές αυτές έχουν μηδενικά κόστη και μεταφορική ικανότητα 1.
5. Δημιουργούμε την κορυφή tπου είναι η κορυφή sink. Στην ακμή αυτή εισέρχεται μόνο μία ακμή από την κορυφή t' με μηδενικό κόστος και μεταφορική ικανότητα k. Με αυτόν τον τρόπο περιορίζουμε max flow του γραφήματος έτσι ώστε να μην μπορεί να είναι μεγαλύτερο από k.

Παρατηρήσεις:

- Το γράφημα που προκύπτει έχει $n+2$ κόμβους.
- Το γράφημα που προκύπτει στα βήματα 1 και 2 είναι DAG. Αυτό οφείλεται στο γεγονός ότι για κάθε ακμή (u,v) ισχύει $t_u < s_j$. Άρα η ακμή με το μικρότερο t_i μπορεί να συνδεθεί το πολύ με $n-1$κορυφές, η ακμή με το επόμενο μικρότερο t_i μπορεί να συνδεθεί με $n-2$ κλπ. Αυτό σημαίνει ότι συνολικά οι ακμές του γραφήματος που προκύπτει από ολόκληρη την παραπάνω διαδικασία είναι:
$$O\left(n\frac{n+1}{2}\right)+2n$$
- Από το παραπάνω προκύπτει και ότι η χρονική πολυπλοκότητα της κατασκευής του γραφήματος αυτού είναι $O(n^2)=O(m)$.

Λύνοντας το $M \in -Cost-Flow$ πρόβλημα στο παραπάνω γράφημα θα προκύψει ένα ελάχιστο κόστος C_{min}.Το μέγιστο ποσό που μπορεί να εισπράξει η εταιρία από τις προσφορές αυτές είναι $P_{max} = -C_{min}$. Επίσης, το κάθε μονοπάτι του $M \in -Cost-Flow$ γραφήματος είναι οι ενοικιάσεις ενός από τα k διαθέσιμα αυτοκίνητα.

- **Πολυπλοκότητα.** Όπως αναφέρθηκε και παραπάνω η χρονική πολυπλοκότητα της κατασκευής του γραφήματος αυτού είναι $O(n^2)$. Συνολικά, η πολυπλοκότητα καθορίζεται από τον αλγόριθμο υπολογισμού $Min-Cost-Flow$. Για να πετύχουμε πολυπλοκότητα που είναι ανεξάρτητη του p_i θα πρέπει να χρησιμοποιήσουμε κάποιον αλγόριθμο $min-cost-flow$ που δεν εξαρτάται η τιμή του από την μέγιστη απόλυτη τιμή των κοστών p_i. Για παράδειγμα ο αλγόριθμος των Goldberg και Tarjan, που χρησιμοποιεί τον αλγόριθμο του Karp για να βρίσκει κύκλους ελαχίστου μήκους, έχει πολυπλοκότητα $O(n|E|^2\log(n))=O(n^5\log(n))$, όπου nο αριθμός των προσφορών.