Κατ'οίκον Εργασία 2 – Σκελετοί Λύσεων

1. Ο αλγόριθμος κτίζει όλες τις δυνατές αναθέσεις εργασιών στους φοιτητές (υπάρχουν n! διαφορετικές αναθέσεις) και επιστρέφει εκείνη με το μέγιστο βαθμό καταλληλότητας.

   Το κτίσιμο των αναθέσεων γίνεται ως εξής. Ξεκινούμε με την κενή ανάθεση αξίας 0 (jobs[n], val = 0 στον κώδικα) και την επεκτείνουμε σε μία 1-υποσχόμενη ανάθεση, 2-υποσχόμενη ανάθεση, και ούτω καθεξής, όπου μία ανάθεση είναι i-υποσχόμενη αν αναθέτει στους i πρώτους φοιτητές i διαφορετικές εργασίες. Η επέκταση μιας ανάθεσης υλοποιείται κατά τις επαναλήψεις του βρόχου στην περίπτωση i < n, όπου η εργασία που ανατίθεται στον i-οστό φοιτητή είναι η k. Ταυτόχρονα ο βαθμός καταλληλότητας val της ανάθεσης αυξάνεται κατάλληλα.

   Όταν μία n-υποσχόμενη ανάθεση ολοκληρωθεί (συνθήκη i == n στον κώδικα) συγκρίνεται με την καλύτερη ανάθεση που έχει βρεθεί μέχρι στιγμής (στον κώδικα η best[n] αξίας max) και, αν είναι καλύτερη, η τιμή της best ανανεώνεται κατάλληλα.

   Για υπολογισμό της επόμενης ανάθεσης, εφαρμόζεται οπισθοδρόμηση. Στόχος της οπισθοδρόμησης είναι η εύρεση του πρώτου φοιτητή (ξεκινώντας από τον τελευταίο και προχωρώντας προς τον πρώτο) στον οποίο μπορεί να δοθεί κάποια άλλη εργασία. Παρατηρήστε ότι, σε κάθε φάση, στον φοιτητή i δίνεται η εργασία με τον μικρότερο αριθμό από εκείνες που δεν του έχουν ήδη δοθεί και που δεν είναι δεσμευμένες από τους φοιτητές 1..i-1. Κατά συνέπεια η συνθήκη k > n στον κώδικα εκφράζει ότι δεν υπάρχουν άλλες δυνατές αναθέσεις για τον φοιτητή i. Σε τέτοια περίπτωση οπισθοδρομούμε στον φοιτητή i-1. Αν το i πάρει την τιμή 0 συμπεραίνουμε ότι ο αλγόριθμος έχει κτίσει όλες τις δυνατές αναθέσεις, και επιστρέφουμε την ανάθεση που είναι αποθηκευμένη ως best.

```
int jobs[n];
int val = 0;
int best[n];
int max = 0;
int i = 1;

while (i <= n){

  if (i < n)
       k = smallest job not already in jobs[1..i-1];
       jobs[i] = k;
       val =+ compatibility (i,k);
       i++;

  if (i == n)

       if (val > max)
            max = val;
            best = jobs;

       while (i > 0 AND k > n)
            k = smallest job not already in jobs[1..i-2]
```

```
                        greater than jobs[i-1];
                  i--;
            if (i == 0)
                  return best;
            else
                  jobs[i] = k;
                  val =+ compatibility (i,k);
                  i++;
      }
```

Ο χρόνος εκτέλεσης του αλγορίθμου είναι της τάξης O(n!).

2. (α) Χρησιμοποιούμε ένα πίνακα $V[1..n, 1..W, 1..R]$. Στη θέση $V[i,j,k]$ του πίνακα θα αποθηκεύουμε τη μέγιστη αξία συλλογής αντικειμένων από το σύνολο $1..i$ της οποία το βάρος δεν ξεπερνά το βάρος j και τον όγκο k.

Έχουμε τη σχέση

$$V[i,j,k] = \begin{cases} v_i, & \text{if } i = 1, w_i \le j, r_i \le k \\ \max(V[i-1,j,k], & \\ \quad V[i-1,j-w_i,k-r_i]+v_i), & \text{if } i > 1, w_i \le j \text{ and } r_i \le k \\ V[i-1,j,k], & \text{if } i > 1, w_i > j \text{ or } r_i > k \\ 0, & \text{otherwise} \end{cases}$$

Υπολογίζουμε το $V[n,W,R]$, ξεκινώντας από το $V[1,1,1],\ldots,V[1,1,R]$, $V[1,2,1],\ldots,V[1,2,R]$, …, $V[1,W,1],\ldots,V[1,W,R]$, και προχωρώντας στο $V[2,\_]$, $V[3,\_],\ldots$.

(β) Δημιουργούμε ένα γράφο με βάρη (V,E), οι κόμβοι του οποίου είναι τριάδες της μορφή (S, w, r). Θέτουμε

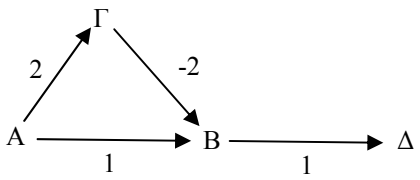
- $(S, w, r) \in V$, αν S είναι κάποιο υποσύνολο των αντικειμένων με συνολικό βάρος $w \le W$ και συνολικό όγκο $r \le R$, και
- $((S, w, r), (S', w', r')) \in E$, αν $S' = S \cup \{i\}$, $w' = w + w_i$, $r' = r + r_i$. Θέτουμε ως βάρος της ακμής την τιμή $-v_i$.

Παρατηρούμε ότι ο γράφος δεν περιέχει κύκλους.

Με αυτή τη μοντελοποίηση, το πρόβλημα μας ανάγεται στην εύρεση του βραχύτερου μονοπατιού από τον κόμβο $(\varnothing, 0, 0)$ και μπορεί να λυθεί με τον αλγόριθμο των Bellman-Ford. (Αφού ο γράφος είναι άκυκλος, το πρόβλημα ορίζεται).

Η χρονική πολυπλοκότητα του αλγορίθμου είναι $V \cdot E$. Παρατηρούμε ότι, στη χείριστη περίπτωση, όπου όλες οι δυνατές επιλογές αντικειμένων ικανοποιούν τους περιορισμούς βάρους και όγκου, ο γράφος περιέχει 2^n κόμβους (κάθε αντικείμενο μπορεί είτε να βρίσκεται είτε να μην βρίσκεται σε κάποια συλλογή) και ο αλγόριθμος έχει εκθετικό χρόνο εκτέλεσης.

3. (α) Ο πιο κάτω γράφος, χωρίς να έχει κύκλους αρνητικού μεγέθους, προκαλεί την αποτυχία του αλγορίθμου του Dijkstra.



Η αποτυχία προκαλείται λόγω του ότι, στην παρουσία ακμών αρνητικού μεγέθους, είναι δυνατόν μία κορυφή *v* να μπει στο σύνολο *S* (δες αλγόριθμο) χωρίς να έχει υπολογιστεί η μικρότερη προς αυτήν απόσταση, δηλαδή, χωρίς να έχει τεθεί $d[v] = \delta(s,v)$ (Θεώρημα 2). Αυτό έχει ως αποτέλεσμα, αφού κάθε κορυφή τυγχάνει επεξεργασίας ακριβώς μία φορά, ακόμα και αν το $d[v]$ πάρει στη συνέχεια την ορθή του τιμή, να μην ενημερωθούν κατάλληλα οι γείτονες του *v*. Στο παράδειγμα, η *B* τυγχάνει επεξεργασίας όταν $d[B] = 1$, δίνοντας $d[\Delta] = 2$, το οποίο δεν αλλάζει στη συνέχεια.

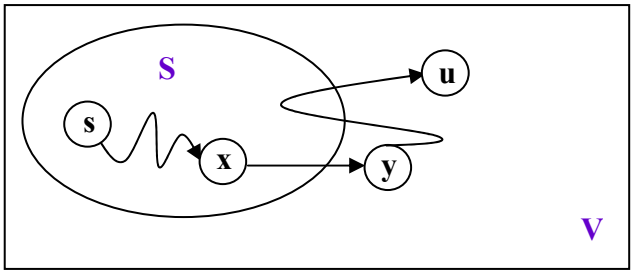
(β) Σε αυτή την περίπτωση ο αλγόριθμος είναι ορθός. Αρκεί να δείξουμε ότι όταν μια κορυφή u 'μπαίνει' στο S, τότε $d[u] = \delta(s,u)$.

<u>Απόδειξη</u>

Υποθέτουμε, για να φτάσουμε σε αντίφαση, ότι η *u* είναι η πρώτη κορυφή η οποία, κατά την εκτέλεση του αλγόριθμου, μπαίνοντας στο *S* έχει $d[u]$ μεγαλύτερο από το βάρος του βραχύτερου μονοπατιού μεταξύ της *u* και της *s*.

Έστω $p = s \rightarrow x \rightarrow \ldots \rightarrow u$ το βραχύτερο μονοπάτι από την *s* στη *u*. Αν $x = u$, δηλαδή το μονοπάτι έχει μήκος 1, τότε είναι προφανές ότι κατά την επεξεργασία της *s* από τον αλγόριθμο το $d[x]$ παίρνει την ορθή του τιμή η οποία δεν αλλάζει στη συνέχεια.

Διαφορετικά, έστω *y* η `πρώτη΄ κορυφή στο *V-S* που ανήκει στο βραχύτερο μονοπάτι από την *s* στη *u*. (Αν δεν υπάρχει τέτοια κορυφή, τότε, το αποτέλεσμα αποδεικνύεται όπως πιο πάνω.)



Αφού η *x* μπήκε στο *S* πριν από την *u*, $d[x]=\delta(s,x)$.

Επίσης, με την εισαγωγή του x στο S, ετέθει $d[y]=d[x]+w(x,y)$, το οποίο είναι το κόστος του υπομονοπατιού στο σχήμα.

Αφού το μονοπάτι από το *s* στο *u*, είναι βραχύτερο, τότε από τη δομή βέλτιστης λύσης συνεπάγεται ότι το υπομονοπάτι $s \Rightarrow x \rightarrow y$ από το *s* στο *y* είναι επίσης βραχύτερο. Άρα $d[y] = \delta(s,y)$.

Έτσι:

$d[u]$	$> \delta(s,u)$	(αρχική υπόθεση)
	$= \delta(s,y) + \delta(y,u)$	(δομή βέλτιστης λύσης)
	$= d[y] + \delta(y,u)$	($d[y] = \delta(s,y)$)
	$\geq d[y]$	(τα βάρη στο μονοπάτι $\delta(y,u)$ είναι ≥ 0)

Αφού $d[u] > d[y]$, ο αλγόριθμος θα διάλεγε και θα εισήγαγε τη y στο S και όχι τη u. **Αντίφαση!**

(γ) Ακολουθεί ο ζητούμενος αλγόριθμος. Μετά από τις απαραίτητες αρχικοποιήσεις, αν κατά την εκτέλεση του αλγορίθμου, κάποιος κόμβος u επιτρέψει μονοπάτι προς κάποιο κόμβο v με μικρότερο βάρος από αυτό που ήδη γνωρίζουμε, τότε ο αριθμός των μονοπατιών προς τον v γίνεται όσος και τα μονοπάτια προς τον u. Αν ο u επιτρέψει μονοπάτι προς τον v με ίδιο βάρος με αυτό που ήδη γνωρίζουμε, τότε ο αριθμός των μονοπατιών προς τον v αυξάνεται κατά τον αριθμό μονοπατιών προς τον u.

```
heap Q;
for all v∈V
      d[v]=∞;
      num[v]=0;
d[s]=0;
num[s]=1;

S=∅;
Q=V;
while (Q ≠ ∅){
      u=DeleteMin(Q);
      S=S∪{u};
      για κάθε γείτονα v του u
            if d[v]>d[u]+w(u,v)
                  d[v]=d[u]+w(u,v);
                  num[v]=num[u];
            if d[v]== d[u]+w(u,v)
                  num[v]=+num[u];
}
```

4. Για επίλυση του προβλήματος, επεκτείνουμε κατάλληλα τον αλγόριθμο των Floyd-Warshal. Συγκεκριμένα χρησιμοποιούμε εκτός από τον πίνακα c, τον πίνακα nf, όπου $nf[i,j,m]$ συμβολίζει τον αριθμό των πτήσεων στη φθηνότερη διαδρομή από την πόλη i στην πόλη j με ενδιάμεσες στάσεις που ανήκουν στο σύνολο $\{1,2,\ldots,m\}$.

```
for all i,j
   c [i,j,0] = w(i,j);
   nf[i,j,0] = 1;

   for(m=1; m≤n; m++)
       for (i=1; i≤n; i++)
            for (j=1; j≤n; j++)
```

```
if (c[i,j,m-1] > c[i,m,m-1]+c[m,j,m-1])
                    OR
    [(c[i,j,m-1] == c[i,m,m-1] + c[m,j,m-1])
                    AND
      (nf[i,j,m-1] > nf[i,m,m-1] + nf[m,j,m-1])]

    c[i,j,m] = c[i,m,m-1]+ c[m,j,m-1]);
    nf[i,j,m] = nf[i,m,m-1]+ nf[m,j,m-1]);
```