Φροντιστήριο 10 – Σκελετοί Λύσεων

Άσκηση 1

Μας δίνεται ένας γράφος G και θέλουμε να αποφασίσουμε κατά πόσο είναι 2-χρωματίσιμος. Για να το πετύχουμε, επιχειρούμε χρωματισμό του γράφου ως εξής: Ξεκινούμε με κάποιο τυχαίο κόμβο και τον χρωματίζουμε με το χρώμα 1. Στη συνέχεια παίρνουμε όλους τους γείτονες του και τους χρωματίζουμε με το χρώμα 2. Συνεχίζουμε με τους γείτονες αυτών και τους χρωματίζουμε με το χρώμα 1, και ούτω καθεξής. Αν συναντήσουμε κάποιο κόμβο ο οποίος είναι ήδη χρωματισμένος υπάρχουν δύο περιπτώσεις: αν είναι χρωματισμένος με το χρώμα που θα θέλαμε να τον χρωματίσουμε, τότε προχωρούμε κανονικά, αν όμως είναι χρωματισμένος με το άλλο χρώμα, συμπεραίνουμε ότι ο γράφος μας δεν είναι 2-χρωματίσιμος και τερματίζουμε την εκτέλεση της διαδικασίας.

Η διαδικασία αυτή είναι επέκταση της διαδικασίας κατά-πλάτος διερεύνησης σε γράφους:

```
2-colorable(graph G, vertex v){

   Q=MakeEmptyQueue();

   for each w in G
        color[w] = 0;

   color[v] = 1;
   Enqueue(v,Q);

   while (!IsEmpty(Q)){
        w = Dequeue(Q);
        for each u adjacent to w
             if (color[u] != 0)
                  if (color[w] == color[u])
                     report THE GRAPH IS NOT 2-COLORABLE
             else
                  if (color[w] == 1)
                       color[u] = 2;
                  else
                       color[u] = 1;
                  Enqueue(u,Q);
}
```

Άσκηση 2

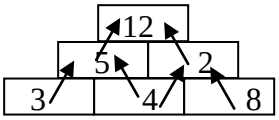
Το πρόβλημα μπορεί να μεταφραστεί σε πρόβλημα εύρεσης του μονοπατιού με το μέγιστο κόστος σε γράφο: Θεωρήστε τον γράφο με βάρη G=(V,E) όπου

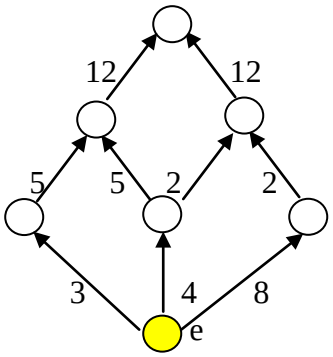
V={ $s_1,...,s_n$ }, δηλαδή υπάρχει μία κορυφή για κάθε πέτρα και η επιπλέον κορυφή e (η γή),

E={(s_i, s_j)| αν $s_i \prec s_j$ }∪{(e, s_i) | αν δεν ισχύει $s_i \prec s_j$ για κανένα j}, και

βάρη w(u,v) = t(v).

Για παράδειγμα,





α. Ο ελάχιστος χρόνος που χρειάζεται για την ολοκλήρωση της πυραμίδας, (υποθέτοντας ότι πέτρες μπορούν να τοποθετηθούν στη θέση τους παράλληλα) είναι ίσος με το μέγιστο κόστος μονοπατιού από την κορυφή e στον πιο πάνω γράφο.

Για το i ισχύει το ακόλουθο:

- earliestEventTime(0) = 0
- earliestEventTime(i) = max {earliestEventTime(j) + length(j,i)}, (j,i) ϵ E,
  - E είναι το σύνολο των ακμών
  - length(j,i) είναι ο χρόνος που χρειάζεται για την δραστηριότητα (j,i)

Για να το υπολογίσουμε, χρησιμοποιούμε την πιο κάτω αναδρομική συνάρτηση:

```
int earliestEventTime(graph GRAPH [][], int vertex) {
        int i;
        int max=0, val;
        for (i=0; i<N; i++) {
                if (GRAPH[i][vertex] != 0) {
                        // filter out the max
                        val = earliestEventTime(i) + GRAPH[i][vertex];
                        if (val>max)
                                max = val;
                }
        }
        return max;
}
```

β. Έστω ότι latestEventTime(i) είναι ο πιο αργός χρόνος για τον οποίο το γεγονός i μπορεί να συμβεί. Εάν το γεγονός i δεν συμβεί μέχρι τον χρόνο αυτό, τότε το έργο δεν μπορεί να ολοκληρωθεί σε χρόνο ίσο με το μήκος έργου. Άρα το latestEventTime του τελικού γεγονότος είναι ίσο με το earliestEventTime του τελικού γεγονότος. Για το i ισχύει το ακόλουθο:

- latestEventTime(n-1) = earliestEventTime(n-1)
- latestEventTime(i) = min {latestEventTime(j) - length(i,j)}, (i,j) ϵ E,
  - E είναι το σύνολο των ακμών
  - length(j,i) είναι ο χρόνος που χρειάζεται για την δραστηριότητα (j,i)

Για να το υπολογίσουμε, χρησιμοποιούμε την πιο κάτω αναδρομική συνάρτηση:

```
int latestEventTime(graph GRAPH [][], int vertex) {
        int j;
        int min = INT_MAX; // in order to recognize the final vertex
        int val;

        for (j=0; j<N; j++) {
                if (GRAPH[vertex][j] != 0) {
                        val = latestEventTime(j) - GRAPH[vertex][j];
                        if (val<min)
                                min = val;
                }
        }

        if (min == INT_MAX)
                return earliestEventTime(vertex);

        return min;

}
```