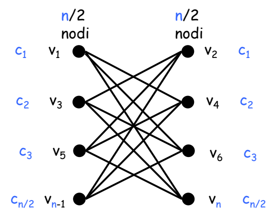
Algoritmi e Strutture Dati - 17/12/15

Esercizio 1

Si consideri il grafo bipartito in figura:



Sappiamo che ogni grafo bipartito è 2-colorabile; ma seguendo l'algoritmo sui nodi nell'ordine $v_1, v_2, \ldots, v_n$, vengono assegnati $n/2$ colori come descritto in figura, e quindi tale algoritmo non è ottimo.

Esercizio 2

Sia $DP[i][j]$ il valore che posso ottenere dai primi i elementi, avendo la disponibilità residua di al più j elementi consecutivi. In altre parole, per via di selezioni precedenti, posso continuare a scegliere fino a j elementi, ma poi dovrò saltare un elemento. La soluzione del problema originale si trova in $DP[n][k]$.
Una formulazione ricorsiva è la seguente:

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ DP[i-1][j] & i > 0 \wedge j = 0 \\ \max\{DP[i-1][j], DP[i-1][j-1] + V[i]\} & i > 0 \wedge j > 0 \end{cases}$$

In pratica, se $j = 0$ si è costretti a saltare, mentre se $j > 0$ si può scegliere di saltare o non saltare un elemento; se si decide di saltare, il numero di elementi consecutivi selezionabili si resetta a k; se si decide di non saltare, restano a disposizione $j - 1$ elementi consecutivi e bisogna sommare il valore selezionato.
È possibile risolvere il problema con memoization nel modo seguente:

```
int kOccurrence(int[] V, int n, int k)
    int[][] DP = new int[1...n][1...n] = {−1}          % Initialized to −1
    return kOccurrenceRec(V, n, k, DP)
```

```
int kOccurrenceRec(int[] V, int i, int j, int[][] DP)
    if i ≤ 0 then
        return 0
    if DP[i][j] < 0 then
        if j == 0 then
            DP[i][j] = kOccurrenceRec(V, i − 1, k, DP)
        else
            DP[i][j] = max(kOccurrenceRec(V, i − 1, k, DP), kOccurrenceRec(V, i − 1, j − 1, D) + V[i])
    return DP[i][j]
```

La complessità è pari a $O(nk)$, in quanto è necessario riempire tutta la tabella.

Esercizio 3

Per calcolare il numero di disposizioni $DP[n]$, è semplice definire una relazione di ricorrenza definita sulla dimensione n. Ci sono due possibilità: viene collocata una tessera verticale, e si calcolano le disposizioni per $n - 1$; oppure due tessere orizzontali, e si calcolano le disposizioni per $n - 2$. I risultati di tali calcoli vanno sommati. Per $n = 1$ e $n = 0$, il numero di disposizioni è pari a 1.

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ DP[n-1] + DP[n-2] & n > 1 \end{cases}$$

Ma questa non è altro che la sequenza di Fibonacci, spostata di un'unità:

```
countDomino(int n)
    return Fibonacci(n + 1)
```

Il costo è quindi $O(n)$.

Esercizio 4

L'algoritmo che proponiamo agisce solo se la capacità dell'arco (u, v) è stata saturata dal flusso che passa attraverso di esso; in caso contrario, non c'è nulla da fare: il flusso massimo definito sulla capacità c è anche il flusso massimo sulla capacità modificata.

Se la capacità è stata saturata, la riduzione di un'unità di capacità richiederà una corrispondente riduzione di un'unità di flusso su (u, v). Ma per la conservazione del flusso, sarà necessario ridurre parimenti il flusso in uno degli archi (v, v') uscenti da v, e in uno degli archi (u', u) entranti in u. Ma questo comporterà un effetto a cascata sugli archi entranti in u' e gli archi uscenti da v', che si interromperà solo nella sorgente e nel pozzo (che non sono soggetti alla conservazione del flusso). È necessario quindi individuare un cammino dalla sorgente a u, e un cammino da v al pozzo, definiti sugli archi tali per cui $f[u, v] > 0$ e diminuire di un'unità tutti gli archi lungo questi due cammini. Questo è realizzato dalla funzione ricorsiva reduceFlow(F, n, x, y), che riduce il flusso di 1 da x a y.

Il flusso così ottenuto è un flusso che rispetta la nuova matrice di capacità; ma non è detto che sia massimo. Occurre quindi lanciare una nuova ricerca del cammino aumentante a partire dalla sorgente, per scoprire eventuali altre strade.

```
reduceFlow(int[][] c, int n, int s, int p, int[][] f, int u, int v)
    c[u][v] = c[u][v] − 1
    if c[u][v] < f[u][v] and f[u][v] > 0 then
        f[u][v] = f[u][v] − 1
        f[v][u] = f[v][u] + 1
        reduceDFS(f, n, s, u)
        reduceDFS(f, n, v, p)
        r = c − f
        g = cammino-aumentante(r, n, s, p)
        f = f + g
```

```
reduceDFS(int[][] f, int n, int x, int y)
    if x == y then
        return true
    for w = 1 to n do
        if f[x][w] > 0 then
            if reduceDFS(f, n, w, y) = true then
                f[x][w] = f[x][w] − 1
                f[w][x] = f[w][x] + 1
                return true
    return false
```

La complessità è pari al costo di tre visite, che realizzate sulla matrici hanno costo $O(n^2)$. È possibile scrivere soluzioni che funzionano in tempo $O(m + n)$.