# Gruppo 20

# progetto 3

## Esercizio 1

## Esercizio 2

La funzione *find\_route()* prende in input l’orario della compagnia, gli aeroporti *a* e *b*, ed un orario di partenza *t*, e trova la rotta che permette di arrivare da *a* a *b* nel minor tempo possibile, partendo ad un orario non precedente a t.

L’algoritmo consiste in una versione modificata dell’algoritmo di Dijkstra in cui il peso di un arco è dato dal costo in termini di tempo della scelta di quel volo. In particolare, il peso è dato dal tempo di coincidenza dell’aeroporto sommato alla durata del volo e al tempo di attesa di quest’ultimo.

Le differenze principali introdotte rispetto all’algoritmo di Dijkstra sono:

* La fase di inizializzazione non prevede l’inserimento di tutti gli aeroporti nella coda ma solo quelli di cui viene scoperta l’esistenza tramite i voli analizzati.
* Si è reso necessario memorizzare per ogni aeroporto il tempo di arrivo per poter calcolare il costo di un volo.
* Per ogni aeroporto incontrato viene inoltre memorizzato il volo che ho preso per arrivarci in modo da semplificare la fase di ricostruzione della soluzione.

### Complessità computazionale

La complessità coincide con quella dell’algoritmo di Dijkstra nel caso di rappresentazione mediante matrici di adiacenza.

O(|E|\*log(|V|)

## Esercizio 3

La funzione seleziona la lista dei voli che, a partire da un certo budget, consente di massimizzare il numero di posti complessivo, ottenuto dalla somma dei posti assegnati a ogni volo. Siccome a ogni volo è associato un costo, la soluzione non deve superare il budget assegnato.

I parametri sono:

* flights: lista dei voli disponibili.
* budget: intero rappresentante il budget a disposizione.

Valore di ritorno:

* tupla contenente sia la lista dei voli, che una mappa in cui per ogni aereoporto, è riportato il budget da assegnare ad ogni aeroporto per finanziare tutti i voli in partenza e appartenenti alla soluzione.

Il problema si può approcciare attraverso la tecnica della programmazione dinamica. Definiamo come *m* il numero di voli della lista, *B* il budget a disposizione, v\_i il budget associato al volo i e c\_i il numero di posti associati al volo i. Si definisce una matrice M di dimensione m \* B+1, in cui in ogni cella viene riportato il numero massimo di posti ottenibili alla disamina del volo i e budget j. Le celle della prima riga della matrice, M[0][j], j=0,…,B, vengono inizializzate con 0, se v\_0 > j, dato che il volo non viene preso, oppure con c\_0, se il volo viene preso e quindi con v\_0 < j. Per le altre celle M[i][j] si considerano i seguenti casi:

se v\_i <= j allora M[i,j] = max{ c\_i+M[i-1][j-v\_i], M[i-1][j] },ovvero il massimo numero di posti considerati al volo i con budget j, viene aggiornato con la somma dei posti del volo i e i posti massimi al volo i-1 e budget rimanente dalla detrazione del costo del volo i, se questo valore è maggiore di quello precedente M[i-1][j], altrimenti viene confermato quello precedente.

Pertanto, l’elemento M[m][B] contiene il numero di posti massimo, considerati tutti i voli e il budget in questione B. Per sapere se il volo i viene preso o non nella soluzione, si utilizza una matrice parallela C, inizialmente inizializzata a False. Quando il volo viene preso al passo M[i][j], la corrispondente cella C[i][j] viene messa a True.

Per trovare dunque la lista dei voli nella soluzione si parte da M[n-1][B] e , se C[n-1][B] = True, allora il volo n-1 è inserito nella soluzione, e si riparte ugualmente con M[n-1][B-v\_n-1] e C[n-1][B-v\_n-1] finche n-1=0.

Una volta ottenuta la lista dei voli della soluzione, si registra con la mappa, per ogni aereoporto, il budget per pagare tutti i voli in partenza.

Si noti come i voli debbano essere ordinati per costo crescente se si vuole ottenere non solo la lista dei voli che massimizza il numero di posti ma che ne minimizzi anche il costo complessivo.

### Complessità computazionale

La complessità per la compilazione delle matrici di ricorrenza è O(m\*B) mentre la scrittura del dizionario è fatta in O(n). Per cui la complessità della select\_flight è

O(n+m\*B)

## Esercizio 4

Data la seguente dimostrazione:

Date le tre affermazioni

1. G e bipartito;
2. G e 2-colorabile e
3. G non contiene cicli di lunghezza dispari,

dimostriamo (i) ⇒ (ii), (ii) ⇒ (iii), (iii) ⇒ (i). Questo dimostra che (i) ⇔ (ii) e (i) ⇔ (iii).

1. Se G e bipartito, ` e 2-colorabile. Semplicemente, diamo colore ` 1 a tutti i nodi in una partizione, diamo colore 2 a tutti i nodi nell’altra. Non essendoci archi fra i nodi di una partizione, la colorazione è valida.

2. Se G e 2-colorabile, non contiene cicli di lunghezza dispari. Supponiamo per assurdo che esista un ` ciclo (v1, v2),(v2, v3). . . ,(vk−1, vk),(vk, v1), con k dispari. Se il nodo v1 ha colore 1, il nodo v2 deve avere colore 2; il nodo v3 deve avere colore 1, e così via fino al nodo vk, che deve avere colore 1. Poiché` v1 e successore di ` vk, v1 deve avere colore 2, assurdo.

3. Se non esistono cicli di lunghezza dispari, il grafo e bipartito. Dimostriamo questa affermazione ` costruttivamente. Si prenda un nodo x lo si assegna alla partizione S1. Si prendono poi tutti i nodi adiacenti a nodi in S1 e li si assegna alla partizione S2. Si prendono tutti i nodi adiacenti a nodi in S2 e li si assegna alla partizione S1. Questo processo termina quando tutti i nodi appartengono ad una o all’altra partizione. Un nodo può essere assegnato più di una volta se e solo se fa parte di un ciclo. Ma ` affinchè venga assegnato a due colori diversi, deve far parte di un ciclo di lunghezza dispari, e questo non è possibile.

Utilizziamo l’implicazione ii => i per trovare soluzione all’esercizio.

In particolare, viene utilizzata una versione modificata della BFS per verificare la 2-colorabilità del grafo. L’algoritmo utilizzato è il seguente:  
1. Assegna colore ROSSO (o “0”) al vertice sorgente (inserendolo nella partizione X).   
2. Colora tutti i vicini con colore BLU (o “1”)(inserendoli nella partizione Y).   
3. Colora il vicino di tutti i vicini con il colore ROSSO (inserendoli nella partizione X).   
4. Procedendo in questo modo, assegnare il colore a tutti i vertici in modo che vengano soddisfatti tutti i vincoli del problema di colorazione a m dove m = 2.  
5. Assegnando i colori, se troviamo un vicino che è colorato con lo stesso colore del vertice corrente, allora il grafo non può essere 2-colorato e di conseguenza non è bipartito.

La funzione deve funzionare anche su grafi non connessi, restituendo per ogni componente connessa le due partizioni, se è bipartibile. La soluzione complessiva è data dall’unione delle coppie di partizioni. Se almeno una componente connessa non è bipartibile, l’intero grafo non lo è. Per fare questo si utilizza un secondo metodo responsabile della verifica della bipartibilità della singola componente connessa del grafo, memorizzando di volta in volta i vertici visitati. In questo modo è possibile definire un’iterazione su tutte le componenti connesse, arrivando a coprire l’intero grafo e ad avere contezza della bipartibilità di esso.

### Complessità computazionale

La complessità computazionale rispecchia quella della BFS con grafo implementato mediante liste di adiacenza.

O(|V| + |E|)