# Gruppo 20

# progetto 3

## introduzione ai metodi di supporto

### Classi di supporto

Per la risoluzione degli esercizi proposti ci si è avvalso dell’impiego delle classi Airport e Flight, utilizzate per rappresentare rispettivamente gli aeroporti e i voli.

La classe Airport prevede due attributi per la memorizzazione del nome e del tempo minimo di coincidenza. Entrambi gli attributi con i relativi getter.

La classe Flight prevede invece gli attributi di: aeroporto di partenza, aeroporto di arrivo, orario di partenza del volo, orario di arrivo e numero di posti disponibili sul volo. Anche in questo caso sono presenti i relativi getter.

Entrambe le classi definiscono i magic methods \_\_hash\_\_, \_\_eq\_\_, \_\_str\_\_, \_\_repr\_\_ per facilitare le operazioni di uguaglianza e di stampa.

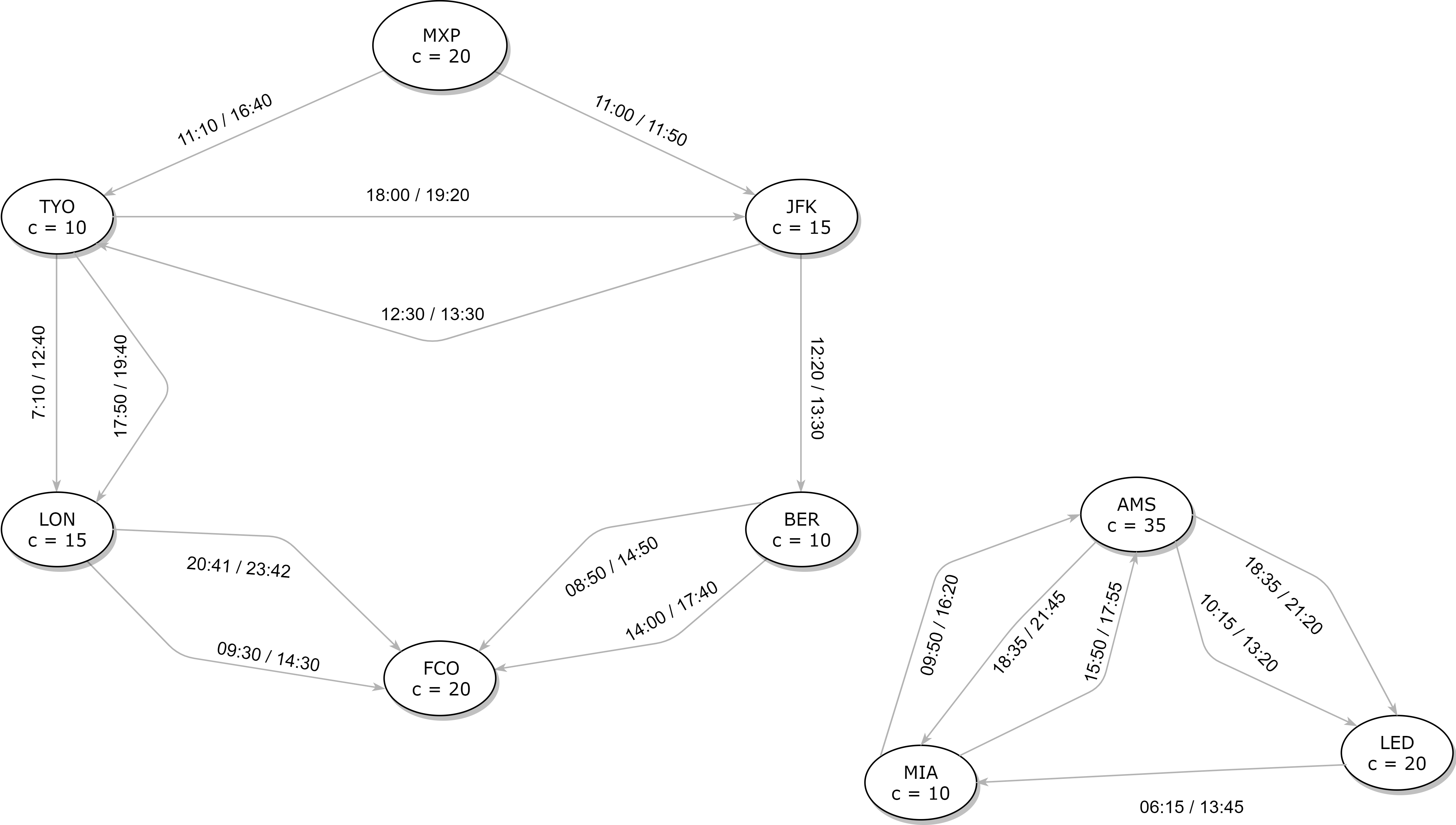
### Metodi di supporto

Oltre ai metodi di supporto enunciati nella traccia, è stato creato un ulteriore metodo per la lettura di due file di configurazione contenenti rispettivamente la lista degli aeroporti e la lista dei voli tra gli aeroporti.

L’orario della compagnia viene infatti considerato composto da due liste distinte. Questa considerazione permetterà ad alcune delle soluzioni proposte di fare un filtraggio sui voli e ridurre mediamente la complessità senza aggiungere eccessivo overhead.

### Supporto ai test

I test sui metodi e sulle soluzioni proposte sono stati condotti considerando diverse situazioni riscontrabili all’interno del seguente grafo:



## Esercizio 1

La funzione *list\_routes* consente di listare tutti i possibili voli per andare dall’aeroporto start all’aeroporto b, in modo che il tempo della rotta sia inferiore a T. Il parametro t è il tempo di partenza desiderato. Il tempo di rotta è definito come la somma delle durate dei voli, più i tempi di coincidenza sugli scali, più i tempi di attesa dei voli. Localmente, ad ogni scalo, si registra l’istante di arrivo e si calcola il peso del volo da prendere, sommando l’istante di arrivo, l’attesa del volo, il tempo di coincidenza e la durata del volo. I parametri sono dunque:

flights: lista dei voli disponibili

start: aeroporto di partenza

b: aeroporto di destinazione

t: orario di partenza richiesto

T: durata desiderabile della rotta

Valore di ritorno: la lista con le rotte che soddisfano le condizioni, o None se nessuna rotta è selezionabile.

Nella prima fase dell’algoritmo si esegue un “filtraggio” dei voli in base alla condizione che il tempo di partenza sia maggiore di t e la durata del volo sia minore o uguale a T. Il nuovo dominio viene organizzato mediante un dizionario in cui, per ogni aeroporto, si memorizza la lista dei voli uscenti che soddisfano le condizioni stabilite. Su questo nuovo dominio, si procede con la risoluzione del problema. Questo approccio consente di ridurre drasticamente i voli da controllare a seconda dei parametri t e T.

Il problema in questione è una ricerca esaustiva di tutti i possibili percorsi per andare dall’origine alla destinazione. Si approccia con una visita in profondità da far partire, ricorrentemente, sugli scali adiacenti a uno scalo di partenza corrente che, inizialmente, coincide con l’aeroporto di partenza. In caso di ciclicità, ovvero voli che portano ad un aeroporto già visitato sono coinvolti nella ricerca di ulteriori soluzioni e eventualmente aggiunti alla soluzione. Nella valutazione di un volo da aggiungere al percorso, si tiene conto del fatto che il costo globale della rotta, correntemente aggiornato ad ogni nuovo volo inserito nella costruzione del percorso, debba essere minore o uguale di T e che l’istante di partenza per il volo successivo, nonché istante di arrivo sullo scalo corrente, sommato al tempo di coincidenza deve essere al più il tempo di partenza del volo successivo che, altrimenti, verrebbe perso e non ammissibile. Quando si arriva alla destinazione, il percorso ricorsivamente costruito viene aggiunto alla soluzione. Al termine di tutte le chiamate ricorsive sugli scali adiacenti, la lista delle rotte è completa.

### Complessità computazionale

La complessità è tipica degli algoritmi di ricerca esaustiva che esplorano tutti i possibili cammini da una sorgente a una destinazione, con un ordine di grandezza esponenziale, rispetto alla dimensione del problema e dipendente dal tipo di grafo. Ad ogni passo il volo può essere preso o non preso, per un totale di m voli, nel caso peggiore. Dato che il grafo in questione si suppone avere m >> n allora la complessità dell’algoritmo è pari a:

O(2^m).

## 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 volo 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, evitando così di riempire eccessivamente la coda nel caso in cui non mi servano determinati aeroporti.
* 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.
* L’algoritmo termina immediatamente una volta trovata la soluzione di interesse.

### Complessità computazionale

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

O(m\*log(n))

## Esercizio 3

La funzione *select\_flights* 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 esso per finanziare tutti i voli in partenza e appartenenti alla soluzione.

Il problema si può approcciare attraverso la tecnica della programmazione dinamica e, in particolare, al problema dello zaino 0-1. 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 i,j rappresenta il costo totale raggiunto considerando l’elemento i e il 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 aeroporto, 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(m). Per cui la complessità della select\_flight è

O(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.

Il metodo deve essere valido anche per 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 certezza della bipartibilità di esso.

### Complessità computazionale

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

O(n+m)