SCRIVERE COME ABBIAMO SRUTTURATO I DATI, CLASSI AIRPORT E FLIGHT E COME E’ FATTO IL “GRAFO”

**Esercizio 1**

Il problema affrontato è di tipo enumerativo per cui è stata scelto come soluzione un algoritmo brute-force corredato da una funzione di *prune* per ridurre lo spazio delle soluzioni esplorate.

***list\_routes(schedule: Dict[Airport, list], source: Airport, dest: Airport, t: timedelta, T: timedelta)*** : prende in ingresso l’orario della compagnia, nella forma di lista degli aeroporti e di dizionario che lega ad ogni aeroporto l’insieme di voli partenti da tale scalo, gli aeroporti sorgente e destinazione, e il tempo massimo di volo. Restituisce tutte le rotte (insieme di voli) che conducono da *source* a *dest* in meno di *T*.

Allo scopo la funzione effettua una visita di tutti i possibili percorsi che partono da *source* alla ricerca di quelli che conducono a *dest* visitando ricorsivamente la struttura dati. La visita viene effettuata richiamando la funzione ausiliaria

***recursive\_visit(schedule: Dict, source: Airport, sink: Airport, arrival\_time: timedelta, T: timedelta, solution: List, paths: List)*** : dato l’orario della compagnia, gli aeroporti sorgente e destinazione, il tempo di arrivo e la durata massima di viaggio, costruisce una lista (*solution*) contenente il costo parziale della tratta e i voli presi. Ricorsivamente chiama sé stessa aggiornando la soluzione che ad ogni passo costruisce ed eventualmente la accetta, salvandola nelle rotte possibili (*paths*) se si giunge a destinazione rispettando i vincoli. Le chiamate ricorsive sono condizionate dalla funzione di *prune* che scarta le soluzioni parziali che non rispettano i vincoli.

***backtracking\_prune(arrival: timedelta, departure: timedelta, coincidence: timedelta, time\_spent: timedelta, total: timedelta) -> bool*** : date le informazioni sui tempi correnti e sulla durata parziale del viaggio verifica se un volo è ammissibile (è possibile prenderlo in tempo dato l’orario di arrivo e la coincidenza dell’aeroporto) e se la durata parziale di viaggio non supera la massima consentita.

SCRIVERE SULLA COMPLESSITA’

**Esercizio 2**

La richiesta del problema si può ricondurre alla più ampia categoria del problema *Single-source shortest path*. Per cui la soluzione proposta è una variante dell’algoritmo di Dijkstra che tiene conto delle specifiche richieste della traccia.

***find\_route(schedule: Dict, airports: List, start: Airport, dest: Airport, t: timedelta) -> List*** : prende in ingresso l’orario della compagnia, nella forma di lista degli aeroporti e di dizionario che lega ad ogni aeroporto l’insieme di voli partenti da tale scalo, gli aeroporti sorgente e destinazione e l’orario di partenza e restituisce la rotta più breve per arrivare da *start* a *dest*. Nella procedura di rilassamento dell’arco, nel caso l’arco è rappresentato dal volo che connette due aeroporti, la funzione verifica che il volo sia ammissibile (è possibile prenderlo in tempo dato l’orario di arrivo e la coincidenza dell’aeroporto).

L’implementazione dell’algoritmo prevede una *AdaptableHeapPriorityQueue*, per cui la complessità computazionale sarà pari a O((m+n)logn), in cui n sono i nodi del grafo e m sono gli archi. Per la natura del problema si suppone che m >> n, per cui una buona approssimazione è O(mlogn). La funzione termina non appena viene aggiunta la destinazione all’insieme dei nodi nel *cloud* dell’algoritmo, sfruttando la proprietà dell’algoritmo che assicura che il *path* più breve per un certo nodo è stato trovato quando quest’ultimo è stato inserito in *cloud*. Questa scelta permette di ridurre la complessità della funzione, anche se non nel worst case.

**Esercizio 3**

NON SACCIO NIENTE

**Esercizio 4**

Un grafo è bipartito se è 2-colorabile, ovvero se è possibile dare un colore a ogni suo nodo senza che due nodi adiacenti abbiano lo stesso colore. Questa proprietà può essere verificata visitando la struttura dati e definendo in maniera alternata il colore dei nodi e valutando nel processo se si incontrano due nodi connessi da un arco con stesso colore.

***bipartite(G: Graph)*** : dato il grafo di partenza restituisce le sue partizioni se è bipartito, None altrimenti. Questa funzione implementa una visita DFS completa, per tener conto di possibili grafi in input non connessi.

Poiché il problema è risolto con visita DFS completa, *bipartite* utilizza una ulteriore funzione per la visita:

***color\_dfs(g, u, discovered, color)*** : dati il grafo, un nodo di partenza, l’insieme dei nodi visitati, e un dizionario che mappa ogni nodo al suo colore, verifica se la componente del grafo visitata da *u* è bipartita restituendo la colorazione dei nodi in tal caso, o None altrimenti.

La complessità della soluzione equivale a quella di una visita DFS completa, cioè O(m+n) dove m sono gli archi del grafo ed n i nodi, dato che l’aggiunta delle colorazioni dei nodi e il loro controllo sono operazioni a tempo costante.