**Introduzione**

L’aeroporto è stato rappresentato con una classe Airport, che conserva il nome dell’aeroporto come *str* e il tempo minimo di coincidenza come *datetime.timedelta.* Il volo è stato definito come una classe Flight che conserva l’aeroporto di partenza come *Airport*, l’aeroporto di destinazione come *Airport*, l’orario di partenza come *datetime.timedelta*, l’orario di arrivo come *datetime.timedelta*, il numero di posti disponibili sul volo come *int*. Entrambe le classi implementano i magic method necessari alle operazioni utilizzate su di esse nei tre esercizi, cioè l’operatore di comparazione “==” e la hash function per le operazioni su collezioni hashed (cioè dict).

Per rappresentare l’orario della compagnia aerea, lo schedule, si è scelto di utilizzare un dict built-in di python con chiave l’aeroporto e valore la lista di voli che partono da tale aeroporto, in quanto le operazioni utilizzate su questa struttura dati hanno complessità temporale O(1), ossia dalla documentazione ufficiale:

|  |  |  |
| --- | --- | --- |
| **dict** | | |
| **Operation** | **Average Case** | **Amortized Worst Case** |
| Get Item | O(1) | O(n) |
| Set Item | O(1) | O(n) |
| Delete Item | O(1) | O(n) |
| Iteration | O(n) | O(n) |

Per rappresentare la lista di voli si è scelto di usare il tipo *list* (e quindi un array) o *deque* (e quindi una doubly linked list) built-in di python a seconda che fosse necessario ottimizzare la complessità spaziale in vista di operazioni (efficienti) di sola lettura, nel primo caso, oppure ottimizzare le operazioni di inserimento e rimozione agli estremi della collezione (inserimento in particolare), e quindi *deque*. Si riporta la documentazione ufficiale:

|  |  |  |
| --- | --- | --- |
| **list** | | |
| **Operation** | **Average Case** | [**Amortized Worst Case**](http://en.wikipedia.org/wiki/Amortized_analysis) |
| Append | O(1) | O(1) |
| Insert | O(n) | O(n) |
| Get Item | O(1) | O(1) |
| Set Item | O(1) | O(1) |
| Delete Item | O(n) | O(n) |

|  |  |  |
| --- | --- | --- |
| **deque** | | |
| **Operation** | **Average Case** | **Amortized Worst Case** |
| append | O(1) | O(1) |
| appendleft | O(1) | O(1) |
| pop | O(1) | O(1) |
| popleft | O(1) | O(1) |
| remove | O(n) | O(n) |

Per l’algoritmo di Dijkstra modificato dell’esercizio 2 si utilizza una AdaptableHeapPriorityQueue della libreria TdP\_collections per mantenere gli elementi (cioè gli aeroporti) da valutare in base al costo, opportunamente inizializzato e poi ricalcolato, che viene utilizzato come chiave.

Inoltre per utilizzare coerentemente con le specifiche le strutture dati descritte precedentemente, sono stati forniti dei metody di utility nel file utils.py, che sono:

|  |  |  |
| --- | --- | --- |
| **utils** | | |
| **Operation** | **Average Case** | **Description** |
| c(a: Airport) | O(1) | Restituisce il tempo minimo di coincidenza per un aeroporto |
| s(f: Flight) | O(1) | Restituisce l'aeroporto di partenza di un volo |
| d(f: Flight) | O(1) | Restituisce l'aeroporto di arrivo di un volo |
| l(f: Flight) | O(1) | Restituisce l'orario di partenza di un volo |
| a(f: Flight) | O(1) | Restituisce l'orario di arrivo di un volo |
| p(f: Flight) | O(1) | Restituisce il numero di posti di un volo |
| initialize\_schedule (airports\_file, flights\_file) | O(n) | Implementa la lettura da file dei dati riguardanti aeroporti e voli e restituisce le strutture dati inizializzate che servono per il testing dei tre esercizi |

Il quarto esercizio prescinde dalle strutture descritte finora e utilizza la struttura *Graph* della libreria TdP\_collections. Si propone, in aggiunta al test di unità della funzione implementata, anche un test grafico che mostra l’esattezza (o meno) dei risultati della bipartizione. Questo test ha come prerequisito l’installazione della libreria networkx e matplotlib, ma sono stati inserite le immagini risultanti dall’esecuzione di tale test nel package pkg\_4.

**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 *pruning* 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 come dizionario con chiave un aeroporto e valore la lista di voli che partono da tale scalo, gli aeroporti sorgente e destinazione, l’orario di partenza e il tempo massimo di volo. Restituisce tutte le rotte (insieme di voli) che conducono da *source* a *dest* con un durata complessiva del viaggio non superiore a *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()*.

***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 se stessa, aggiornando la soluzione che ad ogni passo aggiorna ed, eventualmente, accetta, salvandola tra le rotte possibili (*paths*) se si è giunti a destinazione. Le chiamate ricorsive sono condizionate dalla funzione di *pruning*,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.

La complessità computazionale del problema non è esprimibile in via polinomiale, in quanto si tratta di una quantità esponenziale; ad esempio, nel caso di struttura dati che simula un grafo completo con n nodi, la complessità è dell’ordine di “n!”.

**Esercizio 2**

Il problema si può ricondurre alla categoria del *Single-source shortest path*, per cui la soluzione proposta è una variante dell’algoritmo di Dijkstra che tiene conto delle specifiche richieste.

***find\_route(schedule: Dict, start: Airport, dest: Airport, t: timedelta) -> List*** : prende in ingresso l’orario della compagnia, come dizionario, con chiave l’aeroporto, che lega ad ogni aeroporto l’insieme di voli che partono da tale scalo, rappresentato come lista, 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 dell’algoritmo di Dijkstra (nel caso in questione, è ragionevole considerare l’arco come il volo che connette due aeroporti) la funzione verifica che il volo sia ammissibile (ossia 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 sarebbero i “nodi” del grafo (nel nostro caso, n è il numero di aeroporti) e m sarebbero gli archi (nel nostro caso, il numero di voli). Per la natura del problema, si suppone che m >> n, per cui una buona approssimazione è O(mlogn). La funzione termina non appena la destinazione è aggiunta all’insieme dei “nodi” nel *cloud* dell’algoritmo, sfruttando la proprietà degli algoritmi greedy per cui il *path* più breve per raggiungere un “nodo” è stato trovato quando quest’ultimo è stato inserito in *cloud* e non ne esiste uno migliore. Questa scelta permette di ridurre la complessità della funzione, fatta eccezione per il worst case.

**Esercizio 3**

Il problema affrontato è di ottimizzazione (in particolare della famiglia dei problemi dello zaino), per cui la soluzione proposta prevede un algoritmo di programmazione dinamica.

***select\_flight(airports, flights, B: int)*** : dato l’orario della compagnia, in forma di dizionario come negli altri esercizi, la lista di aeroporti e il budget massimo disponibile, calcola i voli da far partire per massimizzare il numero di passeggeri in volo e calcolare l’esatta quantità di denaro assegnata ad ogni aeroporto. Utilizza come struttura di memorizzazione una matrice n x B, dove n è il numero di aeroporti, compilata con i costi aggiornati a ogni iterazione secondo l’equazione di ricorrenza seguente:

* Sia T[][] la matrice n x B, allora vale che, detti k il costo del volo e p i suoi posti

Al termine della procedura di compilazione viene ricostruita la quantità di denaro da assegnare agli aeroporti ripercorrendo la struttura dati e applicando in maniera inversa l’equazione precedente. Per il calcolo del costo del volo si utilizza la funzione ausiliaria

***get\_cost(time\_departeur, time\_arrive)*** : dati orario di partenza e di arrivo del volo ne calcola il costo in termini di tempo di viaggio.

La complessità computazionale della funzione è pari a O(nB), cioè il costo di compilazione della matrice T[][]. Il costo per ricostruire i costi è di ordine inferiore, O(max{n, B}), e quindi assorbito nella quantità precedente.

**Esercizio 4**

Un grafo è bipartito se è 2-colorabile, cioè 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, valutando, nel corso della visita, se si incontrano due nodi connessi da un arco che hanno lo stesso colore.

***bipartite(G: Graph)*** : dato il grafo di ingresso G, restituisce le sue (due) partizioni se è bipartito, None altrimenti. Questa funzione implementa una visita DFS completa, per tener conto di possibili grafi 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 a partire da *u* è bipartita restituendo la colorazione dei nodi in tal caso, 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.