Relazione

Nome: Domenico Cognome: Citera

Possiamo risolvere il problema proposto utilizzando la teoria dei grafi diretti.

Le stazioni ferroviarie saranno i nodi del grafo, ed ogni tratta effettuata da un treno che collega una stazione a quella immediatamente successiva sarà l'arco diretto che collega rispettivamente i due nodi.

Per individuare gli archi del grafo è necessario quindi individuare una lista di fermate per ogni treno ed ordinarla temporalmente in modo da ottenere la sequenza di fermate effettuate dal treno nelle varie stazioni.

Essendo i treni sparsi nel file di input, per costruire queste liste è stata utilizzata una *mappa (numero treno, lista fermate).* In seguito le liste sono state ordinate temporalmente in ordine di partenza del treno (arrivo nel codice), in base alla prima partenza.

Ultimato questo lavoro è stato possibile individuare gli archi da inserire nel grafo insieme al loro rispettivo peso. Il peso dato all'arco al momento della creazione riguarda il tempo che impiega il treno dalla partenza nella stazione *S* all'arrivo nella stazione immediatamente successiva*,* senza tener conto del tempo speso in stazione (questo verrà calcolato durante l'esecuzione dell'algoritmo di *Dijkstra*).

Il grafo risultante, probabilmente avrà cicli, ma sicuramente non avrà pesi negativi. Quindi per soddisfare le richieste *MINORARIO* e *MINTEMPO* è stato utilizzato l'algoritmo di *Dijkstra*, utilizzato appunto per la ricerca dei cammini minimi in un grafo con o senza ordinamento, ciclico e con archi aventi pesi non negativi. Essendo il tempo il nostro peso infatti, un peso minore indicherà un tempo di percorrenza minore.

STRUTTURA DATI ED EFFICIENZA

Considerando una rete ferroviaria di 5000 treni con una media di 200 fermate per treno (per esagerare), il grafo risulta essere sparso (poiché $|E|<<|V|^2$). Quindi, con un costo in memoria di $O(|V|+|E|)$ le liste di adiacenza risultano essere il modo migliore per la rappresentazione del grafo.

Il grafo è stato implementato tramite una mappa *(numero stazione, vertice),* ed ogni vertice contiene una lista dei suoi adiacenti.

La classe *Edge* rappresenta l'arco, e contiene informazioni quali

- Il vertice di destinazione,
- Il numero del treno che percorre la tratta,
- Il tempo di percorrenza impiegato,
- L'orario di partenza da S,
- L'orario di arrivo nella stazione successiva.

La classe *Vertex* rappresenta il vertice, e contiene informazioni quali

- Il nome della stazione
- La lista dei suoi adiacenti
- Il puntatore al vertice padre
- Il puntatore all'arco che ha permesso di raggiungere quel vertice
- La distanza minima da un ipotetica sorgente
- Ora di arrivo nella stazione rappresentata.

Le ultime quattro informazioni saranno variabili nel corso dell'esecuzione delle richieste elaborate tramite *Dijkstra*.

La classe *Graph*, oltre a fornire i metodi per la creazione e la gestione del grafo si occupa di fornire metodi per il calcolo del peso di ogni arco, ed implementa l'algoritmo di *Dijkstra*.

Questo prende in input oltre al vertice di partenza un intero *start* che indica l'orario di arrivo nella stazione di partenza, ed in base a questo calcola il costo totale degli archi. Infatti, durante l'esecuzione, oltre a sommare i pesi degli archi precedenti, aggiunge il

peso relativo al tempo speso dall'arrivo nel vertice alla partenza indicata sull'arco preso in considerazione, calcolando così i cammini minimi.

Per quanto riguarda la complessità, avendo usato un min-heap per la gestione delle distanze, ogni arco verrà esaminato una sola volta, e quando è esaminato il costo sarà al più *O(logn).* Quindi il costo del *WHILE* risulta *O(nlogn+mlogn),* allora la complessità dell'implementazione di Dijkstra è *O((n+m)logn).*

Il metodo per risalire al cammino di costo minimo, si occupa poi di restituire un itinerario contenente tutte le informazioni relative al tragitto trovato.

La classe *Itinerary* infatti, contiene informazioni quali

- La distanza (espressa in minuti) dalla stazione S1 alla stazione S2
- Il numero delle tratte totali del viaggio
- La sequenza delle tratte relative al percorso individuato.

La classe *Treno* contiene informazioni come il numero del treno, il numero di riferimento della stazione, l'orario di arrivo in quella stazione, e l'orario di partenza da quella stazione. Questi oggetti creati al momento della scansione del file di input ci saranno utili per facilitare la creazione del grafo.

La classe *Utility* contiene dei metodi di supporto, come *mapOrder* e *stopsOrder*, necessari per ordinare temporalmente le informazioni relative alle tratte dei treni raccolte dal file di input. Verranno esaminati per ogni treno una lista contenente le sue fermate. L'ordinamento per ogni lista richiede *O(mlogm)*, quindi la complessità è *O(nm logm)*, dove *n* è il numero delle liste e m il numero delle fermate della lista.

Utility contiene inoltre, il metodo che si occupa di gestire le richieste e creare un itinerario di viaggio. Questo, in base alla richiesta ricevuta in input, esegue le operazioni richiamando l'algoritmo necessario, e ritorna l'itinerario del percorso richiesto. Per quanto riguarda la complessità notiamo che per eseguire una richiesta *MINTEMPO*,

l’algoritmo di *Dijkstra* viene ripetuto per ogni vertice adiacente al vertice di origine, quindi la complessità è *O(n(n+m)logn).*

WriteFile è la classe di appoggio che ci consente di scrivere sul file. Questa contiene un metodo che prende in input un risultato e il nome di un file, e si occupa di trascrive i risultati dell’itinerario sul file indicato.