Università degli Studi di Milano
Corso di Laurea Triennale in Matematica (classe L-35)

Esame di Algoritmi (9 CFU) - Sessione di Giu 2019

Relazione del progetto “TAXI”

Studente: **Fabio Caironi**

Data di consegna: 13-06-19

1. INTRODUZIONE

Il programma si compone di 11 files: il file principale, *sim_taxi.c*, contenente il main e diverse altre funzioni per le stampe ordinate e la simulazione (vedi ¶ 4); le seguenti librerie:

- *listaarchiFS.h* *listaarchiFS.c* — È la libreria dove sono definite la lista di archi e le funzioni che svolgono operazioni su liste di archi, utili per l'implementazione di grafi Forward Star con lati pesati. Conoscendo sempre il nodo di origine dal quale diparte un arco, grazie alle liste FS, è necessario per il record arco contenere soltanto il nodo di destinazione e il peso dell'arco, oltre che ovviamente i puntatori agli elementi adiacenti della lista.

- *grafoFS.h* *grafoFS.c* — È la libreria che contiene il record *grafo* implementato tramite le liste Forward Star e le funzioni di lettura e scrittura di un grafo. Alcune di queste funzioni (come *canc_arco*) sono riportate solo per completezza della libreria, ma non saranno usate nel nostro programma.

La scelta di implementare il grafo con le liste di adiacenza è funzionale all'algoritmo di Dijkstra, utilizzato per il calcolo dei cammini minimi, in quanto ottimizza la scansione degli archi uscenti da un nodo dato, rispetto alle alternative "lista di archi" (più costosa in termini di tempo) e "matrice di adiacenza" (più costosa in termini di spazio). Detto ciò, di queste prime librerie non si parlerà ulteriormente per ragioni di economia di spazio e perché già ampiamente trattate in classe.

- *heapdinE.h* *heapdinE.c* — In questa libreria sono definite le strutture e le funzioni atte a conservare gli eventi e a gestirne l'ordine. Vedi ¶ **2**.

- *dijkstra.h* *dijkstra.c* — È la libreria dove è eseguito il calcolo dei cammini minimi tra coppie di nodi del grafo, per mezzo dell'algoritmo di Dijkstra e di altri algoritmi di mia ideazione. Vedi ¶ **3**.

- *zaino.h* *zaino.c* — Qui sono definite le funzioni per la soluzione del problema dello zaino con la tecnica euristica Greedy, utili a rispondere all'ultima richiesta del progetto.

2. GESTIONE DEGLI EVENTI

Un evento deve essere in qualche modo rappresentato come un'unità sintattica che fornisca le informazioni sull'ora in cui si è verificato (un intero da 1 a *T*), su che tipo di evento (*CHIAMATA, FINE_SERVIZIO, RIENTRO_SEDE, FINE_RICARICA)* è e su quale taxi coinvolge (un intero da 1 a *nv*). Per far ciò si è definito un record *evento* contenente, innanzi tutto, i sopraccitati campi. Per quanto riguarda il *tipo*, esso è stato definito come short e si intende che possa assumere i valori interi da 1 a 4, corrispondenti ai rispettivi tipi di evento indicati nelle direttive *#define* all'inizio del file *heapdinE.h*. Questo modo di distinguere gli eventi torna utile nel confronto di eventi, implementato dalla funzione *ConfrontaEventi*, dove la gerarchia degli ordinamenti prevede che, ad ugual tempo, prevalga l'evento con tipo "minore".
Se l'evento è di tipo CHIAMATA, è necessario immagazzinare nell'evento o rendere disponibili tramite l'evento stesso le informazioni della chiamata. Per questo motivo è stato definito un record

chiamata, con tutti i campi utili, ed è stato aggiunto al record evento un campo **chiamata*. La presenza all'interno di un "record 1" di puntatori ad altri "record 2,3,ecc." rende necessaria l'allocazione esplicita della memoria per contenere i "record 2,3,ecc." ogni volta che si alloca un "record 1". Da qui la distinzione tra le funzioni *CreaHeapDinE e CreaChiamata*.

Passiamo alla gestione dell'ordine degli eventi. Dato che, in ogni dato istante, bisogna conoscere solo il primo evento che si verificherà in ordine cronologico (o, se se ne verifica più di uno simultaneamente, il primo secondo i criteri di ordine forniti), non è necessario ordinare tutti gli eventi, ma basta conoscerne sempre e solo il primo. Ciò detto, sarebbe vantaggioso tenere gli eventi in uno heap, aggiornato ad ogni aggiunta o cancellazione, in modo tale da poter ricavare il primo evento con la funzione *TestaHeapDinE*. Tra questa idea e la soluzione c'è solo un ostacolo: non si conosce a priori il numero di eventi memorizzati nello heap, cioè gli eventi che devono ancora essere processati, dato che ogni evento genera un numero variabile di altri eventi. Fortunatamente, nel nostro caso il numero di eventi generati da un evento è inferiore o uguale ad 1: la lunghezza massima dello heap non può dunque superare il numero delle chiamate iniziali e inoltre non serve una funzione ' *AggiungeHeapDinE* ' perché le uniche operazioni necessarie sono la sostituzione e la cancellazione dell'elemento in testa allo heap.
Perciò, è stato definito il record *heapdine* (Heap Dinamico di Eventi) contenente un vettore *V* di eventi in cui lo heap sarà mantenuto, da *V[1]* fino a *V[last]*, per una lunghezza massima di *size*. L'operazione di sostituzione, svolta da *SostituisceHeapDinE*, sovrascrive un evento in testa allo heap e in seguito aggiorna lo heap con *AggiornaHeapDinE*, che opera ricorsivamente sui due sottoalberi fino alla foglia con indice minore o uguale a *last*. Per la cancellazione, invece, si usa *CancellaHeapDinE,* che riduce la lunghezza *last* dello heap, dopo aver posizionato in fondo l'evento in testa da cancellare, e aggiorna. Infine, per creare uno heap dal generico vettore di eventi, si usa la funzione *CreaHeapDinE,* implementata come la classica procedura di creazione heap, ma con funzione di aggiornamento *AggiornaHeapDinE*.

3. Calcolo dei cammini minimi

Il problema del calcolo dei cammini minimi tra tutte le coppie di nodi della rete stradale è quello a cui ho dedicato più tempo e idee, tentando di migliorare il tempo di calcolo rispetto ad un'implementazione standard come l'algoritmo di Floyd-Warshall o l'iterazione su tutti i nodi del grafo dell'algoritmo di Dijkstra. Tali implementazioni, infatti, richiederebbero tempi, rispettivamente, nell'ordine di $\Theta(n^3)$ e $\Theta(mn \log n)$, che molto facilmente dominano la complessità temporale del progetto (quando il numero di chiamate non è troppo elevato, cfr. **¶ 5**).

Per il posizionamento dei taxi è necessario conoscere le distanze minime tra ogni coppia di nodi, quindi si può escludere il calcolare ciascuna distanza minima di volta in volta: conviene memorizzarle tutte in una matrice *C* che sarà disponibile per tutto il progetto. Tale matrice, poiché il grafo è non orientato, sarà ovviamente simmetrica: per risparmiare spazio la si memorizza in una matrice triangolare inferiore, allocata dinamicamente, da cui poter leggere o scrivere solo elementi $C[i][j]$ con $i \geq j$ (v. *AllocaMTI, LeggeMTI, ScriveMTI* in *dijikstra.c*). Oltre alla matrice *C* è necessario mantenere la matrice dei predecessori *P*, la quale però non è simmetrica. Le due matrici si leggono nel seguente modo: l'elemento $C[i][s]$ è il costo del cammino minimo dal nodo sorgente s al nodo i e l'elemento $P[i][s]$ è il predecessore di i nello stesso cammino.
Dato che, in modo plausibile, in una rete stradale il numero di strade m è $\ll \frac{n(n+1)}{2}$ (in un grafo a maglia quadrata, per esempio, vale $m = 2n - 2\sqrt{n} = \Theta(n)$ e la rete stradale delle grandi

città presenta solitamente una conformazione del genere, v. **Fig 1**), si è scelto l'algoritmo di Dijkstra per la soluzione dei cammini minimi.

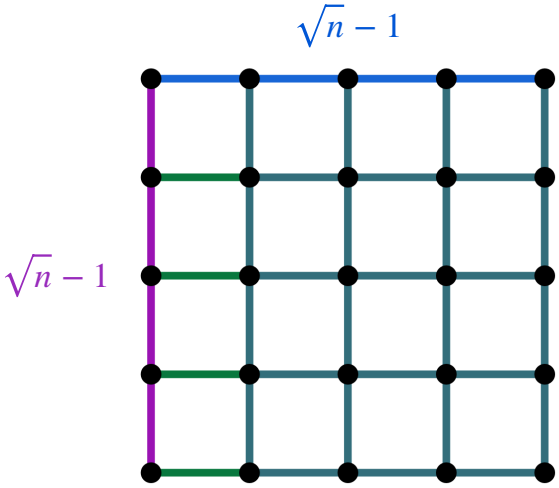
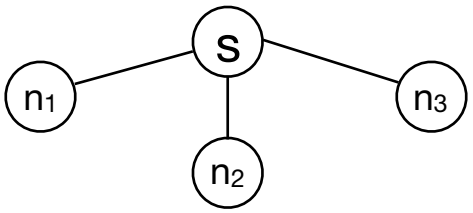


Fig. 1 - Grafo a maglia quadrata

Accanto all'algoritmo di Dijkstra, ho ideato un algoritmo che riempie o 'rinfoltisce', per quanto possibile, le matrici C e P. Tale algoritmo è stato implementato nella funzione denominata *RiempieMatrici*, per l'appunto. L'idea è la seguente: dato l'albero dei cammini minimi da un nodo sorgente s a tutti gli altri nodi del grafo — albero che è ottenuto con la procedura *Dijkstra* invocata sul nodo s ed è conservato nella colonna s-esima dei predecessori, $P[][s]$ — è noto anche ogni cammino intermedio tra la sorgente e ciascuna foglia. Tali cammini intermedi sono in realtà cammini minimi tra coppie di nodi, come dimostra la correttezza dell'algoritmo di Dijkstra (l'albero dei cammini minimi rappresenta proprio questo concetto e il non orientamento del grafo permette di considerare le coppie di nodi in entrambi gli ordini). È possibile allora determinarli, e cioè calcolarne il costo e ciascun predecessore in ambo le direzioni, a partire solo dalla conoscenza delle colonne s-esime delle matrici C e P. Questo lavoro è svolto dalla funzione *CamminiRicorsiva*, ripetuta per ogni foglia dell'albero dei cammini minimi da s, trovata da *TrovaFoglie*.
È ragionevole abbinare questo algoritmo all'algoritmo di Dijkstra: più precisamente, dopo ogni esecuzione di Dijkstra sul nodo sorgente s si chiama la funzione *RiempieMatrici* sull'omonimo nodo. Bisogna osservare che tale procedura non garantisce, in generale, il riempimento totale delle matrici, ma garantisce il riempimento delle **righe *s*-esime**. Nel caso peggiore, quando s può raggiungere tutti gli altri nodi tramite archi diretti a costo minimo, l'albero dei cammini di costo minimo ha la forma:

Esempio 1:



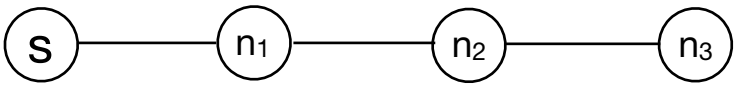
pertanto la procedura *RiempieMatrici* individua i seguenti cammini minimi:

- n_1 -> s
- n_2 -> s
- n_3 -> s

e con ciò è determinata solo la riga s-esima delle matrici C (costi minimi per andare dalla sorgente n_i al nodo s, per i = 1,2,3) e P (predecessori di s in un cammino minimo che parte dalla sorgente n_i,

i = 1,2,3). In tutti gli altri casi è comunque noto il cammino per andare da un nodo *n* alla sorgente *s*, poiché è il cammino inverso per andare da *s* ad *n;* dunque, ancora, è scrivibile almeno la riga *s*-esima delle matrici *C* e *P*.
Nel caso migliore si possono invece riempire tutte le matrici *C* e *P* in una sola esecuzione congiunta di *Dijkstra* e *RiempieMatrici*! Se, infatti, dopo l'esecuzione di *Dijkstra* su un nodo sorgente *s* l'albero dei cammini minimi ha la forma di un cammino:

Esempio 2:



sono determinabili da *RiempieMatrici* i cammini minimi tra ogni coppia di nodi del grafo come sottocammini di quello sopra. Ciò semplifica notevolmente i costi computazionali (v. ¶ **5**).

Affinché l'algoritmo di *Dijkstra* continui a funzionare correttamente per ogni iterazione su un nuovo nodo $s' \neq s$, è necessario che le colonne *s*-esime delle matrici *C* e *P* siano inizializzare ai valori base (*INFINITY* e 0). Ciò significa, essenzialmente, che non si può sfruttare l'informazione fornita da *RiempieMatrici,* invocata su nodi sorgente precedenti, per agevolare l'esecuzione di *Dijkstra* su nuovi nodi, ma solo per rinfoltire le matrici sperando che, dopo un numero $i < n$ (n = numero di nodi del grafo) di esecuzioni in coppia di *Dijkstra* e *RiempieMatrici* risultino riempite più di i colonne delle matrici *C* e *P*, cosicché per riempirle tutte non servano effettivamente *n* iterazioni. Si sono visti prima i casi peggiore e migliore per questo tipo di algoritmo. Il caso medio è indeterminato, ma giace certamente tra i due, quanto a complessità temporale. Per forzare la randomizzazione del processo, si è deciso di invocare *CalcolaCamminiMinimi* (la funzione che raccoglie i due algoritmi in coppia appena visti) ogniqualvolta si tenti di accedere ad un elemento delle matrici *C* o *P* che non è ancora stato inizializzato. A dire il vero, non si tenterà mai di accedere in lettura a *P* se non su una colonna che si conosce già: perciò quanto detto si applica alla sola matrice *C*, tramite la funzione *LeggeCompletataC*.

Entriamo più nel dettaglio con l'algoritmo di *Dijkstra.* Per poter ottenere una complessità temporale di $\Theta(m \log n)$, è stato necessario utilizzare uno heap per rappresentare l'insieme D dei nodi raggiungibili da nodi di cui è stata calcolata la soluzione. Similmente allo heap dinamico di eventi (v. ¶ **2**), lo heap dei nodi in D ha lunghezza variabile da 1 a *size* (= *n* qui), per cui ho utilizzato la stessa strategia per mantenerlo. In più, questo heap non è aggiornato grazie ad informazioni contenute nei nodi stessi, ma tramite la matrice *C* dei costi dei cammini minimi, da cui la necessità di rendere lo heap indiretto (Heap Indiretto di Cammini).
Inizialmente, tutti i nodi stanno nello heap ed hanno valore associato (in *C*) pari a *INFINITY*: un nodo entra, concettualmente, in D quando il suo valore passa da *INFINITY* ad un valore "finito": praticamente, però, quel nodo si trova già nello heap, ma sicuramente sarà collocato in modo ordinato, alla radice di un sottoalbero con tutti elementi pari a *INFINITY*.

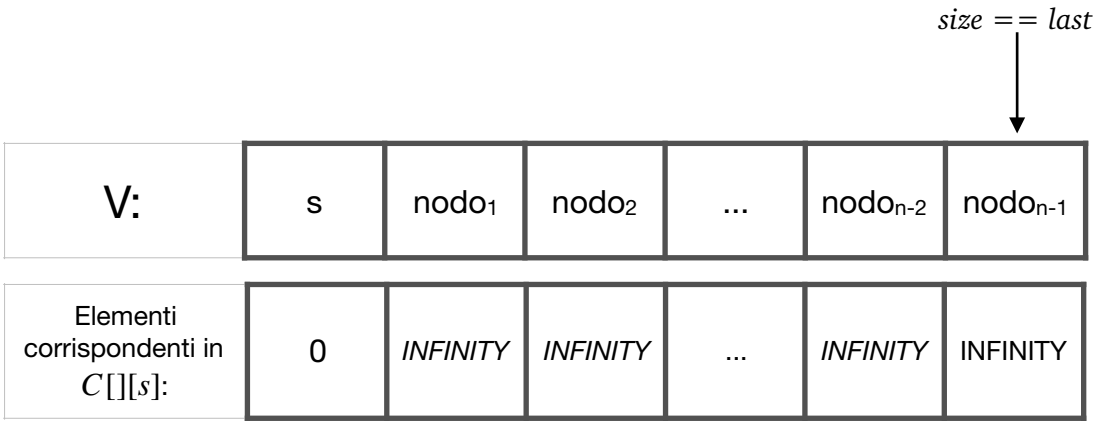


Fig. 2: Situazione di uno *heapdic* all'inizio della procedura *Dijkstra*

A differenza dello heap dinamico di eventi, inoltre, in questa struttura è necessario eseguire, oltre all'operazione di aggiornamento in direzione delle foglie, utile per la procedura di cancellazione in testa (*CancellaHeapDIC*), anche l'operazione di aggiornamento in direzione della radice, o "*AggiornaSopraHeapDIC*". Infatti, quando ad un nodo j di costo $C[j][s]$ corrente pari a c_0 viene sostituito un costo $c_1 < c_0$, allora j deve **salire** lungo l'albero con un algoritmo ricorsivo analogo, ma basato su confronti con il padre.

Si può notare che, accanto allo *heapdic,* nell'algoritmo di *Dijkstra* è mantenuto un vettore di indici *Index*. Questo vettore ha il significato e lo scopo seguenti: l'elemento *Index[j]* è l'indice del nodo j nel vettore V dello *heapdic* **pH* e viene usato per risalire da un nodo al rispettivo indice in V tutte le volte che la scansione della lista di adiacenza produce un nodo di destinazione. L'alternativa sarebbe stata utilizzare una funzione *TrovaIndiceNodo,* che avevo precedentemente scritto e usato e quindi l'ho lasciata, commentata, nel codice, che scandisce il vettore V restituendo l'indice, ma ha lo svantaggio di lavorare in tempo $\Theta(n)$, appesantendo complessivamente *Dijkstra*. Il metodo del vettore di indici costa invece n celle di spazio in più e uno scambio in più nelle procedure *AggiornaSopraHeapDIC*, *AggiornaSottoHeapDIC*, *CancellaHeapDIC*, il quale va solo ad incrementare di 1 la costante moltiplicativa della complessità di *Dijkstra*, senza cambiarne l'ordine di grandezza.

4. Main e simulazione

Il file *sim_taxi.c* raccoglie il main e una serie di funzioni, troppo assortite per essere ripartite in una libreria a parte. Il main si divide in una prima sezione di chiamate a funzioni (per lo più vuote, cfr. sotto), che modularizzano il codice iniziale. Vi è poi la simulazione, costituita essenzialmente da un blocco *while* seguito da un blocco *if* ed infine avviene la deallocazione di tutte le zone di memoria dinamiche usate nel progetto.
Si è scelto di dichiarare le variabili che verranno utilizzate dall'inizio alla fine dell'esecuzione come variabili globali, ovvero fuori dal main: ciò ha innanzi tutto il senso di dare una gerarchia alle variabili del progetto, ma serve anche a contenere la lunghezza della segnatura di molte funzioni, alle quali, se definite nel main, bisognerebbe altrimenti passare in argomento le sopraccitate variabili come parametri appesantendo anche il programma con eccessive copie di indirizzo. Al contrario, le segnature di svariate funzioni definite nel main appaiono con questo accorgimento più snelle e per questo potrebbero sembrare incomplete, ma non lo sono.
Le variabili globali sono il grafo R della rete stradale, lo heap dinamico di eventi H, le matrici C e P dei cammini minimi, i dati sul servizio come l'orizzonte temporale e il numero di veicoli, le informazioni istantanee dei taxi come la batteria e la posizione, le variabili per la gestione della coda di ricarica e i contatori globali. Per quanto riguarda le informazioni istantanee sui taxi, vale a dire quelle variabili che in ogni momento del progetto e della simulazione devono indicare la posizione, la batteria, l'incasso dell'ultimo viaggio e la disponibilità di ciascun taxi, si è scelto di memorizzarle nei vettori *pos, batt, inc, disp*, di lunghezze *nv* (numero di veicoli) e di tipi opportuni, in modo tale che, ad esempio, *batt*[i] indichi la batteria residua del taxi i e *disp[k]* indichi la disponibilità del taxi k, in ogni istante della simulazione.

Il main si apre con il caricamento dei dati del progetto: viene caricato il grafo in R, poi vengono lette le informazioni globali sui taxi e inizializzati i vettori ad essi associati, di cui si parlava sopra, infine vengono lette e caricate le chiamate. Come si può notare nel codice, *AggiungeChiamataHeapDinE,* invocato in *CaricaChiamate* nel main, aggiunge le chiamate una dopo

l'altra nel vettore, futuro heap di eventi, senza mantenere la struttura di heap, la quale sarà utile solo a partire dalla simulazione.

La prima richiesta del progetto è svolta interamente dalla funzione *StampaClientiOrdinati.* Dato che costerebbe troppo e sarebbe inutile ordinare gli eventi direttamente sul vettore in cui sono conservati e dato che l'ordinamento HeapSort è il più veloce, si sono ordinati alfabeticamente e stampati gli eventi chiamata tramite uno heap indiretto. La funzione *AggiornaHeapIndSTR* aggiorna uno heap di indici (verso il basso), tramite le stringhe *cliente* contenute negli eventi mantenuti in *heapdine *pH,* secondo l'ordinamento lessicografico crescente e la funzione *CreaHeapIndSTR* crea inizialmente uno heap. La procedura di HeapSort è svolta da *StampaClientiOrdinati* congiuntamente con la stampa: si prende la testa dello heap, che ha un indice associato all'evento con cliente "minimo", lo si stampa, lo si scambia con l'ultimo, si aggiorna e si itera. Effettivamente, il vettore finale risulterà ordinato alfabeticamente in maniera decrescente, ma le stampe sono state effettuate invece in ordine crescente e questo è quello che importa.

Successivamente si procede alla stampa dei viaggi ordinati per durata decrescente.
Prima di tutto, si aggiungono con *AggiornaChiamateHeapDinE* le informazioni sulla durata del viaggio richiesto e il percorso da effettuare di ciascuna chiamata, ottenute grazie alle matrici *C* e *P*. Si sottointende, dal momento che se n'è già discusso nel **¶ 3**, che, ogni volta che si tenta di leggere la durata di un cammino minimo non noto da una sorgente *s,* tramite la matrice *C,* si esegue *CalcolaCamminiMinimi*, la quale completa, almeno, la colonna e la riga *s*-esime delle matrici *C* e *P* (nel modo in cui ho organizzato il codice, il primo nodo sorgente su cui essa è invocata nel progetto risulta essere il nodo di origine della corsa della prima chiamata pervenuta). C'è da fare una precisazione riguardo alla funzione *AggiornaChiamateHeapDinE:* conoscendo la colonna *nodoOrig*-esima della matrice *P,* è possibile determinare con un ciclo while tutti i nodi toccati da un cammino minimo da *nodoOrig* a *nodoDest*, ma in senso inverso, salvandoli nel vettore *nodiVis;* così le componenti di *nodiVis* per un viaggio da *nodoOrig* a *nodoDest* saranno, nell'ordine: (*nodoDest* , n_1 , n_2 , ... , *nodoOrig*) e per stamparlo al contrario è stata creata appositamente la funzione *StampaCapovoltoVINT*.
Completata la procedura *AggiornaChiamateHeapDinE*, si procede a stampare i viaggi ordinatamente, in una maniera del tutto analoga a quella della stampa dei clienti: in una funzione chiamata *StampaViaggiOrdinati* si stampa iterativamente il viaggio in testa ad uno heap indiretto, che è il viaggio "massimo" secondo l'ordinamento previsto dalla funzione *ConfrontaViaggi* ($v_1 > v_2$ se e solo se v_1 ha durata maggiore di v_2, oppure v_1 ha durata uguale a v_2 e orario di chiamata minore). Il risultato sarà proprio l'elenco stampato dei viaggi ordinati in modo decrescente.

La terza e ultima richiesta prima della simulazione è la collocazione iniziale dei taxi: per essa è bastata la breve ed economica funzione *CollocaTaxi*. Questa funzione alloca un vettore *B* di dimensione *n*, la cui componente *i*-esima sarà la somma delle distanze tra il nodo *i* (quello in cui collocare il nuovo taxi) e i nodi in cui i taxi precedenti sono stati collocati. Ogni taxi da 2 a *nv* è assegnato iterativamente, aggiornando *B* ed estraendone l'elemento massimo con la funzione *TrovaIndiceMax* di complessità temporale $\Theta(n)$. Altrettanta è la complessità di *SommaC*, per l'aggiornamento del vettore *B*. Il costo temporale complessivo di *CollocaTaxi* è, pertanto, $\Theta(n \cdot n_v)$ e, dato che il numero dei veicoli è verosimilmente inferiore ad *n*, questa grandezza non reca fastidi. Come si osservava in **¶ 3**, a questo punto del programma le matrici *C* e *P* saranno senz'altro riempite. Le posizioni dei taxi sono poi banalmente stampate con *StampaPosizTaxi*.

Prima di iniziare la simulazione, si svolge l'ultima richiesta del progetto e cioè eseguire una stima per eccesso del guadagno totale ottenibile tramite un problema di knapsack. Il motivo per cui lo si fa in questo punto del codice è che servono tutte le chiamate iniziali e, al termine della simulazione, esse saranno completamente distrutte. Si è utilizzato un algoritmo Greedy perché non è particolarmente rilevante ottenere una soluzione **esatta** del problema dello zaino, dato che per

l'impostazione del problema stesso si sono fatte ipotesi aggiuntive (il fatto che ogni corsa garantisca un premio) e valicati limiti imposti nel progetto (la durata non illimitata delle batterie, il potersi "prestare tempo" tra taxi). Con l'euristica Greedy si ottiene comunque, ad esclusione di una limitata gamma di casi patologici, una soluzione vicina all'ottimo e per questo l'ho scelta.

Inizia dunque la simulazione. Si rende il vettore degli eventi uno heap con *CreapHeapDinE*, in un tempo di $O(n_c)$ (n_c = numero di chiamate). Ogni evento che viene letto in testa allo *heapdine* può essere di quattro tipi, quindi è opportuno eseguire uno *switch-case* od una serie di *if-else* per smistare l'evento letto e processarlo. La simulazione si interromperà quando lo heap si svuoterà oppure il tempo del prossimo evento eccederà l'orario finale del servizio. Ecco come ho gestito ciascun tipo di evento:

- *CHIAMATA*. Una funzione *AssegnaTaxi* (v. sotto per maggiori dettagli) svolge contemporaneamente la ricerca di un taxi che possa servire il cliente dell'evento *e* secondo i criteri richiesti e l'aggiornamento di tutti i contatori utili. Se la funzione restituisce 0 nessun taxi è stato trovato e dunque la chiamata è rimossa dallo *heapdine* con *CancellaHeapDinE*. Altrimenti, viene costruito un nuovo evento di tipo *FINE_SERVIZIO* il cui campo *taxi* è il risultato di *AssegnaTaxi* e il cui campo *c* (chiamata) viene fatto puntare al campo chiamata dell'evento *CHIAMATA* appena processato, per poi sostituire in testa allo heap il nuovo evento e aggiornare lo heap con la funzione *SostituisceHeapDinE*.
- *FINE_SERVIZIO*. Si distinguono qui i casi "batteria residua $<$ 20%" e "batteria residua $\geq$ 20%". Nel primo caso bisogna rispedire il taxi in sede, quindi si calcola la durata del tragitto di rientro, si aggiornano le informazioni del taxi e il contatore di movimento totale. Infine si costruisce un nuovo evento di *RIENTRO_SEDE* ma, prima di sostituirlo al posto del vecchio evento *FINE_SERVIZIO*, si distrugge la chiamata contenuta in quest'ultimo: il nuovo evento, infatti, non ha un campo chiamata (o meglio, vale *NO_CALL*) e non fare ciò porterebbe ad un caso di memory leakage.
- *RIENTRO_SEDE*. Vengono aggiornate le variabili che mantengono la coda (v. sotto) e si costruisce (sempre) un nuovo evento di *FINE_RICARICA* sostituendolo in testa allo heap.
- *FINE_RICARICA*. Questo è il tipo evento che, se verificatosi, termina la catena di inneschi consecutivi: dev'essere stato generato infatti da *RIENTRO_SEDE*, il quale è generato da *FINE_SERVIZIO*, il quale a sua volta viene innescato da *CHIAMATA;* ma non genera nessun altro evento (quindi ci assicura lo svuotamento dello heap). Le uniche operazioni da fare in questo blocco *else if* sono il ripristino dell'autonomia e della disponibilità del taxi, l'incremento del contatore delle ricariche e l'aggiornamento delle variabili della coda, infine cancellare l'evento dallo heap.

La procedura di assegnamento di un taxi idoneo a servire un evento *e* di tipo chiamata è completata dalla funzione *AssegnaTaxi*, che prende in ingresso il suddetto evento ed un puntatore ad intero e restituisce il numero del veicolo assegnato, o 0 se non è stato possibile assegnare alcun veicolo. Il passaggio di un intero tramite indirizzo serve a 'restituire' un ulteriore dato: si tratta dell'orario a cui finirà la corsa, se attivata. La funzione prima di tutto controlla se gli orari di partenza e arrivo richiesti dal cliente sono compatibili con la durata della corsa: se, infatti, quest'ultima durasse più dell'arco temporale entro cui desidera viaggiare il cliente, non sarebbe ovviamente possibile e verrebbe rifiutata. Successivamente, la scansione e valutazione dei taxi avviene nel modo seguente:

A. Si crea un vettore *vtaxi* in cui vengono posizionati i soli indici dei taxi disponibili ed un vettore *tdao* che semplicemente raccoglie l'informazione dei tempi necessari a ciascun taxi ad arrivare al nodo di origine della corsa.

B. Il vettore *vtaxi* viene ordinato per tempo crescente di arrivo al nodo di origine, sfruttando *tdao*. Ciò è eseguito tramite un SelectionSort indiretto, di complessità $\Theta(n_{vd}^2)$, con n_{vd} il numero di veicoli disponibili: questo per non introdurre un'ulteriore funzione di aggiornamento di heap indiretto (non è una complessità particolarmente rilevante dato che $n_{vd} \leq n_v \ll n$).

C. Si scandisce il vettore ordinato *vtaxi* fino a quando l'orario di arrivo del taxi *i*-esimo al nodo di origine non supera l'orario minimo di partenza richiesto dal cliente e si assegna un indice *iPuntuale* a quest'ultima posizione. Notare che, in questo modo, si è diviso il vettore in due: i taxi che stanno prima di *iPuntuale* sono candidati a servire il cliente con puntualità, mentre quelli dopo *iPuntuale* sono candidati a servirlo senza puntualità.

D. A questo punto bisogna cercare il 'primo' tra i taxi così ripartiti che possa effettivamente servire il cliente, tenendo in considerazione che deve poter arrivare al nodo di origine, poi al nodo di destinazione e poi alla sede centrale per ricaricarsi (anche se non necessario). Secondo le specifiche del progetto, 'primo' significa il primo in ordine decrescente di arrivo, se prima di *iPuntuale*, e il primo in ordine crescente di arrivo, se dopo *iPuntuale*. Per questo motivo si esegue prima una scansione dei taxi disponibili da *iPuntuale* ad 1 e poi, se nessun assegnamento è possibile, da *iPuntuale* alla fine del vettore *vtaxi*. Qualora, in uno dei due blocchi, si trovi un taxi che soddisfa tutte le richieste, si aggiornano le informazioni globali su quel taxi, i contatori globali, l'orario di fine servizio e si restituisce il numero del taxi assegnato. Altrimenti si restituisce 0 e si incrementa *crif* (contatore dei rifiuti di chiamata).

Per quanto riguarda la gestione della coda di ricarica, ho optato per mantenerla solo tramite due variabili intere: *tfr1,* che indica il *t*empo di *f*ine *r*icarica del primo taxi in testa alla coda e *ntc*, che conta il numero di taxi in coda. L'unico dato che è necessario calcolare quando un taxi rientra in sede è l'orario in cui terminerà la ricarica, per poter innescare appunto il successivo evento di fine ricarica e le sole due variabili introdotte sono sufficienti a determinarlo. Esso è infatti pari a

$$hFine = tfr1 + (ntc \cdot dr)$$

dove *dr* è la durata di una ricarica e *ntc* va inteso come $(ntc - 1) + 1$: "-1" per il fatto che quando il taxi in testa alla coda avrà finito di ricaricarsi non sarà più in coda, mentre "$+1$" perché dopo aver aspettato gli altri taxi anche il corrente taxi dovrà usare un tempo di *dr* per ricaricarsi. In realtà il precedente assegnamento di *hFine* è valido solo quando $ntc > 0$, cioè quando c'è già almeno un taxi in coda; altrimenti, il tempo di fine ricarica è semplicemente la durata di una ricarica più l'orario a cui il taxi rientra in sede. Naturalmente, le variabili $tfr1$ e *ntc* vanno opportunamente aggiornate durante le modifiche della coda. Nel blocco *else if* di *RIENTRO_SEDE* la variabile *ntc* è in ogni caso incrementata, perché un taxi si è accodato, mentre $tfr1$ è aggiornata solo se effettivamente l'accodamento del nuovo taxi determina un nuovo tempo di fine ricarica del taxi in testa e ciò avviene solo quando è esso stesso la testa della coda (cioè $ntc = 0$ prima di arrivare). In quel caso $tfr1$ viene aggiornato con l'orario di fine ricarica del taxi appena arrivato. Nel blocco *else if* di *FINE_RICARICA*, invece, dato che il primo taxi in coda ha appena finito la ricarica, bisogna sempre sia decrementare *ntc* sia aggiornare $tfr1$ con il tempo di fine ricarica del prossimo taxi, pari all'orario dell'evento attuale di fine ricarica più la durata di una ricarica.

Terminata la simulazione, se c'è ancora qualche evento nello heap, cioè se l'arresto della simulazione è avvenuto perché si è superato l'orario di fine servizio, allora si procede a svuotare ordinatamente lo heap stampando gli eventi rimasti, ma senza processarli più.

Infine si deallocano tutte le zone di memoria precedentemente allocate dinamicamente.

5. ANALISI DEL TEMPO DI ESECUZIONE

Si vuole studiare la complessità temporale degli algoritmi presenti in questo progetto, assumendo il **criterio del costo uniforme**, in dipendenza delle variabili che indicano la grandezza dei dati, ovvero:

n	il numero di nodi del grafo
m	il numero di archi del grafo
n_v	il numero di veicoli
n_c	il numero di chiamate

Si omette pertanto, perché banale, l'analisi dei costi di funzioni che terminano in tempo 'costante', per esempio le funzioni di lettura/scrittura su grafo, su lista di archi, le funzioni di test di vuotezza, di estrazione della testa, la funzione che interpreta la linea di comando, ecc. Inoltre, anche se la funzione *CalcolaCamminiMinimi* è nascostamente invocata ad ogni lettura della matrice *C* tramite *LeggeCompletataC*, nelle procedure che contengono quest'ultima funzione non considererò la complessità di *CalcolaCamminiMinimi*, che verrà invece studiata a parte (considerando sempre che è invocata in modo aleatorio). Bisogna tener conto, infine, dei possibili limiti dei rapporti con i quali le variabili sopra possono comparire: si può supporre ragionevolmente che $n_v < n$, altrimenti la richiesta di posizionare i veicoli in nodi distinti non avrebbe senso, poi $m = \Theta(n)$ per quanto osservato nel ¶ **3**, e in qualche modo anche n_c e n sono collegati (basti pensare ad una città con 10^4 nodi ed un sobborgo di 10^2 nodi: è plausibile che in città arrivino almeno 500 chiamate in una giornata lavorativa, ma è poco probabile che più dello stesso numero di chiamate arrivi nel piccolo sobborgo).

Passiamo brevemente in rassegna le funzioni di complessità minore, tra le quali molte sono di complessità al più lineare. Nella libreria *heapdinE* le procedure che non terminano in tempo costante sono: *DistruggeHeapDinE*, di complessità $\Theta(n_c)$; *AggiornaChiamateHeapDinE*, la cui complessità è dominata dalla costruzione del cammino da origine a destinazione per ogni chiamata ed è quindi certamente $O(n \cdot n_c)$, anche se tale stima è, nella maggior parte dei casi pratici, grossolana (nei dati forniti, il valore $\max_{e \in H.V} (e.c \rightarrow nNodi)$ raramente e di poco supera $\sqrt{n}$); infine, per il mantenimento e l'aggiornamento dello *heapdine*, le funzioni *AggiornaHeapDinE*, *CancellaHeapDinE, SostituisceHeapDinE* costano $O(\log n_c)$ e *CreaHeapDinE* costa $O(n_c)$. Non sorprende che tutte le altre funzioni di aggiornamento di uno heap, nel progetto, abbiano complessità analoga: in *sim_taxi.c* le procedure *AggiornaHeapIndSTR*, *AggiornaHeapIndINT* sono di complessità $O(\log n_c)$ e *CreaHeapIndSTR, CreaHeapIndINT* di $O(n_c)$. Vi sono poi le funzioni di inizializzazione, *CaricaGrafo* di complessità $\Theta(m)$, *InizializzaTaxi* di $\Theta(n_v)$ e *CaricaChiamate* di $\Theta(n_c)$. Per quanto osservato sui rapporti con cui le variabili compaiono, neanche la complessità di *AssegnaTaxi,* $\Theta(n_v^2)$, è particolarmente incisiva nel nostro caso (motivo per cui si è scelto un SelectionSort anziché un altro ordinamento). Un ragionamento simile si può fare di *CollocaTaxi*, che costa $\Theta(n \cdot n_v)$ come visto nel ¶ **4.**

Vi sono poi procedure di complessità leggermente più rilevante. Si tratta delle funzioni che coinvolgono gli ordinamenti: gli HeapSort con stampa, *StampaClientiOrdinati* e *StampaViaggiOrdinati* costano $\Theta(n_c \log n_c)$ nel caso peggiore. Anche la simulazione ha un costo simile, anche se si mantiene solo uno heap e non un vettore ordinato: nel peggiore dei casi possibili, infatti, a partire da ogni evento *CHIAMATA* vengono innescati, l'uno dopo l'altro, tutti e tre gli altri tipi di evento e con ciò lo l'aggiornamento dello heap dovrà essere eseguito $4n_c$ volte su un'istanza

di dimensione al più n_c e in seguito, sempre nel caso peggiore, dovranno essere cancellati ordinatamente n_c eventi di tipo *FINE_RICARICA* per svuotare lo heap. Dunque:

$$T^p_{sim}(n_c) = 4n_c \cdot O(\log n_c) \; + \sum_{i=1}^{n_c} O(\log i) = O(n_c \log n_c)$$

infatti, l'espressione $\sum_{i=1}^{n_c} \log i$ è pari a $\log(n_c!)$ e, dalle disuguaglianze $n! \leq n^n$, $(n!)^2 \geq n^n$ segue che

$$\frac{1}{2} n_c \log n_c \leq \log(n_c!) \leq n_c \log n_c \,, \quad \text{cioè} \quad \log(n_c!) = \Theta(n_c \log n_c)$$

Accanto alle procedure di ordinamento, un'altra procedura che ha una complessità di $O(n_c \log n_c)$ è *GreedyKP*. Se, infatti, tutti gli "oggetti" dovessero stare nello "zaino", ovvero la somma della durate dei viaggi fosse inferiore al tempo totale del servizio, allora in tal caso il ciclo *while* di questa funzione verrebbe eseguito il numero massimo di volte e cioè n_c volte, ed insieme ad esso *AggiornaHeapIndiretto,* che costa $O(\log i)$ su un'istanza di dimensione i. Quindi:

$$T^p_{GKP}(n_c) = \sum_{i=1}^{n_c} O(\log i) = O(n_c \log n_c) \,,$$

dimostrando l'ultima uguaglianza come sopra.

L'algoritmo che però domina la complessità dell'intero progetto, quando incontra il suo caso peggiore, è *CalcolaCamminiMinimi,* nella libreria *dijkstra*. Avendo già analizzato a lungo il funzionamento di questo algoritmo, analizziamone semplicemente i costi.
È evidente che la parte più costosa dell'algoritmo sono le funzioni *Dijkstra* e *RiempieMatrici*. La prima, essendo stata implementata con uno heap, ha una complessità di

$$T_D(n, m) = \Theta(n + m) + O(m \log n)$$

come dimostrato a lezione. L'aspetto interessante è che, come osservato nel ¶ **3**, il numero m di archi complessivi in una rete stradale è solitamente nell'ordine di $\Theta(n)$, per cui l'algoritmo *Dijkstra*, anche se iterato n volte senza l'utilizzo di *RiempieMatrici*, è più vantaggioso dell'algoritmo di Floyd-Warshall. Se avessimo invece che $m = \Theta(n^2)$, sarebbe allora più vantaggioso quest'ultimo.
Passiamo all'algoritmo *RiempieMatrici*: il motivo della sua ideazione è agevolare il tempo di calcolo complessivo. Si è osservato, infatti, che nel migliore dei casi in una sola iterazione di *Dijkstra+RiempieMatrici* si completano C e P, mentre nel peggiore servono n iterazioni. Teniamo quindi conto di questi due casi limite e studiamo il costo di *RiempieMatrici* in riferimento ad essi.
Il cuore di *RiempieMatrici* è costituito dalla funzione *CamminiRicorsiva*, anche in termini di complessità: *TrovaFoglie* termina in meno di $3n$ passi, mentre *CamminiRicorsiva*, in generale, impiega più tempo. Osserviamo infatti che il tempo di esecuzione di *CamminiRicorsiva* $T_{CR}(i)$ quando è invocata su una foglia i è pari a

$$T_{CR}(i) = \sum_{j=0}^{p_i} \sum_{k=0}^{j} \Theta(1) = \Theta(p_i^2)$$

dove p_i è la profondità della foglia i. La precedente espressione è stata ottenuta considerando che *CamminiRicorsiva* è stata architettata proprio in modo da scandire tutte le coppie di nodi di un cammino la cui lunghezza è $p_i + 1$. D'altronde, la procedura *CamminiRicorsiva* è di tipo terminale ed è immediato verificare che ogni volta viene eseguita su un'istanza più piccola di una 'unità', ovvero il cammino che va da s al padre di f. In questo modo il costo di n_f iterazioni di *CamminiRicorsiva* su ciascuna foglia è uguale a

$$\sum_{i=1}^{n_f} T_{CR}(i) = \Theta\left(\sum_{i=1}^{n_f} p_i^2 \right).$$

Tale grandezza è minima quando l'albero è nella forma dell'**Esempio 2** del ¶ **3**, che avviene in concomitanza del minimo riempimento possibile delle matrici. In quel caso infatti $p_i = 1 \quad \forall i = 1,\ldots,n_f$ e $n_f = n - 1$, cosicché

$$T^m_{RM} = \Theta(n) \quad \text{(Tempo migliore di esecuzione di } \textit{RiempieMatrici)}$$

Nel caso opposto, cioè quello dell'**Esempio 1**, che dà luogo al riempimento totale delle matrici, vale $n_f = 1$ e $p_1 = n - 1$ quindi

$$T^p_{RM} = \Theta(n^2) \quad \text{(Tempo peggiore di esecuzione di } \textit{RiempieMatrici)}$$

Quindi, il numero T di passi necessari a riempire le matrici C e P ha un ordine di grandezza inferiormente e superiormente limitato, rispettivamente, dai seguenti casi:

Caso peggiore - Tutti gli alberi dei cammini minimi hanno altezza 1	Caso migliore - Esiste un albero dei cammini minimi che è un cammino, ed è il primo albero esplorato.
$T = n \cdot (T_D + T^m_{RM}) =$	$T = T_D + T^p_{RM} =$
$= n \cdot (O(m \log n) + \Theta(n+m) + \Theta(n)) =$	$= O(m \log n) + \Theta(n+m) + \Theta(n^2) =$
$= O(mn \log n) + \Theta(n(n+m)) =$	$= O(m \log n) + \Theta(n^2) =$
$\overset{m=\Theta(n)}{=} O(n^2 \log n) + \Theta(n^2)$	$\overset{m=\Theta(n)}{=} \Theta(n^2)$

Notare che nel caso peggiore prevale la complessità dovuta all'algoritmo di *Dijkstra*, mentre nel caso migliore quella di *RiempieMatrici*.

In definitiva,

6. Analisi dello spazio di memoria occupato

Nel corso dell'esecuzione, lo spazio allocato per eseguire gli algoritmi e mantenere le strutture dati varia. C'è un limite inferiore allo spazio utilizzato dal programma, che è lo spazio occupato dalle variabili globali, allocate all'inizio del progetto e distrutte alla fine. Si riportano tali quantità in una tabella, utilizzando le variabili dimensionali n, m, n_c, n_v usate anche nell'analisi dei tempi. Per indicare la quantità di memoria occupata da, rispettivamente, un *int*, un *long*, un *double*, un *char,* un *puntatore*, si useranno le notazioni i, l, d, c e pt.

Variabili	Spazio
nv, T, aut, percaut, dr, tfr1, ntc, crif, cric, tmov, guad	$11i$
inc, pos, batt, disp	$(4 \cdot n_v)i$
C	$\left(\sum_{j=1}^{n}(j+1)\right)i = \frac{n^2+3n}{2}i$
P	$(n+1)^2\ i$
R	$2i + 2m \cdot \texttt{sizeof(arco)} =$ $= 2i + 2m \cdot (2i + 2pt) =$ $=(2+4m)i\ +\ (4m)pt$
H	$n_c \cdot (3i + \texttt{sizeof(chiamata)}) =$ $n_c \cdot (3i + \texttt{ROW\_LENGTH} \cdot c + 7i + \texttt{nNodi}\,i) =$ $=[n_c(10 + \texttt{nNodi})]i\ +\ (n_c \cdot \texttt{ROW\_LENGTH})c$

Totale dello spazio occupato dalle variabili globali:

$$S_{v.gl.} = \left(11 + 4n_v + \frac{1}{2}(3n^2 + 5n + 2) + (2 + 4m) + n_c(10 + \texttt{nNodi})\right) i\ + \\ +(4m)pt\ +\ (n_c \cdot \texttt{ROW\_LENGTH})\ c$$

Oltre a queste variabili, nel *main* vengono mantenute le strutture dati per il problema dello zaino (con uno spazio di $\Theta(n_c) \cdot l\ +\ \Theta(n_c) \cdot i$), pochi interi locali ai blocchi *if-else,* due eventi *e* ed *enew* per la simulazione e tutto lo spazio necessario ad eseguire le funzioni. Dato che si può tracciare l'inizio e la fine di ogni procedura eseguita nel *main*, si possono anche individuare quelle a cui corrisponde il maggiore spazio di memoria utilizzato istantaneamente. Da una breve analisi si può constatare che la maggior parte delle procedure nel *main* richiedono spazio costante e limitato. Ecco un elenco di quelle che, invece, allocano una quantità di memoria che dipende dalle variabili dimensionali (vengono considerate solo le macro-funzioni, chiamanti di altre funzioni che operano con spazio costante o non maggiore alle prime):

CalcolaCamminiMinimi	$\Theta(n)i\ +\ \Theta(1)pt$
StampaClientiOrdinati	$\Theta(n_c) \cdot i + \Theta(1)pt$

StampaViaggiOrdinati	$\Theta(n_c) \cdot i \ + \ \Theta(1)pt$
CollocaTaxi	$\Theta(n + n_v) \cdot i + \Theta(1)pt$
GreedyKP	$\Theta(n_c) \cdot l \ + \ \Theta(n_c) \cdot d \ + \ \Theta(1)(pt + i)$

In conclusione, è possibile determinare anche un limite superiore allo spazio occupato del progetto, ma questo, in termini di ordini di grandezza, non supererà comunque lo spazio inizialmente allocato per conservare le variabili globali, che, ricordiamo, era nell'ordine di

$$\left(O(n_v + n^2 + n_c + m + n_c \cdot n)\right) \cdot i \ + \ \Theta(m) \cdot pt \ + \ \Theta(n_c) \cdot c.$$

Milano, 13 Giugno 2019