Università degli Studi di Pisa
Corso di Laurea Specialistica in Informatica

Tesi di Laurea

"Algoritmi per l'ottimizzazione simultanea di orari e turni nel trasporto pubblico urbano"

Candidato:
Alessandro Bertolini

Relatore:
Prof. Antonio Frangioni

In collaborazione con:
M.A.I.O.R. S.r.l.

Contents

1 Introduzione

Il processo di pianificazione per un servizio di trasporto pubblico si pone il fine ultimo di raggiungere simultaneamente due obiettivi in contrasto tra loro:

- il soddisfacimento dei bisogni della clientela, mettendo a disposizione mezzi adeguati alla copertura del territorio e organizzando le corse dei suddetti mezzi in modo tale da sopperire alla diversa richiesta di copertura durante l'arco del periodo esaminato (giornate/fasce orarie più o meno impegnative);
- la gestione oculata delle risorse, ovvero ci si prefigge l'utilizzo del minor numero di mezzi possibile, la ricerca di viaggi che minimizzino i percorsi e l'impiego del minor numero di dipendenti (piloti, autisti, etc.) possibile nel tentativo di ridurre i costi.

Questi obiettivi sono ovviamente generali e valgono per una qualsiasi azienda di trasporti (bus, taxi, treni, aerei) che si impegna a fornire un servizio soddisfacente al pubblico. È bene premettere, tuttavia, che il modello e l'algoritmo proposti in questo documento sono specifici per il contesto riguardante il Trasporto Pubblico Locale (TPL). Infatti, sebbene gli obiettivi e le considerazioni qui introdotte possono essere ritenuti validi in tutti i casi, addentrandosi nel problema le differenze tra i vari contesti diventano sempre più importanti e sempre meno sintetizzabili in un unico ragionamento nonché in un unico modello e in un unico algoritmo di risoluzione. Nel proseguio di questo lavoro quindi si circoscrivono i ragionamenti al TPL.

Il processo di pianificazione per un servizio di TPL si può suddividere in cinque fasi:

- *la definizione di un insieme di percorsi* per i bus al fine di coprire particolari aree del territorio: consequenzialmente la definizione, per ogni percorso, delle fermate dei bus in zone strategiche per la salita/discesa dei passeggeri e delle relative stime di affluenza-passeggeri sulle fermate stesse;
- *la definizione delle frequenze richieste*, per ogni percorso e per ogni periodo di tempo; in questa fase si effettua una stima del tempo che dovrebbe intercorrere tra una corsa di un bus su un percorso e la corsa successiva (sul medesimo percorso) per fornire un servizio ottimale dal punto di vista dei passeggeri;
- *timetabling*, ovvero la ricerca di una soluzione, intesa come insieme di corse coperte dal servizio offerto, che soddisfi il più possibile le frequenze definite nella fase precedente;
- *scheduling delle vetture*, cioè si definisce l'impiego di vetture in maniera ottimale (dal punto di vista del risparmio delle risorse) e coerente con le decisioni prese in fase di timetabling;
- *formazione dei turni degli autisti*, ovvero si definiscono i turni degli autisti in modo da assicurare la copertura delle corse e l'utilizzo delle vetture decise ai passi precedenti.

Le fasi sopra descritte possono essere ulteriormente partizionate in due sottoinsiemi: il primo comprende le fasi che definisono i percorsi e le frequenze desiderate, cercando di descrivere la realtà tramite la creazione di un modello semplificato della stessa. Il secondo

sottoinsieme comprende le fasi che lavorano sul modello realizzato al fine di trovare una soluzione ottimale al problema descritto dal modello stesso.

Il modello e l'algoritmo trattati in questa sede propongono un particolare approccio alla risoluzione combinata del *timetabling* e dello *scheduling delle vetture*: essi cercano una soluzione ottimale a partire da un input e tralasciano per il momento la fase di *formazione dei turni degli autisti*. Tale input non è determinato dal modello in questione in quanto emerge dalle prime due fasi che vengono indagate da soggetti appositamente preposti. La combinazione delle due fasi genera un problema molto complesso la cui risoluzione non è banale. L'approccio storicamente più utilizzato per questo tipo di lavoro consiste nel pianificare in prima istanza le corse basandosi sulle frequenze richieste e a partire da queste ultime si costruisce uno scheduling ottimizzato dei veicoli: ad esempio Kliever [KN05] descrive un sottomodello ottimizzato per lo scheduling dei veicoli a partire da un insieme di corse già decise in sede di timetabling. Il vantaggio di questo tipo di soluzione consiste nel ritorno ai problemi "originali" (il timetabling prima, lo scheduling delle vetture dopo) che sono più facilmente risolvibili se presi singolarmente. Risulta evidente, tuttavia, che l'ottimizzazione disgiunta delle due fasi non garantisce di ottenere la soluzione di costo *globale* minore. Nella fattispecie una scelta sub-ottima delle corse nella fase iniziale potrebbe portare ad una soluzione globalmente migliore perché conveniente per lo scheduling dei veicoli. Nell'approccio di tipo sequenziale appena esposto è abbastanza palese che la ricerca della soluzione è maggiormente orientata ad ottimizzare la fase di timetabling piuttosto che lo scheduling dei veicoli che, essendo risolto in seconda istanza, deve adattarsi ad una soluzione grandemente delineata.

Un esempio inverso lo si può indagare attraverso il lavoro di Michaelis [MS09] nel quale vengono utilizzate euristiche in modo sequenziale che privilegiano la decisione riguardo a quanti e quali veicoli utilizzare, determinano in seguito il percorso che ognuno di essi deve coprire ed infine eseguono il timetabling. È chiaro come l'aspetto fondamentale in questo tipo di risoluzione sia il tentativo di aggirare l'alta complessità del problema congiunto tramite forti assunzioni e/o compromessi a monte, azioni che tuttavia eliminano la garanzia di ottimalità della soluzione trovata nonché creano squilibrio tra i due obiettivi perseguiti (soddisfazione dei clienti vs ottimizzazione dell'impiego delle risorse). In sintesi: porre in ordine sequenziale i sottoproblemi riduce la complessità ma pone un'enfasi maggiore su uno dei due.

La ricerca in questi ultimi anni ha provato ad eludere le problematiche emerse con un cambio di prospettiva. Altri tipi di approcci tentano di confinare la sequenzialità delle due fasi all'interno dei passi d'iterazione di un algoritmo di tipo euristico, cercando di affrontare i due problemi quasi simultaneamente: tale modus operandi è proposto, seppure in modalità diverse, da VanDerHakken [VVV08], Hao [HG08], Fleurent [FL09]. In questi approcci le valutazioni riguardanti le soluzioni trovate (ovvero i calcoli delle varie funzioni obiettivo) agglomerano il timetabling con lo scheduling, fornendo così un metro univoco ed omnicomprensivo con il quale confrontare le soluzioni trovate tra loro. Le euristiche che governano l'esecuzione sono di *ricerca locale*, ovvero modificano leggermente soluzioni già trovate in precedenza (ad esempio modificando di poco la timetable e ricalcolando l'adeguato scheduling dei veicoli), osservano se hanno ottenuto dei miglioramenti tenendo sempre in considerazione la possibilità di poter incappare in un ottimo locale nel corso della ricerca.

Il modello e l'algoritmo presentati in questo lavoro si propongono di risolvere il problema tramite un approccio integrato in accordo alla tendenza ad eliminare la sequenzializzazione dei sottoproblemi. Questi sono stati sviluppati in collaborazione con la *M.A.I.O.R. S.r.l.*, azienda che fornisce strumenti informatici a coloro che operano nel settore del trasporto pubblico. Gli strumenti che la *M.A.I.O.R.* ha fornito fino a questo momento per la risoluzione del problema esposto sono basati su euristiche guidate da insiemi di regole dettate dall'esperienza: questi algoritmi operano una risoluzione sequenziale dei due sottoproblemi, dando maggiore priorità alla parte di timetabling (che è eseguita prima) rispetto allo scheduling delle vetture (calcolato a partire dalle corse trovate nel timetabling). È emersa conseguentemente l'esigenza di un algoritmo che non attribuisca a priori una maggior importanza ad uno dei due sottoproblemi e che sia incentrato non su un approccio euristico ma piuttosto su un modello matematico in grado di integrare timetabling e scheduling.

Il modello matematico è stato quindi ideato come composizione di due sottomodelli, due grafi formati da insiemi di nodi e insiemi di archi pesati e limitati superiormente, sui quali si applicano tre differenti algoritmi di risoluzione:

- il problema del timetabling è stato modellato in modo tale da essere risolto tramite un algoritmo per la ricerca di cammini minimi sul relativo grafo;
- il problema dello scheduling dei veicoli è stato modellato in modo da essere risolto tramite un codice esterno preesistente (di proprietà dell'Università di Pisa) specializzato nella ricerca del flusso di costo minimo sul relativo grafo;
- il compito di far collimare gli obiettivi dei due sottoproblemi è ottenuto tramite l'utilizzo di tecniche Lagrangiane di rilassamento di vincoli, implementate in un codice esterno specializzato, anch'esso di proprietà dell'Università di Pisa, e integrate quindi nel nuovo codice.

Il lavoro svolto non si è esaurito con l'ideazione e l'implementazione del modello integrato. È stato infatti eseguito uno studio approfondito sulle potenzalità del modello stesso indagando quanto fosse in grado di fornire soluzioni adeguate al problema descritto . A tal fine si e deciso di escludere gli algoritmi implementati all'interno della prima versione funzionante del codice e si è scelto di utilizzare un software esterno (*Cplex*) la cui validità è ampiamente riconosciuta nel settore. In base ai risultati di questo studio è stato quindi implementato un aggiornamento sostanziale della struttura del codice, volta ad affiancare alle tecniche Lagrangiane già presenti un approccio di tipo euristico-greedy di costruzione progressiva della soluzione, ai fini di migliorare la qualità delle soluzione trovate e le prestazioni fornite dal codice.

È necessario inoltre premettere che il codice è stato implementato in modo da utilizzare un ampio set di parametri che permettono di regolare sia l'equilibrio tra timetabling e scheduling (agendo direttamente sulla funzione obiettivo del modello) sia l'esecuzione stessa del codice. Infatti sia la qualità di una soluzione (valutabile ad esempio su quanto essa rispecchi il tipo di equilibrio tra timetabling e scheduling dei veicoli che s'intende raggiungere) sia il modo di utilizzo del codice stesso (si può preferire un'esecuzione rapida ad un'analisi più approfondita ma anche più dispendiosa in termini di tempo) sono concetti strettamente legati alla volontà e alle esigenze del singolo utente. Una parte del lavoro è consistita per l'appunto nella

ricerca, in collaborazione stretta con gli esperti M.A.I.O.R., di insiemi di valori da attribuire ai parametri così da ottenere, nel maggior numero possibile di casi, soluzioni "equilibrate" in tempi ragionevoli.

Infine il modello è stato più volte esteso con l'obiettivo di descrivere problemi di natura complessa (frequenze frazionarie, controllo del flusso dei veicoli nella giornata operato da input, percorsi plurimi) senza modificare la natura dell'approccio algoritmico.

Senza entrare prematuramente nei dettagli, si può comunque anticipare che il lavoro svolto ha permesso di raggiungere gli obiettivi prefissati. Il modello concepito e implementato riesce a descrivere scenari, tipici del TPL, mediamente complessi, garantendo soluzioni di buona qualità (secondo gli esperti M.A.I.O.R.) in tempi ragionevoli.

2 Il problema

Sia dato un insieme di stazioni di partenza/arrivo/fermata dei bus, che da qui in poi saranno chiamate genericamente *nodi*. Nell'insieme dei nodi sono individuati due sottoinsiemi di *capolinea*, $A_1, ...A_n$ e $B_1, ...B_m$. Presso tali capolinea i bus partono o arrivano. Si assume che per ogni possibile coppia di capolinea (A_i, B_j) esista un unico *percorso* possibile. Sul suddetto percorso le *corse*, ovvero un singolo viaggio di un singolo bus sul percorso in un preciso orario, possono transitare nei due versi: da A_i a B_j (verso *Ascendente*) o da B_j ad A_i (verso *Discendente*).

Caratteristica fondamentale di questa topologia è il tratto centrale, comune a tutti i percorsi.

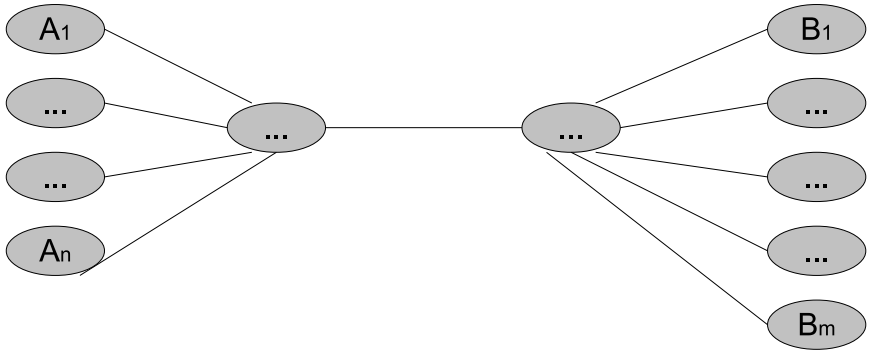

Figure 1: Caso generico (non tutti i nodi sono stati disegnati)

Nel caso più semplice esistono ovviamente soltanto due capolinea, A e B, collegati da due percorsi, uno per ogni verso.

Figure 2: Caso semplice (non tutti i nodi sono stati disegnati)

Sul tratto centrale comune a tutti i percorsi sono presenti due nodi cosidetti *pilota*, decisivi per stabilire le frequenze desiderate. Può essere designato un unico nodo come pilota per entrambi i versi: in questo caso potranno comunque essere stabilite frequenze diverse a seconda del verso. È anche fornito un *orizzonte temporale*; un esempio tipico è costituito dall'orizzonte 5:00 – 24:00, ma sono possibili altre configurazioni, quali 0:00 – 40:00 per tener conto degli "effetti di bordo" della schedulazione sulla giornata successiva. Per questo l'orizzonte temporale *non* è ciclico, il che significa che, ad esempio, 5:00 e 29:00 *non* indicano lo stesso istante di tempo; ogni giornata è trattata in modo indipendente. L'orizzonte temporale è diviso in *fasce orarie*; per ciascuna di queste su ogni nodo pilota è definito il cosidetto *intertempo desiderato* ovvero il tempo, calcolato a priori, che si vorrebbe intercorresse tra il passaggio di una corsa sul nodo pilota al passaggio della successiva corsa sempre sul nodo pilota. La corsa precedente e la corsa successiva passanti dallo stesso nodo pilota in una data fascia oraria non devono necessariamente coprire il medesimo percorso: è sufficiente che passino dal nodo pilota e seguano il rispettivo percorso nello stesso identico

verso, Ascendente o Discendente. Un veicolo (bus) percorre quello che è definito Turno Macchina (TM) cioè una sequenza di corse e di *tratte fuori linea* (i *fuori linea* permettono al veicolo di andare dal *deposito* ad un capolinea e viceversa). Si definisce invece *viaggio* una sequenza di corse successive tra loro e non interrotte da nessun rientro al deposito: un TM è quindi costituito da una sequenza di viaggi (eventualmente uno) intramezzati eventualmente da rientri al/uscite dal deposito.

Oggetto del problema di base che ci prefiggiamo di risolvere è la pianificazione degli istanti di partenza delle corse dai capolinea (in altri temini: la scelta delle corse) e la loro partizione in viaggi perseguendo i seguenti obiettivi contrastanti:

1. minimizzazione delle differenze tra gli intertempi desiderati e gli intertempi pianificati;
2. ottimizzazione dell'uso del parco veicoli, ottenibile tramite:
   (a) minimizzazione del *numero di turni macchina complessivi* necessario per coprire il servizio;
   (b) minimizzazione del *numero di turni macchina in circolo ad ogni istante*, ossia (idealmente) la somma su tutti gli istanti di tempo del numero di macchine che *non* sono al deposito in quell'istante;
   (c) minimizzazione del *tempo totale speso dai bus fuori dal deposito*, eventualmente escludendo quello speso ad effettuare le corse (che è inevitabile) e quindi conteggiando solamente quello speso nelle soste ai capolinea e nei viaggi da/per il deposito;
   (d) minimizzazione del *numero dei viaggi*.

Tutto ciò nel rispetto dei seguenti vincoli:

- inserimento nella soluzione di eventuali *corse fisse*, ovvero corse che sono state pianficate a priori;
- per ogni nodo pilota la distanza temporale tra una corsa e la sua successiva deve rientrare in un certo range;
- per ogni nodo pilota la prima (e l'ultima) corsa che passano per esso deve rientrare in un sottoinsieme (prestabilito) dell'insieme delle corse;
- ogni sosta ad un capolinea effettauata da un mezzo deve rientrare in un prestabilito range temporale;
- ogni sosta al deposito effettuata da un mezzo non può durare meno di una certo periodo di tempo;
- il numero di mezzi che possono entrare nel (uscire dal) deposito può essere soggetto a limitazioni poste a priori.

Per quanto riguarda l'obiettivo 1., è necessario misurare in qualche modo lo spostamento complessivo tra gli intertempi desiderati e quelli realizzati. Ci sono molte norme possibili: per convenienza modellistica ed algoritmica si utilizzerà una norma separabile, ossia data dalla somma di un termine per ciascun intertempo. Questo termine sarà zero se l'intertempo è quello desiderato, e aumenterà con una formula arbitraria al crescere del (valore assoluto del)la differenza tra l'intertempo ottenuto e quello desiderato.

Gli obiettivi elencati nei punti del tipo "2." invece possono trovarsi in forte correlazione tra loro. Gli obiettivi 2.(b) e 2.(c) sono sostanzialmente equivalenti; poiché 2.(c) è più facile da gestire dal punto di vista algoritmico e modellistico, nel seguito ci riferiamo solamente a questo, intendendo tacitamente che esso "rappresenta bene" anche 2.(b). I due obiettivi 2.(a) e 2.(c) non sono in contrasto tra loro: 2.(c) è sostanzialmente una funzione obiettivo a "granularità più fine" di 2.(a), nel senso che tiene conto non solo del numero di TM complessivo, ma anche di come essi vengono usati. In questo senso, 2.(a) si può considerare come un obiettivo "di primo livello"' mentre 2.(c) lo si può considerare "di secondo livello". Ciò si può ottenere, come al solito, mediante opportuna scalarizzazione (coefficienti di costo molto più grandi per 2.(a) rispetto a 2.(c)). Viceversa, gli obiettivi 2.(c) e 2.(d) sono contrastanti: minimizzare il numero di viaggi porta ad evitare i rientri al deposito utilizzando ove possibile al loro posto le soste al capolinea. Ovviamente 2.(a) e 2.(d) sono sostanzialmente allineati, infatti se si evitano i rientri al deposito il numero di turni macchina è uguale al numero di viaggi. Per questo è possibile, in linea di principio, scegliere 2.(a) o 2.(d) come obiettivo "di primo livello"; però, come già esposto, scegliendo 2.(d) si "va contro" a 2.(c), per cui in questo caso non avrebbe senso avere la prima come funzione "di secondo livello".

Come vedremo meglio in seguito con la descrizione dettagliata del modello, si è scelto di provare a perseguire in primo luogo l'obiettivo 2.(a), ovvero la minimizzazione dei Turni Macchina complessivi. La scelta invece tra il perseguire, ulteriormente al 2.(a), l'obiettivo 2.(c) piuttosto che 2.(d) (o viceversa, ricordiamo che sono obiettivi in contrasto tra di loro) è insita nella taratura di alcuni dei pesi che regolano, come vedremo, la funzione obiettivo del modello ma è meno esplicita rispetto al 2.(a) (regolato da un peso univoco) e quindi ottenibile solo dopo aver compreso a fondo la logica dietro all'equilibrio dei parametri in gioco. Per quanto riguarda invece l'obiettivo 2.(b), ovvero la minimizzazione dei Turni Macchina in circolo ad ogni istante, come vedremo non è ottenibile a pieno: si può però imporre dei vincoli sul flusso entrante o uscente dei mezzi dal deposito (vedi Paragrafo 6) simulando parzialmente tale obiettivo.

3 Il modello "base"

Il modello *base* permette la risoluzione della *versione semplice del problema*, ovvero quella mostrata in Figura 2 che prevede due soli capolinea. L'esistenza di un modello base implica che il problema è stato privato, in prima istanza, di alcune parti al fine di renderlo più semplice; queste parti furono comunque analizzate teoricamente (vedi Paragrafo 3.5) allo scopo di avere idee dalle quali iniziare qualora fosse stato ritenuto opportuno estendere il modello.

Sia $N = \{A, B\}$ l'insieme dei capolinea; si definisce quindi come $N_+ = N \cup \{O\}$ l'insieme che contiene anche il deposito O da cui i bus partono all'inizio del TM e ritornano alla sua fine. Il deposito non è in pratica necessariamente unico, ma assumiamo che la scelta tra quale dei depositi venga utilizzato per raggiungere un dato capolinea sia esogena al modello, e quindi che sia possibile considerare nel modello un unico deposito. È specificato, come input del modello, un insieme C di possibili corse, a sua volta partizionato nei sottoinsiemi C_A delle corse in partenza da A verso B (verso Ascendente) e C_B delle corse in partenza da B verso A (verso Discendente). Ciascuna corsa $c \in C$ è caratterizzata dall'istante di partenza e dall'istante di arrivo, ossia $c = (i, j)$ dove i e j sono istanti all'interno dell'*orizzonte temporale* T. Gli istanti sono dati in secondi, e potranno essere indicati rispettivamente come s_c ed e_c per $c \in C$; alternativamente potremo dire direttamente $(i, j) \in C$. Ogni corsa sarà caratterizzata anche dagli istanti d'arrivo a tutti i nodi intermedi che compongono il percorso. Tipicamente l'orizzonte temporale T sarà una giornata lavorativa 5:00 – 24:00, ma possono essere considerate corse che iniziano o finiscono dopo la fine della giornata, il cui l'orario finale potrebbe essere 25:00, 26:00, ... fino a 40:00. Indichiamo con $\bar{t}$ l'istante finale di T, mentre per definizione l'istante iniziale è 0. L'orizzonte temporale T è suddiviso in k fasce orarie, caratterizzate da $k+1$ istanti $t_0 = 0$, $t_1 > t_0$, $t_2 > t_1$, ..., $\bar{t} = t_k > t_{k-1}$. Per ciascuna fascia oraria h dell'orizzonte temporale T e per ciascun verso (Ascendente, Discendente) è definito un intertempo desiderato I_h tra passaggi di corse successive per il relativo *nodo pilota* in quella fascia oraria. L'intertempo tra due passaggi per il nodo pilota potrebbe *non* coincidere con l'intertempo tra le due partenze successive, nel caso in cui l'aumentare o il diminuire della congestione del traffico porti a velocità misurabilmente diverse. Assumiamo però (ovviamente) che sia noto, per ciascuna corsa $c \in C$, l'istante p_c di passaggio per il nodo pilota.

Il problema richiede di determinare contemporaneamente *due* diverse cose:

- il sottoinsieme $C^* \subset C$ delle corse da effettuare che "rispettino il più possibile" gli intertempi desiderati;
- un insieme di TM che permettano di coprire C^* ottimizzando le funzioni obiettivo 2.(a)–2.(d).

Per questo il modello è composto di due sotto-strutture sostanzialmente separate, una relativa alla determinazione di C^* e l'altra relativa alla determinazione dei TM. Saranno quindi per prima cosa presentati separatamente i due sottomodelli relativi a ciascuna delle sotto-strutture, quindi sarà affrontata la questione di come congiungere i due sottomodelli per esprimere il problema complessivo.

3.1 Il (sotto)modello di schedulazione dei bus (modello BS)

Per ogni $n \in N_+$ e per ogni fascia oraria h sono definite la *sosta minima* $\delta^h_{min,n}$ e la *sosta massima* $\delta^h_{max,n} \geq \delta^h_{min,n}$; si noti che può accadere che la sosta al capolinea sia *fissata*, ossia che risulti $\delta^h_{max,n} = \delta^h_{min,n}$ per qualche h ed $n \in N$. Si noti inoltre che si può assumere che $\delta^h_{max,O} = \infty$, ovvero che non esiste un limite di sosta massima per un bus all'interno del deposito O. Sono anche definiti i tempi di viaggio t^h_{n+} e t^h_{n-} rispettivamente dal capolinea $n \in N$ al deposito e viceversa; sono dati due tempi diversi perché non si può assumere $t^h_{n+} = t^h_{n-}$ (per via di sensi unici, ecc.).

Tra due corse $c \in C$ e $d \in C$ sono definiti due diversi concetti di compatibilità:

- *compatibilità in linea*: la corsa d è compatibile con la corsa c se c termina nella fascia oraria h allo stesso capolinea n,lo stesso in cui d inizia nella fascia oraria $h' \geq h$ (ossia $c \in C_A$ e $d \in C_B$ o viceversa), $s_d - e_c \geq \delta^h_{min,n}$ (il bus arriva al capolinea prima del momento in cui ci dovrebbe ripartire, e resta al capolinea almeno il tempo di sosta minima), e $s_d - e_c \leq \delta^h_{max,n}$ (il bus non resta al capolinea un tempo superiore alla sosta massima);

- *compatibilità fuori linea*: la corsa d è compatibile con la corsa c se c termina nella fascia oraria h ad un qualsiasi capolinea n, d inizia nella fascia oraria $h' \geq h$ ad un qualsiasi capolinea n' e $s_d \geq e_c + t^h_{n-} + \delta^h_{min,O} + t^{h'}_{n'+}$, ossia il bus ha il tempo necessario per tornare al deposito, restarci almeno il tempo minimo e poi raggiungere il capolinea.

Scopo del modello è determinare un insieme di corse e l'insieme dei TM che le coprono in modo da minimizzare il numero totale di bus utilizzati rispettando i vincoli corrispondenti alle compatibilità (derivanti dai tempi di viaggio e dalle soste minime e massime). Ciò può essere formulato per mezzo di un problema di *Flusso di Costo Minimo* (MCF).

Nei prossimi sottoparagrafi è riportata l'evoluzione "storica" del grafo BS e quindi del sottomodello stesso: una descrizione di tipo incrementale permetterà tramite il modello di base di comprendere i concetti fondamentali ed essere quindi in grado di capire più agevolmente le estensioni del modello fino all'ultima di queste, implementata nella prima versione dell'algoritmo.

3.1.1 Il modello BS base

Il problema può essere formulato come un problema di MCF sul *grafo della compatibilità*, un grafo bipartito strutturato come segue. Per ciascuna corsa $c \in C$ (quindi sia le corse in partenza da A che quelle in partenza da B) sono presenti nel grafo due diversi nodi, che indicheremo con c^- e c^+. Per aiutare l'intuizione, c^- può essere identificato con l'inizio della corsa e c^+ con la sua fine.

Nel grafo esistono quindi quattro diversi tipi di archi:

1. *Archi corsa*: sono gli archi (c^-, c^+) che uniscono i due nodi della stessa corsa c, con costo zero e capacità uno. Un'unità di flusso che circola sull'arco relativo alla corsa c indica che la corsa sarà effettuata da un bus.

2. *Archi di compatibilità in linea*: sono gli archi (c^+, d^-) che uniscono il nodo "termine" di una corsa c con il nodo "inizio" di una corsa d se e solo se c e d sono compatibili in

linea. Un'unità di flusso che circola sull'arco (c^+, d^-) indica che il bus che ha effettuato la corsa c attenderà al capolinea e da lì ripartirà per effettuare la corsa d (si noti che questo ovviamente significa che $c \in C_A$ e $d \in C_B$ o viceversa). Questi archi hanno capacità uno e costo proporzionale al tempo di attesa usato in più rispetto al tempo di sosta minimo sul capolinea c^+ nella fascia h, ossia a $s_d - e_c - \delta^h_{min,n}$, moltiplicato per un opportuno fattore γ_I. Il costo può aumentare nel caso in cui $s_d - e_c \geq I_h$, dove I_h è l'intertempo desiderato (concetto che chiariremo più avanti nel Paragrafo 3.2) per quanto riguarda la fascia oraria h entro la quale si trova e_c, ovvero l'istante d'arrivo della corsa c. Se tale condizione si verifica, il costo di compatibilità in linea aumenta di $\frac{\gamma_M}{stima_{numVetture}}$[1], dove $stima_{numVetture}$ è calcolata tramite $\frac{TG_c}{I_h}$[2] e rappresenta una stima puntuale del numero di vetture necessario a soddisfare la frequenza richiesta.

3. *Archi di compatibilità fuori linea*: sono gli archi (c^+, d^-) che uniscono il nodo "termine" di una corsa c con il nodo "inizio" di una corsa d se e solo se c e d sono compatibili fuori linea. Un'unità di flusso che circola sull'arco (c^+, d^-) indica che il bus che ha effettuato la corsa c andrà dal capolinea al deposito, attenderà una quantità di tempo opportuna (non inferiore alla sosta minima al deposito) e ne ripartirà in tempo per raggiungere il capolinea della corsa d ed effettuarla. Gli archi hanno capacità uno e costo proporzionale al tempo di viaggio tra il capolinea terminale di c ed il deposito più il tempo di viaggio tra il deposito ed il capolinea iniziale di d, calcolati nelle fasce orarie in cui avvengono (il tempo speso al deposito non conta in quanto i bus in deposito non sono "circolanti"), moltiplicato per un opportuno fattore γ_F. Ponendo $\gamma_I \neq \gamma_F$ si può esprimere una preferenza tra l'attesa in linea ed il rientro al deposito.

4. *Archi di rientro*: sono gli archi (c^+, d^-) per tutte le coppie di corse (d, c) nella chiusura della relazione di compatibilità, ossia tali che esiste un cammino $d = c_0, c_1, \ldots, c_h = c$ sul grafo degli archi di compatibilità (di entrambe i tipi). In altri termini, esiste l'arco di rientro tra c^+ e d^- se la corsa c potrebbe essere effettuata dallo stesso bus che ha *precedentemente* effettuato la corsa d. Tali archi hanno costo γ_M molto grande e capacità uno; alternativamente si può dire che questi archi hanno costo 1, mentre γ_I e γ_F sono $<< 1$, per cui è sempre preferibile usare un arco di rientro di meno, ove possibile, rispetto ad usare una qualsiasi combinazione di archi di compatibilità. Un'unità di flusso che circola sull'arco (c^+, d^-) indica che: *d è la prima corsa effettuata da un bus appena uscito dal deposito, e c è l'ultima corsa che lo stesso bus effettua prima di rientrare definitivamente (ossia non per effettuare una compatibilità fuori linea) al deposito.* Si noti che tra gli archi di rientro sono compresi anche gli "archi tripper" (c^+, c^-), che sono gli "opposti" agli archi di corsa, e che significano che un bus esce dal deposito per effettuare solamente la corsa c, e che vi rientra immediatamente dopo averla terminata senza più uscirne (all'interno dell'orizzonte T).

Tutti i nodi del grafo hanno deficit nullo; il problema è quindi una *circolazione*. È ovvio che qualsiasi flusso ammissibile sul grafo compone un insieme di cicli orientati disgiunti,

[1] γ_M è il costo di ogni vettura, come si può notare alla voce "Archi di rientro"

[2] TG_c, ovvero il *tempo giro* riferito alla corsa c, è il tempo impiegato ad effettuare la corsa stessa, una sosta minima al capolinea d'arrivo, la corsa di ritorno al capolinea di partenza e infine la sosta minima al capolinea di partenza

ciascuno dei quali rappresenta un TM. Si consideri ad esempio il frammento di grafo rappresentato in Figura 3 (non sono mostrati tutti gli archi, in particolare non sono rappresentati gli archi tripper ed è mostrato un unico arco di ritorno $(5^+, 1^-)$).

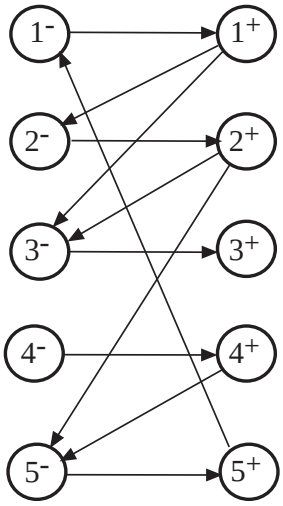

Figure 3: (parte di) Un grafo della compatibilità

Un'unità di flusso che circoli lungo il ciclo $(1^-, 1^+)$, $(1^+, 2^-)$, $(2^-, 2^+)$, $(2^+, 5^-)$, $(5^-, 5^+)$, $(5^+, 1^-)$ indica che un unico bus parte dal deposito, effettua per prima la corsa 1, poi la corsa 2 ed infine la corsa 5 prima di tornare al deposito. Il costo del ciclo è pari ad γ_m (se si escludono eventuali costi sugli archi $(1^+, 2^-)$ e $(2^+, 5^-)$ nel caso in cui rappresentassero compatibilità in linea), il che riflette il fatto che un solo bus è utilizzato per compiere tutte e tre le corse. In questo senso l'uso degli archi tripper è sconsigliato perché corrisponde ad un numero maggiore di bus per coprire lo stesso numero di corse.

Così come è descritto, questo modello minimizza le funzioni obiettivo 2.(a) e 2.(c) (vedi Paragrafo 2). È però immediato vedere che un opportuna scelta dei parametri di costo realizza un modello che minimizza la funzione obiettivo 2.(d). Se infatti si pone $\gamma_F = 1$, il costo degli archi di compatibilità fuori linea diviene superiore al costo degli archi di rientro. Poiché è sempre possibile rimpiazzare un arco di compatibilità fuori linea con un arco di rientro (sostanzialmente spezzando un TM formato da due viaggi in due TM formati ciascuno da un solo viaggio), è chiaro che in questo caso la soluzione ottima del problema non userà mai gli archi di compatibilità fuori linea; in effetti, in questo caso è possibile evitare del tutto di costruirli. Essendo quindi vietate le compatibilità fuori linea il concetto di TM e quello di viaggio coincidono, e la funzione obiettivo risulta quindi minimizzare il numero di viaggi come richiesto da 2.(d).

È abbastanza semplice intuire che la soluzione ottima di questo problema è *il flusso tutto nullo*, in quanto non c'è niente che costringa o renda conveniente coprire le corse (percorrere gli archi corsa), mentre il farlo comporta sicuramente un costo per via della necessità di usare gli archi di ritorno. È utile ricordare che il grafo MCF non è che una parte del modello; esiste infatti la parte di modello dedicata al timetabling (vedi Paragrafo 3.2), la cui soluzione sarà integrata (vedi Paragrafo 3.3) con quella ricavata dalla risoluzione del MCF. Inoltre potrebbero esserci *corse fissate*, ossia un sottoinsieme $CF \subset C$ di corse che devono

necessariamente essere effettuate. Ciò può essere facilmente ottenuto nel modello fissando ad *uno* il *flusso* sui corrispondenti archi corsa. Alternativamente (e più semplicemente) per ciascun $c \in CF$ si *elimina* l'arco corsa di c ma si pone deficit -1 su c^+ (trasformandolo in una sorgente di un'unità di flusso) e deficit 1 su c^- (trasformandolo in un pozzo di un'unità di flusso).

3.1.2 Eliminare gli archi di rientro

Un primo problema che si riscontra sul modello di base consiste nella creazione degli archi di rientro, che possono essere moltissimi: per via delle soste al deposito infatti quasi tutte le corse potrebbero essere compatibili con una corsa c che "termina molto tardi".

Per questo motivo si può costruire una variante del grafo della compatibilità in questo modo:

- si eliminano gli archi di rientro;
- si aggiungono due nodi O^- (uscita dal deposito) ed O^+ (ritorno al deposito);
- si creano archi *di inizio TM* (O^-, c^-) per ogni $c \in C$, di costo zero e capacità uno;
- si creano archi *di fine TM* (c^+, O^+) per ogni $c \in C$, di costo zero e capacità uno;
- si crea un unico arco di rientro (O^+, O^-) di costo γ_m e capacità infinita.

Un esempio della costruzione, costruito utilizzando lo stesso (frammento di) grafo mostrato in Figura 3, è illustrato in Figura 4.

Il ciclo precedentemente descritto è ora equivalentemente rappresentato da $(O^-, 1^-)$, $(1^-, 1^+)$, $(1^+, 2^-)$, $(2^-, 2^+)$, $(2^+, 5^-)$, $(5^-, 5^+)$, $(5^+, O^+)$, (O^+, O^-).

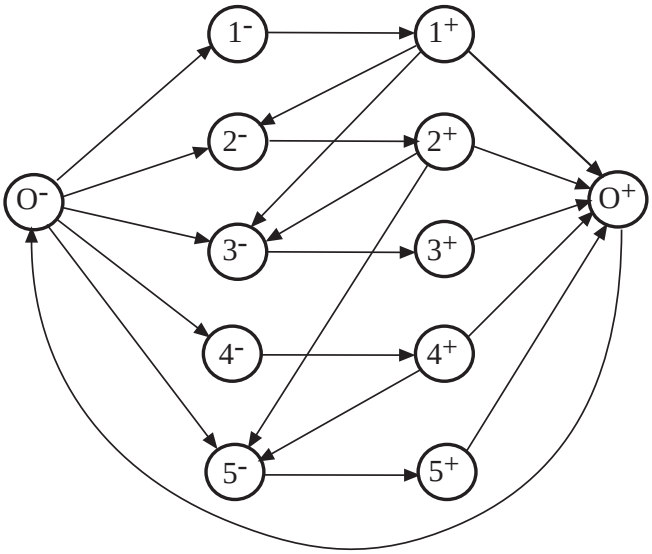

Figure 4: (parte di) Un grafo della compatibilità semplificato

Tale modello ha due nodi in più, ma tipicamente un numero di archi sostanzialmente inferiore. Un ulteriore vantaggio di questa soluzione consiste nella possibilità di inserire *vincoli di cardinalità della flotta* sotto forma della capacità dell'arco (O^+, O^-).

3.1.3 Eliminare gli archi di compatibilità fuori linea

Analogamente al caso precedente, gli archi di compatibilità fuori linea possono essere "molti", perché una corsa che "finisce presto" è compatibile fuori linea con quasi tutte le altre.

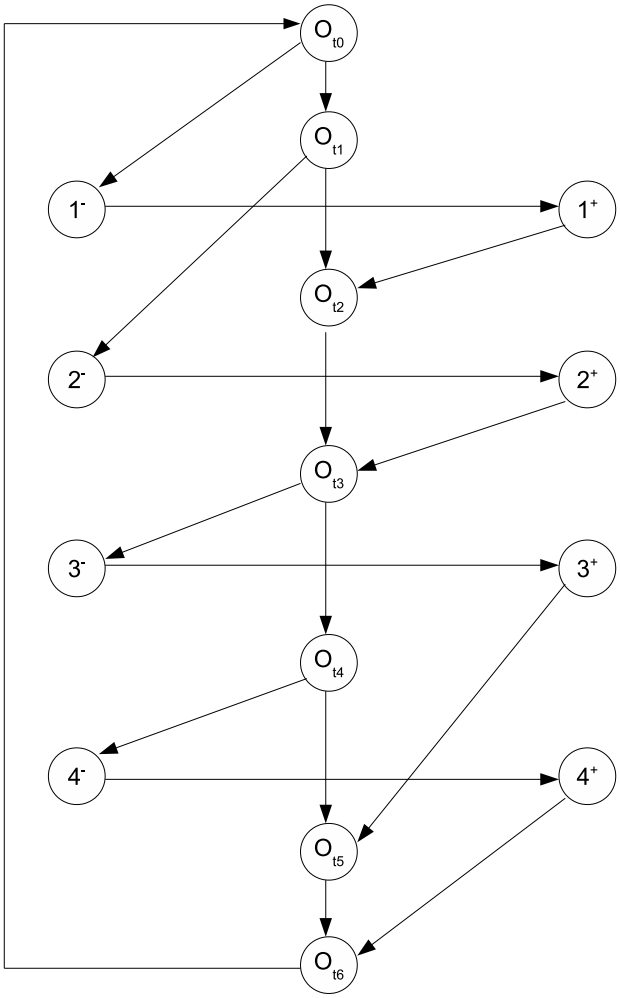

Figure 5: Possibile grafo con presenti i nodi O_t (per chiarezza sono esclusi gli archi di compatibilità in linea)

Sotto l'ipotesi (già peraltro assunta) che *non ci sia un massimo tempo di attesa al deposito*, ciò può essere fatto come segue (e come si può vedere nella Figura 5):

- Al posto dei soli O^- ed O^+, è creata una sequenza di nodi O_t corrispondenti ciascuno ad un istante $t \in T$. Non tutti gli istanti di tempo esistenti (può essere necessario utilizzare come granularità il secondo, quindi sarebbero molti istanti nell'arco di una giornata) sono effettivamente utili. Per ogni corsa c infatti occorre (ed è sufficiente) rappresentare al massimo due istanti come nodi O_t, uno per s_c e uno per e_c: la semantica di questi nodi è trattata nel prossimo punto. Ovviamente se il medesimo istante di tempo si presenta più volte (nel senso che è utile, per questo scopo, a più corse) verrà comunque

rappresentato nel modello da un nodo unico. Si identifica quindi $O^- = O_0$, $O^+ = O_{\bar{t}}$, e si mantiene l'arco di rientro (O^+, O^-).

- Gli archi di fine TM (c^+, O^+) sono eliminati.
- Gli archi di compatibilità fuori linea sono trasformati in archi "di ritorno al deposito" (c^+, O_t), dove t è il momento in cui il bus, dopo essere arrivato al deposito, è pronto per ripartire ($e_c + t^h_{n+} + \delta^{h'}_{min,O}$, dove h è la fascia oraria a cui appartiene e_c, n è il capolinea della corsa c, ed h' è la fascia oraria a cui appartiene $e_c + t^h_{n+}$), di capacità uno e costo proporzionale a t^h_{n+} (con fattore γ_F).
- Per ciascun $t \in T \setminus \{\bar{t}\}$ sono aggiunti archi (O_t, O_{t+1}) (dove "$t+1$" significa "il più piccolo istante $t' > t$ per cui è stato costruito un nodo $O_{t'}$"); tali archi hanno costo zero ma capacità infinita (o pari alla cardinalità delle flotta).
- Gli archi di inizio TM (O^-, c^-) sono eliminati. Per ciascun t, vengono aggiunti archi (O_t, c^-) per tutte le corse c tali che $t \leq s_c - t^h_{n+} < t + 1$ (ossia tali che il bus deve partire "nell'istante t" per poter effettuare la corsa c), sempre con capacità uno e costo proporzionale a t^h_{n+} (con fattore γ_F).

Gli archi di compatibilità fuori linea sono quindi rimpiazzati con cammini che descrivono esplicitamente il ritorno al deposito, l'attesa al deposito, e le possibili ripartenze. Ciò richiede l'introduzione di nuovi nodi (al massimo due volte il numero delle corse) e di un nuovo arco per ciascun nodo; in compenso per ciascun nodo c^+ c'è un solo arco di ritorno al deposito, che assume anche il ruolo dell'arco di fine TM, al posto di un insieme (presumibilmente grande) di archi di compatibilità.

3.1.4 Differenziare la compatibilità fuori linea dall'inizio e dalla fine dei TM

Un'ulteriore modifica del grafo prevede di introdurre una netta distinzione tra l'inizio e la fine di un turno macchina, ovvero il primo e l'ultimo viaggio di un bus dal (o verso il) deposito, e la compatibilità fuori linea che può esistere tra due corse.

Analogamente al caso precedente, sotto l'ipotesi sempre verificata che *non ci sia un massimo tempo di attesa al deposito*, ciò può essere fatto come segue:

- Al fianco della sequenza di nodi O_t si reintroducono i nodi O^- ed O^+. Ovviamente la precedente identificazione che prevedeva $O^- = O_0$, $O^+ = O_{\bar{t}}$ non vale più; si mantiene ancora l'arco di rientro (O^+, O^-), riferito quindi ai nuovi nodi reintrodotti.
- Si reinseriscono gli archi di inizio TM (O^-, c^+) e di fine TM (c^+, O^+), con capacità infinita e costo proporzionale al tempo di percorrenza, moltiplicato per un opportuno fattore γ_S.
- I nodi O^- e O^+ non sono collegati alla catena di nodi O_t se non tramite il passaggio dai nodi che formano gli archi-corsa

Quest'ultima versione del modello è un ibrido tra i due precedenti modelli sopra descritti. Reintroduce la differenziazione tra inizio (fine) TM e compatibilità fuori linea, aggiungendo un ulteriore parametro γ_S che affiancato al parametro γ_F rende differenziabili anche i costi dei relativi insiemi di archi.

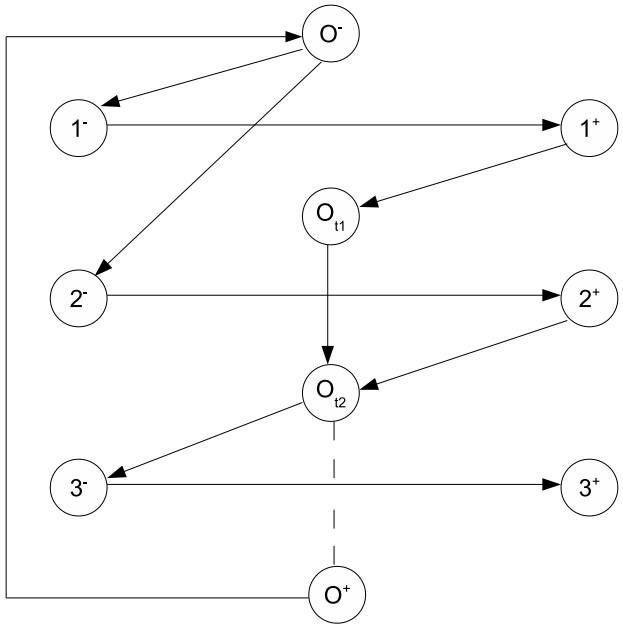

Figure 6: Possibile grafo con presenti sia i nodi O_t che i nodi O_- e O_+ (per chiarezza sono esclusi gli archi di compatibilità in linea)

3.2 Il (sotto)modello di determinazione delle corse (modello TT)

A differenza del modello di schedulazione dei bus, il modello per la determinazione dell'insieme di corse che rispettano in modo ottimale gli intertempi desiderati è *separabile per nodo pilota*. Ciò significa che per ciascun nodo pilota $n \in N$ si considerano le sole corse C_n che passano per esso; pertanto si risolvono *indipendentemente* tanti problemi quanti sono i nodi pilota (al massimo due, come vedremo). Per questo nel seguito di questo paragrafo considereremo il grafo relativo ad un singolo nodo pilota n, dando per inteso che il modello debba essere replicato per ciascun nodo pilota.

Il problema di determinazione dell'insieme di corse può essere formulato come un *cammino minimo* su di un opportuno grafo $G^n = (N^n, A^n)$, orientato ed aciclico. Il grafo è costruito come segue:

- I nodi corrispondono ciascuno ad una delle potenziali corse passanti per n, tranne per un nodo origine O ed un nodo destinazione D (quindi, $N^n = C_n \cup \{O, D\}$).

- In linea di principio, esiste un arco (c, d) tra due nodi ordinari (corse) se $d \in C_n$ passa per n dopo che vi è passato $c \in C_n$, ossia $s_c < s_d$. Tali archi hanno un costo che dipende dalla differenza tra i momenti p_d e p_c di passaggio dal nodo pilota delle due corse e dall'intertempo desiderato I_h della fascia oraria a cui appartengono p_d e p_c. Il costo sarà *zero* se $p_d - p_c = I_h$, e crescerà (molto velocemente) al crescere del valore assoluto di $p_d - p_c - I_h$. La specifica formula di costruzione dei costi è irrilevante per il modello, nel senso che essa è tranquillamente rimpiazzabile a seconda delle specifiche esigenze. La formula comunque deve tenere conto della possibilità che p_c e p_d appartengano a due fasce orarie diverse h ed h', tipicamente $h' = h + 1$: in aggiunta infatti è definito il

massimo valore di $|p_d - p_c - I_h|$ per cui l'arco (c, d) esiste, mentre per valori maggiori l'arco non è definito.

- Esistono archi tra O ed "i primi" nodi corsa c, così come esistono archi tra "gli ultimi" nodi corsa d e D. Opportune formule (anch'esse irrilevanti per la descrizione del modello e discusse nel Paragrafo 3.2.1) stabiliranno quale sia l'insieme dei nodi accettabili, e quale sia il costo di tali archi.

Il problema di determinare l'insieme di corse che rispettano in modo ottimale gli intertempi desiderati è quindi espresso come il problema di determinare il cammino minimo da O a D su G^n. Essendo questo un grafo orientato aciclico, l'algoritmo ha un costo $O(|A^n|)$ lineare sul numero di archi del grafo.

3.2.1 Le formule di costo degli archi

Le formule di costo per il generico (c, d) tra due nodi ordinari (corse) tali che $s_c < s_d$ sono determinate tenendo in considerazione le due possibili configurazione con le quali ogni fascia oraria può essere modellata. Ogni fascia oraria infatti può prevedere un intertempo desiderato che l'algoritmo cerca di soddisfare, oppure può non averlo; nell'ultimo caso le corse consecutive all'interno della fascia in questione potranno avere un qualsiasi intertempo. Le fasce orarie senza un intertempo desiderato sono determinate a livello di input ponendo a 0 l'intertempo desiderato.

Entrando nel dettaglio, consideriamo inizialmente il caso semplice in cui p_d e p_c fanno parte della stessa fascia h, ossia risulta $t_{h-1} \leq p_c \leq p_d \leq t_h$. In caso di intertempo desiderato pari a zero i costi degli archi sono sempre nulli. Nel caso invece di intertempi desiderati non nulli dobbiamo considerare innanzitutto che oltre all'intertempo desiderato I_h sono definiti dall'input anche l'*intertempo massimo* IM_h e *l'intertempo minimo* Im_h per la fascia; di conseguenza, l'arco è costruito solo se $Im_h \leq p_d - p_c \leq IM_h$. Qualora tale relazione sia soddisfatta, si deve verificare un'ulteriore condizione. È specificato in input un ulteriore parametro α, valido per tutte le fasce orarie, che fornisce un ulteriore filtro di ammissibilità: per ogni fascia h non vengono costruiti gli archi che hanno $\frac{p_d - p_c}{I_h} > \alpha$, anche se ammissibili per IM_h e Im_h[3].

Se l'arco rispetta entrambe le condizioni sopra descritte ed è quindi costruito, il suo costo è così calcolato:

- per prima cosa si calcola $\delta = |p_d - p_c - I_h|$;
- quindi

$$\frac{a\delta^2 + b\delta}{I_h} + \begin{cases} c & \text{se } \delta > 0 \\ 0 & \text{altrimenti} \end{cases} \tag{1}$$

.

I parametri a, b e c sono tutti maggiori di 0. In altre parole, il costo dell'arco è una funzione quadratica a carico fisso della deviazione relativa dall'intertempo desiderato. Adesso

[3]Il parametro α funge quindi da ulteriore controllo sul numero degli archi qualora i parametri IM_h e Im_h fossero stati inseriti con superficialità

consideriamo il caso in cui il viaggio "attraversa il confine tra due fasce", ossia risulti $t_{h-1} \leq p_c \leq t_h$ ma $t_h < p_d$ (si assume che risulterà sempre $p_d \leq t_{h+1}$). In questo caso l'arco è costruito se $\min\{Im_h, Im_{h+1}\} \leq p_d - p_c \leq \max\{IM_h, IM_{h+1}\}$.

Per il costo dell'arco si distinguono tre casi:

1. *Entrambe le fasce hanno un intertempo*: si calcola un nuovo intertempo I_{new}, relativo solo a questo arco, tenendo in considerazione i due intertempi delle due fasce originali ed una "percentuale di presenza" dell'arco all'interno delle stesse: in formule, detto t l'istante del cambio di fascia, allora

$$I_{new} = I_h \cdot \frac{t - p_c}{p_d - p_c} + I_{h+1} \cdot \frac{p_d - t}{p_d - p_c} \quad .$$

   Tramite I_{new} si calcola δ e quindi il costo seguendo la formula scelta.

2. *Una fascia ha un intertempo, l'altra no*: si usa la formula del caso normale calcolando δ tramite l'intertempo della fascia che ha un intertempo.

3. *Entrambe le fasce non hanno un intertempo*: il costo dell'arco è nullo.

Rimangono infine da inserire gli archi (O, c) e gli archi (d, D): determinando quali di questi archi esistano (e quale sia il loro costo) si pongono le basi per sciegliere la prima e l'ultima corsa della giornata passante per il nodo pilota. Per ogni nodo pilota sono definiti sei diversi coefficienti per il costo degi archi, ossia la terna a_i, b_i e c_i per la parte iniziale e a_f, b_f e c_f per la parte finale. Sono inoltre fissati il cosidetto *range di ammissibilità iniziale* r_i e il cosidetto *range di ammissibilità finale* r_f. Detto N_o l'insieme dei nodi pilota si calcola, $\forall n \in N_o$, t_n come l'istante iniziale della prima fascia oraria per la quale sul nodo pilota n è richiesto un intertempo desiderato non nullo, per poi calcolare $t = min(t_n : n \in N_o)$.

Quindi $\forall n \in N_o$ si procede come segue.

- Se $t_n = t$ vi sarà un solo intervallo di ammissibilità. Si calcola, per ogni corsa, δ come la differenza tra t e l'orario di passaggio della corsa dal nodo pilota. Si crea quindi l'arco verso la corsa se e solo se δ rientra nel range d' ammissibilità iniziale: il costo dell'arco è quindi calcolato mediante l'uso di δ e dei coefficienti a_i, b_i e c_i con modalità analoghe al calcolo del costo degli archi "normali".

- Se t_n è successivo a t, allora sono previsti effettivamente due intervalli di ammissibilità di natura diversa. Il primo intervallo $[t, t_n - r_i[$ contiene archi a costo nullo, mentre il secondo intervallo $[t_n - r_i, t_n + r_i]$ contiene archi con costo calcolato come sopra, tramite δ e la terna di parametri a_i, b_i e c_i prendendo però come riferimento t_n.

La descrizione della formulazione degli intervalli temporali e dei costi degli archi per la parte finale è analoga a quella della parte iniziale, quindi viene omessa.

3.3 Il modello complessivo

Il modello complessivo si ottiene semplicemente considerando:

- il modello di schedulazione dei bus;
- i modelli (in questa versione del modello sono due) di determinazione delle corse per nodo pilota $n \in N$;
- *vincoli accoppianti* che legano tutti i modelli tra loro.

Il modello può essere descritto come un unico grafo composto da tre grafi disgiunti sul quale calcolare un flusso di costo minimo che deve rispettare, oltre ai classici vincoli che ne definiscono la natura, i vincoli accoppianti sopra citati. In realtà, come abbiamo visto e come rivedremo, i modelli di determinazione delle corse (per nodo pilota) sono risolvibili calcolando su di essi il cammino minimo.

Il modello descrive un problema di *Programmazione Lineare Intera*.

$$\min k_{BS}x + k_{TT_1}y + k_{TT_2}z \tag{2}$$

$$\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = b_i \qquad i \in N_{BS} \tag{3}$$

$$\sum_{(j,i)\in BS(i)} y_{ji} - \sum_{(i,j)\in FS(i)} y_{ij} = b_i \qquad i \in N_{TT_1} \tag{4}$$

$$\sum_{(j,i)\in BS(i)} z_{ji} - \sum_{(i,j)\in FS(i)} z_{ij} = b_i \qquad i \in N_{TT_2} \tag{5}$$

$$0 \le x_{ij} \le u_{ij} \qquad (i,j) \in A_{BS} \tag{6}$$

$$y_{ij} \in \{0,1\} \qquad (i,j) \in A_{TT_1} \tag{7}$$

$$z_{ij} \in \{0,1\} \qquad (i,j) \in A_{TT_2} \tag{8}$$

$$\sum_{a\in BS(c)} y_a = x_c \qquad c \in \overline{N}_{TT_1} \tag{9}$$

$$\sum_{a\in BS(c)} z_a = x_c \qquad c \in \overline{N}_{TT_2} \tag{10}$$

Nella formulazione sopra riportata le tre componenti adesso rappresentate da due distinti nodi (BS, TT_1, TT_2) del grafo sono state descritte in maniera disgiunta per una migliore comprensione. Il sottografo BS è perciò composto dai nodi N_{BS} e dagli archi A_{BS} sulle quali sono state fissate le variabili x_{ij} e analogamente gli altri sottografi. L'insieme $\overline{N}_{TT_1}$ invece (e analogamente l'insieme $\overline{N}_{TT_2}$) rappresenta l'insieme dei nodi del sottografo TT_1 privato del relativo nodo radice e del relativo nodo pozzo. La funzione obiettivo (2) è un costo tripartito (poiché esistono tre insiemi di variabili) da minimizzare. I primi tre insiemi di vincoli (3), (4) e (5) osservabili sono i cosidetti *vincoli di bilanciamento*, tipici di un problema di flusso di costo minimo; l'insieme di vincoli (6) sono *di capacità* mentre gli insiemi (7) e (8) impongono ai sottografi di tipo TT di usare variabili binarie. Infine gli ultimi due insiemi di vincoli (9) e (10) sono i sopracitati vincoli accoppianti. I vincoli accoppianti costringono i tre

sottomodelli, di per sé indipendenti, a trovare soluzioni compatibili tra di loro; in particolare ogni corsa deve risultare coperta (o non coperta) in ogni soluzione di ogni sottoproblema. La descrizione di tali vincoli è complicata dalla diversa rappresentazione che le corse stesse hanno nei vari sottomodelli: sono archi nel modello di schedulazione dei veicoli, sono nodi nel modello di determinazione delle corse.

Nella formulazione data ogni corsa è un nodo di TT_1 e quindi un elemento dell'insieme $\overline{N}_{TT_1}$ oppure è un nodo di TT_2 e quindi un elemento dell'insieme $\overline{N}_{TT_2}$, mentre sul sottografo BS essa è rappresentata da un arco x_c. Il singolo vincolo d'accoppiamento sulla corsa c impone quindi che il flusso sull'arco "corsa" x_c, forzatamente 0 o 1, sia pari alla somma dei flussi sugli archi entranti nel nodo y_c (o z_c).

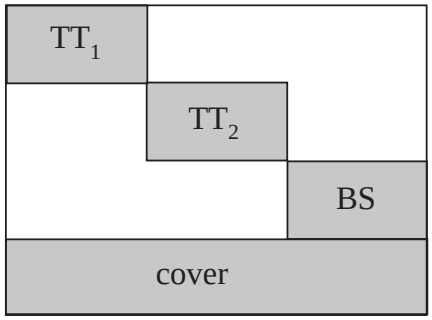

Figure 7: Il problema

Per comprendere meglio quanto grande può essere il problema presentato basta osservare la modellazione di un caso reale, uno di quelli sui quali faremo girare il codice. L'istanza reale presa in considerazione prevede 2642 possibili corse da poter scegliere; i due grafi TT sono composti entrambi da 1323 nodi e 27168 archi, mentre il grafo BS è composto da 6697 nodi e 74809 archi. Questi dati, trasportati nel modello sopra descritto, portano ad avere 129145 variabili e altrettanti vincoli di capacità / binari, 9339 vincoli di bilanciamento, 2642 vincoli accoppianti. È però la presenza dei (2642) vincoli accoppianti che implica, purtroppo, un'alta complessità computazionale rendendo quindi il problema globale di difficile risoluzione. Questa difficoltà non è incontrata solo dal nostro codice: come vedremo nei successivi paragrafi anche un software potente, specializzato nella risoluzione di problemi matematici, come *Cplex* non riesce a ridurre il gap tra *lower bound* e l'*upper bound* (vedremo cosa questo significhi nel Paragrafo 4.3) ad un livello che si può considerare accettabile.

3.4 L'approccio algoritmico

L'approccio algoritmico scelto per la risoluzione del problema può essere visto come innovativo all'interno del particolare settore entro il quale stiamo operando. Come già accennato nell'introduzione, per la risoluzione del problema combinato "timetabling + scheduling dei veicoli" sono emersi storicamente due tipi di approcci:

- risoluzione sequenziale dei sottoproblemi: in generale gli algoritmi che seguono questa politica prima trovano un insieme di corse che risolve al meglio il problema del timetabling, quindi costruisono il miglior scheudling dei veicoli possibile per coprire le corse che fanno parte della soluzione trovata per il timetabling;

- risoluzione "simultanea" dei due sottoproblemi.

L'approccio scelto fa parte sicuramente della seconda categoria, differenziandosi però dai lavori precedentemente presentati abbastanza nettamente. Nel lavoro di VanDerHakken [VVV08] ad esempio è presente un unico grafo chiamato *time-space network*, in certi aspetti simile al nostro grafo per il Bus Scheduling, sul quale sono calcolati sia lo scheduling che il timetabling attraverso la risoluzione su di esso di un problema di flusso di costo minimo. La ricerca della soluzione migliore si svolge tramite un processo iterativo dove, ad ogni iterazione, viene calcolata una soluzione che fornisce un certo timetabling e un certo scheduling e che fornisce una base di partenza per l'iterazione successiva: l'algoritmo in questione infatti applica un procedimento del tipo *ricerca locale* modificando di poco la soluzione precedentemente trovata attraverso lo spostamento di qualche minuto in avanti o all'indietro di una linea di un bus (quindi piccole modifiche alla timetabling), ricalcolando la soluzione e infine confrontandola con quelle già trovate. Una logica basata sulla ricerca locale iterativa volta a migliorare la soluzione tramite progressive piccole permutazioni di soluzioni già trovate è adottata anche nel lavoro di Hao [HG08].

Il modello adottato qui invece distingue nettamente i due sottoproblemi tramite la creazione di grafi distinti e la risoluzione separata di sottoproblemi diversi (un flusso di costo minimo e due cammini minimi). Questa particolarità è dovuta principalmente al fatto che sul problema, per come è stato qui formulato, è possibile adottare una tecnica Lagrangiana di rilassamento dei vincoli: il problema infatti è decomponibile in $|N| + 1$ problemi indipendenti ($|N|$ cammini minimi ed un MCF, esattamente come sono stati descritti nei precedenti paragrafi) a bassa complessità rilassando i vincoli accoppianti, ovvero non considerando più gli stessi come vincoli ma come elementi da aggiungere al computo della funzione obiettivo del problema stesso.

In tal senso occorre formare il cosidetto *rilassamento Lagrangiano* (il problema che sarà trattato) associando a ciascuno dei vincoli acoppianti un moltiplicatore λ_c e aggiungendo il corrispondente termine

$$\sum_{c \in C} \lambda_c \left(\sum_{a \in BS(c)} y_a \; - \; x_c \right)$$

alla funzione obiettivo, e rimuovendo proprio i vincoli accoppianti dalla formulazione.

Fissato il valore del vettore $\lambda = [\lambda_c]_{c \in C}$ di moltiplicatori Lagrangiani, per la linearità del termine precedentemente introdotto nella funzione obiettivo il problema si decompone nuovamente in $|N| + 1$ problemi indipendenti.

Tali problemi sono identici a quelli descritti nei Paragrafi 3.1 e 3.2, tranne per il fatto che (per ciascun $c \in C$)

- al costo di tutti gli archi $a \in BS(c)$ nei problemi di cammino minimo è *aggiunto* λ_c;
- il costo dell'arco corsa (c^-, c^+) nel problema di schedulazione dei veicoli diviene $-\lambda_c$.

Una volta formato il rilassamento Lagrangiano occorre risolvere (almeno approssimativamente) il corrispondente *duale Lagrangiano*. Il problema del duale Lagrangiano consiste nel determinare il vettore λ^* dei moltiplicatori Lagrangiani in modo da massimizzare la funzione

obiettivo del problema rilassato (che è una valutazione inferiore sul valore della funzione obiettivo del problema originale). L'approccio algoritmico è basato su uno schema iterativo: il *solutore del Duale Lagrangiano* fissa un vettore $\bar{\lambda}$ che modifica il costo di alcuni archi, come è stato detto sopra, nei grafi che descrivono i vari sottoproblemi.

A questo punto il *solutore del rilassamento* deve risolvere il rilassamento Lagrangiano, e fornire al solutore del Duale

- il valore ottimo del rilassamento, ottenibile semplicemente sommando i valori ottimi di tutti i sottoproblemi come riportati dai rispettivi solutori;
- un *subgradiente*, ossia il vettore

$$\left[\sum_{a \in BS(c)} y_a^* \; - \; x_c^* \right]_{c \in C}$$

  dove x^* e y^* denotano le soluzioni ottime dei sottoproblemi (per il fissato $\bar{\lambda}$).

Partendo da questi elementi il solutore del Duale determina un miglior vettore $\bar{\lambda}$ per poi giungere iterativamente a (una buona approssimazione di) λ^*. Da notare come tipicamente, durante il processo iterativo, i moltiplicatori Lagrangiani di molte corse risulteranno positivi, il che corrisponderà all'"apparizione" di archi di costo negativo nel MCF; ancora una volta, questo è un elemento che "forza" (o almeno rende possibile) la circolazione del flusso nel modello di schedulazione dei bus. Questo approccio determina solo una valutazione inferiore sul valore ottimo del problema, ma non una soluzione ammissibile (che è invece una valutazione superiore sull'ottimo). Per questo sono necessarie *euristiche* che determinano (buone) soluzioni ammissibili per il problema globale sfruttando le soluzioni generate dai sottoproblemi ad ogni iterazione. Le soluzioni dei sottoproblemi tipicamente non rispetteranno i vincoli accoppianti (se lo facessero si otterrebbe il subgradiente nullo e si sarebbe determinata la soluzione ottima del problema globale), pertanto è necessario modificarle euristicamente in modo tale da ottenere il rispetto dei vincoli accoppianti.

Una soluzione ammissibile si può trovare in molti modi. Ad esempio:

- Utilizzando la soluzione del MCF per determinare il costo (in termini di soddisfacimento degli intertempi) dei TM contenuti in tale soluzione. Tale costo potrebbe risultare (e tipicamente risulterà, almeno nelle prime iterazioni) *infinito*, nel senso che i TM risultano in corse "troppo distanziate" e la soluzione del sottoproblema dello scheduling non sarà compatibile coi vincoli che formano i sottoproblemi delle frequenze: in questo caso non si può parlare di soluzione ammissibile. Questo ad esempio è tipicamente vero alla prima iterazione, in cui $\bar{\lambda} = 0$, perché in questo caso nessun bus circola.
- Determinando i TM che meglio coprono le corse decise dai cammini minimi. Ciò richiede di risolvere un modello analogo a quello del Paragrafo 3.1 ma in cui un sottoinsieme delle corse è fissato, e le altre sono eliminate. Si tratta quindi di un MCF di dimensioni ridotte rispetto a quello che viene comunque risolto ad ogni iterazione. Questa tecnica, pur risultando più costosa della precedente, ha il vantaggio di garantire

la produzione di una soluzione ammissibile ad ogni iterazione qualora i vincoli inseriti nella fase di *Controllo del flusso* (Paragrafo 6) fossero ragionevoli o che non fossero inseriti.

- Utilizzando schemi più sofisticati in cui si conservano le soluzioni dei sottoproblemi di ogni iterazione per costruirne versioni "convessificate" con opportuni moltiplicatori prodotti dal solutore del Duale e quindi fissare iterativamente variabili (corse) in modo opportuno. In questa maniera si giungerebbe più velocemente ad una buona soluzione per il problema originale, in quanto fissare le corse comporta una riduzione delle dimensioni dei sottoproblemi (con evidenti vantaggi computazionali).

3.5 Sviluppi previsti del modello

Come già detto all'inizio del Paragrafo 3, il modello base risolve una versione semplificata del problema. Nel momento in cui furono delineate quali parti del problema generale "inserire" nella versione semplificata, furono comunque discussi a livello teorico alcuni degli obiettivi che non erano stati inclusi nella prima versione del modello.

In particolare furono analizzate le seguenti possibili estensioni del modello:

1. La formazione di *Turni Guida* in contemporanea ai TM, il che rende il problema di schedulazione dei bus non più risolubile attraverso il semplice modello di MCF presentato nel Paragrafo 3.1. In linea di principio la formazione di TG viene affrontata inserendo opportuni vincoli nei TM in modo da rendere coincidenti, a meno di facili elaborazioni, i due concetti.

2. Il trattamento di depositi multipli.

3. L'esistenza eventuale di *frequenze "frazionarie"* rispetto ad un dato *fattore di scala*.

4. Il trattamento di linee di forma più complessa, come quelle mostrate in Figura 10.

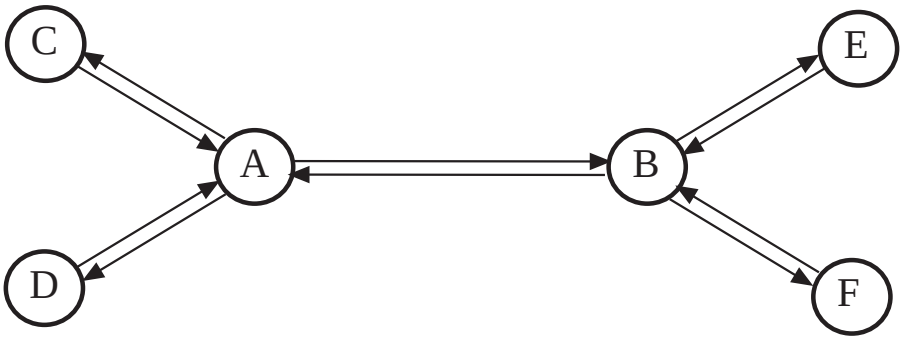

Figure 8: Una linea "complessa"

3.5.1 Formazione di Turni Guida

L'analisi teorica affermò che per questo particolare punto sarebbe bastato rimpiazzare la parte "MCF" con un opportuno modello (di generazione di colonne o altro) in grado di tenere in considerazione gli ulteriori vincoli sui TM. In questo caso l'analisi teorica non ha avuto alcuna conferma a livello sperimentale.

3.5.2 Depositi Multipli

L'analisi teorica affermò che il caso in cui le corse possano essere effettuate da bus provenienti da più depositi diversi sarebbe potuto essere integrato facilmente nel modello base. Poiché i concetti espressi all'interno di questa particolare analisi non sono stati implementati e quindi sperimentati, essi restano senza una conferma pratica ma anche senza una smentita e quindi ancora plausibili. Dato D l'insieme dei depositi, potrebbe infatti bastare la definizione di un diverso problema di schedulazione per ciascuno dei depositi $o \in D$.

Il problema complessivo si potrebbe quindi modellare come fosse il caso standard, eccezion fatta per:

- la presenza di più di un problema di schedulazione (MCF), e quindi di diversi vettori x^o che rappresentano le variabili di flusso corrispondenti a ciascun deposito $o \in D$;
- i vincoli accoppianti divengono

$$\sum_{a \in BS(c)} y_a \; = \sum_{o \in D} x_c^o \qquad\qquad c \in C \; . \tag{11}$$

  in quanto ciascuna corsa deve essere effettuata da un solo bus, indipendentemente dal deposito da cui proviene.

L'approccio algoritmico basato sul Rilassamento Lagrangiano si estenderebbe quindi facilmente, almeno per quanto riguarda la computazione della valutazione inferiore. Da notare che, una volta fissato un insieme di corse, il problema di determinare la miglior schedulazione diverrebbe un flusso Multicommodity intero, quindi un problema NP-hard.

Utilizzando l'approccio Lagrangiano infatti non è detto che (anzi, tipicamente ci si può aspettare che non) risulti

$$\sum_{a \in BS(c)} x_c^o \leq 1 \;\; .$$

In altri termini, ci possono essere corse coperte da più di un bus. Chiaramente è necessario che questo venga evitato per costruire una soluzione ammissibile, e ciò richiederebbe l'utilizzo di opportune euristiche.

3.5.3 Frequenze "frazionarie"

Un'ulteriore possibile sofisticazione del modello riguarda la possibilità di specificare un *fattore di scala* F_s globale e di richiedere che le frequenze siano multipli interi di questo fattore di scala. Sono valori ragionevoli per F_s, ad esempio, 60, 30, 15, 10, 5 e 1 secondi. Può però capitare che l'intertempo desiderato per un certo nodo pilota in una certa fascia oraria non

sia un multiplo del fattore di scala globale definito. Ad esempio si può ipotizzare che il fattore di scala F_s sia 60 (1 minuto), che nell'insieme delle corse possibili la distanza tra una corsa e la sua successiva (misurata sul nodo pilota) sia anch'essa di 1 minuto, che l'intertempo desiderato I_h sia 270 (4.5 minuti). In questo caso non esisterebbe nessun arco a costo nullo nel grafo: nessuna corsa potrebbe essere distante 4,5 minuti da un'altra, anche perché le corse possibili sono distanziati in termini di minuti interi. In questo scenario scegliere corse distanziate di 4 minuti costerebbe uguale a scegliere corse distanziate 5 minuti: gli archi relativi avrebbero lo stesso costo. Ciò porterebbe, per esempio, a considerare sequenze di intertempi quali $4, 4, 4, 5, 5, 5$ qualitativamente identiche a sequenze del tipo $4, 5, 4, 5, 4, 5$. L'esperienza "trasportistica" insegna invece che la seconda sequenza è più desiderabile per l'utente: così facendo le corse "pari" (la seconda con la quarta, etc.) e analogamente le corse "dispari" hanno maggiori possibilità di avere come intertempo $540 = 270 * 2$ secondi (9 minuti), ossia un multiplo intero ($540/60 = 9$) dell'intertempo desiderato. Occorre quindi estendere il (sotto)modello di determinazione delle corse in modo da riuscire a trovare tipi di soluzioni aderenti agli schemi che gli utenti ritengono migliori. Supponendo che $F_s > 1$, una data fascia oraria h può accadere che I_h/F_s non sia un numero intero, ossia $I_h/F_s = Q*F_s+R$ con $R > 0$.

Si definisce $\alpha = \frac{R}{F_s}$, $I_h^- = Q * F_s$ e $I_h^+ = I_h^- + F_s$; così facendo risulterà sempre $\alpha < 1$. A seconda del valore di α si possono sviluppare diversi schemi. L'esempio fatto inizialmente (con intertempo desiderato uguale a 270 secondi) comporterebbe che $\alpha = 0.5$.

Nel caso invece in cui $F_s = 60$ $(1m)$ e $I_h = 260$ allora $I_h^- = 240$ $(4m)$ ed $\alpha = \frac{1}{3}$; ciò significa che gli intertempi desiderati sono ripetizioni della sequenza $240, 240, 300$ ($4m$, $4m$, $5m$) con la quale si ottiene che $\frac{240+240+300}{260} = 3$.

Se invece $F_s = 60$ e $I_h = 280$ allora $I_h^- = 240$ $(4m)$, ma stavolta $\alpha = \frac{2}{3}$; ciò significa che gli intertempi desiderati sono ripetizioni della sequenza $240, 300, 300$ ($4m$, $5m$, $5m$) tale che $\frac{240+300+300}{280} = 3$.

Permettere l'utilizzo di questi schemi comporta una modifica sostanziale del grafo su quale calcolare il cammino minimo. Il costo di attraversare un certo arco non è più fisso, ma dipende ora anche da quali archi sono stati attraversati precedentemente durante il cammino. Considerando l'esempio della sequenza "$4, 5$", il costo di un arco corrispondente ad un intertempo 5 è nullo se l'arco precedente nel cammino ha intertempo 4, mentre è > 0 se l'arco precedente nel cammino ha intertempo 5, e viceversa. In figura 9 qui sotto è mostrato l'esempio che prevede la sequenza "$4, 5$", quindi con $\alpha = 0.5$. Nella parte alta della Figura 9 è mostrato un frammento del grafo dell'SPT "classico", ossia corrispondente a $F_s = 1$. Sono visualizzate una corsa che passa per il nodo pilota all'istante t, e corse che vi passano agli istanti $t+3$, $t+4$, $t+5$ e $t+6$ (il displacement è misurato in minuti). Dato che $I_h = 270$ $(4.5m)$, entrambi gli archi da $(t, t+4)$ e $(t, t+5)$ hanno costo non nullo (pari ad una deviazione di 30 secondi dall'intertempo desiderato); ovviamente il costo degli archi $(t, t+3)$ e $(t, t+6)$ è maggiore (pari ad una deviazione di 90 secondi dall'intertempo desiderato).

Nella parte bassa della Figura 9 è invece mostrato un frammento del grafo dell'SPT "adattato" al caso delle frequenze frazionarie. Ciascuna corsa (caratterizzata dall'istante t di passaggio per il nodo pilota) è adesso rappresentata da due distinti nodi: $(t, 4)$ e $(t, 5)$.

Il significato di questi è:

- se il cammino passa per $(t, 4)$ viene effettuata la corsa corrispondente sapendo che la

corsa immediatamente precedente ha avuto da questa un'intertempo *pari o inferiore* al primo intertempo della sequenza desiderata, ossia a $4m$;

- se il cammino passa per $(t, 5)$ viene effettuata la corsa corrispondente sapendo che la corsa immediatamente precedente ha avuto da questa un'intertempo *pari o superiore* al secondo intertempo della sequenza desiderata, ossia a $5m$.

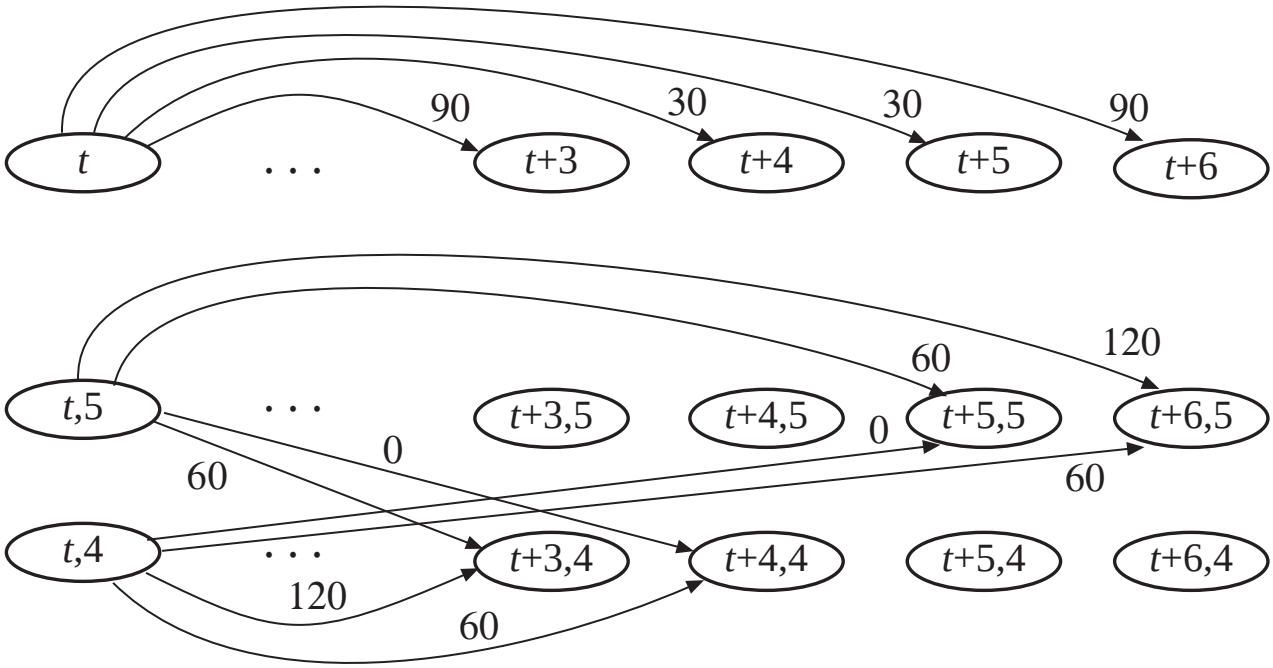

Figure 9: Modifica all'SPT per gestire le frequenze frazionarie (sono rappresentati solo alcuni archi)

Esaminiamo tutti gli archi del (frammento di) grafo per verificarne il senso:

- L'arco $((t, 4), (t+3, 4))$ indica che dopo aver effettuato la corsa (che passa al nodo pilota al tempo) t si effettua la corsa (che passa al nodo pilota al tempo) $t+3$. Poiché la corsa precedente aveva avuto un intertempo di $4m$ o inferiore, questa corsa ha un intertempo desiderato pari a $5m$; di conseguenza la violazione dell'intertempo è di $2m = 120s$. Il prossimo intertempo desiderato sarà $5m$.

- L'arco $((t, 5), (t + 3, 4))$ indica ancora che dopo aver effettuato la corsa t si effettua la corsa $t + 3$. Siccome la corsa precedente aveva avuto un intertempo di $5m$ o superiore, questa corsa ha un intertempo desiderato pari a $4m$ e di conseguenza la violazione dell'intertempo è di $1m = 60s$. Il prossimo intertempo desiderato sarà $5m$.

- L'arco $((t, 4), (t+4, 4))$ indica che dopo aver effettuato la corsa (che passa al nodo pilota al tempo) t si effettua la corsa (che passa al nodo pilota al tempo) $t+4$. Poiché la corsa precedente aveva avuto un intertempo di $4m$ o inferiore, questa corsa ha un intertempo desiderato pari a $5m$; di conseguenza la violazione dell'intertempo è di $1m = 60s$. Il prossimo intertempo desiderato sarà $5m$.

- L'arco $((t, 5), (t + 4, 4))$ indica ancora che dopo aver effettuato la corsa t si effettua la corsa $t + 4$. La corsa precedente aveva avuto un intertempo di $5m$ o superiore, quindi questa corsa ha un intertempo desiderato pari a $4m$. Di conseguenza questa corsa ha esattamente l'intertempo desiderato e il suo costo è 0. Il prossimo intertempo desiderato sarà $5m$.

- L'arco $((t, 4), (t+5, 5))$ indica che dopo aver effettuato la corsa t si effettua la corsa $t+5$. La corsa precedente aveva avuto un intertempo di $4m$ o inferiore, quindi questa corsa ha esattamente l'intertempo desiderato di $5m$e il suo costo è 0. Il prossimo intertempo desiderato sarà $4m$.

- L'arco $((t, 5), (t + 5, 5))$ indica ancora che dopo aver effettuato la corsa t si effettua la corsa $t + 5$. La corsa precedente aveva avuto un intertempo di $5m$ o superiore, quindi questa corsa ha un intertempo desiderato pari a $4m$, che viene quindi violato di $1m = 60s$. Il prossimo intertempo desiderato sarà $4m$.

- L'arco $((t, 4), (t + 6, 5))$ indica che dopo aver effettuato la corsa t si effettua la corsa $t + 6$. Poiché la corsa precedente aveva avuto un intertempo di $4m$ o inferiore, questa corsa ha un intertempo desiderato di $5m$, che viene quindi violato di $1m = 60s$. Il prossimo intertempo desiderato sarà $4m$.

- L'arco $((t, 5), (t + 6, 5))$ indica ancora che dopo aver effettuato la corsa t si effettua la corsa $t + 5$. La corsa precedente aveva avuto un intertempo di $5m$ o superiore, quindi questa corsa ha un intertempo desiderato pari a $4m$, che viene quindi violato di $2m = 120s$. Il prossimo intertempo desiderato sarà $4m$.

- Non esistono gli archi $((t, 4), (t + 5, 4))$ e $((t, 5), (t + 5, 4))$ in quanto questi archi indicherebbero che dopo l' intertempo desiderato sarebbe $5m$, quando invece sarebbe giusto aspettarsi $4m$.

- Non esistono gli archi $((t, 4), (t + 4, 5))$ e $((t, 5), (t + 4, 5))$ in quanto questi archi indicherebbero che dopo l' intertempo desiderato sarebbe $4m$, quando invece sarebbe giusto aspettarsi $5m$.

Chiaramente l'esempio sopra analizzato non è vincolante rispetto alle potenzialità di questo tipo di approccio. Si può infatti, come vedremo nel Paragrafo 7, aumentare il numero di nodi che rappresentano una singola corsa ben oltre la soglia di 2 qui proposta in modo da aumentare così l'espressività del modello e quindi il numero di situazioni (ovvero il numero di tipologie di intertempi) descrivibili.

3.5.4 Trattamento di linee complesse

In questo caso l'analisi teorica fatta nel momento della definizione del modello base definisce un'estensione del modello che non coincide con l'estensione realizzata e quindi implementata in un secondo momento. L'estensione effettivamente realizzata (vedi Paragrafo 8), di natura abbastanza dissimile rispetto all'approccio presentato in questo Paragrafo, è stata infatti costruita in base ad uno nozione di schema che qui non sarà documentata (ma sarà

documentata nel relativo Paragrafo) e che, al momento della stesura del modello base, non era emerso. La presentazione delle ipotesi fatte in precedenza e poi non implementate non è comunque un esercizio didascalico fine a se stesso, sia perché mostrano in un certo misura il grado di versatilità del modello ideato sia (e soprattutto) perché le considerazioni qui sotto riportate possono in un eventuale futuro essere la base di partenza per lo sviluppo di un modello diverso da quello successivamente realizzato e implementato qualora fosse rituenuto necessario.

L'analisi teorica mise in luce come fosse possibile modellare una "linea complessa" (come quella in Figura 10) semplicemente assumendo che, al posto di avere tre modelli legati dai vincoli accoppianti come abbiamo nel caso semplice, ce ne possano essere di più per via della presenza di un maggior numero (fino a 10 nell'esempio in Figura 10) di sottolinee, con l'ovvia conseguenza di dover gestire un modello più ampio dove, a fronte di un unico MCF, vi sono molti cammini minimi. Sarebbe semplice infatti inserire nel modello *vincoli di intertempo su sottoinsiemi arbitrari di corse*, indipendentemente dall'effettivo arrangiamento topologico delle stesse. Il fatto che ogni sottolinea sia definita come l'insieme di corse in partenza da un certo capolinea sarebbe del tutto irrilevante semplicemente perché una siffatta estensione del modello necessita solo dell'indicazione di come l'insieme C delle corse disponibili sia suddiviso in un numero arbitrario k di sottoinsiemi, ossia $C = C^1 \cup C^2 \cup \ldots \cup C^k$. Definiti i sottoinsiemi di corse si possono scrivere i vincoli accoppianti e quindi applicare l'approccio algoritmico. Da notare che la stessa corsa $c \in C$ può far parte di più sottoinsiemi diversi, sottostando quindi a più vincoli.

Considerando ad esempio la rete rappresentata in Figura 10, dove le corse sono definite sui 4 cammini C — A — B — E, C — A — B — F, D — A — B — E, D — A — B — F, l'insieme delle corse sarebbe quindi "naturalmente" partizionato in 8 sottoinsiemi (ciascun cammino ha corse nelle due direzioni). Ciò è possibile usando come punti di osservazione C, D, E ed F (usati in entrambe i versi).

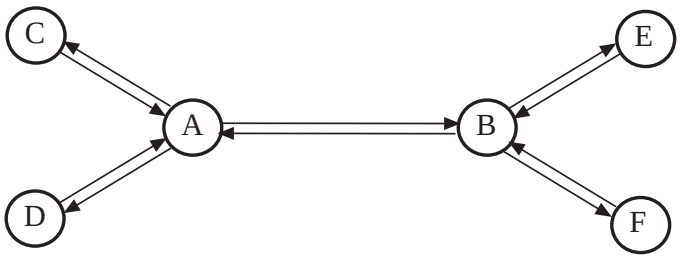

Figure 10: Una linea "complessa"

Nel caso in cui invece si voglia (come è ragionevole) tenere sotto controllo anche la frequenza nel tratto condiviso A — B, si possono utilizzare A e/o B come ulteriori punti di osservazione, definendo i sottoinsiemi di corse che passano per essi (tutte). Il meccanismo di selezione degli insiemi di corse, così come è stato presentato, è limitato: ad esempio, non si può tenere sotto controllo la frequenza delle corse che vanno da C ad E escludendo quelle che vanno da D ad E. Sarebbe possibile però definire nodi non capolinea, eventualmente anche "fantasma", con l'unico scopo di aiutare a partizionare l'insieme delle corse. Nonostante questa limitazione, l'ipotetico modello avrebbe un alto potere espressivo. Ad esempio è interessante notare che una linea complessa può essere *descritta* in modi diversi.

In particolare, facendo riferimento all'esempio in Figura 10:

1. come le 5×2 sottolinee semplici C — A, D — A, A — B, B — E, B — F;
2. come le 4×2 sottolinee "complesse" C — A — B — E, C — A — B — F, D — A — B — E, D — A — B — F.

Ai fini del modello, le due descrizioni non sono equivalenti, nel senso che:

- Nel caso 1, i nodi intermedi (A e B) sono capolinea. Ciò significa che il modello può scegliere la lunghezza delle soste che i bus fanno in tali nodi. Il caso in cui essi non siano "veri" capolinea, nel senso che non sia possibile per i bus effettuare (lunghe) soste in corrispondenza ad essi, può essere modellato semplicemente dando ad $\delta^h_{max,A}$ e $\delta^h_{max,B}$ nel modello di schedulazione dei bus valori piccoli (eventualmente zero) per ogni h. Ciò corrisponde ad inserire "vincoli di continuità della corsa" in cui lo stesso bus che arriva in A deve necessariamente (ed immediatamente) proseguire o verso C o verso D. Si può anche porre ad infinito il tempo per l'andata a/il rientro da questi capolinea (A e B) al deposito, per evitare "sostituzioni al volo" di un bus con uno arrivato esattamente in quel momento dal deposito.

- Nel caso 2, i nodi intermedi (A e B) non sono (necessariamente) capolinea. Questo significa che la lunghezza delle soste a tali nodi è fissata dai dati in input, e viene *ignorata dal modello*. Infatti, in input è noto il tempo di arrivo e di partenza di un bus nel nodo, ma il tempo di arrivo viene ignorato; in altri termini, si considera come se il bus arrivasse e partisse al nodo nello stesso istante, di fatto interpretando il tempo di attesa forzata al nodo (se ve n'è alcuna) come se fosse tempo di viaggio.

In altri termini, la differenza tra i due casi sarebbe la *gestione delle soste ai nodi intermedi*: nel primo si potrebbe avere un certo grado di flessibilità (che però può essere ridotto a zero), mentre nel secondo la flessibilità sarebbe per definizione nulla. È però importante notare che la flessibilità sarebbe nulla in termini di scelta ma più elevata in termini di definizione. Infatti, nel caso 2 la durata della sosta è fissata corsa per corsa, ma può (in principio) variare arbitrariamente da una corsa alla successiva. Nel caso 1, invece, la durata della sosta può essere scelta, ma se viene fissata ciò si ottiene ponendo (es.) $\delta^h_{max,A} = \delta^h_{min,A} = \delta$, e quindi *è la stessa per tutte le corse nella stessa fascia*. Si noti che nel caso 1 aumenta considerevolmente il numero delle corse, ossia il numero di archi nel modello MCF. Inoltre due corse con un vincolo di compatibilità molto stretto, nei fatti, corrispondono ad un solo arco: nel caso una fosse utilizzata allora lo sarebbe anche l'altra (e viceversa).

Un'ulteriore possibile estensione sarebbe quella di permettere *viaggi a vuoto tra capolinea diversi*, in cui es. un bus giunto in C si sposti vuoto verso D, senza passare né per A né per O, per effettuare una corsa della linea $D \to A$. Ciò dovrebbe solamente richiedere l'aggiunta un opportuno insieme di archi al modello di MCF.

4 La versione base del codice

La versione base del codice implementa il modello appena presentato e risolve il problema in una sua forma semplificata, ovvero imponendo il vincolo di poter gestire solamente due percorsi, uno per ogni verso (vedi Figura 2). In realtà il modello implementato, almeno in linea teorica, avrebbe potuto gestire situazioni con più percorsi. È stato consapevolmente scelto invece un percorso "a piccoli passi", con l'implementazione di una versione base che permettesse lo svolgimento di una serie di test atti a verificare se il modello formulato teoricamente fosse in grado di pervenire a soluzioni valide in tempi ragionevoli. Si decise così di rinviare ad un secondo momento l'inserimento di tutte le possibili estensioni del modello e la gestione delle relative problematiche connesse.

Il codice prodotto è stato sviluppato in C++; oltre ai vantaggi noti, la scelta di tale linguaggio ha permesso l'utilizzo di due software preesistenti, di proprietà dell'Università di Pisa, specializzati nel risolvere due specifici sottoproblemi:

- *MCFSimplex* [BF11], per risolvere il problema di flusso di costo minimo alla base del sottomodello di schedulazione dei veicoli (Paragrafo 3.1.1);
- *Bundle* [F10], per risolvere il problema *Duale Lagrangiano* che permette il soddisfacimento dei vincoli accoppianti (Paragrafo 3.3).

Il codice ha, fin dalla sua versione di base, due possibili utilizzi: ricerca di una soluzione buona per un problema dato (standard, la funzionalità per cui è stato creato), valutazione in termini di costo per una soluzione già fatta. Il programma, nel suo funzionamento standard, riceve in input due files:

- la rete, ovvero i capolinea, gli intertempi, le durate delle soste, il tempo di viaggio dei vari segmenti di percorso, le corse possibili da poter inserire in soluzione; la rete è descritta in un unico file tramite un formalismo (detto TTD) usato all'interno della *M.A.I.O.R.*;
- un file di configurazione che setta alcuni parametri (come, ad esempio, il costo di ogni vettura o il costo di ogni secondo di sosta al capolinea) necessari al funzionamento dell'algoritmo.

quindi restituisce un file, simile al file d'input che descrive la rete, che incapsula al suo interno il risultato della computazione. Nella modalità "valutazione" invece il programma riceve i due files d'input come sopra e in più, ovviamente, il file con all'interno la soluzione da valutare.

Una considerazione a parte la merita la gestione di alcuni parametri, operata in un apposito file separato dalla descrizione della rete. La scelta di non inglobare questi parametri all'interno del file che descrive la rete è dovuta a due motivazioni. Innanzitutto estendere il protocollo TTD con l'inserimento dei suddetti parametri avrebbe comportato un dispendio di tempo nonché sarebbe risultato macchinoso e addirittura controproducente, dal momento che tale formalismo fornisce una struttura a files che sono usati in molteplici codici per molteplici tipi di problemi. In secondo luogo i due files rappresentano fisicamente due insiemi di dati

eterogenei anche a livello logico. La rete fornita in input è infatti il frutto della modellazione della realtà che si vuole descrivere e per la quale si vuole trovare una soluzione. Tale realtà si può considerare immutabile, poichè descrive definitivamente il problema che il programma cerca di risolvere. L'insieme di parametri che si trova all'interno del file di configurazione invece non fa parte della descrizione del problema: il suo ruolo infatti è esclusivamente quello di influenzare il comportamento del programma nella ricerca della soluzione. In altre parole, il programma trova soluzioni diverse per lo stesso problema al variare dei parametri suddetti.

La ricerca del giusto set di parametri diventa quindi fondamentale per:

- ottenere soluzioni che non siano, ad esempio, sbilanciate a favore della schedulazione ottimale dei veicoli piuttosto che del problema del timetabling (o viceversa);
- trovare buone soluzioni per un vasto numero di problemi (e non uno solo).

È evidente quindi il vantaggio di avere a disposizione un file separato, più semplice e piccolo rispetto al file che descrive la rete, così da poter modificare tali parametri con più facilità e chiarezza.

4.1 L'architettura logica del codice

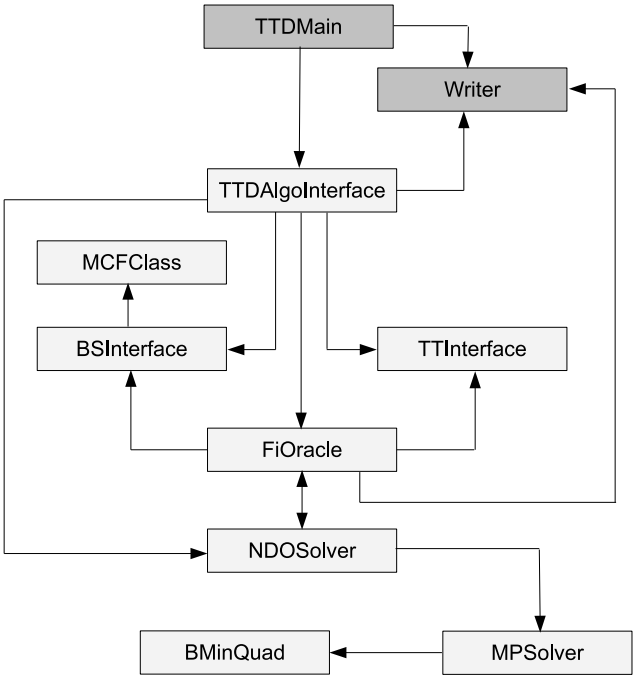

Figure 11: L'architettura del codice espressa tramite le interfacce

Il codice è strutturato secondo lo schema mostrato in Figura 11, dove si è voluto mostrare la suddivisione del codice stesso in *moduli*, rappresentati dalle relative interfacce astratte. Ogni modulo infatti, eccezion fatta per *TTDMain* e *Writer*, è composto da un'interfaccia astratta, che determina il protocollo tramite il quale gli altri moduli comunicano con il

modulo in questione, e quindi da una classe concreta che implementa l'interfaccia astratta. Questo tipo di approccio permette, eventualmente, la sostituzione di una classe concreta con un'altra simile (cioè che implementa la stessa interfaccia) senza che sia necessario apportare modifiche alle altre classi "adiacenti". Ad esempio è possibile sostituire il solutore *MCFSimplex* con un altro solutore che, come quest'ultimo, implementa la stessa interfaccia astratta (*MCFClass*). Questo vantaggio è stato utilizzato, per fare un altro esempio, in una fase più avanzata (Paragrafo 5) dello sviluppo del codice: è stata insereta un'ulteriore classe di tipo *TTDAlgoInterface* tra *TTDMain* (che non ha risentito dell'inserimento) e la già presente classe di tipo *TTDAlgoInterface*. Vi sono poi i vantaggi "classici" derivati da un codice diviso in moduli, vantaggi che si trovano sia nella fase di sviluppo (più semplice perché il problema è partizionato in pezzi più piccoli e gestibili) sia nella fase di debugging (più semplice localizzazione di eventuali bug o malfunzionamenti più rilevanti).

Passando ad una breve descrizione della struttua logica, si può subito notare come la gestione dell'input e quella dell'output siano separate dal resto del programma, ovvero il suo *core*. Tale separazione rende il core stesso non sensibile alle eventuali modifiche che può subire il protocollo *TTD* e/o il file di configurazione. I dati letti nei files d'input sono preprocessati all'interno di *TTDMain* in modo tale da renderli conformi con quello che il core del codice pretende. Il core vero e proprio inizia nello "strato" di codice appena sottostante a *TDMain*, ovvero da *TTDAlgoInterface* e la classe concreta che lo implementa. Questa ha il controllo dell'intera esecuzione della parte "operativa" del codice, ovvero quella che si occupa della ricerca della soluzione o della valutazione di una soluzione preesistente senza preoccuparsi della fase d'input o di output su file. I suoi metodi pubblici raccolgono i dati inviati da *TTDMain* e li processano in modo da essere a sua volta inviati ai moduli sottostanti. I suoi metodi privati invece gestiscono i solvers dedicati ai sottoproblemi (scheduling, timetabling, duale Lagrangiano), inviandogli i dati di input, controllandone l'esecuzione, raccogliendo i loro output ed aggregando in modo opportuno i risultati. Attraverso le proprie strutture dati si preoccupa inoltre di effettuare un'ulteriore controllo di ammissibilità sulle eventuali soluzioni trovate.

TTInterface si occupa di risolvere il sottoproblema relativo al timetabling. Il modulo riceve i dati necessari dal modulo *TTDAlgoInterface* con i quali crea un grafo orientato e aciclico, dotato di un nodo radice e di un nodo pozzo (vedi Paragrafo 3.2), che si preoccupa di modificare quando occore e sopra il quale risolve, tramite un piccolo solver incapsulato all'interno del modulo stesso, il problema di trovare un cammino minimo dalla suddetta radice al suddetto pozzo. Il risultato di questo lavoro è determinare l'insieme delle corse (rappresentate dai nodi del grafo) che formino il cammino di costo globale minimo rispetto ai costi attribuiti dall'algoritmo alle corse stesse, variabili all'interno dell'esecuzione. La variabilità dei costi delle corse, laddove invece il resto degli archi è caratterizzato da costi che rimangono invariati nel corso dell'esecuzione, è la caratteristica che permette l'integrazione tra la soluzione qui calcolata del problema del timetabling con la soluzione del problema dello scheduling dei veicoli, come già spiegato nel Paragrafo 3.3. Il modulo *TTInterface* è inoltre in grado di valutare, in accordo con la doppia funzionalità del programma (ricerca di una soluzione + valutazione di una soluzione preesistente), prima l'ammissibilità poi eventualmente il costo di una soluzione già formata.

Il modulo *BSInterface* invece si occupa di risolvere il sottoproblema relativo allo scheduling dei veicoli. Anch'esso riceve i dati dal modulo *TTDAlgoInterface* e tramite questi crea

una grafo orientato e aciclico, che si preoccupa di modificare quando serve e sopra il quale risolve un problema di flusso di costo minimo. In questo caso però non è direttamente il modulo stesso che risolve il problema, bensì delega il lavoro al software esterno *MCFSimplex*, una classe che implementa l'interfaccia *MCFClass*.

MCFSimplex è un software di proprietà dell'Università di Pisa che è stato sviluppato dal sottoscritto sotto la supervisione del Prof. Antonio Frangioni e che è specializzato proprio nella risoluzione di problemi di Flusso di Costo Minimo (MCF) tramite gli algoritmi del Simplesso Primale e Simplesso Duale.

Riguardo alla risoluzione del MCF il modulo *BSInterface* quindi limita la propria azione alla gestione del grafo definito all'interno della sua struttura dati, al caricamento del grafo stesso all'interno di *MCFSimplex*, ed infine alla ricezione dei risultati calcolati dal software esterno chiamato in causa. Il risultato di questo lavoro è determinare il comportamento dei veicoli (ovvero quando escono/rientrano al deposito, qundo sostano, quando effettuano una corsa) usati per fornire il servizio richiesto durante il periodo di tempo prestabilito. Anche in questo caso, come per il modulo *TTInterface*, il modulo è in grado di valutare (tramite *MCFSimplex*) una soluzione preesistente e di variare i costi delle corse interagendo sempre com *MCFSimplex* (che permette la modifica del grafo inseritogli in un primo momento e quindi il ricalcolo di una nuova soluzione al problema MCF).

Si giunge infine agli ultimi due moduli, *NDOSolver* e *FiOracle* (i moduli *MPSolver* e *BMinQuad* sono legati a *NDOSolver*, ne tralasciamo la trattazione). I due moduli risolvono, lavorando a stretto contatto, il problema cosidetto *Duale Lagrangiano*, così come descritto nel Paragrafo 3.3. Le quattro interfacce elencate fanno parte di un software di proprietà dell'Università di Pisa, sviluppato dal Prof. Antonio Frangioni. Insieme alle interfacce sono fornite anche tre classi concrete che implementano *NDOSolver*, *MPSolver* e *BMinQuad*: chiameremo l'intero pacchetto delle tre interfacce con le tre classi concrete *modulo Bundle*.

Il *modulo Bundle* si occupa, ad ogni iterazione compiuta all'interno dell'esecuzione del programma, di risolvere il problema del Duale Lagrangiano; questi utilizza come input il valore ottimo del *Rilassamento Lagrangiano* (che si ottiene tramite la risoluzione indipendente dei due sottoproblemi di scheduling e timetabling) e il subgradiente (generato anch'esso dalle soluzioni dei sottoproblemi, mettendole a confronto).

Il modulo *FiOracle* invece ha il compito di rapportarsi sia col *Bundle* sia coi solvers dei sottoproblemi (i moduli *TTInterface* e *BSInterface*) passando e ricevendo da questi i dati fondamentali per l'esecuzione dell'algoritmo. In pratica il *Bundle* comunica col resto del codice tramite *FiOracle*: esso fornisce al *Bundle* l'input (valore ottimo e subgradiente) che ricava dai solvers TT e BS, attende l'esecuzione del *Bundle*, quindi raccoglie l'output del Bundle che rigira ai solvers. È chiaro quindi il motivo per cui il modulo *FiOracle* è a stretto contatto coi solvers TT e BS, come si può vedere in Figura 11. Una tipica iterazione consiste nella risoluzione del rilassamento Lagrangiano da parte dei solvers (quindi i due moduli *BSInterface* e *TTInterface*), dai quale *FiOracle* raccoglie le soluzioni e i valori delle funzioni obiettivo aggregandole e formando quindi il valore ottimo del rilassamento e il subgradiente da passare al *Bundle*; finita l'esecuzione, il *Bundle* restituisce il vettore λ che suggerisce la miglior combinazione di costi sulle corse in modo da poter, alla successiva iterazione, soddisfare il più possibile quei vincoli accoppianti (vedi Paragrafo 3.3) che sono stati rilassati.

Durante le iterazioni quindi i solvers TT e BS si troveranno a risolvere, rispettivamente,

problemi di cammino minimo e MCF sempre diversi, poiché cambiano i costi sulle corse secondo le indicazioni fornite dalle esecuzioni del *Bundle*. Le soluzioni aggregate trovate durante l'esecuzione diventano sempre più costose, in virtù del fatto che tramite l'azione del *Bundle* il peso dei vincoli accoppianti nelle funzioni obiettivo (rappresentato proprio da quei costi sulle corse che inizialmente sono nulli, vedi Paragrafo 3.3) diventa sempre più grande. Tale livello è definito come *lower bound*: durante l'esecuzione esso salirà progressivamente. Inoltre il modulo *FiOracle* fornisce al *Bundle*, durante ogni ogni iterazione, un ulteriore valore, il cosidetto *upper bound*, rappresentato dal costo di una soluzione ammissibile calcolata sul momento in maniera euristica: in pratica, ad ogni iterazione, si prende la parte di soluzione del rilassamento relativa al timetabling e si risolve un nuovo problema BS di scheduling con le corse trovate dal timetabling fissate "a 1", garantendo così il soddisfacimento dei vincoli accoppianti e quindi l'ammissibilità della soluzione.

La presenza di un lower bound e di un upper bound assicurano che la soluzione ottima abbia sicuramente un costo che si trova tra di essi: il *Bundle* quindi tenta di ridurre tale gap fino al punto in cui questo sarà ritenuto piccolo, quindi la soluzione ammissibile calcolata a quel punto sarà considerata quella finale da stampare sul file d'output poichè presumibilmente molto simile alla soluzione ottima.

4.2 L'architettura fisica del codice

In questo Paragrafo sono presentati più dettagliatamente i moduli. Di ogni modulo sono descritte l'interfaccia, la classe concreta che la implementa con la/e classe/i che la istanzia e la/e classe/i a sua volta istanziate; inoltre saranno menzionate a grandi linee, laddove è necessario un approfondimento, alcune delle scelte implementative fatte riguardo ai metodi e alle strutture dati private all'interno della classe. Si escludono da questa trattazione i moduli che riguardano *MCFSimplex* e *Bundle* (ma non *FiOracle*, la cui classe sarà analizzata), poiché già dotati, in quanto software esterni, di documentazioni propria.

4.2.1 Il modulo *TTDMain*

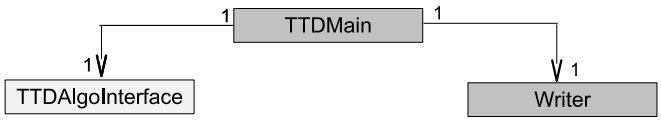

Figure 12: Il frammento di struttura con *TTDMain*

Il modulo *TTDMain* è composto di un unico file (*TTDMain.C*). *TTDMain* rappresenta il vero e proprio "main" per il codice, ed è effettivamente la radice dell'intera struttura: esso infatti carica al suo interno il modulo *TTDAlgoInterface* che a sua volta caricherà il resto del programma. *TTDMain* istanzia anche il modulo *Writer* per passarlo, in riferimento, alla struttura sottostante. Come già accennato, *TTDMain* si preoccupa di leggere i dati dai due files di input (il file che descrive la rete e il file di configurazione) e, qualora l'utente lo richiedesse, di leggere anche il file che racchiude una rete già risolta. I dati raccolti nella fase di lettura dell'input vengono memorizzati in apposite strutture dati all'interno del

modulo, quindi sono svolte tutta una serie di operazioni di *preprocessing* sui dati prima di passarli al modulo sottostante, il *TTDAlgoInterface*. Questa ulteriore memorizzazione può sembrare ridondante e inutile, quindi persino dannosa. In realtà tale caratteristica risulta imprescindibile: non si può infatti presumere che i dati all'interno delle strutture dati del modulo *TTDMain* siano assimilabili a quelli che serviranno al modulo *TTDAlgoInterface*. Il processing che *TTDMain* svolge sui dati è mirato semplicemente ad adattarli a quello che il core del codice pretende come input: tale preprocessing deve essere quindi sensibile solo al cambio di protocollo interno dei files. Un esempio di preprocessing è la selezione dei nodi (stazioni) significativi da considerare. Un altro esempio consiste nel calcolo della divisione della giornata in fasce orarie: all'interno del file che descrive la rete ne esistono diverse (a seconda del contesto) ed eterogenee tra loro, mentre il programma ne pretende una unica e totale, sul quale sono modellati tutti i dati che riceve.

4.2.2 Il modulo *Writer*

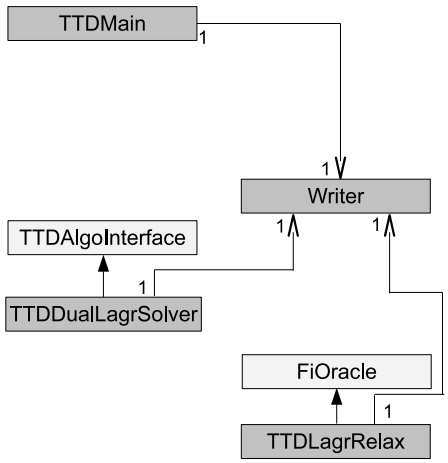

Figure 13: Il frammento di struttura con *Writer*

Il modulo *Writer*, formato dall'header *Writer.h* e dal file *Writer.C*, si occupa della scrittura delle soluzioni trovate dal programma all'interno di un file scritto nello stesso formato (*TTD*) usato per i files d'input che descrivono le reti. Nel frammento di struttura qui in Figura 13 possiamo osservare come *Writer* sia collegato a *TTDMain*, *TTDAlgoInterface* (in modo da poter scrivere la soluzione finale alla fine dell'esecuzione) e *FiOracle* (in modo da poter scrivere le soluzioni trovate durante l'esecuzione). In realtà lo stesso oggetto *Writer* è istanziato in *TTDMain* e passato per riferimento agli altri moduli che lo usano. Il file d'output, nella sua struttura, è molto simile ad un file d'input. La differenza più importante consiste nel fatto che, mentre nel file d'input sono presenti tutte le possibili corse, nel file d'output sono presenti solo le corse fatte, e sono raggruppate secondo il veicolo che le effettua. *TTDMain* si preoccupa, nel momento in cui istanzia *Writer*, di passargli tutte le informazioni utili estratte dal file d'input, che in questo modo non deve essere aperto e letto nuovamente ogni volta che *Writer* necessita di scrivere un nuovo file d'output.

4.2.3 Il modulo *TTDAlgoInterface*

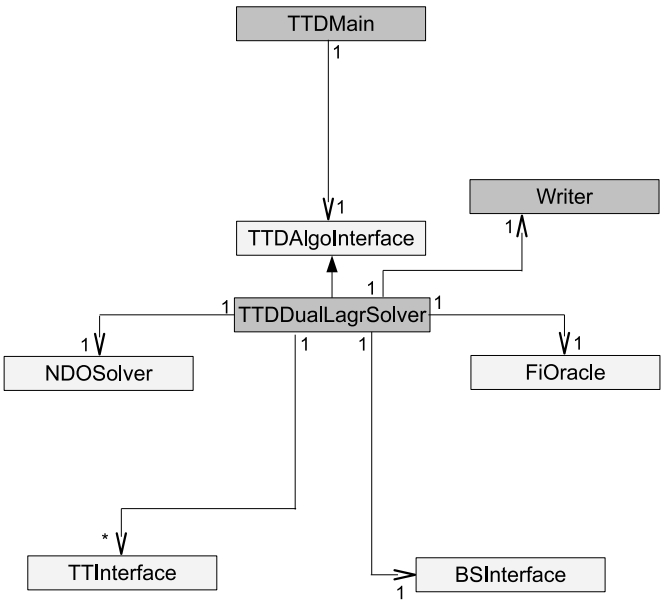

Figure 14: Il frammento di struttura con *TTDAlgoInterface*

Il modulo *TTDAlgoInterface* è il primo strato dell'algoritmo. È composto dall'interfaccia *TTDAlgoInterface.h* e dalla classe concreta formata dai files *TTDDualLagrSolver.h* e *TTDDualLagrSolver.C*. Il modulo si occupa della raccolta e dell'organizzazione dei dati forniti dal modulo *TTDMain*, attraverso i metodi pubblici dichiarati nella sua interfaccia *TTDAlgoInterface*. I dati raccolti sono memorizzati in strutture dati interne. Il modulo inoltre si preoccupa di istanziare i vari solver (*TTInterface*, *BSInterface*, *NDOSolver*, *FiOracle*) adibiti alla risoluzione dei sottoproblemi. Tramite il metodo pubblico *Solve* ne controlla l'esecuzione.

La particolarità di questo metodo consiste nella possibilità, concessa all'oggetto chiamante (in questo caso *TTDMain*, più avanti, come vedremo, da un ulteriore oggeto di tipo *TTDAlgoInterface*) di poter ottenere il controllo dell'esecuzione durante il (presumibilmente) lungo processo iterativo che porta alla soluzione migliore. In pratica *TTDMain* chiama il metodo *Solve*, questo esegue qualche passo in avanti quindi resituisce a *TTDMain* il controllo: *TTDMain* ha quindi la possibilità di effettuare qualche operazione per poi richiamare il metodo *Solve*, finché non si è trovata la soluzione migliore. Questa caratteristica (per ora non sfruttata) può permettere, ad esempio, al modulo chiamante di aggiornare in tempo reale un'eventuale controllo grafico che fornisce all'utente la possibilità di visionare lo stato d'avanzamento dei lavori e la possibilità ulteriore di interrompere manualmente l'esecuzione del programma.

Infine il modulo si occupa della raccolta dei risultati, sia che si tratti della classica soluzione sia che si tratti della valutazione di una soluzione preconfezionata.

4.2.4 Il modulo *TTInterface*

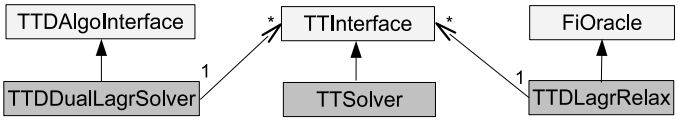

Figure 15: Il frammento di struttura con *TTInterface*

Il modulo *TTInterface* è composto dall'interfaccia *TTInterface.h* e dalla classe concreta formata dai files *TTSolver.h* e *TTSolver.C*. Si occupa della risoluzione del problema del timetabling. Il modulo è istanziato dal modulo *TTDAlgoInterface*, che gli passa tutti i dati necessari affinché possa costruire il grafo orientato e aciclico, dotato di nodo radice e di nodo pozzo, sul quale lo stesso modulo può risolvere il problema del *Cammino minimo* di un'unità di flusso tramite il solver interno implementato nei metodi *Solve* e *SPTSolver*. Il processo di creazione del suddetto grafo è svolto all'interno del metodo *CreateGraph*, che è invocato dal modulo *TTDAlgoInterface* dopo che tutti i dati sono stati passati al modulo sottostante. Come già accennato, la risoluzione del problema prevede la risoluzione del *Rilassamento Lagrangiano* (quindi dei due sottoproblemi di timetabling e scheduling dei veicoli) effettuata molteplici volte, ognuna con un diverso set di costi applicati alle corse e derivati dalla risoluzione simultanea del problema del *Duale Lagrangiano* effettuata dal modulo *FiOracle*. Per questo motivo l'interfaccia del modulo presenta metodi pubblici che permettono la modifica di tali costi delle corse. Nel grafo ogni corsa è rappresentata da un nodo (o, come vedremo in seguito, da un insieme di nodi), mentre il costo di ogni singola corsa lo si trova su ogni arco entrante al nodo relativo. I metodi che modificano i costi delle corse in realtà non agiscono direttamente sul grafo: essi modificano la struttura dati che tiene traccia dello stato delle corse, mentre un ulteriore metodo *UpdateGraph*, da invocare alla fine delle modifiche, si preoccupa di trasferire le suddette modifiche sul grafo stesso. Gli stessi principi si applicano nel caso in cui le corse vengano, dall'esterno, *fissate ad 1* o *fissate a 0* (ovvero le corse devono o non possono far parte della soluzione): nella fattispecie la struttura dati relativa alle corse si preoccupa di rappresentare con sufficiente dettaglio il tipo di fissaggio effettuato. Nel "momento storico del codice" descritto in questo Paragrafo, esistono già due tipi di fissaggio diversi, ovvero quello derivato da una precisa scelta "a monte" effettuata dal cliente e descritta nel file d'input e il fissaggio che serve per la costruzione delle soluzioni temporanee rilasciate dal programma durante le iterazioni che scandiscono la sua esecuzione. I due tipi di fissaggio seguono una loro gerarchia: può capitare infatti che una corsa non fissata in input (quindi "libera") sia fissata ad 1 ad una data iterazione per la costruzione di una soluzione temporanea, mentre invece non può capitare che una corsa fissata in input non faccia parte di qualsiasi soluzione rilasciata e quindi il livello inferiore di gerarchia viene ignorato. Vedremo in seguito che questa gerarchia di fissaggio sarà ulteriormente estesa. Il modulo prevede inoltre, in linea con la duplice funzionalità del programma stesso, la possibilità di valutare il costo (parziale, ovviamente rigurdante solo la parte timetabling) una soluzione già fatta tramite il metodo *GetCostFixedSolution*, che si appoggia alla gerarchia dei fissaggi sopra spiegata e usa il metodo *Solve* sopra citato.

4.2.5 Il modulo *BSInterface*

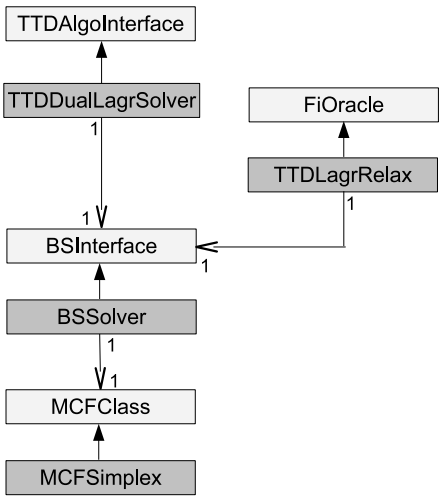

Figure 16: Il frammento di struttura con *BSInterface*

Il modulo *BSInterface* è composto dall'interfaccia *BSInterface.h* e dalla classe concreta formata dai files *BSSolver.h* e *BSSolver.C*. Si occupa della risoluzione del problema dello scheduling dei veicoli. Il modulo è istanziato dal modulo *TTDAlgoInterface*, che gli passa tutti i dati necessari affinché possa costruire il grafo orientato sul quale il solver dedicato *MCFimplex* può risolvere il problema del *Flusso di costo minimo*. Il processo di creazione del suddetto grafo è svolto, così come per il modulo *TTinterface*, all'interno del metodo *CreateGraph*, che è invocato dal modulo *TTDAlgoInterface* dopo che tutti i dati sono stati passati al modulo sottostante. Le analogie con il modulo *TTInterface* non terminano col metodo *CreateGraph*, ma sono estendibili a tutte le considerazioni fatte nel paragrafo precedente. I due moduli, per quanto trattino sottoproblemi diversi e quindi vi siano differenze anche sostanziali, agiscono simultaneamente dovendo rispondere a sollecitazioni esterne simili (cambio dei costi delle corse che comporta la modifica dei grafi durante l'esecuzione, identica gerarchia per i fissaggi, valutazione delle soluzioni già trovate). La differenza più eclatante è già stata citata: mentre il modulo *TTInterface* ha al suo interno un solver per il problema che deve risolvere, *BSInterface* è costretto ad appoggiarsi ad un codice esterno per risolvere il problema di flusso di costo minimo, sicuramente più complesso del problema di trovare un cammino minimo per un'unità di flusso all'interno di un grafo orientato e aciclico dotato di radice e di pozzo.

4.2.6 Il modulo *FiOracle*

Il modulo *FiOracle* è composto dall'interfaccia *FiOracle.h* e dalla classe concreta formata dai files *TTDLagrRelax.h* e *TTDLagrRelax.C*. *FiOracle* si occupa di raccogliere la soluzione fornita dal *Bundle* passandola ai solvers dei sottoproblemi come costi da attribuire alle corse di fornire al software esterno *Bundle* che risolve il Duale Lagrangiano, quindi raccoglie dai solvers stessi il valore in termini di costo della soluzione del Rilassamento Lagrangiano e calcola il subgradiente. *FiOracle* fornisce quindi il metodo pubblico *Fi* che il *Bundle*

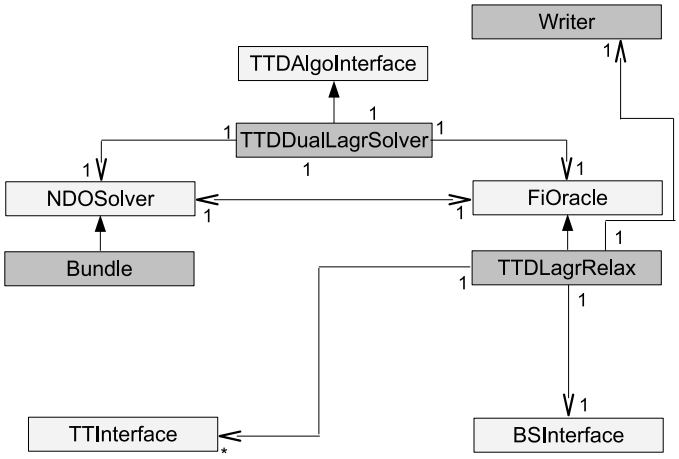

Figure 17: Il frammento di struttura con *FiOracle*

chiama ogni qual volta ha finito la sua computazione all'interno di ogni singola iterazione dell'algoritmo: *Fi* modifica i costi delle corse chiamando i relativi metodi forniti dalle interfacce *TTInterface* e *BSInterface*, quindi raccoglie da questi le soluzioni parziali, calcola il suo valore e il subgradiente relativo inserendoli in apposite strutture dati che il *Bundle* leggerà. Il metodo *Fi*, una volta raccolte le soluzioni parziali, chiama un altro metodo di *FiOracle*, ovvero *HeuristicSolution*. Questo metodo estrapola dalla soluzione parziale fornita da *TTInterface* l'insieme delle corse usate, fissandole a 1 temporaneamente (cioè tale fissaggio vale solo per questa specifica operazione) all'interno di *BSInterface* e quindi fa calcolare a *BSInterface* uno scheduling che combacia perfettamente con la parte timetabling: si crea così una soluzione ammissibile il cui valore sarà memorizzato in un apposito campo e usato dal *Bundle* come upper bound.

4.3 Risultati ottenuti

L'obiettivo finale da raggiungere tramite la realizzazione della prima versione del codice era, come già spiegato, quello di capire come il modello teorico, una volta realizzato, riusciva a risolvere il problema; lo studio del comportamento del programma durante l'esecuzione, l'analisi della qualità delle soluzioni trovate e dei tempi d'esecuzione riscontrati avrebbero infatti fornito indicazioni importanti sulle successive modifiche da apportare al modello e/o all'algoritmo implementato.

I primi test effettuati sulla versione base dell'algoritmo hanno evidenziato alcuni comportamenti standard del programma durante l'esecuzione e, come era prevedibile, alcuni problemi.

- Il primo problema riscontrato è risultato essere il tempo impiegato in ogni esecuzione, che poteva protrarsi per ore se non per giorni. Il programma infatti, anche per istanze di problema di media complessità, non riusciva a ridurre in maniera sensibile il gap tra il suo lower bound e il suo upper bound. Nella Figura 18 sono elencati i risultati ottenuti da questa versione del codice su tre diverse istanze di problema tramite 24 ore di esecuzione per ciascuna istanza: il gap è stato calcolato usando la formula $\frac{UB-LB}{LB}$.

Istanza	lower bound	upper bound	gap (%)
n. 1	108	378	250,00%
n. 2	176	324	84,09%
n. 3	607	793	30,64%

Figure 18: Esempi di esecuzioni del programma

- Tipicamente il programma riusciva a trovare quasi subito una buona soluzione ammissibile, quindi un upper bound; difficilmente però durante l'esecuzione codesta veniva rimpiazzata con una migliore in termini di costo.

- Come conseguenza dei dilatati tempi d'esecuzione fin troppo spesso non era possibile, soprattutto per i problemi più difficili come quelli riportati nella tabella qui sopra, arrivare alla soluzione ottima; si era costretti a troncare "a metà" le esecuzioni e, quindi, a valutare soluzioni ammissibili ma non ottime.

- Le soluzioni analizzate erano spesso "non buone" dal punto di vista qualitativo. Si riteneva in via ipotetica che la causa potesse dipendere fortemente da un improprio settaggio dei parametri di configurazione, poichè variando questi le soluzioni (ammissibili ma parziali) trovate erano spesso sbilanciate a favore del timetabling piuttosto che dell'ottimizzazione del numero di veicoli o viceversa. I tempi d'esecuzione dilatati non permettevano però di stabilire la veridicità dell'ipotesi fatta. Innanzitutto il gap tra lower bound e upper bound troppo grande, causa dei tempi d'esecuzione alti, impediva una corretta valutazione, in termini di distanza dall'eventuale (ma sconosciuta) soluzione ottima, delle soluzioni trovate. Inoltre i tempi d'esecuzione alti non rendevano agevole effettuare il necessario numero di prove con diversi set di parametri al fine di trovare il set giusto per bilanciare il soddisfacimento dei due obiettivi (timetabling buono, ottimizzazione dell'impiego dei veicoli) all'interno delle soluzioni stesse.

- Le soluzioni trovate erano spesso "facilmente migliorabili" degli esperti *M.A.I.O.R.* che, tramite semplici operazioni fatte "a mano" riuscivano a migliorare la soluzione sia a livello qualitativo sia per quanto riguarda il suo costo. Anche per questa situazione il gap tra lower bound e upper bound monitorato durante l'esecuzione era troppo alto per poter dare una sufficiente garanzia di bontà alla soluzione ammissibile trovata dal programma: in pratica era facilmente ipotizzabile che il programma fosse ancora troppo lontano dal suo ottimo e quindi poco buona la soluzione trovata al momento dell'interruzione.

La difficoltà incontrata dal programma nel ridurre il gap in tempi ragionevoli è parsa naturalmente il problema più importante. Non era solo un mero problema d'efficienza: i tempi troppo dilatati impedivano di raggiungere le soluzioni ottime e costringevano a studiare soluzioni di cui non era possibile ipotizzare la distanza dall'ottimo a causa del gap altissimo. È chiaro infatto che se il gap fosse stato decisamente più basso si sarebbe potuto analizzare le soluzioni ammissibili trovate con la consapevolezza della loro distnza dall'ottimo. Rimaneva anche da capire la ragione di gap così alti, capire cioè se tale problema potesse essere causato

da upper bound troppo alti e/o di lower bound troppo bassi. Ad esempio nel caso l'upper bound fossero risultati troppo alti si sarebbe potuto intervenire cercando di migliorare (o cambiando in toto) il processo euristico di costruzione delle soluzioni ammissibili. Era chiaro che un'analisi completa non poteva prescindere dal raggiungimento della soluzione ottima, obiettivo che la prima versione del codice troppo spesso non riusciva a soddisfare.

Fu deciso quindi di utilizzare *IBM Cplex*, un software specializzato nella risoluzione di molteplici problemi matematici (quindi anche di ricerca operativa), concesso in licenza all'Università di Pisa, al fine di risolvere al meglio e in tempi ragionevoli gli stessi problemi che avevano evidenziato le problematiche fin qui esposte e trarre da questo studio indicazioni utili per lo sviluppo del codice. Furono quindi create routines nel codice per unire i sottografi creati (a partire sia dalla rete che dai parametri di configurazione) durante l'esecuzione in un unico grafo da passare a *Cplex*, in modo tale che questi avrebbe dovuto risolto un problema di flusso di costo minimo su un grafo composto da molti sottografi disgiunti: aggiungendo opportunamente alla formulazione i vincoli accoppianti, Cplex avrebbe risolto il problema nella sua interezza. Gli studi effettuati grazie all'utilizzo di *Cplex* non hanno fornito indicazioni precise sulle motivazioni alla base dell'esistenza del gap. Sono stati effettuati test sulle medesime istanze e con gli stessi vincoli (interruzione dell'esecuzione dopo 24 ore) imposti precedentemente al nostro programma; questi, come si può osservare nella tabella comparativa in Figura 19, hanno evidenziato come il potente software abbia spesso ridotto maggiormente il gap rispetto alle corrispondenti prestazioni del nostro programma, ma non abbastanza da poter ricavare informazioni precise su dove potesse essere la soluzione ottima e quindi su quale punto (lower bound o upper bound) intervenire.

	Algoritmo			Cplex		
Istanza	lower bound	upper bound	gap (%)	lower bound	upper bound	gap (%)
n. 1	108	378	250,00%	110	212	92,73%
n. 2	176	324	84,09%	184	262	42,39%
n. 3	607	793	30,64%	619	643	3,88%

Figure 19: Comparazione tra le esecuzioni del programma e quelle di Cplex sulle medesime istanze di problema

Questo tipo di risposta suggeriva sia che il modo di costruire (per via euristica) soluzioni ammissibili era migliorabile sia che il problema stesso nella sua interezza era effettivamente di difficile risoluzione per via del fatto che il gap, nonostante l'utilizzo di un software potente come *Cplex*, non era stato comunque ridotto a livelli accettabili.

Simultaneamente allo "studio del gap" furono studiate anche la qualità delle soluzioni ammissibili che *Cplex* restituiva. Furono variati opportunamente i parametri di configurazione (che ovviamente influivano sul processo di costruzione del grafo sul quale Cplex lavorava), quindi furono eseguite tre batterie di tests, provando ogni volta un diverso set di parametri di costo su ognuna delle tre istanze di problema finora prese in considerazione e facendo "girare" *Cplex* sempre un giorno per ciascuna istanza: un totale quindi di nove esecuzioni, un giorno ciascuna, da svolgere in simultanea con gli altri tipi di tests riguardanti il gap citati sopra. Fortunatamente questi tests fornirono precise indicazioni e buone prospettive

per il proseguio del lavoro:

- il gap rimaneva comunque abbastanza alto. Se da un lato questa non era una buona notizia, è stato importante apprendere che il gap non dipendeva affatto dai parametri di configurazione usato, ma era insito nella natura stessa del problema.
- Fu trovato un set di parametri di configurazione per il quale *Cplex* riusciva a fornire soluzioni ammissibili ritenute comunque, a prescindere dall presenza del gap alto, qualitativamente buone.

Il lavoro svolto con *Cplex* ha quindi, nella sua interezza, indicato che era possibile costruire soluzioni buone dal punto di vista qualitativo nonostante il gap tra lower bound e upper bound rimanesse alto. Era naturale a questo punto provare a modificare la natura stessa del codice integrando l'esistente approccio Lagrangiano al problema con un meccanismo di tipo euristico che emulasse in qualche maniera il tipo di scelte fatte da *Cplex* in sede di costruzione delle soluzioni ammissibili e contemporaneamente abbattesse i tempi di esecuzione.

5 L'approccio euristico

La bontà delle soluzioni trovate tramite *Cplex*, come già spiegato nel paragrafo precedente, ha confermato la validità del modello formulato, che si dimostrava quindi in grado di descrivere il problema in modo che fosse possibile trovare buone soluzioni. I dilatati tempi d'esecuzione del nostro codice e la minore validità delle soluzioni da questi trovate invece imponevano un'evoluzione abbastanza importante nell'algoritmo implementato, affiancando un approccio di tipo euristico a quello già esistente che abbattesse i tempi d'esecuzione.

L'idea di base è abbastanza semplice: al contrario dell'algoritmo base, che durante tutta la sua esecuzione aveva la facoltà di scegliere liberamente quali corse, all'interno dell'enorme insieme di corse possibili, inserire nella/e soluzione/i, la nuova euristica avrebbe guidato e facilitato l'esecuzione dell'algoritmo "fissando" progressivamente insiemi di corse opportune. L'algoritmo avrebbe quindi iniziato la sua esecuzione con la possibilità di scegliere le corse all'interno dell'intero insieme delle corse possibili; man mano che l'esecuzione fosse andata avanti, l'insieme delle corse possibili dal quale scegliere sarebbe diventato sempre più piccolo perché alcune di queste sarebbero state definitivamente scelte (o scartate) per la composizione della soluzione. Osservando la situazione da un'altra angolazione, l'algoritmo avrebbe dovuto costruire la sua soluzione progressivamente: questo avrebbe ridotto sempre di più le dimensioni del problema (e quindi, in linea teorica e come vedremo anche pratica, abbattuto i tempi d'esecuzione) poiché la porzione di soluzione da costruire sarebbe stata via via sempre più piccola. Se il principio di fondo dell'euristica risultava chiaro e abbastanza scontato, meno ovvio era comprendere quale meccanismo di scelta delle corse usare. Una scelta non consona della "regola di fissaggio" avrebbe ovviamente inficiato la riuscita stessa dell'upgrade dell'algoritmo, sia nell'obiettivo di abbattimento dei tempi d'esecuzione sia nell'obiettivo (primario) di ricerca di una buona soluzione: immaginando, ad esempio, una "cattiva decisione" nelle prime fasi dell'esecuzione, è facile prevedere che l'algoritmo sarebbe stato costretto a costruire una soluzione a partire da una base non buona che inoltre, molto probabilmente, avrebbe compromesso anche la successiva ricerca. Il fatto, acclarato, di avere buone soluzioni provenienti da *Cplex* non forniva, in questo senso, un aiuto dato che le tecniche usate da *Cplex* erano (e sono) complesse da reimplementare e comunque anch'esse inefficienti dal punto di vista dei tempi d'esecuzione: per raggiungere infatti le soluzioni in seguito ritenute buone *Cplex* doveva girare almeno 24 ore, decisamente di più rispetto ai tempi d'esecuzione medi che era ritenuto necessario raggiungere (nell'ordine delle decine di minuti). Inoltre la regola di fissaggio doveva essere gioco forza ricavabile dai dati interni all'algoritmo, ovvero da quelli forniti dal *Bundle*, in modo da permettere una naturale e (relativamente) comoda estensione dello stesso.

All'interno di ogni sua esecuzione l'algoritmo (o meglio, il *Bundle*) fornisce ad ogni iterazione:

- Un vettore di moltiplicatori Lagrangiani applicati ai vincoli del problema rilassati (quindi ad ogni corsa) che abbiamo chiamato λ: i valori λ non sono delimitati all'interno di un dominio, e quindi possono essere positivi o negativi con un valore assoluto qualsiasi.

- La soluzione del rilassamento Lagrangiano sottoforma di soluzioni dei sottografi, indipendenti tra loro se non per la presenza all'interno delle relative funzioni obiettivo

dei moltiplicatori Lagrangiani ovviamente in comune.

- La soluzione al problema del *rilassamento convessificato* (vedi [F05] oppure [FK00]) che sappiamo essere, dalla teoria della Ricerca Operativa, il duale lineare al problema del Duale Lagrangiano che il *Bundle* è chiamato a risolvere. La soluzione del rilassamento convessificato, che per semplicità da qui in avanti chiameremo *soluzione continua*, può essere vista come una "somma pesata" di alcune delle soluzioni del rilassamento Lagrangiano (quindi di flussi sui vari sottografi) operata attraverso opportuni moltiplicatori generati dal *Bundle* nella sua ricerca della soluzione ottima del Duale Lagrangiano. Ad ogni passo dell'algoritmo quindi la soluzione continua rappresenta lo stato d'avanzamento del lavoro compiuto dal Bundle e fornisce informazioni importanti sulla soluzione che si sta cercando di generare. In particolare per ogni corsa sono forniti tre valori (uno per ogni sottografo) compresi tra 0 e 1 (quindi la soluzione continua non è ammissibile, mentre se fosse composta di valori interi sarebbe ammissibile e ottima) che indicano quanto la corsa stessa sia considerata attualmente per la formazione di ogni "sotto-soluzione".

Si decise di effettuare uno studio di tipo "comparativo" (ovviamente sulle stesse istanze di problema) tra le soluzioni che *Cplex* forniva, ritenute qualitativamente buone dagli esperti *M.A.I.O.R.*, e i dati che il codice calcolava via via all'interno delle sue esecuzioni. La speranza era quella di trovare una qualche correlazione tra le soluzioni continue e/o i λ calcolati dal *Bundle* e gli insiemi di corse che formavano le "soluzioni buone". I criteri con i quali è stato effettuato lo studio sono i seguenti:

- i dati riguardanti i λ e le soluzioni continue sono estratti in momenti precisi dell'esecuzione del codice: in particolare, come vediamo negli esempi sotto descritti, la prima estrazione è effettuata alla 250esima iterazione dell'algoritmo, la seconda alla 500esima, quindi alla n.1000, n. 2000 e n. 5000;

- ad ogni estrazione l'insieme delle corse è partizionato in sottoinsiemi secondo il criterio d'appartenenza di ciascuna corsa alle diverse fasce orarie che suddividono l'intero periodo temporale analizzato;

- su ciascuno di questi sottoinsiemi di corse è definito il relativo sottoinsieme delle corse che fanno parte anche della soluzione buona già trovata;

- per ogni estrazione e per ogni fascia oraria sono calcolate, sull'insieme C delle corse e il suo sottoinsieme C' delle corse di C facenti parte della soluzione buona, le seguenti misure aggregate:

  - rapporto tra la media dei λ positivi delle corse in C' e la media dei λ positivi delle corse in C;
  - rapporto tra la media dei λ negativi delle corse in C' e la media dei λ negativi delle corse in C;
  - rapporto tra il numero di corse in C' con λ nullo e il numero delle in C con λ nullo;

– rapporto tra la media dei valori nella parte della soluzione continua riguardante il sottografo BS sulle corse in C' e la media dei valori nella parte della soluzione continua riguardante il sottografo BS sulle corse in C;
– rapporto tra la media dei valori nella parte della soluzione continua riguardante il sottografo SPT-1 sulle corse in C' e la media dei valori nella parte della soluzione continua riguardante il sottografo SPT-1 sulle corse in C;
– rapporto tra la media dei valori nella parte della soluzione continua riguardante il sottografo SPT-2 sulle corse in C' e la media dei valori nella parte della soluzione continua riguardante il sottografo SPT-2 sulle corse in C.

Ogni misura aggregata descrive, almeno per l'istanza presa in considerazione, quanto una particolare caratteristica dei dati calcolati dall'algoritmo (dal *Bundle*) è in grado di distinguere le corse che *Cplex* ha usato per costruire una soluzione ritenuta buona. Se un qualsiasi rapporto tra quelli presi risulta sensibilmente maggiore di 1, significa che la media di tale valore calcolato solo sulle corse della soluzione buona è più alta della stessa media calcolata su tutte le corse: questo comporta che in un dato momento dell'esecuzione e su una certa fascia oraria se una corsa ha, ad esempio, un valore BS nella soluzione continua alto allora la corsa probabilmente faceva parte anche della soluzione buona.

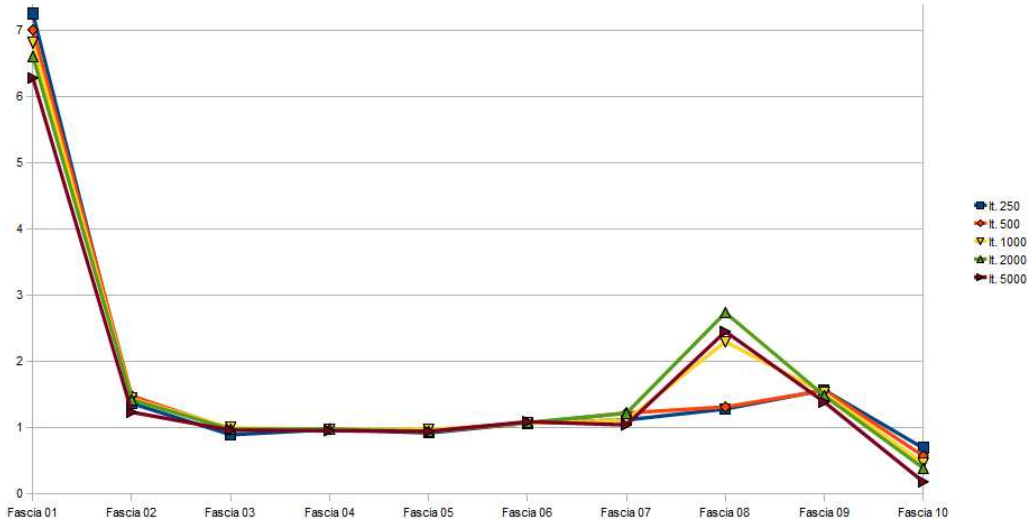

Figure 20: Rapporto tra le medie sui λ positivi

Lo studio è stato effettuato su tre diverse istanze di problema, con ovviamente le stesse modalità: i risultati ottenuti sono simili a prescindere dall'istanza analizzata, quindi in questa sede sono presentati solo i risultati riguardanti una di queste istanze. Ognuno dei grafici qui illustrati separa le diverse estrazioni dei dati, effettuate in punti prestabiliti dell'esecuzione del codice, con linee dai colori diversi. Ogni grafico quindi osserva, tramite la relativa linea, come la misura aggregata analizzata si sviluppi lungo l'arco della giornata, che è diviso in maniera discreta in fasce orarie. Come possiamo vedere dal grafico in Figura

20, sembra che per quanto riguarda soprattutto la prima fascia oraria possa esistere una correlazione tra valori alti (e positivi) dei λ e appartenenza alla soluzione buona: detta in altri termini, le corse che sappiamo essere delle buone scelte risultano essere contraddistinte, soprattutto sulla prima fascia oraria e in ogni rilevazione effettuata, da alti e positivi valori di λ all'interno dell'esecuzione del nostro codice. Il trend mostrato è confermato anche da altre prove empiriche che, per ragioni di spazio, qui non mostriamo.

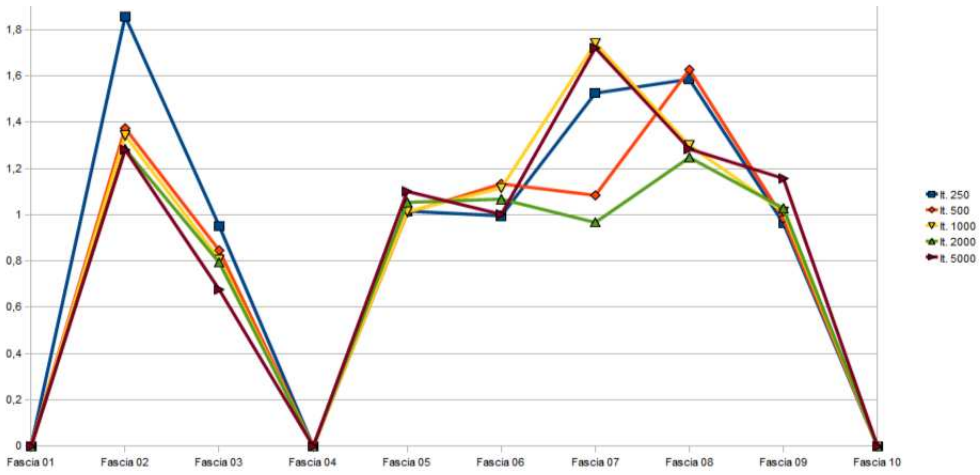

Figure 21: Rapporto tra le medie sui λ negativi

Il trend visto sui λ positivi non sembra ripetersi per i valori λ negativi, per i quali sembra esserci correlazione su altre fasce orarie rispetto alla prima; la correlazione trovata però non trova riscontro (come possiamo vedere nel grafico in Figura 22) nelle altre prove effettuate, dove emergono altri schemi.

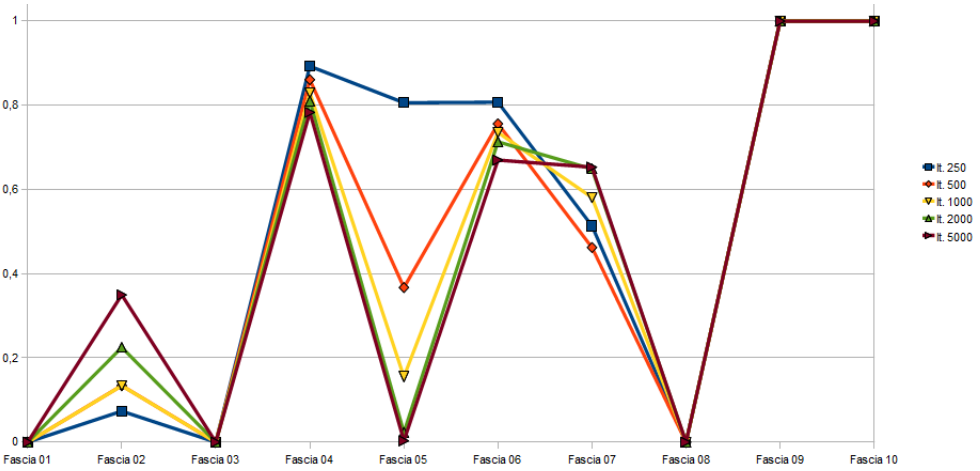

Figure 22: Altro esempio di rapporto tra le medie sui λ negativi

Al contrario quindi dei λ positivi che mostrano sempre una correlazione sulla prima fascia oraria, per i λ negativi si riscontra un'assenza di coerenza tra i trend mostrati.

Nel grafico in Figura 23 invece la misure aggregata analizzata non è un rapporto tra medie bensì un rapporto tra quante corse della soluzione buona abbiano λ nullo rispetto a

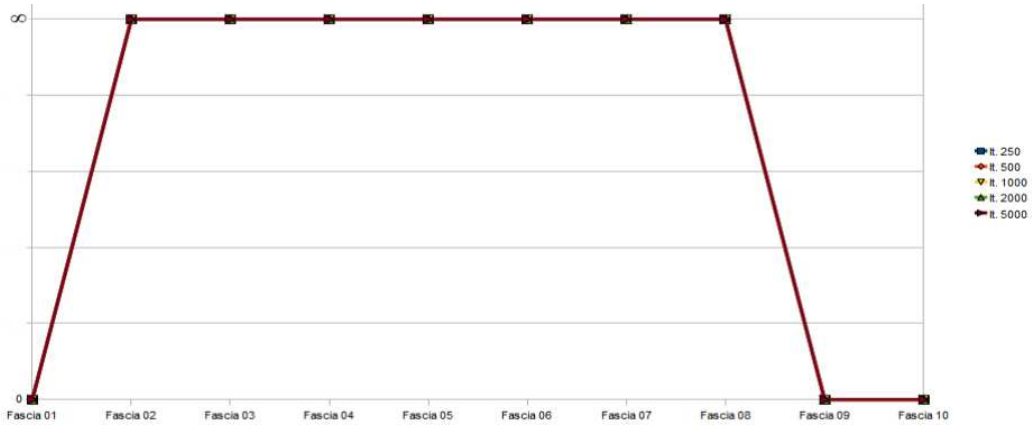

Figure 23: Rapporto tra le medie sui λ nulli

quante corse in genere abbiano λ nullo. Possiamo vedere che le possibilità emerse sono due: nelle fasce orarie "esterne" qualche corsa è caratterizzata da λ nullo ma nessuna di queste è tra quelle "buone", mentre nelle fasce interne nessuna corsa in genere ha λ nullo. I dati suggeriscono chiaramente come i λ nulli siano una caratteristica da non considerare.

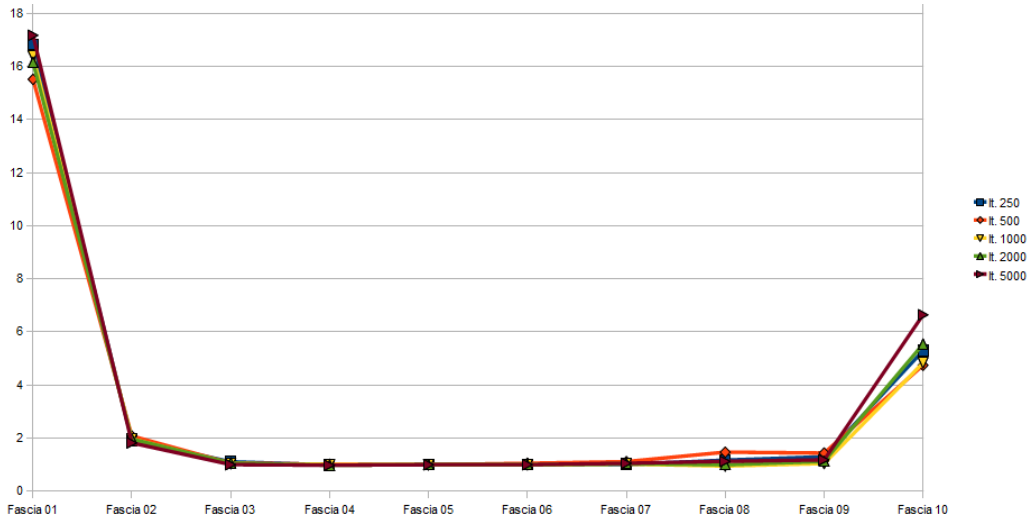

Figure 24: Rapporto tra le medie sui valori BS della soluzione continua

Riguardo ai valori espresssi dalla soluzione continua la situazione è risultata decisamente promettente.

Come vediamo nel grafico in Figura 24, che descrive l'andamento della misura aggregata calcolata tramite i valori espressi dalla parte BS della soluzione continua, e come possiamo vedere nei due grafici in Figura 25 e 26, che analizzano la medesima cosa ma sui sottografi TT, si vede che valori alti di tali misure comportano, soprattutto nella prima fascia oraria e poi a scendere andando avanti nel tempo, una probabile appartenenza della corsa al set di corse ritenute buone.

Dall'analisi di questi dati sembrò plausibile l'ipotesi di basare la regola di fissaggio su valori alti riscontrati nei λ e/o nei dati estratti dalla soluzione continua, limitando la fase di

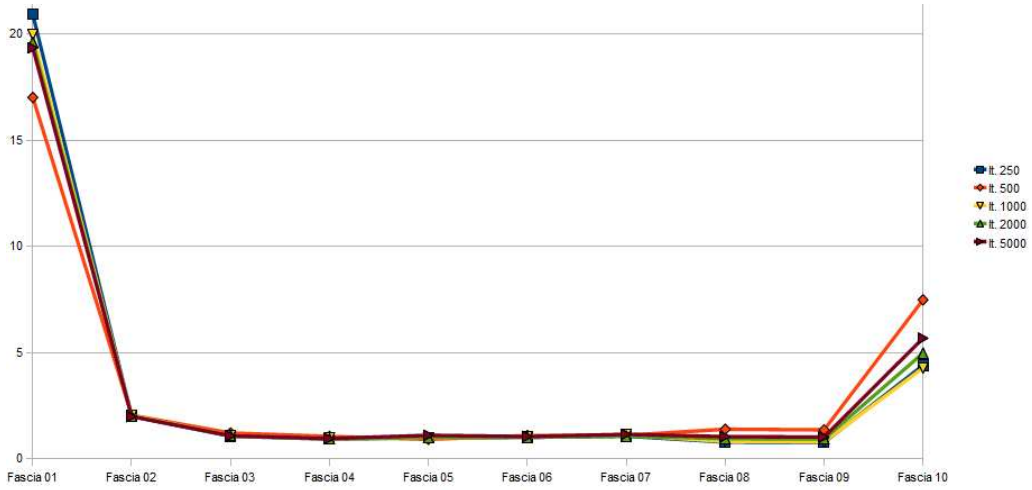

Figure 25: Rapporto tra le medie sui valori TT-0 della soluzione continua

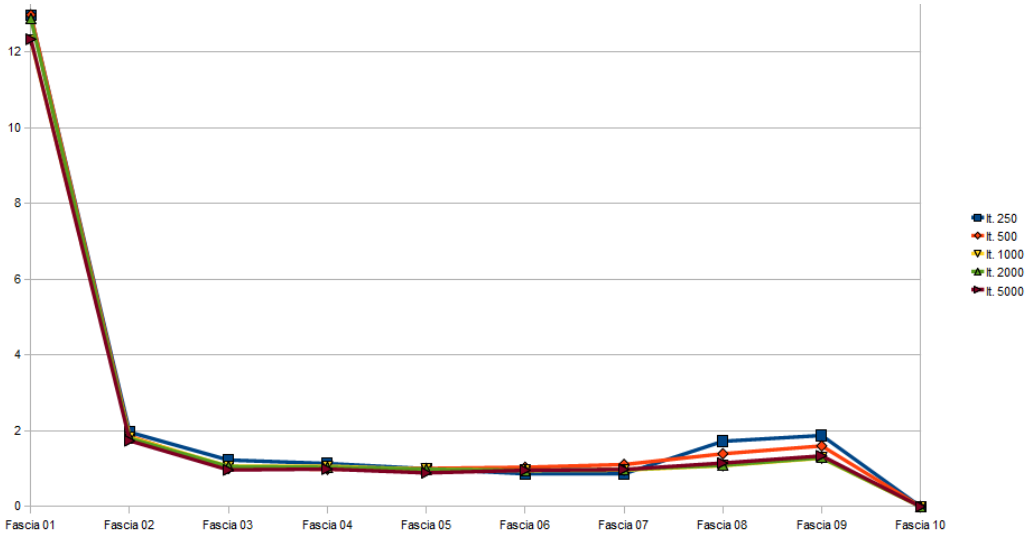

Figure 26: Rapporto tra le medie sui valori TT-1 della soluzione continua

fissaggio inizialmente alla prima fascia oraria per poi proseguire progressivamente in avanti. Risultava chiaro comunque come quest'ipotesi fosse basata su un'altra ipotesi, verificata empiricamente ma non certo dimostrata in maniera assoluta, ovvero che *Cplex* fosse in grado di trovare soluzioni ritenute buone. Inoltre lo studio fatto osservava il comportamento del codice per come, allora, era stato implementato, sollevando un problema di scarsa previdibilità del comportamento dello stesso una volta aggiornato con il meccanismo di fissaggio delle corse: non era assolutamente detto che il meccanismo di fissaggio, progressivo lungo l'arco della giornata a partire dall'inizio della stessa, avesse mantenuto la stessa aderenza alle soluzioni buone così come risultava dallo studio fatto sulla prima fascia.

Si decise in ogni caso di seguire l'ipotesi evidenziata, passando quindi alla fase di implementazione dell'euristica e quindi della regola di fissaggio così come è stata sopra spiegata.

5.1 L'implementazione dell'approccio euristico

Il nuovo approccio euristico fu implementato sfruttando la natura modulare e le interfacce astratte appositamente sviluppate all'inizio proprio allo scopo di poter in seguito modificare il codice in modo semplice, riducendo quindi il piu' possibile la possibilita' di introdurre errori in un codice che, pur palesando limiti in fase di raggiungimento di soluzioni buone in tempi ragionevoli, risultava comunque ben funzionante. Fu deciso quindi di inserire nella struttura già esistente una nuova classe, come si può osservare in Figura 27 (il "modulo rosso"), che implementasse l'interfaccia astratta *TTDAlgoInterface* e si inserisse tra *TTDMain* (vedi Paragrafo 4.2.1) e la classe sottostante (che implementa sempre l'interfaccia *TTDAlgoInterface*, vedi Paragrafo 4.2.3).

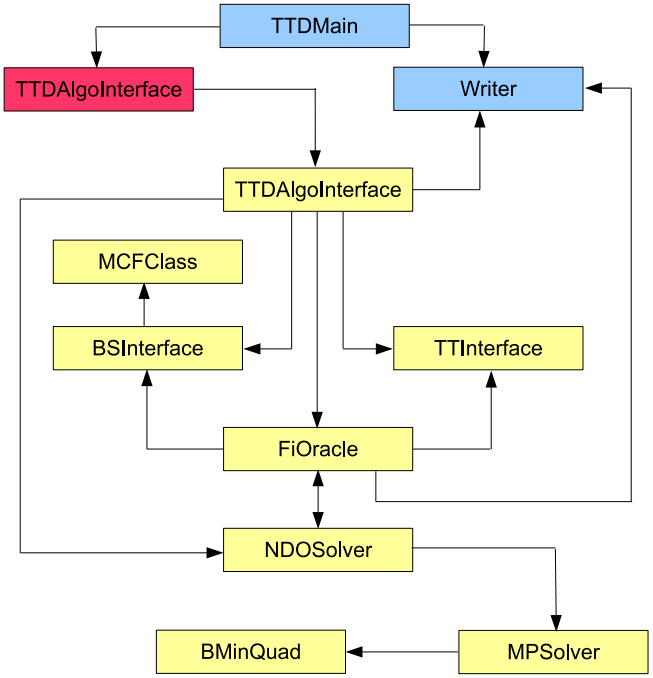

Figure 27: L'attuale struttura del codice

Se nella vecchia struttura *TTDMain* raccoglieva i dati d'input per passarli alla classe di tipo *TTDAlgoInterface* che gestiva i vari solutori dei sottoproblemi, ora *TTDMain* passa i suoi dati alla nuova classe addetta all'implementazione dell'euristica: il fatto che la nuova classe è anch'essa di tipo *TTDAlgoInterface* rende assolutamente indolore, dal punto di vista del codice insito in *TTDMain*, il cambio di struttura. La nuova classe ha quindi il compito di raccogliere i dati d'input e passarli proprio a quella classe di tipo *TTDAlgoInteface* che prima era collegata direttamente a *TTDMain*, istanzialdola per poi quindi controllarla secondo l'approccio euristico voluto. Prima di addentrarsi nei dettagli dell'implementazione di questa nuova classe è bene descrivere brevemente le modifiche apportate al codice preesistente.

- *TTDMain*, come già detto, è rimasto sostanzialmente inalterato, se non per un piccolo upgrade nella gestione della lettura del file di configurazione.

- Al file di configurazione (vedi Paragrafo 12 ogni qualvolta si fa riferimento ad esso o ad un suo parametro), per l'appunto, è stato aggiunto un set di nuovi parametri (vedi Paragrafo 12.6) atti a governare dall'esterno il funzionamento dell'euristica.

- La classe preesistente di tipo *TTDAlgoInterface* ha subito piccole modifiche dovute all'aggiunta di metodi pubblici (che la nuova classe avrebbe utilizzato) che gestissero il flusso di alcuni dati (ad esempio la soluzione continua e i λ oppure il fissaggio di determinate corse deciso proprio dalla euristica stessa) in un senso o nell'altro.

- I solvers dedicati di tipo *TTInterface* e *BSInterface* hanno subito un'estensione della "gerarchia di fissaggio delle corse" (vedi Paragrafo 4.2.4 e Paragrafo 4.2.5): i fissaggi ("a 1" o "a 0") decisi dall'euristica si pongono in mezzo gerarchicamente tra i fissaggi decisi a priori in input (i più importanti) e quelli "volatili" dovuti alla compilazione di soluzioni temporanee ammissibili.

- Infine la classe di tipo *FiOracle*, che lavora a stretto contatto col *Bundle*, è stata estesa con l'inserimento di nuove politiche per il calcolo (e la trasmissione) della soluzione continua, per la trasmissione dei λ, per la riduzione delle dimensioni del sottoproblema dovuta alle scelte operate dalla nuova euristica.

Detto questo, resta da descrivere nel dettaglio come l'intuizione, figlia dello studio spiegato nel paragrafo precedente, sia stata implementata all'interno della nuova classe. Lo studio statistico evidenziò come sia i valori (BS o TT) derivati dalla soluzione continua sia i λ (positivi) potessero fungere da guida per un fruttuoso fissaggio delle corse. In fase d'implementazione fu deciso che la regola di fissaggio avrebbe utilizzato uno solo dei tre set di valori, e la scelta sarebbe stata demandata all'utente stesso tramite il settaggio di un apposito parametro all'interno del file di configurazione. Lo sviluppo dell'euristica e lo studio sul suo possibile modo di utilizzarla compiuto da parte nostra in collaborazione con la M.A.I.O.R. fu però basato sulla scelta, fatta a priori, di utilizzare i valori derivati dalla parte BS della soluzione continua.

Prima di addentrarsi nei dettagli dell'euristica implementata, occorre premettere il fondamentale concetto di *finestra temporale*. Lo studio descritto nel Paragrafo 5 ha mostrato non solamente il tipo di valori che possono essere utili per il nostro scopo, ma ha anche suggerito come la correlazione tra codesti valori e corse che formano soluzioni valutate come buone sia più alta all'inizio della giornata, nello specifico all'interno della prima fascia oraria significativa (ovvero con intertempo desiderato non nullo), e decresca man mano si vada avanti nell'esaminare fasce orarie sempre più vicine al termine della giornata. Va precisato però che l'ipotesi di continuare ad utilizzare la suddivisione in fasce orarie anche all'interno della definizione della regola di fissaggio, centrale in questo approccio euristico, nascondeva delle possibili insidie. Innanzitutto vi era la possibilità che uno studio effettuato con una grana più fine, probabilmente scomodo per una valutazione sulle misure aggregate e quindi scartato in quel momento, avrebbe scoperto una correlazione più precisa: in altre parole l'ipotesi (empiricamente dimostrata più avanti proprio con studi fatti ad implementazione avvenuta) era che la correlazione poteva essere più forte non tanto sulle corse all'interno delle prime fasce orarie ma sulle corse più vicine all'inizio della giornata. Inoltre lo studio effettuato aveva portato ai risultati sopra mostrati ragionando su input caratterizzati da

fasce orarie che suddividevano in parti abbastanza simili e relativamente non molto grandi la giornata, ma non esisteva nessuna garanzia sugli input che tali condizioni fossero sempre verificate: poteva benissimo capitare una suddivisione della giornata con fasce orarie molto grandi. L'eventualità di fissare su fasce orarie in modo sequenziale, ovvero prima si fissa liberamente su una fascia per poi passare alla successiva una volta "esaurite" le corse papabili sulla fascia stessa, diventa sfavorevole in caso di fasce orarie grande perché può capitare di fissare simultaneamente corse a notevole distanza temporale e che eventualmente possono risultare per qualche motivi incompatibili tra loro. Per questi motivi è stato ideato il concetto di *finestra temporale*, ovvero un periodo di tempo che va dall'inizio della giornata fino ad un certo punto: la finestra è centrale perché l'euristica può fissare solo all'interno di essa. La finestra temporale sarà inizialmente piccola (in proporzione con la prima fascia oraria significativa) per poi crescere, gradualmente durante l'esecuzione in relazione a quanto l'euristica riesca a fissare corse, fino a raggiungere la fine della giornata: questo meccanismo garantisce la coerenza tra i fissaggi operati e quindi una migliore qualità delle soluzioni trovate.

Si può ora passare ad una sintetica descrizione del comportamento della nuova classe tramite pseudocodice.

```
Inizializza Moduli Inferiori;                                          (1a)
Preprocessing;                                                         (2a)
do {
    modInf.Solve(numIterazioni);                                       (3a)
    lambda = modInf.GetLambda();                                       (4a)
    solCont = modInf.GetSolCont();                                     (5a)
    solAmmiss = modInf.GetSolAmmiss();                                 (6a)
    if ( solCont è intera ) { esci dal ciclo; }                        (7a)
    if ( solAmmiss ha solo corse fissate qui ) { esci dal ciclo; }     (8a)
    if ( tutte le corse sono state fissate ) { esci dal ciclo; }       (9a)
    corseFixed0.Add( corse a 0 da defixZeroTrip turni );               (10a)
    RidimensionaFinestra();                                            (11a)
    FissaggioCorse();                                                  (12a)
    } while (condizioni d'uscita non verificate)
```

Una volta inizializzati i moduli inferiori (istruzione 1a) si esegue un piccolo preprocessing (istruzione 2a) volto a togliere tutte le corse che risultano antecedenti all'insieme delle corse papabili come "prima corsa" o postcedenti all'insieme delle corse papabili come "ultima corsa".

Fatto ciò si entra in un ciclo iterativo. La prima operazione svolta dall'euristica all'interno del ciclo (istruzione 3a) consiste nel passaggio del testimone ai livelli inferiori della struttura, che eseguono un po' del lavoro di risoluzione del Duale Lagrangiano a partire dall'insieme (inizialmente vuoto) di corse fissate finora calcolato. Nei moduli inferiori vengono quindi eseguiti `numIterazioni` risoluzioni dei sottoproblemi, con `numIterazioni` definito nel file di configurazione, prima che il comando ritorni all'euristica stessa. Un altro modo di vedere la cosa consiste nel concepire una intera esecuzione del vecchio algoritmo per la risoluzione del Duale Lagrangiano, così come accadeva prima dell'inserimento dell'approccio euristico, intervallato da alcuni interventi dell'euristica che, fissando progressivamente le corse, riducono

le dimensioni del problema. Il numero di iterazioni che il resto dell'algoritmo può eseguire all'interno di quello che chiamiamo *turno di fissaggio* (ovvero la singola iterazione del ciclo che stiamo descrivendo) è determinato da alcuni parametri nel file di configurazione (*firstIter-Bundle, parIter1*), uno che determina il numero di iterazioni al primo turno di fissaggio, l'altro che regola il numero di iterazioni nei rimanenti turni di fissaggio. A livello sperimentale è risultato che far girare di più i solvers negli strati inferiori al primo fissaggio rispetto che nei successivi migliora sia la qualità delle soluzioni trovate sia le prestazioni dell'algoritmo. Una volta che i solvers sotto hanno lavorato, l'euristica scarica nelle sue strutture dati quelle informazioni che le servono per procedere coi fissaggi, ovvero la soluzione continua, λ, la migliore soluzione ammissibile (istruzioni 4a, 5a, 6a).

A questo punto il controllo ritorna all'euristica stessa, che con le sue valutazioni terminerà il turno di fissaggio per passare, eventualmente, al prossimo. Uno dei compiti che ha l'euristica è quello di riuscire a capire se la soluzione ottima (il termine "ottima" sarebbe in realtà improprio) è già stata trovata per uscire quindi dal ciclo iterativo.

Vengono quindi esaminate diverse condizioni d'uscita (istruzioni 7a, 8a, 9a).

- La soluzione continua, sia nella parte BS che in quella TT, presenta valori tutti interi (o 0 o 1): questo significa che essa è già una soluzione ammissibile ed è anche ottima.
- La migliore soluzione ammissibile trovata è composta solamente da corse fissate dalla nuova euristica: questo significa che l'euristica ha fissato (a 1) un numero di corse sufficiente per formare una soluzione ammissibile, e quindi non esiste più la necessità di prendere ulteriori decisioni.
- Tutte le corse esistenti sono state fissate, a 0 o a 1.

Nel caso che una delle condizioni d'uscita fosse vera, l'iterazione termina senza fare nient'altro e, con essa, l'esecuzione stessa dell'euristica e dell'intero codice.

In caso contrario il prossimo passo (istruzione 10a) è volto a migliorare l'efficienza, sfoltendo un po' l'insieme delle corse sul quale l'euristica sceglie quelle da fissare: le corse il cui valore di riferimento risulta a 0 da `defixZeroTrip` turni di fissaggio consecutivi vengono automaticamente fissate a 0 e quindi escluse: il valore scelto per `defixZeroTrip` è 3.

Il passo successivo (istruzione 11a) porta ad aggiornare la dimensione della finestra temporale. Come già detto, all'interno del primo turno di fissaggio la finestra temporale è determinata in proporzione (il valore scelto per il parametro `percFirstLapTime` è 0.5) alla dimensione della prima fascia oraria significativa. Più complicato invece è l'espansione della suddetta finestra nei turni di fissaggio successivi al primo. Il meccanismo infatti ha come obiettivo l'allargamento della finestra temporale in funzione dei nuovi fissaggi fatti dall'euristica, ma deve tener di conto che non tutti i turni di fissaggio sono in grado di fissare corse. Può benissimo capitare che i valori calcolati dal *Bundle* siano privi di picchi significativi: esiste quindi, come vedremo, un meccanismo all'interno dell'euristica che scarta la corse, scelte per essere fissate, se il loro valore di riferimento è ritenuto basso rispetto ad una certa soglia prestabilita tramite il parametro `thresholdValue`. Tale meccanismo evita di fissare in base a valori troppo bassi e quindi poco significativi ma può, nel nostro caso, portare alla situazione in cui un turno di fissaggio non ha corse da fissare. Il meccanismo di ridimensionamento della finestra temporale si comporta diversamente a seconda dello scenario.

```
void RidimensionaFinestra() {
    if ( numFissaggiUltimoTurno > 0 )
       window = lastFixedTrip + intertempoRelativo * parWin1;
    else
       if ( numTotaleFissaggi > 0 )
          window = window + intertempoRelativo * parWin1;
    }
```

Se nell'ultimo turno di fissaggio alcune corse sono state fissate allora la finestra viene ricalcolata in base alla corsa fissata più vicina alla fine della giornata (`lastFixedTrip`), l'intertempo desiderato sul nodo pilota e sulla fascia oraria relativi a tale corsa ed a un parametro `parWin1` (il valore scelto per `parWin1` è 2) definito nel file di configurazione. Nell'altro scenario si procede invece ad un allargamento della finestra ma solo se l'euristica, nel corso dell'esecuzione, sia riuscita a fissare qualche corsa. Determinata la nuova finestra, sappiamo quale sottoinsieme delle corse possiamo considerare.

```
void FissaggioCorse() {
    corseFixed1 = empty; corseFixed0 = empty;                              (1b)
    listaCorseMigliori = TrovaCorseMigliori( maxFixTrips, window );        (2b)
    for each c in listaCorseMigliori {
       if ( value of c >= thresholdValue ) {                               (3b)
          corseFixed1.Add( c ); corseFixed0.Add( intorno di c );           (4b)
          }
       }
    if ( modInf.Test( corseFixed1, corseFixed0 ) == false ) {              (5b)
       corseFixed1 = empty; corseFixed0 = empty;                           (6b)
       for each c in listaCorseMigliori {
          if ( value of c < thresholdValue ) { continue; }
          fix1 = corseFixed1 + c; fix0 = corseFixed0 + intorno di c;       (7b)
          if ( modInf.Test( fix1, fix0 ) == false ) { exit for each; }     (8b)
          corseFixed1 = fix1; corseFixed0 = fix0;                          (9b)
          }
       }
    if ( corseFixed1.isEmpty() )                                           (10b)
       numEmptyFix++;                                                      (11b)
    else
       numEmptyFix = 0;                                                    (12b)
    if ( corseFixed1.isEmpty() ) && ( numEmptyFix == forcedFix ) {         (13b)
       foreach c in orderedTrip {                                          (14b)
          if ( modInf.Test( c, intorno di c ) == true ) {                  (15b)
             corseFixed1 = c; corseFixed0 = intorno di c;                  (16b)
             exit for each;                                                (17b)
             }
          }
       }
```

```
modInf.SetFixedTrips( corseFixed1, corseFixed0 );                    (18b)
}
```

L'ultima fase del ciclo iterativo (istruzione 12a, nel primo pseudocodice presentato) si occupa direttamente del fissaggio delle corse e la possiamo osservare nel dettaglio nell'ultima porzione di pseudocodice presentata. In base al valore di riferimento scelto, l'euristica stila una classifica di `maxFixTrips` corse (il valore scelto per `maxFixTrips` è 20) papabili per essere fissate a 1, ordinate in modo decrescente sul valore di riferimento scelto (istruzione 2b). Sulla scelta delle corse da fissare a 1 è stato imposto un vincolo: un valore troppo basso rispetto ad un valore `thresholdValue` prestabilito è infatti motivo di esclusione, quindi non è detto che le corse da fissare ad 1 siano alla fine proprio n (vedi istruzione 3b e 4b).

Per ognuna delle corse scelte per essere fissate a 1 (e quindi far parte della soluzione in costruzione) si fissa a 0 (quindi si esclude dalla soluzione in costruzione) un insieme di corse attigue, definito *intorno*, alla corsa appena fissata ad 1 (istruzione 4b). Questo perché è noto dalla definizione di grafo TT (vedi Paragrafo 3.2) che esiste la nozione di "intertempo minimo" tra corse, ovvero che due corse adiacenti hanno una distanza minima (nel tempo) da rispettare. È chiaro quindi che, ad esempio, se si sceglie una determinata corsa non si potrà in alcun modo scegliere la corsa che agisce dopo un minuto sullo stesso nodo pilota se l'intertempo minimo è definito maggiore di un minuto. Come possiamo vedere nello pseudocodice l'operazione di fixing a 0 sull'intorno della corsa è immediatamente successiva al fixing a 1, ovvero se abbiamo due corse da fissare a 1 si fisserà ad 1 la prima, quindi si fisserà a 0 sull'intorno della prima, infine si ripeterà il tutto sulla seconda corsa. Questo escamotage evita una possibile situazione di empasse: può capitare infatti che corse molto vicine tra loro abbiano valori alti; se l'euristica fosse implementata in modo da fissare prima a 1 per poi passare a fissare a 0 sugli intorni, potrebbe capitare che vengono fissate a 1 corse incompatibili tra loro.

Le decisioni prese in questa fase non sono definitive, nel senso che ancora devono essere comunicate ai moduli inferiori. È necessario infatti validarle, ovvero controllare che le eventuali decisioni di fixing prese non portino il sistema in uno stato in cui non sia più andare avanti, per un'eventuale violazione di un vincolo o per l'impossibilità di trovare una soluzione ammissibile a partire dai fissaggi fin qui decisi. Per scongiurare questa eventualità le decisioni prese vengono passate ai moduli inferiori che le testano "in sicurezza", ovvero senza che nessuna modifica ai grafi sia effettivamente compiuta. Come vediamo nello pseudocodice (istruzione 5b) nel caso che la valutazione preveda uno stato di inconsistenza il set di decisioni viene azzerato (istruzione 6b) quindi ricreato pezzo per pezzo testando se la singola decisione presa non porti in uno stato di inconsistenza: in pratica si fissa ad 1 la prima corsa nella "classifica" e a 0 il suo intorno e si esegue il test e se tutto va bene si passa alla seconda corsa e via dicendo, fermandosi nel momento in cui emergono dei problemi (istruzioni 7b, 8b, 9b).

Può capitare comunque che, nonostante tutto, l'euristica non possa, per via dei vincoli imposti, fissare nessuna corsa all'interno del turno di fissaggio corrente e sia quindi ricostretta a passare al prossimo turno di fissaggio ripresentando agli strati inferiori del codice il problema nella medesima forma del turno precedente. Può capitare inoltre che l'euristica si trovi nella condizione di non poter fissare niente per diversi turni di fissaggio consecutivi, facendo quindi lavorare il resto del codice senza apportare quell'aiuto per la quale è stata ideata e

implementata. Per scongiurare questo è stato creato un meccanismo per il quale se dopo `forcedFix` turni di fissaggio consecutivi (il valore scelto per `forcedFix` è 15), l'euristica non ha fissato niente allora viene fissata ad 1 un'unica corsa (a prescindere quindi da un eventuale valore di riferimento più basso della soglia) a patto che, ovviamente, questa operazione non porti in uno stato inconsistente (istruzioni da 13b a 17b). Eseguite le verifiche sopra spiegate si passa alla vera e propria fase di fixing, passando l'insieme delle corse da fissare ai moduli sottostanti, quindi si reinizia il ciclo passando al prossimo turno di fissaggio (istruzione 18b).

Al lavoro di definizione e implementazione dell'approccio euristico è stato affiancato uno studio simultaneo, in stretta collaborazione con gli esperti M.A.I.O.R., delle soluzioni che l'euristica stava producendo. Lo studio svolto riguardava il giusto set di parametri da passare tramite file di configurazione, studiando i valori sia dei parametri che regolano il funzionamento stesso dell'euristica sia dei costi che formano la funzione obiettivo. La regolazione dei parametri strettamente legati al funzionamento dell'euristica fu volto alla ricerca del giusto equilibrio tra accuratezza della ricerca (e quindi, consequenzialmente, qualità delle soluzioni trovate) ed efficienza.

Ad esempio:

- Si è scoperto come il lasciar "girare molto" il *Bundle* nel primo turno di fissaggio alzi in modo consistente il tempo d'esecuzione delle prime fasi (dove tra le altre cose il problema non è ancora stato ridotto dai fissaggi), ma complessivamente abbassi i tempi d'esecuzione e migliori la qualità delle soluzioni trovate in fondo: è evidente che porre buone basi all'inizio dell'esecuzione migliori la qualità del lavoro anche delle fasi successive: è stato quindi deciso di far girare il *Bundle* per 3000 iterazioni nel primo fissaggio e per 100 iterazioni nei successivi.

- È importante definire la soglia riguardante i valori di riferimento in modo equilibrato: una soglia troppo alta è dispendiosa in termini di tempo perché limita notevolmente il numero di corse che si possono scegliere per il fixing, una soglia troppo bassa invece migliorerebbe le performance in termini di tempo ma inficerebbe la qualità delle scelte e quindi la qualità delle soluzioni trovate. È stato inoltre verificato che, a causa di scelte non proprio ottime sui primi fissaggi per via di una soglia troppo bassa, con l'avanzare dell'esecuzione si verificano sempre più spesso stati di inconsistenza che finiscono per penalizzare oltre che la qualità delle soluzioni anche l'efficienza stessa. In base a queste considerazioni è stato deciso di definire il valore della soglia pari a 0.45, tenendo ovviamente di conto che era già stato prestabilito di fissare usando i valori BS della soluzione continua.

- Anche l'incremento della finestra temporale risulta molto importante, poiché incrementare troppo significa esporsi sempre più a possibili scelte di fissaggio in contrasto tra loro, mentre incrementare poco significa negare le condizioni per le quali l'euristica può effettuare nuovi fissaggi: si è deciso quindi, come già abbiamo detto, di definire il relativo parametro `parWin1` pari a 2.

In concomitanza sono stati ricalibrati i parametri di costo che formano la funzione obiettivo del problema, sfruttando il sopravvenuto miglioramento esponenziale dell'efficienza algoritmica che permetteva di trovare soluzioni in tempi ragionevoli. Sono stati quindi trovati

valori per codesti parametri in modo tale che il codice potesse trovare soluzioni ritenute soddisfacenti dagli esperti M.A.I.O.R. in termini di qualità e quindi di equilibrio tra i diversi obiettivi da perseguire. Il significato dei parametri qui sotto elencati lo si può trovare nel Paragrafo 12 oppure, volendo entrare nel dettaglio, nei Paragrafi 3.1 e 3.2 che descrivono rispettivamente i sottomodelli BS e TT.

`gammaM = 90, gammaI = 0.07, gammaF = NULL`[4]`, gammaS = 0.00001`

`alfa = 2`

`a = 1, b = 0.6, c = 0.01`

`aI = 1, bI = 0.1, cI = 1`

`aF = 1, bF = 0.1, cF = 1`

Concludendo, il risultato del lavoro di implementazione dell'approccio euristico ha permesso al codice di effettuare il cosidetto "salto di qualità". Il nuovo codice è stato infatti più volte testato su molte istanze di problema diverso. È stato quindi trovato un set di valori per i parametri di configurazione che andasse bene un po' in tutti i casi, in modo tale che il nuovo codice fosse in grado di trovare soluzioni equilibrate e quindi buone in tempi ritenuti ragionevoli (nell'ordine delle decine di minuti e non delle giornate intere).

[4]Un'opzione del codice prevede che se gammaF = NULL allora non si può fare compatibilità fuori linea, ovvero i bus non possono sostare al deposito: in altre parole ogni bus può fare un solo viaggio all'interno del proprio Turno Macchina.

6 Il controllo del flusso dei veicoli

Per quanto con l'introduzione dell'euristica la qualità delle soluzioni trovate fu ritenuta nel complesso buona, emerse nei molteplici test fatti un possibile problema, "limitato" a piccole porzioni della soluzione ma comunque abbastanza importante all'interno di una valutazione che tenesse pienamente conto delle aspettative dei possibili utenti.

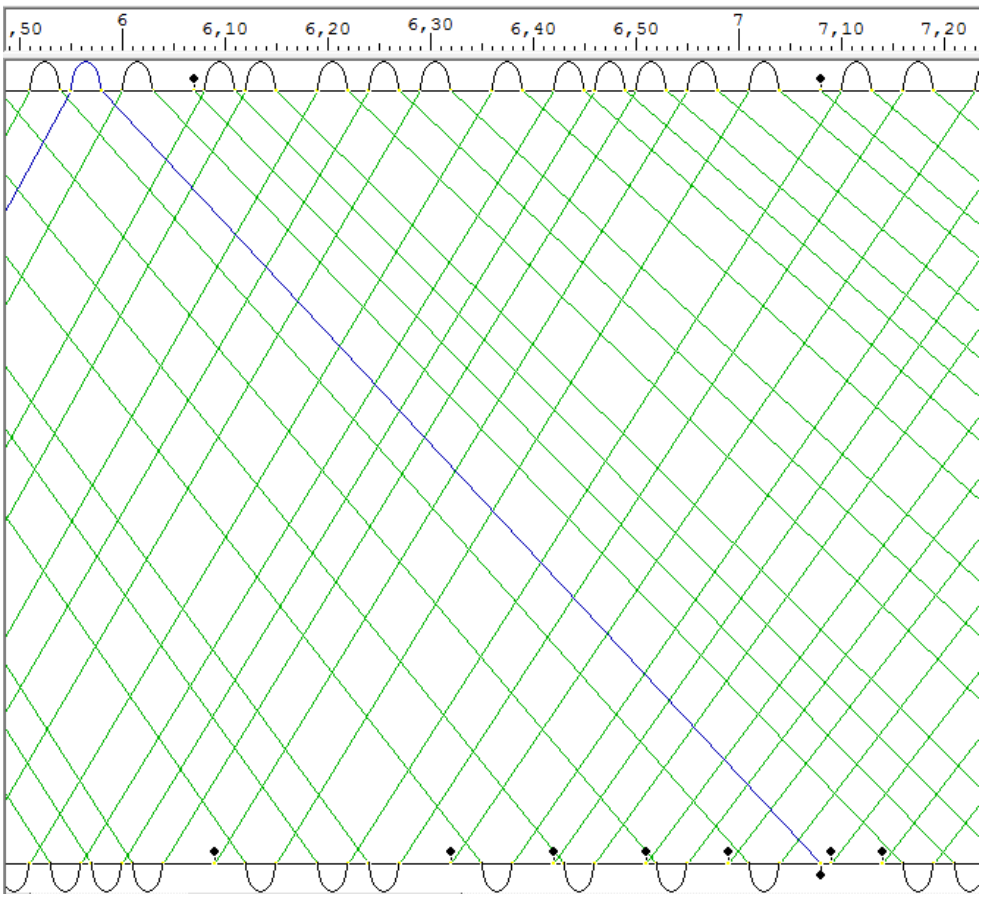

Figure 28: Un esempio del problema emerso

Nella figura 28, realizzata tramite un software di proprietà della M.A.I.O.R., è mostrato in forma grafica un esempio di scenario dove si verifica il problema suddetto.

Nella figura possiamo osservare molte linee verdi (e una blu) andare in diagonale da una prima linea orizzontale verso una seconda linea orizzontale (e viceversa): le due linee orizzontali rappresentano due capolinea, che chiameremo A (quello di sopra) e B. Ogni linea verde (blu) "spezzata" (ovvero il singolo segmento che va da un capolinea all'altro) è una corsa coperta, nella soluzione esaminata, da un bus. Ad esempio, osservando il segmento blu che si riesce ad osservare per intero, si può vedere come un bus copra la corsa che parte da A alle 5:58 e termina in B alle 7:07. Possiamo inoltre osservare come non esista un unico segmento blu: questo perché il bus suddetto non copre solamente quella corsa, ma in precedenza ne copre almeno un'altra. Osservando quindi la figura si può affermare che il "bus blu" copre una corsa che non si può dire quando parte da B ma si può dire che arriva

in A alle 5:55, quindi sosta 3 minuti su A (la sosta al capolinea è indicata graficamente dall'archetto) e riparte alle 5:58 per arrivare in B, come detto sopra, alle 7:07; a questo punto, come indicato dal rombo nero che si stacca dalla linea del capolinea B scendendo in basso, il bus ritorna al deposito. Analoghe considerazioni valgono per le linee verdi in figura: ognuna di esse rappresenta un bus che copre alcune corse, fa alcune soste, esce (rombo nero che si stacca in basso dalla linea) o entra (il rombo si stacca in su) in servizio andando verso il (venendo dal) deposito.

Detto questo si può facilmente notare come alle 7:07 un bus parta dal capolinea B e rientri al deposito, mentre alle 7:08 e alle 7:13 altri due bus arrivino al medesimo capolinea (iniziando quindi il loro lavoro) partendo in precedenza dal deposito. È facile immaginare come una situazione simile, dove un bus va via da B mentre un altro bus arriva in B poco dopo, sia ritenuta sconveniente da un utente per via di un non proprio ottimale utilizzo dei mezzi. Basterebbe ad esempio (ma è solo uno dei modi possibili per correggere tale situazione) ridurre un po' (se possibile) la sosta al capolinea A del bus blu in modo tale che esso, coprendo una corsa precedente a quella delle 5:58 - 7:07, possa sostare al capolinea B quel tanto che basta per non violare il vincolo di sosta minima al capolinea (che probabilmente ha costretto il codice a far entrare il nuovo bus) e quindi sostituire il nuovo bus nel lavoro che la soluzione in esame gli avrebbe fatto fare.

È ipotizzabile che tale accorgimento "manuale", nei casi in cui risulta possibile (ammissibile rispetto ai vincoli), non è scelto dall'algoritmo per via di un equilibrio tra i costi della funzione obiettivo probabilmente sbilanciato (ma non troppo, visto che comunque le soluzioni erano ritenute buone). È stato ipotizzato quindi che la funzione obiettivo fosse settata in maniera tale che l'algoritmo preferiva in una qualche (piccola) misura ottimizzare le tabelle orarie piuttosto che lo scheduling dei veicoli, preferendo creare una situazione come quella descritta sopra piuttosto che compilare orari peggiori.

Le numerose verifiche sperimentali, guidate da una ricerca approfondita di un set di parametri per la funzione obiettivo che fosse ancora più equilibrato, non hanno però portato i risultati sperati. Nella maggior parte dei test infatti piccoli spostamenti dei valori dei parametri a favore dello scheduling portavano all'eliminazione del problema ma anche ad un peggioramento globale della qualità della soluzione. Nei pochi casi in cui le modifiche migliorarono la soluzione (eliminando quindi il problema), esse risultarono poco generali e molto legati alla specifica istanza di problema: poiché era necessario un set di parametri che andasse bene per molti casi e non per istanze particolari, la strada delle modifiche ai costi risultò fallimentare.

Particolarmente utile invece risultò essere il fatto che gli utenti, al momento della compilazione dell'input, sono in grado di effettuare stime ragionevoli sul numero di bus, in ogni momento della giornata, necessari per effettuare il servizio.

Tale stima, come già anticipato nel Paragrafo 3.1.1, è calcolata mediante il concetto di *tempo giro*, che qui andiamo a spiegare in maniera più estesa: il tempo giro, in un dato momento della giornata, è il tempo totale che impiega un bus a partire da A verso B, effettuare una sosta minima al capolinea B (quella imposta dai vincoli del problema), quindi ritornare in A ed effettuare anche su A una sosta minima. In pratica il tempo giro, come si evince anche dal nome stesso, definisce il minor tempo necessario ad un bus per compiere un giro su due capolinea (andata e ritorno) per quindi ritornare al punto di partenza. Chiaramente il tempo giro dipende dai tempi di percorrenza, diversi nei vari momenti della giornata a causa

del diverso traffico che si può incontrare, e dalle soste minime ai capolinea imposte come vincoli al problema. La stima del numero di veicoli è data quindi dal rapporto tra tempo giro e intertempo desiderato in quel dato istante t ($\frac{TG_t}{I_t}$): è facile intuire infatti che se un bus impiega, ad esempio, 20 minuti per fare un "avanti-indietro" tra A e B allora esso passerà dal nodo pilota del tratto "A-B" una volta ogni 20 minuti, ed è altrettanto facile presumere che se l'intertempo desiderato su quel nodo pilota fosse di 10 minuti allora servirebbero almeno 2 vetture per effettuare un servizio ragionevole. È necessario quindi considerare periodi di tempo dove il tempo giro e l'intertempo desiderato non siano mutevoli: qualora cambiasse uno dei due valori, cambierebbe di conseguenza la stima delle vetture necessarie. Poiché si sta parlando di una stima, peraltro con buona possibilità di essere un numero "con la virgola" quando invece serve chiaramente un numero intero, ci si aspetterebbe che essa venga arrotondata per eccesso o comunque aumentata nel momento in cui la si voglia imporre misura di valutazione o addirittura, come vedremo, come vincolo del problema: è chiaro però che un arrotondamento per eccesso contrasta con il sotto-obiettivo del problema stesso che vuole un'ottimizzazione del numero di veicoli impiegati. La tendenza quindi è quella di arrotondare per eccesso nei momenti "di punta" della giornata laddove l'intertempo richiesto è più breve e il traffico previsto è più congestionato, privilegiando quindi la precisione nel timetabling a scapito dell'ottimizzazione dello scheduling dei veicoli, mentre nei momenti più "tranquilli" si preferisce un arrotondamento per difetto. Tramite queste stime gli utenti controllano la bontà delle soluzioni: anche in base a queste stime quindi le soluzioni dell'algoritmo sono state giudicate buone o meno buone (nel caso si presenti il problema suddetto) dagli esperti M.A.I.O.R.

Fu ipotizzata perciò una piccola modifica al modello stesso. Il modello descritto fino a quel momento permetteva solamente di porre un limite superiore sul numero di vetture da poter usare. Lo sviluppo ipotizzato, mirato a modificare il modello di schedulazione dei bus descritto nel Paragrafo 3.1, consiste nell'effettuare il cosidetto *controllo del flusso*, ovvero inserire nel modello la suddetta conoscenza a priori della stima del numero di vetture necessari nei vari periodi della giornata. L'idea è cercare di avvicinarsi ad un obiettivo che resta comunque impossibile da conseguire nella sua totalità, ovvero controllare con precisione il numero di vetture in un dato istante della giornata. Si cerca quindi di controllare, in modo più o meno dettagliato, il numero di bus che possono entrare o uscire dal deposito in un determinato momento della giornata, forzando l'algoritmo a trovare soluzioni più in linea con la stima del numero di vetture sopracitata. Facendo un esempio, se a priori sappiamo che da una certa ora in avanti il numero di vetture necessario sale da 5 a 9 si può, nel periodo dove si richiedono 9 vetture, impedire qualsiasi tipo di entrata verso il deposito e contemporaneamente permettere l'uscita dei veicoli dal deposito; in più, laddove il modello lo permette, si può vincolare il modello per permettere l'entrata di massimo 4 bus (quelli che servono).

Restano da stabilire alcuni punti importanti:

- in quale "zona" del modello si inserisce tale controllo;
- il livello di controllo che si intende implementare;
- come gestire la combinazione del controllo del "flusso in entrata" (bus che partono dal deposito) con il "flusso in uscita" (bus che tornano al deposito) nell'arco della giornata

in relazione alla stima a priori delle vetture necessarie e del livello di controllo scelto.

Considerando operativa la differenza tra inizio (fine) TM e compatibilità fuori linea introdotta nel Paragrafo 3.1.4, il controllo del flusso sarà effettuato proprio su queste due zone del modello: il livello di controllo però sarà differenziato, poiché a inizio/fine giornata si può essere più raffinati. Per quanto riguarda gli archi di compatibilità fuori linea infatti è possibile solo un controllo di livello più basso: se tra un periodo della giornata e il successivo la stima delle vetture non aumenta allora si impedisce l'uscita dei bus dal deposito, altresì se la stima delle vetture tra un periodo e il successivo non diminuisce allora si impedisce l'entrata al deposito dei bus che stanno circolando. Impedire l'uscita dal deposito significa non inserire nel grafo, nel momento della sua costruzione, gli archi che collegano i nodi c^- delle corse c appartenenti al periodo indicato ai relativi nodi O_t[5]; analogamente impedire l'entrata dal deposito significa non inserire alcuni archi che collegano i nodi $c+$ di alcune corse c ai nodi O_t. Nella zona d'inizio/fine dei TM invece è possibile un controllo più raffinato, tramite una piccola modifica del modello finora descritto. Oltre a impedire il flusso in entata/uscita, come si faceva con gli archi di compatibilità fuori linea, si aggiunge un controllo sul numero massimo di bus che possono entrare/uscire dal deposito. Ad esempio se in un periodo sono previste 3 vetture e il periodo successivo 4, allora per il periodo successivo si prevede che non entrerà in deposito nessun bus e ne potrà uscire al massimo 1.

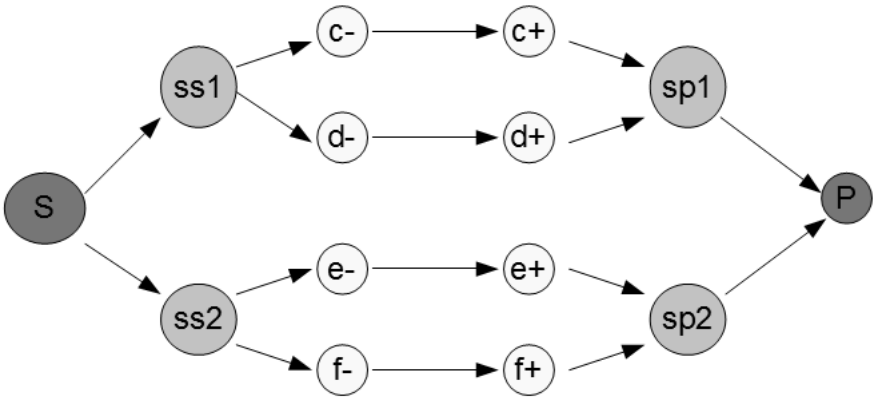

Figure 29: L'estesione della parte del grafo che modella l'inizio e la fine dei TM

Analogamente se tra un periodo e l'altro il numero di vetture usate diminuisse, allora durante il secondo periodo non potranno uscire vetture dal deposito ma potranno entrare nel deposito un numero di vetture che al massimo potrà essere la differenza tra le vetture richieste prima e quelle richieste dopo. Questo meccanismo si attua inserendo, per ogni periodo di tempo[6], due nodi che chiameremo *sorgente locale* e *pozzo locale*.

Se prima per ogni corsa c vi era un arco che collegava la sorgente S al nodo c^- e un altro arco che collegava il nodo c^+ al pozzo P, ora si sostituiscono questi due archi con le seguenti

[5]il verso dell'arco va dal nodo O_t al nodo c^-

[6]la giornata sarà suddivisa in periodi di tempo, dove per ognuno di questi abbiamo una stima delle vetture che servono; non necessariamente tali periodi di tempo devono coincidere cone le fasce temporali definite in altri punti del modello.

"catene" di archi: dalla sorgente alla sorgente locale e infine al nodo c^- per il controllo sul flusso in uscita dal deposito per fare (come prima corsa) la corsa c, più dal nodo c^+ al pozzo locale e quindi al pozzo per il controllo sul flusso in entrata al deposito (c è l'ultima corsa eseguita prima della fine del TM). Nella Figura 29 si vede bene come le due sorgenti locali e i due pozzi locali si inseriscono, rispettivamente, tra la sorgente e le corse e tra le corse e il pozzo. Con questa particolare configurazione si può controllare agevolmente il flusso in entrata/uscita dal deposito di un intero periodo di tempo semplicemente modificando la capacità dell'arco che dalla sorgente porta alla sorgente locale o dell'arco che dal pozzo locale porta al pozzo: ovviamente ponendo a 0 la capacità di uno dei due archi si impedisce alla vettura di entrare (o uscire) in quel determinato periodo. Resta da capire come si possono calcolare per ogni periodo, partendo dalla suddivisione della giornata in periodi temporali e dalla stima delle vetture necessarie in ognuno di questi periodi, la capacità massima in entrata e in uscita.

La spiegazione proposta qui sotto è uno dei possibili modi di procedere. Prima di tutto occorre premettere che:

- non possono esistere, in cima o in fondo alla lista delle stime, periodi di tempo che stimano 0 vetture: nel caso esistessero essi vengono accorpati al primo (ultimo) periodo della lista che abbia una stima non nulla, mantenendo la stima non nulla;
- si attribuisce ad ogni corsa un periodo di appartenenza, identico sia per il collegamento alla relativa sorgente locale sia per il collegamento al relativo pozzo locale: il periodo d'appartenenza è scelto utilizzando come criterio l'orario di partenza della corsa stessa.

Detto questo, il numero massimo di vetture che possono uscire dal deposito e il numero massimo di vetture che possono entrare al deposito sono calcolati, per ogni periodo, secondo il meccanismo descritto qui sotto nell'esempio.

Stima numero vetture	Num. Max in uscita dal deposito	Num. Max in entrata verso il deposito
4	4	1
3	0	0
5	2	3
2	0	2

Figure 30: Esempio di controllo del flusso

La stima delle vetture è data in input, mentre le altre due colonne sono calcolate a partire dalla stima delle vetture. Chiamiamo *Stima* il vettore delle stime delle vetture in ogni periodo, ed s l'ultimo elemento di Stima.

Il numero massimo di vetture in uscita dal deposito all'i-esima riga (periodo) è:

$$\begin{cases} Stima[1] & \text{se } i = 1 \\ max(0, Stima[i] - Stima[i-1]) & \text{altrimenti} \end{cases} \tag{12}$$

ll numero massimo di vetture in entrata verso il deposito all'i-esima riga (periodo) è:

$$\begin{cases} Stima[s], & \text{se } i = s \\ max(0, Stima[i] - Stima[i+1]) & \text{altrimenti} \end{cases} \tag{13}$$

I risultati sperimentali hanno mostrato un netto miglioramento della situazione: laddove sussisteva il problema del cattivo utilizzo dei veicoli, il codice con al suo interno la modifica al modello descritta ha saputo trovare soluzioni buone, cioè equilibrate nelle sue componenti e prive del suddetto problema, in tempi ragionevoli.

Tipo d'istanza	Num. Veicoli usati	Valore f.o.	Tempi
senza controllo del flusso	47	7723,84	22 min
con controllo del flusso	28	4787,83	22 min

Figure 31: Tabella comparativa tra i risultati ottenuti senza e con il controllo del flusso sulla medesima istanza di problema

Nella tabella in Figura 31 possiamo osservare come, a parità di tempo d'esecuzione, il codice ha una miglior gestione dei veicoli in presenza del controllo del flusso, vantaggio che si riflette in maniera evidente anche confrontando i valori delle due funzioni obiettivo. Nella Figura 32 invece possiamo osservare come, nel medesimo periodo di tempo dove senza controllo del flusso si verificava la cattiva gestione dei veicoli (vedi Figura 28), il problema scompare.

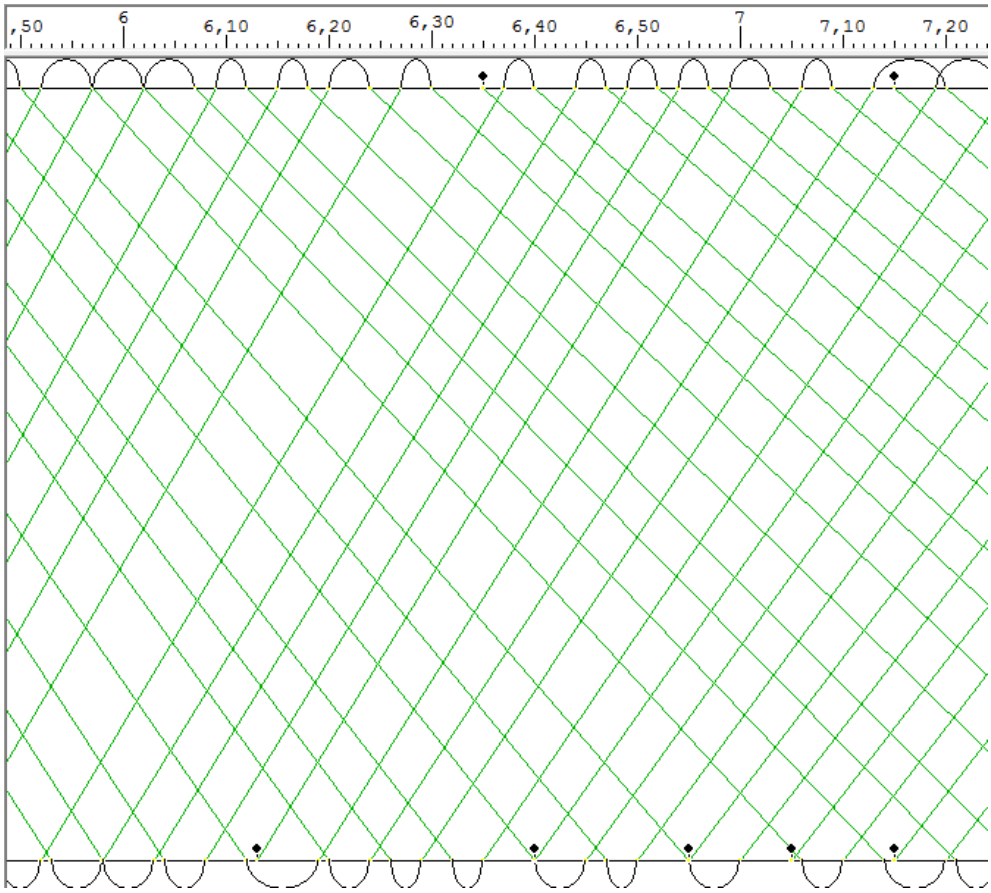

Figure 32: Il problema suddetto è scomparso

Concludendo, si può dire che la modifica al modello ha avuto successo. L'emergere del problema suddetto sull'impiego "localizzato" dei veicoli è stato lo stimolo decisivo per inserire all'interno del modello un meccanismo dedito al tentativo di controllare il numero dei veicoli non su tutto l'arco della giornata ma approfondendo il controllo anche sulle singole porzioni della stessa. Il principale obiettivo, ovvero la risoluzione del problema emerso, è stato raggiunto. Altrettanto importante però è il fatto che l'introduzione del controllo del flusso riesca a modellizzare con buona aderenza l'obiettivo, nella sua totalità irraggiungibile per come è strutturato il sottomodello BS, di permettere all'utente (se vuole) di avere il controllo momento per momento del numero dei veicoli. I test che abbiamo fatto, con il supporto degli esperti M.A.I.O.R., eran volti a scoprire se l'algoritmo, con l'inserimento di un controllo del flusso mirato, fosse in grado di trovare in tempi ragionevoli soluzioni di buona qualità e in linea con stime create per utilizzare il giusto numero di veicoli (calcolato tramite il meccanismo di generazione delle stime sopra illustrato). Niente vieta però all'utente di poter utilizzare il meccanismo di controllo del flusso per cercare soluzioni dalla natura più particolare, senza "impazzire" con modifiche dei pesi in funzione obiettivo che sono sicuramente più complicate e meno precise. Con questo meccanismo, ad esempio, è possibile cercare di ottenere soluzioni che, in uno o più momenti della giornata, siano molto parsimoniose nell'impiego dei veicoli; è altrettanto possibile simulare, in uno o più momenti della giornata, un blocco delle uscite (o entrate) dei mezzi dal deposito. Si può affermare quindi che l'introduzione del controllo del flusso all'interno del modello abbia aggiunto molta versatilità, soprattutto vedendo le cose nell'ottica dell'utente, al modo di utilizzare il codice.

7 Le frequenze frazionarie

Completato il *controllo del flusso* fu possibile dichiarare finita la versione "base" del codice. Rimanevano comunque ancora da studiare i possibili sviluppi descritti nel Paragrafo 3.5: ne sono stati studiati e implementati due, quelli ritenuti più importanti. Il primo di questi consiste nell'introduzione della possibilità di utilizzo di intertempi desiderati non multipli del fattore di scala, ed è l'argomento affrontato in questo paragrafo. Il secondo, che sarà trattato nel Paragrafo 8, consiste nella possibilità di lavorare su n percorsi invece che i 2 permessi finora.

L'utilizzo di intertempi non multipli, che da qui in avanti chiameremo *frequenze frazionarie*, ha comportato una modifica nel modo di costruire il sottomodello TT. Ripetendo per chiarezza espositiva concetti già espressi nel paragrafo 3.5.3, si suppone che se $F_s > 1$ allora può accadere che, in una data fascia oraria h, I_h/F_s non sia un numero intero, ossia $I_h/F_s = Q * F_s + R$ con $R > 0$. Si definisce quindi $\alpha = \frac{R}{F_s}$, $I_h^- = Q * F_s$ e $I_h^+ = I_h^- + F_s$; così facendo risulterà sempre $\alpha < 1$. Teoricamente potrebbero essere implementati in questa maniera tutti i possibili valori di α. Nell'esempio fatto nel Paragrafo 3.5.3, descritto nella Figura 33, si può osservare come il modelloo descriverebbe il caso $\alpha = 0.5$.

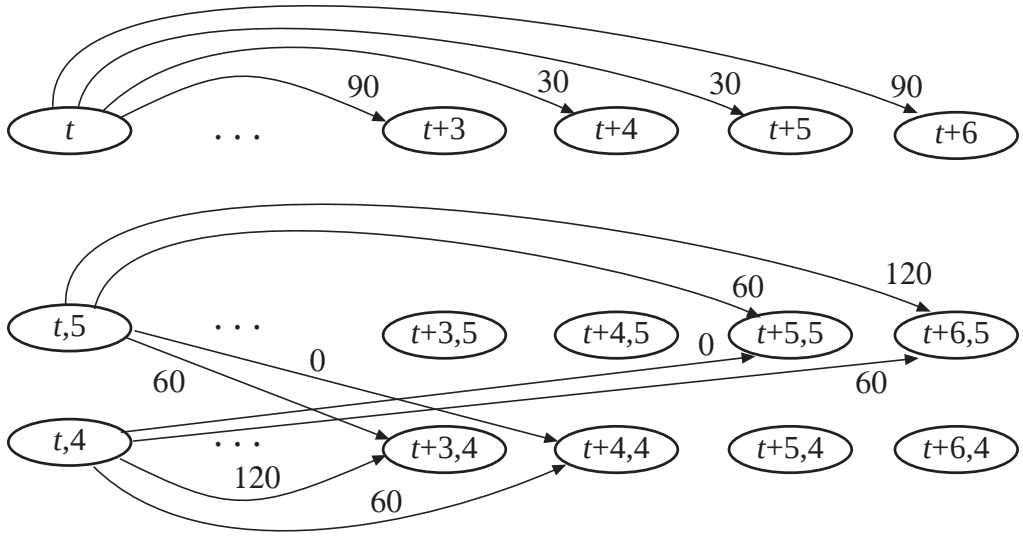

Figure 33: Modifica all'SPT per gestire le frequenze frazionarie (sono rappresentati solo alcuni archi)

In pratica però risulta necessario limitare il numero di valori diversi di α da gestire, per poi ricondursi (tramite approssimazione) sempre ad uno di questi qualora il valore effettivo di α non fosse conforme. Come si può vedere nella Figura 33 ogni corsa è rappresentata nel grafo da due nodi; in generale il numero di nodi per corsa risulta alto in presenza di schemi "complicati". Ad esempio se α fosse 0.1 lo schema relativo sarebbe I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^+. Seguendo questo tipo di logica, $\alpha = 0.01$ prevederebbe uno schema con 99 livelli del tipo I_h^- più uno del tipo I_h^+. È evidente quindi che cercare di rappresentare valori sempre più piccoli di α, nel tentativo di modellare con maggiore precisione la realtà, comporta la creazione di schemi sempre più complicati che rischiano di aumentare notevolmente la

complessità computazionale e l'occupazione di memoria. Il compromesso tra precisione ed efficienza obbliga quindi a scegliere un numero limitato di valori possibili di α, optando per una granularità meno fine.

La scelta che è stata fatta prevede i seguenti valori di *alpha* e, quindi, i seguenti schemi riportati:

- $\alpha = 0.1$: lo schema di intertempi desiderati è I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^-, I_h^+;
- $\alpha = 0.2$: lo schema di intertempi desiderati è I_h^-, I_h^-, I_h^-, I_h^-, I_h^+;
- $\alpha = 0.3$: lo schema di intertempi desiderati è I_h^-, I_h^-, I_h^-, I_h^+, I_h^-, I_h^-, I_h^+, I_h^-, I_h^-, I_h^+;
- $\alpha = 0.4$: lo schema di intertempi desiderati è I_h^-, I_h^-, I_h^+, I_h^-, I_h^+;
- $\alpha = 0.5$: lo schema di intertempi desiderati è I_h^-, I_h^+;
- $\alpha = 0.6$: lo schema di intertempi desiderati è I_h^-, I_h^+, I_h^-, I_h^+, I_h^+;
- $\alpha = 0.7$: lo schema di intertempi desiderati è I_h^-, I_h^+, I_h^+, I_h^-, I_h^+, I_h^+, I_h^-, I_h^+, I_h^+, I_h^+;
- $\alpha = 0.8$: lo schema di intertempi desiderati è I_h^-, I_h^+, I_h^+, I_h^+, I_h^+;
- $\alpha = 0.9$: lo schema di intertempi desiderati è I_h^-, I_h^+, I_h^+, I_h^+, I_h^+, I_h^+, I_h^+, I_h^+, I_h^+, I_h^+;
- $\alpha = \frac{1}{3}$: lo schema di intertempi desiderati è I_h^-, I_h^-, I_h^+;
- $\alpha = \frac{2}{3}$: lo schema di intertempi desiderati è I_h^-, I_h^+, I_h^+.

Ovviamente per $F_s = 1$ tutti gli intertempi sono multipli interi del fattore di scala, e quindi il modello rimane quello descritto in precedenza, e altrettanto ovviamente se I_h/F_s fosse un numero intero allora la porzione di grafo relativa sarebbe costruita seguendo le tecniche precedentemente descritte. Per prima cosa si controlla se $\alpha = \frac{1}{3}$ oppure $\alpha = \frac{2}{3}$: se così non fosse allora α è troncato al primo decimale e ricondotto forzatamente ad uno dei casi sopracitati. L'esempio fatto inizialmente (con intertempo desiderato uguale a 270 secondi) si riconduce allo schema con ad $\alpha = 0.5$.

Nel caso in cui $F_s = 60$ $(1m)$ e $I_h = 260$ allora $I_h^- = 240$ $(4m)$ ed $\alpha = \frac{1}{3}$; ciò significa che gli intertempi desiderati sono ripetizioni della sequenza $240, 240, 300$ $(4m, 4m, 5m)$ con la quale si ottiene che $\frac{240+240+300}{260} = 3$.

Se invece $F_s = 60$ e $I_h = 280$ allora $I_h^- = 240$ $(4m)$, ma stavolta $\alpha = \frac{2}{3}$; ciò significa che gli intertempi desiderati sono ripetizioni della sequenza $240, 300, 300$ $(4m, 5m, 5m)$ tale che $\frac{240+300+300}{280} = 3$.

Permettere l'utilizzo di questi schemi comporta una modifica sostanziale del grafo su quale calcolare il cammino minimo. Il costo di attraversare un certo arco non è più fisso, ma dipende ora anche da quali archi sono stati attraversati precedentemente durante il cammino. Considerando l'esempio della sequenza "$4, 5$", il costo di un arco corrispondente ad un intertempo 5 è nullo se l'arco precedente nel cammino ha intertempo 4, mentre è > 0 se l'arco precedente nel cammino ha intertempo 5, e viceversa. La costruzione osservabile nella Figura 33 è estendibile nche agli altri schemi più complessi. Per fare questo occorre però un meccanismo generico.

Prendiamo come esempio la Figura 34, dove l'intertempo desiderato è 2.4.

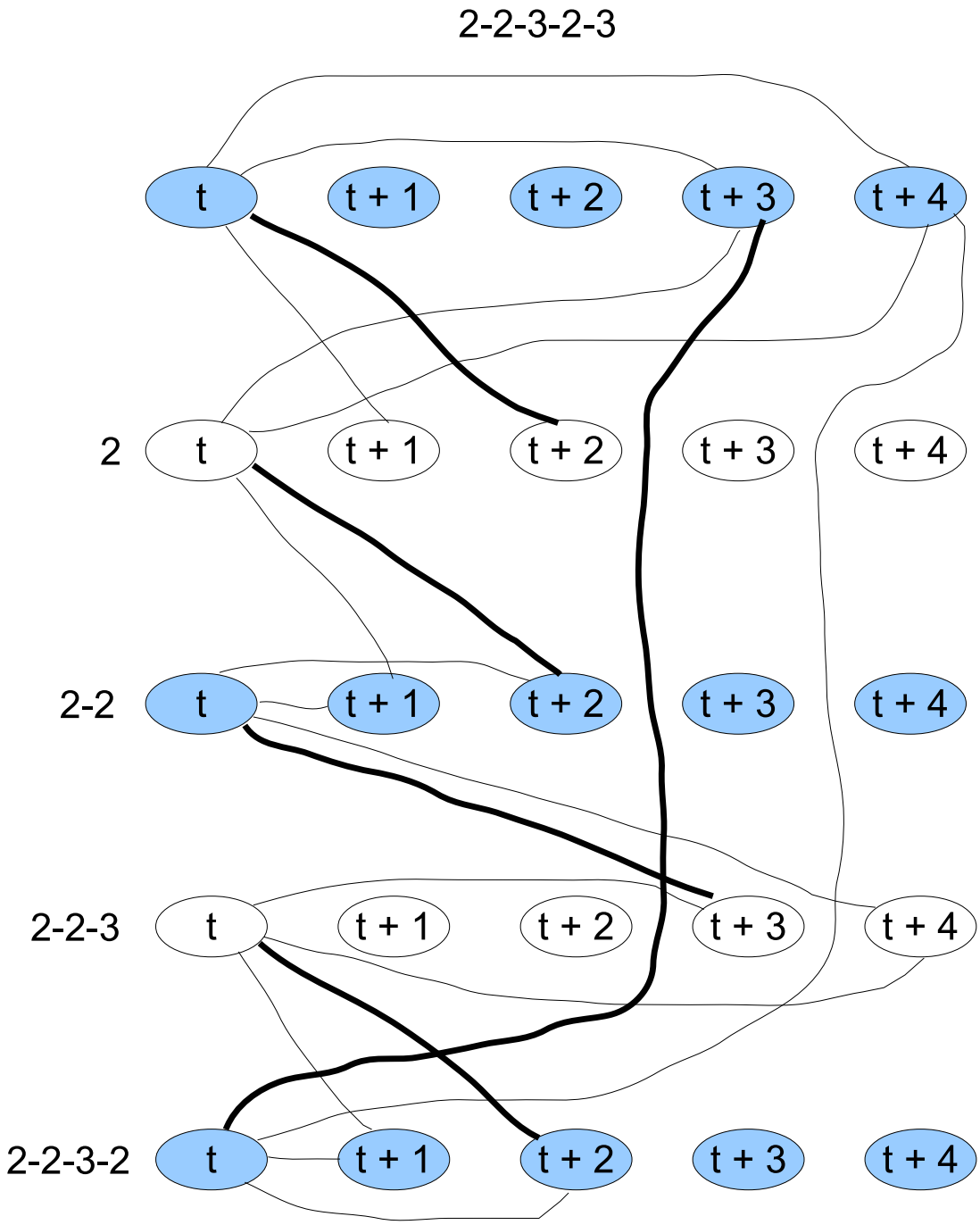

Figure 34: Esempio complesso di frequenza frazionaria

L'intertempo desiderato in questo caso è 2.4, mentre l'intertempo minimo e l'intertempo massimo sono rispettivamente 1 e 4. Ogni corsa (lo si può vedere anche nell'esempio precedente) è rappresentata da tanti nodi quanto è lunga la sequenza. Nella Figura 34 è disegnato solamente l'insieme degli archi che partono dai nodi rappresentanti la corsa dell'istante t. Gli archi in grassetto sono gli archi che rispettano perfettamente l'intertempo desiderato. Nel caso descritto dalla Figura 34 la sequenza è 2-2-3-2-3 vi sono quindi 5 possibili nodi per

ogni corsa, uno per ogni riga: la prima riga indica che la sequenza è reiniziata da capo o sta iniziando (quindi l'intertempo desiderato in questo caso sarebbe 2), mentre ad esempio la terza riga indica che la sequenza ha già coperto la sottosequenza "2-2" e quindi l'intertempo desiderato sarebbe 3. Tale meccanismo di costruzione del grafo è valido per ogni possibile sequenza composta da 2 possibili intertempi diversi, I^- e $I^+ = I^- + 1$.

Detto δ_a il lasso di tempo che intercorre tra il nodo t in questione e il nodo $t + \delta_a$ all'altro capo dall'arco a, e tenendo presente la struttura dell'esempio in Figura 34, il nodo t della riga generica r può avere archi a che si collegano a nodi $t + \delta_a$ appartenti alla riga...

- cosidetta *successiva*, ovvero generalmente alla riga $r + 1$ oppure, nel caso la riga r fosse l'ultima della sequenza, la riga n. 1;

- cosidetta *primaria*, ovvero la prima riga r' della sottosequenza di righe con intertempo desiderato identico alla quale appartiene la riga r; ad esempio in una sequenza "2-2-3-2-2-3-2-2-2-3" le tre righe che compongono la sottosequenza di tre 2 in fondo avranno tutte come *riga primaria* proprio la prima riga che fa parte di questa sottosequenza (la settima riga).

Concettualmente un arco che collega un nodo di una riga ad un nodo della sua *successiva* è un arco che ha rispettato la sequenza data, mentre tornare alla riga *primaria* significa che l'arco ha disatteso completamente le attese, quindi è necessario reiniziare la sottosequenza.

Fatta questa premessa, vi sono due distinti comportamenti a seconda che il nodo considerato abbia come intertempo considerato I^- oppure I^+: nella Figura 34 la prima, al seconda e la quarta riga appartengono al primo caso, la terza e la quinta al secondo caso.

- Se l'intertempo desiderato è I^- allora tutti gli archi a con relativo $\delta_a \leq I^-$ si collegano al nodo $t + \delta_a$ della riga *successiva* alla riga r, mentre tutti gli archi a con relativo $\delta_a > I^-$ si collegano al nodo $t + \delta_a$ che si trova nella riga *primaria* rispetto alla riga r.

- Se l'intertempo desiderato è I^+ allora tutti gli archi a con relativo $\delta_a \geq I^+$ si collegano al nodo $t + \delta_a$ della riga *successiva* alla riga r, mentre tutti gli archi a con relativo $\delta_a < I^+$ si collegano al nodo $t + \delta_a$ che si trova nella riga *primaria* rispetto alla riga r.

In un primo momento la formula per il calcolo del costo degli archi fu lasciata invariata, cioè

$$k_{new} = \frac{a\delta_a^2 + b\delta_a}{I_h} + \begin{cases} c & \text{se } \delta_a > 0 \\ 0 & \text{altrimenti} \end{cases} \tag{14}$$

dove si sostituisce solo il vecchio δ calcolato rispetto all'intertempo desiderato sulla fascia con δ_a, calcolato rispetto all'intertempo desiderato della riga r alla quale appatiene il nodo di partenza t.

Tale soluzione però portava con sé un problema, facilmente descrivibile mediante un esempio pratico. Nel caso si avesse un fattore di scala pari a 1 minuto e un intertempo desiderato (su una generica fascia oraria) pari a 4 minuti e 30 secondi, lo schema per le frequenze da utilizzare sarebbe del tipo "4-5-4-5". In questo scenario, nel momento in cui sarebbe preferibile un intertempo di 4 minuti (analoghe considerazioni possono essere fatte

anche con 5 minuti), risulta chiaro che ottenere un intertempo di 3 minuti o di 5 minuti (qualora fossero entrambi ammissibili) avrebbe la stessa penalizzazione in funzione obiettivo poiché la distanza dall'intertempo desiderato "momentaneo" sarebbe sempre di 1 minuto. È anche vero però che un intertempo di 5 minuti in questo caso sarebbe preferibile perché si manterrebbe in linea con i valori richiesti dallo schema "4-5-4-5" o, per dirla in un'altra maniera, sarebbe più vicino all'intertempo desiderato di 4 minuti e 30 secondi.

Si è deciso quindi di mediare il nuovo costo

$$k_{new} = \frac{a\delta_a^2 + b\delta_a}{I_h} + \begin{cases} c & \text{se } \delta_a > 0 \\ 0 & \text{altrimenti} \end{cases} \tag{15}$$

con il vecchio costo

$$k_{old} = \frac{a\delta^2 + b\delta}{I_h} + \begin{cases} c & \text{se } \delta > 0 \\ 0 & \text{altrimenti} \end{cases} \tag{16}$$

.

Ogni arco quindi costerà $k = \frac{k_{new}+k_{old}}{2}$.

Mettiamo quindi a confronto 3 tipi di esecuzione diversi sulla medesima istanza, esaminando il numero dei veicoli usati e il tempo d'esecuzione. In particolare abbiamo preso un'istanza che aveva definito i propri intertempi in modo tale che non fossero multipli del fattore di scala e abbiamo fatto "girare" l'algoritmo escludendo la prima volta la gestione delle frequenze frazionarie (ma con il controllo del flusso descritto nel Paragrafo 6) e inserendolo nelle due successive esecuzioni, una volta con e una volta senza il "costo mediato".

Tipo d'istanza	Num. Veicoli usati	Tempi
controllo flusso	28	22 min
controllo flusso + frazionaria senza costo mediato	29	38 min
controllo flusso + frazionaria con costo mediato	29	28 min

Figure 35: Tabella comparativa tra i risultati ottenuti senza e con la gestione delle frequenze frazionarie

In conclusione, gli obiettivi prefissi sono stati raggiunti. La nuova estensione del modello TT permette al codice di trovare soluzioni che, proprio dal punto di vista del timetabling, risultano essere più belle. Per quanto possa sembrare strano l'utilizzo del concetto di "bellezza" applicato a soluzioni che generano orari per linee urbane di bus, essa non solo esiste ma, come abbiamo cercato di fare e in questa sede di spiegare (vedi anche Paragrafo 3.5.3), risulta anche essere modellizzabile e codificabile. È chiaro che l'approccio che abbiamo deciso di concretizzare, sia per quanto riguarda i diversi schemi ideati sia per quanto riguarda il modello che li realizza all'interno del sottografo, non sia l'unico possibile: non è però errato affermare che gli schemi usati siano "condivisibili" (a meno di piccole variazioni) e soprattutto che il modello che le implementa nel grafo sia nello stesso tempo semplice da comprendere e soprattutto estendibile ad ogni possibile schema incentrato sulla presenza di I^- e I^+. Un ulteriore obiettivo consisteva, ovviamente, nel riuscire a mantenere ragionevolmente bassi i tempi d'esecuzione: anche tale obiettivo è stato raggiunto, in virtù del fatto che l'estensione alle linee frazionarie è andata a gravare esclusivamente sul modello TT e

quindi, in fase d'esecuzione, sulla parte di codice che implementa l'algoritmo che risolve i cammini minimi su grafi orientiati aciclici e che incideva (e continua ad incidere) molto poco sui tempi d'esecuzione globali del codice.

8 Le linee complesse

Nel Paragrafo 2 il problema è stato introdotto partendo da una ben precisa ipotesi sulla topologia, secondo la quale i percorsi inseriti in un unico modello devono necessariamente ricreare la situazione descritta nella Figura 1, che riproponiamo qui sotto.

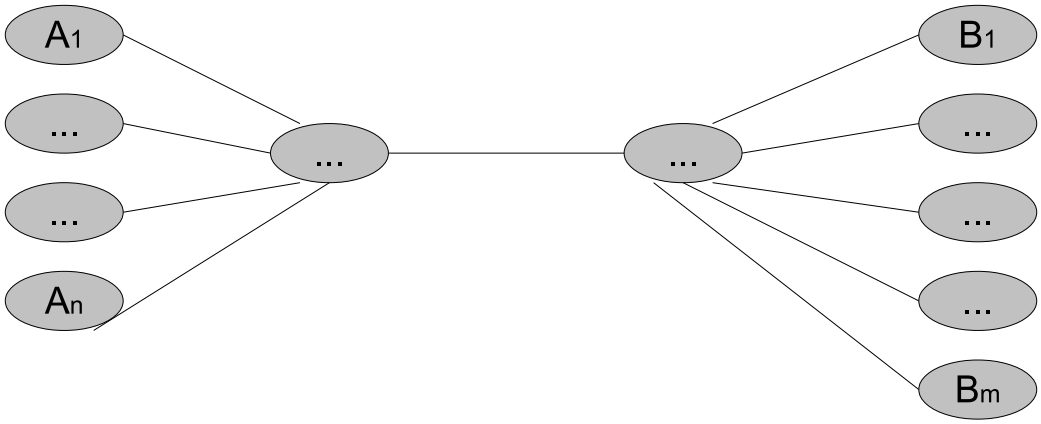

Figure 36: Caso generico (non tutti i nodi sono stati disegnati)

Nel corso della trattazione riguardante i sottoproblemi (e i sottomodelli relativi) la scelta della topologia d'esempio da usare è inevitabilmente caduta, per motivi di semplicità e compattezza, sul caso particolare di due percorsi, uno per ogni verso, che collegano due capolinea A e B. Le descrizioni dei singoli sottomodelli infatti possono prescindere da quelle particolari complicazioni e/o scelte modellistiche che ora saranno affrontate.

È altrettanto vero però che pure lo sviluppo del modello, e quindi dell'algoritmo e del codice, è partito dal caso semplice per arrivare (ora) a quello complesso. Non è solo una questione di trattazione: finora l'algoritmo è stato in grado di risolvere solo problemi che avevano due percorsi, uno in un verso e uno in un altro. Una volta appurato (come abbiamo fatto) che il codice fosse in grado di risolvere i problemi "semplici", trovando buone soluzioni in tempi ragionevoli, è sorta l'esigenza di trattare n percorsi invece di 2. Era chiaro che andasse modificato il modello in modo tale che fosse in grado di descrivere situazioni complicate, ma era altrettanto evidente che sarebbe stata preferibile una modifica al modello tale da non stravolgere la struttura finora utilizzata e tale da non invalidare gli studi finora fatti e i risultati raggiunti. Vedremo in questo paragrafo che questo particolare obiettivo è stato raggiunto.

È facile comprendere che avere n percorsi anziché 2 non comporta nessun tipo di problema dal punto di vista del sottomodello di schedulazione dei veicoli: avere un numero maggiore di percorsi significa semplicemente aumentare il numero degli "archi-corsa" e, conseguentemente, il numero degli archi in generale. Il modello rimane strutturalmente invariato, sarà solo presumibilmente più grande. I veri problemi sorgono sull'altro lato, nel sottomodello TT di determinazione delle corse a partire dagli intertempi desiderati. In presenza di più percorsi, sullo stesso verso di percorrenza, che hanno stazioni in comune il modo di pensare le frequenze è più complicato rispetto al semplice concetto di intertempo desiderato.

Una situazione tipica (anche se non prevista dal nostro modello) può essere questa:

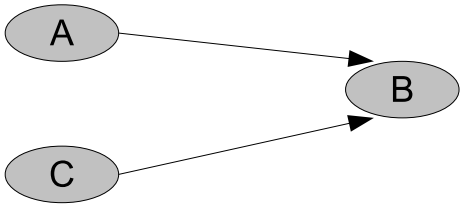

Figure 37: Caso alternativo

Il punto di vista dell'utente in questo caso può essere quello di pretendere simultaneamente:

- di fissare un certo intertempo di arrivo dei bus sulla stazione B, ad esempio un bus ogni 10 minuti;
- di stabilire secondo un qualche tipo di schema il percorso di provenienza del bus che arriva in B: ad esempio l'utente può volere che, ogni tre bus, arrivino due bus da A e solo uno da C.

Modellare ad esempio la strategia sopra descritta tramite l'attuale sottomodello per le frequenze sarebbe abbastanza facile. È sufficiente infatti designare due nodi pilota: il primo sarebbe B e avrebbe un intertempo desiderato uguale a 10 minuti, mentre il secondo sarebbe C e avrebbe un intertempo deisderato pari a 30 minuti. Il modello complessivo, così come è stato finora descritto e con l'aggiunta dei vincoli leganti trattati nel Paragrafo 3.3, permetterebbe la realizzazione di questo scenario. Sul verso Discendente (il verso Ascendente non è stato trattato, ma la sua presenza non porterebbe problemi di sorta) avremmo due sottoproblemi di "determinazione delle corse a frequenza" anziché uno solo e la soluzione trovata alla fine dell'esecuzione dell'algoritmo prevederebbe sostanzialmente tre bus ogni mezzora che arrivano in B (quindi uno ogni 10 minuti come richiesto) di cui uno ogni mezzora partirebbe da C. A questo punto la topologia particolare richiesta, con la presenza inderogabile del tratto centrale e con la particolare distribuzione dei capolinea/nodi indicata in Figura 36 nonché in Figura 1, sembrerebbe ridondante e addirittura limitativa.

Con questo tipo di soluzione più "libera" sorgono purtroppo enormi problemi quando si vogliono inserire sequenze più complesse. Il caso d'esempio sopra descritto era fin troppo semplice e quindi si poteva agevolmente tradurre il doppio obiettivo dell'utente in una lista d'intertempi desiderati su molteplici nodi pilota in modo tale da poter affrontare il problema con il modello semplice finora descritto. Di contro un caso più complicato può essere impossibile o non facilmente gestibile secondo questo tipo di logica. Un altro problema riscontrabile consiste nel fatto che il meccanismo suddetto non garantisce un eventuale ordine (ad esempio prima i bus da A e poi quelli da C) nella sequenza che che l'utente vuole dare, un problema che risulta maggiormente pesante in caso di sequenze lunghe.

Alla luce di queste considerazioni diventa chiaro che lasciare intatta la versione base del modello non è la strada migliore; risulta invece naturale estendere il modello nel tentativo di imitare il più possibile il modo che gli utenti hanno di risolvere questo tipo di situazione, seguendo quindi il ragionamento alla base dello schema sopra illustrato. L'estensione del modello mostrata in Figura 38 cerca di fare questo. Essa si basa sulla presenza di un tratto

centrale comune a tutti i pecorsi, così come illustrato nella Figura 36, che replichiamo per semplicità qui sotto. Sul tratto centrale due nodi vengono designati nodi pilota (uno per ogni verso) e su di essi sono definiti gli intertempi desiderati: come già emerso, esiste la possibilità che un solo nodo funga da pilota per entrambi i versi. Il caso semplice (due capolinea e due percorsi, uno per ogni verso) è un caso particolare proprio perchè il collegamento tra i due capolinea può essere considerato il tratto centrale.

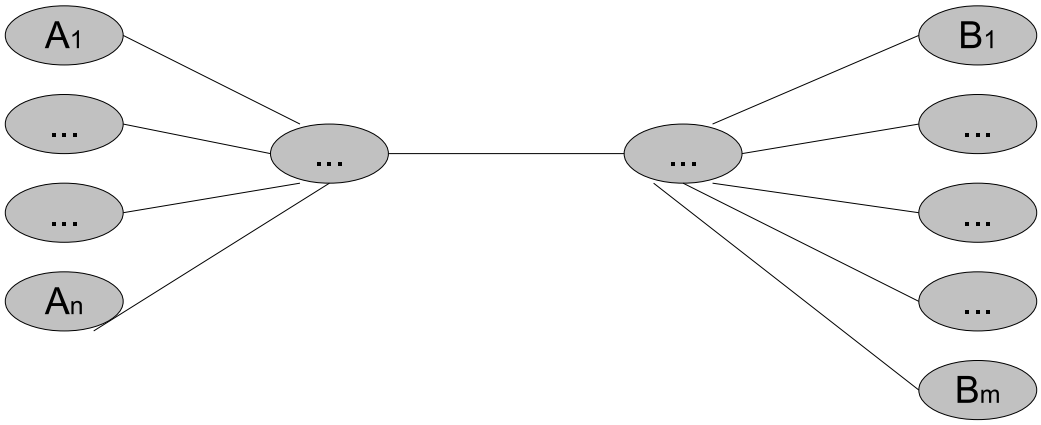

Figure 38: Caso generico (non tutti i nodi sono stati disegnati)

Stabilito che ogni bus passante per il nodo pilota ha un sottopercorso *di provenienza* (il bus parte da A_i, con $i \in 1..n$) e un sottopercorso *di destinazione* (il bus arriva in B_j, con $j \in 1..m$), l'estensione del modello prevede che siano definite due sequenze per ogni verso: la prima sequenza indica l'ordine *delle provenienze* mentre la seconda indica l'ordine *delle destinazioni*. In questo modo l'utente può decidere l'ordine di passaggio dei bus al nodo pilota in base alla tipologia "di provenienza" e "di destinazione". Ad esempio, in riferimento alla Figura 38, l'utente può decidere, oltre l'intertempo sul nodo pilota, che i primi 2 bus partano da A_1, il terzo da A_3, il quarto da A_1, il quinto da A_2, e così via; stesso discorso per le destinazoni.

Per inserire questo tipo di logica (le sequenze ordinate) nel modello occorre riproporre lo stesso stratagemma usato per le frequenze frazionarie (Paragrafo 7), ovvero la sostituzione del singolo nodo rappresentante la generica corsa c con un insieme di nodi: ognuno di essi continuerà ovviamente a riferirsi alla corsa c, ma avrà una sua semantica all'interno del contesto che stiamo analizzando. Nella Figura 39 è rappresentato un possibile scenario e, sotto di esso, come sarebbe rappresentato all'interno del modello. Lo scenario rappresentato, del quale consideriamo per semplicità solo il verso da sinistra a destra, prevede come nodo pilota il nodo N, sul quale saranno definiti gli intertempi. I capolinea "di provenienza" sono i nodi A e B, quelli "di destinazione" C e D. Le sequenze di ordinamento[7] dei bus sono state così definite: un bus proverrà da A e uno proverrà da B, mentre per quanto riguarda la destinazione due termineranno in C e poi uno terminerà in D. La sequenza "delle provenienze" è lunga due, quella "delle destinazioni" tre: ogni corsa (nell'esempio possiamo

[7]È importante precisare che, all'interno delle due sequenze, sarebbe più corretto parlare di *sottopercorsi* piuttosto che di capolinea, visto che è possibile che due bus possono ad esempio partire dal medesimo capolinea ed arrivare al nodo pilota seguendo strade diverse.

osservare tre diversi istanti, quindi tre diverse corse possibili) è quindi rappresentata nel modello da $2x3 = 6$ nodi, uno per ogni possibile combinazione degli elementi delle sequenze. Alcuni elementi possono essere ripetuti (il nodo C nell'esempio qui sopra è indicizzato) per replicare fedelmente le sequenze date.

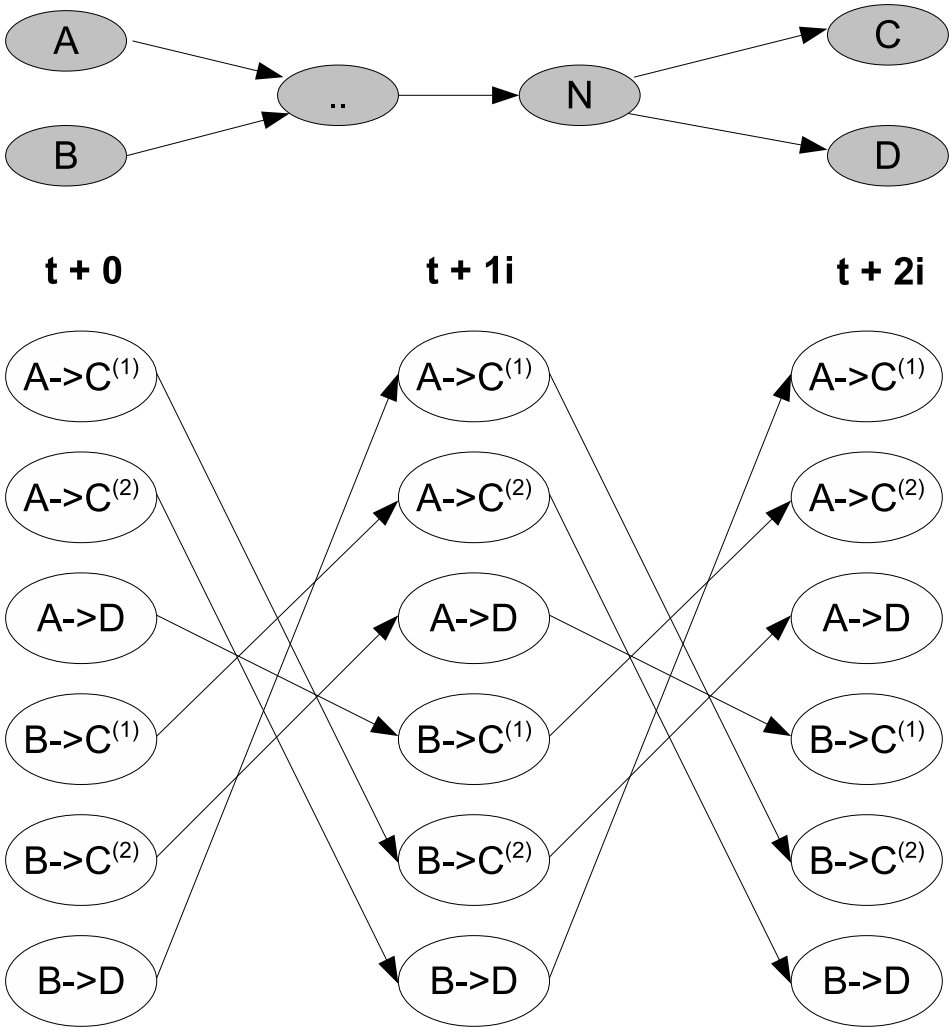

Figure 39: Esempio di modellazione di una linea complessa

Nel modello descritto in Figura 39 è facile notare come le due sequenze siano rispettate con precisione: se la corsa all' istante t proviene da A, la corsa successiva dovrà necessariamente provenire da B e quella dopo (l'ordinamento, è bene ricordarlo, è ciclico) ancora da A, e simultaneamente se una corsa termina in C la successiva terminerà nuovamente in C se eravamo all'inizio della sequenza oppure terminerà in D (per poi ricominciare la sequenza al successivo step) se eravamo alla seconda posizione. Quindi ogni arco che precedentemente avrebbe collegato due corse con tale estensione del modello è replicato 6 volte con la logica suddetta, mantenendo ovviamente le caratteristiche originali (costo, capacità,..). Questo meccanismo non interferisce minimamente con il meccanismo simile che delinea le frequenze frazionarie. Sotto l'ipotesi di avere sia le frequenze frazionarie sia molteplici percorsi, il modello rappresenterà ogni corsa c tramite un insieme di nodi del tipo $c_{i,j}$, con $i \in [1, n]$ e

$j \in [1, m]$ dove n è il numero di livelli che necessitano per realizzare lo schema frazionario e m è il numero di combinazioni per realizzare le linee complesse.

Nel caso in cui la corsa c^1 sia da collegare con la corsa c^2 e nel caso in cui il livello delle linee frazionarie i_1 abbia come suo successore il livello i_2, allora tutti i nodi del tipo $c^1_{i_1,j}$ saranno collegati a nodi del tipo $c^2_{i_2,j}$ con $j \in [1, m]$.

Nel caso in cui la corsa c^1 sia da collegare con la corsa c^2 e nel caso in cui la combinazione (nella logica delle linee complesse) j_1 abbia come suo successore la combinazione j_2, allora tutti i nodi del tipo c^1_{i,j_1} saranno collegati a nodi del tipo c^2_{i,j_2} con $i \in [1, n]$.

Un possibile problema in questo tipo di approccio è il numero di nodi (e archi), che cresce considerevolmente all'aumentare del numero delle combinazioni possibili per le linee complesse e del numero di livelli utili alle frequenze frazionarie. Se all'interno di un orizzonte temporale sono previste t corse, n livelli e m combinazioni, allora il numero dei nodi del grafo è uguale a $tmn + 2$. Parlando comunque di un problema di ricerca del cammino minimo su un grafo aciclico, il numero elevato di nodi e archi non comporta un visibile aumento della complessità computazionale dell'algoritmo.

A livello di codice, tali modifiche al modello sono state ottenute semplicemente estendendo il metodo di creazione del grafo TT.

I risultati ottenuti sperimentalmente, oltre ovviamente a seguire perfettamente le sequenze "di provenienza" e "di destinazione" richieste da input, continuano a risultare equilibrati nel raggiungimento dei vari sotto-obiettivi; i tempi, in caso di percorsi più complessi risultano ovviamente maggiormente dilatati, restando ancora comunque sotto la soglia della ragionevolezza, come possiamo vedere nella Figura 40.

Tipo d'istanza	Num. Veicoli usati	Tempi
controllo flusso + frazionaria con costo mediato	29	28 min
controllo flusso + frazionaria con costo mediato + linee complesse	34	67 min

Figure 40: Tabella comparativa tra un'esecuzione che (non) affronta le Linee Complesse

Infine un po' di "profiling" in modo da evidenziare le criticità del codice riguardo ai tempi d'esecuzione. La tabella in questione si riferisce alla medesima esecuzione nella seconda riga della tabella in Figura 40, quella con al suo interno le linee complesse.

Parte dell'algoritmo		Tempi (%)
Euristica	684 sec	16,9%
Risoluzione Rilassamento Lagrangiano	903 sec	22,2%
Creazione soluzioni ammissibili	238 sec	5,9%
Risoluzione Duale Lagrangiano	2189 sec	53,9%
Altro	41 sec	0,1%
Totale	4055 sec	100%

Figure 41: Esempio di modellazione di una linea complessa

Come emerge dal profiling fatto, il cosidetto "collo di bottiglia" sembra essere la risoluzione del Duale Lagrangiano. Questo molto probabilmente è dovuto al fatto che l'introduzione delle linee complesse grava, oltre che sul sottomodello TT (ma, come si è già detto, l'algoritmo di risoluzione dei cammini minimi non incide molto sui tempi d'esecuzione), sul numero di *vincoli accoppianti* che aumenta in proporzione proprio col numero di "nodi-corsa" all'interno dei sottomodelli TT. In particolare una linea a "doppia Y", come quella usata per il profiling, quadruplica il numero di nodi (già peraltro aumentato dalla presenza delle frequenze frazionarie in alcuni punti della giornata) necessari a rappresentare le corse e quindi quadruplica il numero di vincoli che il *Bundle* è costretto a gestire all'interno del suo sottoproblema di competenza. Fermo restando che, come già detto, gli obiettivi prefissi sono stati raggiunti, è possibile pensare di ottimizzare ulteriormente i tempi d'esecuzione, comunque risultati ragionevoli e in linea di massima soddisfacenti, attraverso un uso più mirato dell'euristica, magari studiando ulteriormente i parametri del file di configurazione nel contesto "linee complesse" che regolano i pesi nella funzione obiettivo e/o il funzionamento dell'euristica stessa.

Vi è inoltre da considerare, come ulteriore possibilità di miglioramento delle prestazioni, ulteriori preprocessing e/o postprocessing all'interno dell'euristica incentrati su ulteriori fissaggi a 0 di insiemi di corse. Con l'introduzione delle linee complesse sono state introdotte infatti nuove corse e quindi nuovi vincoli sul quale il *Bundle* deve lavorare: è altresì vero che tra le corse possono sorgere legami che prima non esistevano, legmi che possono rendere plausibile l'esclusione automatica di una corsa (o più) a causa della preventiva esclusione (o inserimento) di un'altra all'interno della soluzione che l'algoritmo sta formando. Quindi si può ipotizzare che ad un fissaggio a 0 di una corsa, deciso dall'euristica nella sua attuale implementazione, ne possono conseguire altri: è chiaro che più fissaggi si riesce a fare più il problema per i solvers viene rimpicciolito e quindi si possono avere vantaggi in termini di tempi d'esecuzione.

9 Conclusioni

Alla luce della trattazione finora esposta è evidente come gli obiettivi prefissati siano stati raggiunti. In primo luogo il modello ideato, sia nella sua prima versione "base" sia nelle sue diverse estensioni, è riuscito nel doppio intento di:

- descrivere al meglio scenari di diversa complessità del contesto TPL;
- essere la base sulla quale potessero essere usati algoritmi in grado di cooperare con l'obiettivo di trovare soluzioni in linea con le aspettative.

Questo è stato senza dubbio l'obiettivo più importante da raggiungere, così come lo è per qualsiasi tipo di codice; è altrettanto evidente come tale obiettivo sia stato l'incognita più grande, per via del semplice fatto che l'approccio utilizzato nella modellizzazione del problema fosse un approccio nuovo, una strada non ancora battuta. Al raggiungimento dell'obiettivo primario sono stati quindi dedicati gli sforzi maggiori.

Dapprima il modello stesso è stato ottimizzato addirittura prima di essere effettivamente realizzato, quindi è stato effettivamente realizzato; con esso sono stati implementati gli algoritmi dediti alla risoluzione del problema. Sono stati eseguiti numerosi test, prima su una versione "1.0" del codice, funzionante su tipologie di istanze non complesse (due percorsi, due nodi pilota), quindi riproponendo gli stessi problemi al solver *Cplex*: laddove il nuovo codice non riusciva a superare un evidente problema di convergenza, *Cplex* è riuscito a produrre soluzioni di buona qualità e quindi a validare il modello stesso certificando la sua validità. Conseguentemente a ciò è stato necessario rendere il nuovo codice abile nel trovare soluzioni di buona qualità in tempi ragionevoli, superando i problemi di convergenza. Sono state quindi studiate quelle soluzioni ritenute di buona qualità dagli esperti M.A.I.O.R. nella ricerca di una qualche correlazione esistente con i risultati intermedi trovati dal codice: sulla base di questo studio è stato ideato e quindi implementato un approccio euristico in grado di ridurre incrementalmente il dominio del problema prendendo via via decisioni definitive sulle corse da inserire o da escludere dalla soluzione e sgravando quindi i solvers esistenti da codeste scelte. Grazie anche ad uno studio sui parametri di costo della funzione obiettivo, l'approccio euristico è riuscito finalmente a trasformare un codice sostanzialmente non in grado di ottemperare allo scopo per il quale era stato progettato in un software in grado di farlo efficientemente: finalmente infatti è stato eliminato il problema di convergenza, quindi venivano restituite soluzioni che sono state certificate come di buona qualità, ovvero equilibrate nel raggiungimento simultneo dei sotto-obiettivi di un buon timetabling e di un buono scheduling dei veicoli, in tempi di esecuzione ragionevoli secondo le aspettative. Raggiunto sostanzialmente l'obiettivo primario, è stato possibile quindi concentrarsi sui dettagli.

Il codice è stato quindi dotato di un meccanismo che permetteva di controllare, in buona misura, il numero di veicoli in circolo in ogni momento della giornata attraverso il controllo del flusso di veicoli in uscita e in entrata dal deposito. Questa estensione del modello ha permesso di raggiungere, seppur non in maniera totale, un obiettivo (il controllo localizzato del numero di veicoli in servizio) che fu indicato, in sede di analisi preliminare del problema, come impossibile da raggiungere per via della struttura stessa del modello proposto, fornendo ai futuri utenti uno strumento semplice da usare e allo stesso tempo efficace nella gestione

dello scheduling dei veicoli (argomento per il quale chi si accinge ad organizzare orari di bus è sicuramente molto sensibile !).

Il modello è stato quindi esteso ulteriormente per gestire in maniera migliore quegli scenari che prevedono intertempi "frazionari": il codice ritorna, in questi casi, soluzioni con un timetabling aderente (nalla misura in cui è possibile) a schemi d'intertempo precaricati e ritenuti a priori "eleganti" nonché maggiormente funzionali nel descrivere le situazioni suddette, mantenendo inoltre relativamente bassi i tempi d'esecuzione. Infine il modello è stato arricchito con la possibilità di modellare scenari più complessi rispetto alla coppia di percorsi fino a quel momento usata, fornendo in maniera ragionevolmente efficiente soluzioni perfettamente aderenti con gli schemi di alternanza sui percorsi che l'utente precarica tramite input.

Naturalmente il lavoro è passibile di ulteriori aggiornamenti e migliorie. Se gli obiettivi più importanti sono stati sostanzialmente raggiunti, alcuni di essi (quelli di minore importanza) non sono stati affrontati nel lavoro fin qui svolto.

Ad esempio:

- il modello potrebbe essere arricchito con l'inserimento nel novero del calcolo dei turni degli autisti;
- il modello potrebbe prevedere la possibilità di più depositi;
- si possono studiare, all'interno dell'euristica, preprocessing o postprocessing sulle decisioni prese più efficaci di quelle già presenti, soprattutto in caso di linee complesse;
- si potrebbe studiare ulteriormente, attraverso altri test, i parametri di costo della funzione obiettivo e/o i parametri di funzionamento dell'euristica alla ricerca di un set di valori più performante.

In conclusione il lavoro svolto ha mostrato come il modello integrato, equidistante tra la ricerca di un buon timetabling e l'ottimizzazione dell'insieme dei veicoli usati, possa essere un modo efficace di modellare realtà TPL di diversa complessità, fornendo la base per un algoritmo composito, implementato tramite un codice modularizzato, che risolve efficientemente i problemi presentati fornendo soluzioni di buona qualità e allo stesso tempo permette ampio margine di manovra e di scelta agli utenti in fase di generazione dell'input.

10 Bibliografia

[KN05] Kliever, N., Mellouli, T., Shul, L., 2005, *A timespace network based exact optimization model for multi-depot bus scheduling*, European Journal of Operational Research 175, 1616-1627

[MS09] Michaelis, M., Schöbel, A., 2009, *Integrating line planning, timetabling, and vehicle scheduling: a customer-oriented heuristic*, Public Transp (2009) 1: 211-232

[VVV08] Van den Heuvel, A.P.R., Van den Akker, J.M., Van Kooten Niekerk, M.E., 2008, *Integrating timetabling and vehicle scheduling in public bus transportation*, Technical Report UU–CS–2008–003

[HG08] Hao, J., Guihaire, V., 2008, *Transit Network Re-timetabling and Vehicle Scheduling*,

[FL09] Fleurent, C., Lessard, R., 2009, *Integrated timetabling and vehicle scheduling*, GIRO

[BF11] Bertolini, A., Frangioni, A., 2011, *MCFSimplex*

[F10] Frangioni, A., 2010, *Bundle*, SIAM Journal on Optimization 13(1), p. 117 - 156, 2002

[F05] Frangioni, A., 2005, *About Lagrangian Methods in Integer Optimization*, Annals of Operations Research 139, 163-193

[FK00] Feltenmark, S., Kiwiel, K., 2000, *Dual Applications of Proximal Bundle Methods, including Lagrangian Relaxation of Nonconvex Problems*, SIAM J. on Optimization 10(3), 697-721

11 Appendice A: le interfacce del codice

L'Appendice A fornisce una descrizione dettagliata delle tre interfacce (*TTDAlgoInterface*, *TTInterface* e *BSInterface*) presenti nell'architettura del codice la cui particolarità consiste nell'essere state create ex novo. La documentazione fornita la si può trovare all'interno del codice in un formato compatibile per essere estratta (come in questo caso) tramite *Doxygen*, ed è stata mantenuta costantemente aggiornata durante tutto il periodo di sviluppo.

11.1 TTDAlgoInterface

11.1.1 Detailed Description

`TTDAlgoInterface` definisce una classe astratta per il problema "Time Trial". Il problema "Time Trial" ha come input due o più percorsi per bus, un insieme fissato di nodi che definiscono i percorsi, un insieme di depositi dai quali i bus partono verso e/o arrivano dai capolinea, ed infine un sottoinsieme di nodi che fungono da "punti di osservazione".

Per ogni deposito esiste una sosta minima e una sosta massima consentita che ogni bus entrante nel deposito deve necessariamente rispettare. Anche per ogni capolinea esiste un concetto analogo di sosta min/max. Sia nel caso dei depositi che nel caso dei capolinea non esiste un'unica coppia di valori sosta min/max valida per tutta la giornata, che invece è divisa in fasce orarie: per ogni fascia oraria esiste una diversa sosta min/max.

Per ogni "punto di osservazione" (necessariamente un nodo facente parte di almeno un percorso) esiste una direzione che determina l' insieme dei percorsi per i quali il nodo è punto di osservazione (ovviamente tutti i percorsi che hanno questa particolare direzione), ed esiste per ogni fascia oraria esistente una terna di valori: l' intertempo desiderato nella fascia oraria relativa tra due bus che passano in sequenza (anche appartenenti a due corse che coprono due percorsi diversi) dal nodo, l' intertempo minimo e l' intertempo massimo consentito. Inoltre ogni punto di osservazione avrà un range iniziale (rispetto all'inizio della giornata) e uno finale (rispetto alla fine della giornata) entro i quali poter far partire rispettivamente la prima e l'ultima corsa.

I capolinea sono collegati ai depositi, in una direzione e nell'altra. Per ogni coppia capolinea-deposito, per entrambe le direzioni possibili e per ognuna delle fasce orarie esistenti è indicato un tempo di percorrenza.

Ogni percorso è definito dai due capolinea, da una direzione e dagli eventuali nodi di transito. Ogni percorso ha un insieme di possibili corse da usare nella creazione della soluzione finale.

Esistono due direzioni, '`As`' e '`Di`', che partizionano l'insieme dei percorsi (e quindi anche delle corse). Su ogni direzione agisce un unico nodo d'osservazione: in generale ci saranno quindi 2 nodi di osservazione, ma sarà possibile delegare ad un unico nodo la funzione per entrambe le direzioni. Su ogni direzione sarà possibile definire, durante la giornata, uno o più 'schemi-bus' che definiranno come i vari percorsi aventi la medesima direzione concorrano a formare la tabella delle corse.

Una soluzione ottima (o quantomeno buona) è un piano d'azione che permetta di rispettare il più possibile gli intertempi richiesti ed allo stesso tempo di utilizzare il minor numero di bus possibili; le due cose sono ovviamente in contrapposizione, nel senso che usando un numero

opportunamente alto di bus si può certamente garantire il rispetto esatto degli intertempi. È quindi necessario dare pesi opportuni alle due componenti della funzione obiettivo in modo da determinare un giusto bilanciamento tra di esse.

Una soluzione è espressa indicando più insiemi di corse (un bus per ogni insieme) scelte tra le corse possibili indicate dall' input.

11.1.2 Classes

- `struct BusChartType`: struttura che descrive un singolo schema corse.
- `class Inf`: piccola classe per ricavarsi il valore "+ infinito" per un tipo primitivo: basta scrivere `Inf<`*`tipo`*`>()`.
- `struct TripType`: struttura che descrive una singola corsa.
- `class TTDException`: piccola classe per le eccezioni.

11.1.3 Enumeration Documentation

- `enum TTDStatus`

  Tipo enumerato pubblico che descrive i possibili stati del solver.

  Enumerator:

  - `kUnSolved`: nessuna soluzione disponibile
  - `kError`: nessuna soluzione trovata a causa di errore non gestibile
  - `kStopped`: trovata soluzione ammissibile ma non con la precisione richiesta
  - `kOK`: trovata soluzione con la precisione richiesta
  - `kInProgress`: algoritmo in esecuzione

- `enum TTDParam`

  Tipo enumerato pubblico che descrive i parametri da gestire coi metodi pubblici `SetPar()` e `GetPar()`.

  Enumerator:

  - `kMaxTime`: tempo massimo d'esecuzione concesso
  - `kMaxIteration`: massimo num. di iterazioni concesse
  - `kSolPrecision`: soglia di accuratezza della soluzione da raggiungere
  - `kLastParam`: parametro "dummy": qualsiasi classe derivata può aggiungere nuovi parametri avendo cura di dare al suo primo nuovo parametro il valore `kLastParam` +1.

11.1.4 Public types

`TTDAlgoInterface` definisce tre tipi semplici pubblici:

- `Time`, rappresenta sia istanti di tempo (hh:mm:ss, 0 codifica la mezzanotte) sia intervalli di tempo (in secondi), tramite un intero positivo;
- `IndexNodes`, rappresenta gli indici dei capolinea o gli indici dei depositi;
- `TimeBand`, rappresenta un vettore di `Time` utile a descrivere la divisione dell' arco temporale in fasce orarie.

È definito inoltre un tipo strutturato pubblico:

- `TripType`, permette di rappresentare le singole corse.

I tipi pubblici sono pesantemente usati nei parametri dei metodi pubblici di questa interfaccia. Questo significa che chi vuole usare l'interfaccia ha la possibilità di, ma è anche "costretto" ad, usare questi tipi.

11.1.5 Member Function Documentation

- `virtual TTDAlgoInterface ()`

  Costruttore

- `virtual  TTDAlgoInterface ()`

  Distruttore

- `virtual void SetGammaM ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso dei turni macchina all'interno della funzione obiettivo.

  Il numero di turni macchina è il numero totale di bus utilizzati da una soluzione; analogamente, il numero di volte che un bus torna al deposito di partenza per non uscirne più (nell'intervallo temporale del problema). Nella funzione obiettivo, ogni turno macchina utilizzato ha un costo `GammaM`, il cui valore è settato da questo metodo.

- `virtual void SetGammaI ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso della "compatibilità in linea" all'interno della funzione obiettivo. Ogni bus ha la possibilità, una volta terminata la corsa e quindi una volta arrivato al capolinea, di sostare al capolinea per un certo periodo per poi ripartirvi per effettuare un'altra corsa. Nella funzione obiettivo, questa azione (detta compatibilità in linea tra le due corse) ha un costo diviso in due parti: una parte proporzionale al tempo di attesa al capolinea in più rispetto al tempo di sosta minimo previsto moltiplicato per un opportuno fattore `GammaI`, il cui valore è settato da questo metodo, l'altra parte consiste in un costo fisso (che dipende non dalla sosta ma dalla corsa di inizio sosta) da pagare se la sosta è maggiore dell'intertempo

desiderato nella fascia oraria che contiene l'istante di arrivo al capolinea dove è prevista la sosta. Si noti che è sempre possibile evitare di fare compatibilità in linea al costo di usare un numero di bus molto alto: per ogni corsa si usa un bus diverso. Quindi, dando valori molto alti a `GammaI`, per cui il costo della compatibilità in linea sia superiore al costo di un turno macchina (si veda `SetGammaM()`), esiste una soluzione ottima che non ne usa nessuna. Pertanto, in questo caso potrebbe essere possibile semplificare il problema assumendo che nessuna compatibilità in linea venga mai eseguita. Non si ritiene che ciò abbia interesse, per cui questa condizione potrà essere ignorata.

- `virtual void SetGammaF ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso delle "compatibilità fuori linea" all'interno della funzione obiettivo. Ogni bus ha la possibilità, una volta terminata la corsa e quindi una volta arrivato al capolinea, di andare in un deposito, sostarci e poi ripartire da lì per effettuare una nuova corsa. Nella funzione obiettivo, il costo di tornare al deposito (detta compatibilità fuori linea tra due corse) è proporzionale al tempo di viaggio fino al deposito (ma non al tempo di attesa al deposito, in cui il mezzo "non è viaggiante") moltiplicato per un opportuno fattore `GammaF`, il cui valore è settato da questo metodo. Si noti che ponendo `GammaI` $\neq$ `GammaF` (si veda `SetGammaI()`) si può esprimere una preferenza tra l'attesa in linea ed il rientro al deposito. Si noti che è sempre possibile evitare di fare compatibilità fuori linea al costo di usare un numero di bus molto alto: per ogni corsa si usa un bus diverso. Quindi, dando valori molto alti a `GammaF`, per cui il costo della compatibilità fuori linea sia superiore al costo di un turno macchina (si veda `SetGammaM()`), esiste una soluzione ottima che non ne usa nessuna. Pertanto, in questo caso potrebbe essere possibile semplificare il problema assumendo che nessuna compatibilità fuori linea venga mai eseguita. In gergo tecnico, porre `GammaF` ad un valore molto alto, e quindi eliminare la compatibilità fuori linea, significa che nella funzione obiettivo vengono contati i "turni macchina" e non i "viaggi". I viaggi sono il numero di volte che un bus esce dal deposito, mentre i turni macchina sono il numero totale di bus utilizzati (si veda `SetGammaM()`): i viaggi possono essere di più dei turni macchina perché lo stesso bus può uscire più di una volta dal deposito (proprio nel caso di compatibilità fuori linea). Usualmente minimizzare il numero di turni macchina è importante, ma può darsi ed esempio il caso che il numero di turni macchina minimo sia già noto. In questo caso, ponendo `GammaF` ad un valore molto alto si unifica il concetto di "viaggio" e "turno macchina" (che coincidono se non possono essere effettuate compatibilità fuori linea), e quindi minimizzando i secondi si minimizzano i primi. Si ritiene che possa essere interessante esplorare questo tipo di situazione. Pertanto, dare un costo `Inf<double>` a `GammaF` in questo metodo corrisponde ad indicare che non possono essere effettuate compatibilità fuori linea.

- `virtual void SetGammaS ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso delle "compatibilità fuori linea iniziali/finali" all'interno della funzione obiettivo. Per ogni singolo turno macchina il costo di uscire la prima volta dal deposito (rientrare l'ultima volta al deposito) è proporzionale al tempo che occore al bus per percorrere il tragitto deposito–capolinea moltiplicato per un opportuno fattore `gammaS`, il cui valore è settato da questo

metodo.

- `virtual void SetExecutionName ( string name ) [pure virtual]`

  Il metodo permette di settare il nome che identifica l'esecuzione in corso. Ad esempio può essere usato, per questo scopo, il nome del file d'input dal quale sono stati caricati i dati.

- `virtual void SetScale ( Time value ) [pure virtual]`

  Il metodo permette di settare il fattore di scala (in secondi). Il fattore di scala può modificare lo schema degli intertempi desiderati tra le corse nel caso in cui l'intertempo in questione non sia un multiplo del fattore di scala. Ad esempio se il fattore di scala è 1 minuto mentre l'intertempo desiderato su una data fascia oraria è 3 minuti, non ci sono modifiche allo schema che prevede come soluzioni ideali corse che partono a distanza di 3 minuti l'una dall'altra. Ipotizzando invece che l'intertempo I non sia multiplo del fattore di scala F_s, avremo che $I = Q \cdot F_s + r$. Si calcolano $n = Q \cdot F_s$, $a = r/F_s < 1$. Si sceglie quindi un nuovo numero A all'interno di 1/2, 1/3, 2/3, prendendo l'elemento dell' insieme più vicino ad a. A seconda del valore di A viene definito un diverso schema d'intertempi.

  Se A = 1/2 allora si definisce uno schema (ciclico) di intertempi desiderati del tipo n, n + Fs. In pratica tra la prima corsa e la seconda (e fra la terza e la quarta ...) l'intertempo desiderato è $n = Q \cdot F_s$, mentre tra la seconda e la terza corsa (e fra la quarta e la quinta ...) l'intetempo desiderato è $n + F_s$.

  Se A = 1/3 allora lo schema è n, n, n + Fs, mentre se A = 2/3 allora lo schema è n, $n + F_s$, $n + F_s$.

- `virtual void SetFunctionCost ( int what) [pure virtual]`

  Il metodo setta quale usare tra le varie funzioni di costo implementate all'interno del solver TT.

- `virtual void SetCostParamForLapTime ( double alfa, double a, double b, double c, IndexNodes obPoint = Inf< IndexNodes >(), bool dirDi = true, bool dirAs = true, Time startTime = 0, Time endTime = Inf< Time >() ) [pure virtual]`

  Il metodo permette di settare i parametri (non negativi) che determinano il costo di effettuare corse con un intertempo non pari a quello desiderato. Il metodo permette di settare i primi quattro parametri sia universalmente per tutti i punti di osservazione e per tutte (o alcune) le fasce orarie, sia dettagliatamente settando una quadrupla di parametri per ogni punto di osservazione e/o per ogni fascia oraria. Dipendendo strettamente dall'insieme dei punti di osservazione e dalla divisione in fasce orarie della giornata, è obbligatorio chiamare il metodo solo dopo aver finito di inserire i due insiemi di cui sopra coi metodi `AddObservationPoint()` e `SetTimeBand()`.

Parameters:

- `alfa` determina una soglia oltre la quale l'errore relativo fra l'intertempo effettivo della corsa t e l'intertempo desiderato d (per la specifica fascia oraria) diventa intollerabile e quindi le due corse non possono essere usate in sequenza. In particolare se $\delta = \frac{|t-f|}{f} <= \alpha$ allora l'intertempo della corsa è ammissibile.
- `a` è il coefficiente quadratico nella formula di costo per gli intertempi tra le corse
- `b` è il coefficiente lineare nella formula di costo per gli intertempi tra le corse
- `c` è il coefficiente costante nella formula di costo per gli intertempi tra le corse
- `obPoint` è l' identificativo del punto di osservazione al quale vogliamo associare i quattro parametri alfa, a, b, c; se obPoint = `Inf<IndexNodes>()` (default), i quattro parametri sono associati a tutti i punti d'osservazione esistenti
- `dirDi, dirAs` indicano rispettivamente se il metodo effettua o meno modifiche ai costi relativi al/ai punto/i di osservazione indicato/i da `obPoint` per quanto riguarda le due direzioni esistenti '`Di`' e '`As`'. Se `obPoint` indica un singolo nodo e se uno dei due parametri è true laddove però tale nodo non è punto di osservazione per quella specifica direzione, il metodo lancerà un'eccezione con `TTDException`.
- `startTime, endTime` sono rispettivamente l'orario di inizio e l'orario di fine del periodo di tempo al quale associare i quattro parametri `alfa`, `a`, `b`, `c`. Il parametro `startTime` è settato di default a 0, `endTime` a `<Time>()`: i due valori significano rispettivamente l'inizio e la fine della giornata, qualunque siano gli orari d'inizio e fine giornata settati in precedenza da `SetTimeBand()`. I parametri `startTime` e `endTime` possono essere settati diversamente rispetto agli orari di inizio/fine giornata. Gli orari dati non possono essere "casuali": ovviamente `startTime` deve precedere `endTime`, inoltre questi devono essere orari che segnano l'inizio di una fascia oraria esistente e quindi presente nell'insieme di fasce orarie già passato con il metodo `SetTimeBand()`: in caso contrario viene lanciata un'eccezione con `TTDException`.

I tre parametri `a`, `b`, e `c` determinano il costo (in funzione obiettivo), di scegliere una coppia di corse in sequenza. Sono state ipotizzate tre possibili varianti di calcolo del suddetto costo.

Le prime due calcolano per prima cosa un intertempo relativo $\delta = \frac{|t-f|}{f}$

Quindi la prima formula per il calcolo del costo dice che $costo = \begin{cases} a\delta^2 + b\delta + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

mentre la seconda dice che $costo = \begin{cases} a(\delta+1)^2 + b(\delta+1) + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

La terza ipotesi utilizza invece un intertempo assoluto $\delta = |t - f|$

per poi calcolare così il costo: $costo = \begin{cases} \frac{a\delta^2+b\delta}{f} + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

- `virtual void SetFirstTripCost ( double aI, double bI, double cI, Time rifTime,Time startTime = 0, Time endTime = Inf< Time >(), IndexNodes obPoint = Inf< IndexNodes >(), bool dirDi = true, bool dirAs = true ) [pure virtual]`

  Il metodo permette di settare i parametri (non negativi) che determinano il costo di effettuare la prima corsa ad una determinata distanza temporale dall'inizio della giornata.

  Parameters:

  - `aI` è il coefficiente quadratico nella formula di costo della distanza tra `rifTime` e la prima corsa
  - `bI` è il coefficiente lineare nella formula di costo della distanza tra `rifTime` e la prima corsa
  - `cI` è il coefficiente costante nella formula di costo della distanza tra `rifTime` e la prima corsa
  - `rifTime` è l'istante di tempo dal quale determinare la distanza di ogni corsa
  - `startTime, endTime` sono rispettivamente l'orario di inizio e l'orario di fine del periodo di tempo al quale associare i parametri `aI`, `bI`, `cI`, `rifTime`. Il parametro `startTime` è settato di default a 0, `endTime` a `Inf<Time>()`: i due valori significano rispettivamente l'inizio e la fine della giornata, qualunque siano gli orari d'inizio e fine giornata settati in precedenza da `SetTimeBand()`.
  - `obPoint` è l' identificativo del punto di osservazione al quale vogliamo associare i parametri `aI`, `bI`, `cI`, `rifTime`; se `obPoint` = `Inf<IndexNodes>()` (default), i quattro parametri sono associati a tutti i punti d'osservazione esistenti
  - `dirDi, dirAs` indicano rispettivamente se il metodo effettua o meno modifiche ai costi relativi al/ai punto/i di osservazione indicato/i da `obPoint` per quanto riguarda le due direzioni esistenti '`Di`' e '`As`'. Se `obPoint` indica un singolo nodo e se uno dei due parametri è true laddove però tale nodo non è punto di osservazione per quella specifica direzione, il metodo lancerà un' eccezione con `TTDException`.

  I tre parametri `aI`, `bI`, e `cI` determinano il costo (in funzione obiettivo), di scegliere una corsa come prima della giornata avente distanza /delta dall'istante passato col parametro `rifTime`.

  La formula è $costo = \begin{cases} a\delta^2 + b\delta + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

- `virtual void SetLastTripCost ( double aF, double bF, double cF, Time rifTime,Time startTime = 0, Time endTime = Inf< Time >(), IndexNodes obPoint = Inf< IndexNodes >(), bool dirDi = true, bool dirAs = true ) [pure virtual]`

  Il metodo permette di settare i parametri (non negativi) che determinano il costo di effettuare l'ultima corsa ad una determinata distanza temporale dalla fine della giornata.

Parameters:

- `aF` è il coefficiente quadratico nella formula di costo della distanza tra `rifTime` e l'ultima corsa
- `bF` è il coefficiente lineare nella formula di costo della distanza tra `rifTime` e l'ultima corsa
- `cF` è il coefficiente costante nella formula di costo della distanza tra `rifTime` e l'ultima corsa
- `rifTime` è l'istante di tempo dal quale determinare la distanza di ogni corsa
- `startTime, endTime` sono rispettivamente l'orario di inizio e l'orario di fine del periodo di tempo al quale associare i parametri `aF`, `bF`, `cF`, `rifTime`. Il parametro `startTime` è settato di default a 0, `endTime` a `Inf<Time>()`: i due valori significano rispettivamente l'inizio e la fine della giornata, qualunque siano gli orari d'inizio e fine giornata settati in precedenza da `SetTimeBand()`.
- `obPoint` è l' identificativo del punto di osservazione al quale vogliamo associare i parametri `aF`, `bF`, `cF`, `rifTime`; se `obPoint` = `Inf<IndexNodes>()` (default), i quattro parametri sono associati a tutti i punti d'osservazione esistenti
- `dirDi, dirAs` indicano rispettivamente se il metodo effettua o meno modifiche ai costi relativi al/ai punto/i di osservazione indicato/i da `obPoint` per quanto riguarda le due direzioni esistenti '`Di`' e '`As`'. Se `obPoint` indica un singolo nodo e se uno dei due parametri è true laddove però tale nodo non è punto di osservazione per quella specifica direzione, il metodo lancerà un'eccezione con `TTDException`.

I tre parametri `aF`, `bF`, e `cF` determinano il costo (in funzione obiettivo), di scegliere una corsa come prima della giornata avente distanza /delta dall'istante passato col parametro `rifTime`.

La formula è

$$costo = \begin{cases} aF\delta^2 + bF\delta + cF & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$$

- `virtual void GetContinueSol ( vector< vector< double > > & valTTTrips, vector< double > & valBSTrips, vector< double > & Lambda ) [pure virtual]`

  Il metodo restituisce la migliore soluzione continua trovata.

- `virtual void SetPrintScreen ( bool PrintScreen [pure virtual]`

  Il metodo permette di decidere se stampare a video messaggi di output durante l'esecuzione.

- `void SetPar ( int par, double value ) [inline, virtual]`

  Il metodo permette di settare i parametri generali dell'algoritmo che hanno un valore di tipo float.

Parameters:

- `par` è il parametro da settare; si può usare l'enumerato `TTDParam`, anche se il metodo accetta un int (ogni enumerato è un int) in modo da poter essere esteso dalle classi derivate per il settaggio dei loro parametri
- `value` è il valore da assegnare al parametro.

- `virtual void SetPar ( int par, int value ) [pure virtual]`

  Il metodo permette di settare i parametri generali dell'algoritmo che hanno un valore di tipo integer.

  Parameters:

  - `par` è il parametro da settare; si può usare l'enumerato `TTDParam`, anche se il metodo accetta un int (ogni enumerato è un int) in modo da poter essere esteso dalle classi derivate per il settaggio dei loro parametri
  - `value` è il valore da assegnare al parametro.

- `void GetPar ( int par, double & value ) [inline, virtual]`

  Il metodo permette di leggere i parametri generali dell'algoritmo che hanno un valore di tipo float.

  Parameters:

  - `par` è il parametro da leggere; si può usare l'enumerato `TTDParam`, anche se il metodo accetta un int (ogni enumerato è un int) in modo da poter essere esteso dalle classi derivate per la lettura dei loro parametri
  - `value` al ritorno contiene il valore del parametro.

- `virtual void GetPar ( int par, int & value ) [pure virtual]`

  Il metodo permette di leggere i parametri generali dell'algoritmo che hanno un valore di tipo integer.

  - `par` è il parametro da leggere; si può usare l'enumerato `TTDParam`, anche se il metodo accetta un int (ogni enumerato è un int) in modo da poter essere esteso dalle classi derivate per la lettura dei loro parametri
  - `value` al ritorno contiene il valore del parametro.

- `virtual void SetLogPar ( bool showInputBS, bool showInputTT,`
  `bool showGraphBS, bool showGraphTT,`
  `int printTempSolBS, int printTempSolTT,`
  `bool showTempSolBS, bool showTempSolTT,`
  `bool showTempLastSol, bool showTempBestSol, int showTempSolEveryIt,`
  `int showTempFOEveryIt, bool showBetterSol, int printFileBetterSol,`
  `string nameInput, string nameOutput, bool ShowSolution ) [pure virtual]`

Il metodo determina il modo con il quale l'algoritmo interagisce con l'utente, ovvero quali dati mostrare tramite file di log durante l'esecuzione.

Parameters:

- `showInputBS` se settato a true la classe `BSSolver` riporta nel file di Log l'input che riceve dalla classe `TTDDualLagrSolver`
- `showInputTT` se settato a true la classe `TTSolver` riporta nel file di Log l'input che riceve dalla classe `TTDDualLagrSolver`
- `showGraphBS` se settato a true la classe `BSSolver` riporta nel file di Log la struttura del grafo che sta creando
- `showGraphTT` se settato a true la classe `TTSolver` riporta nel file di Log la struttura del grafo che sta creando
- `printTempSolBS` la classe `BSSolver` stamperà in files appositi la struttura delle soluzioni che troverà ogni x iterazioni, dove x è indicato proprio da questo parametro
- `printTempSolTT` la classe `TTSolver` stamperà in files appositi la struttura delle soluzioni che troverà ogni x iterazioni, dove x è indicato proprio da questo parametro
- `showTempSolBS` se settato a true l'algoritmo, ogni x iterazioni, riporta nel file di Log la sol. parziale in quel momento trovata dalla classe `BSSolver`
- `showTempSolTT` se settato a true l'algoritmo, ogni x iterazioni, riporta nel file di Log la sol. parziale in quel momento trovata dalla classe `TTSolver`
- `showTempLastSol` se settato a true l'algoritmo, ogni x iterazioni, riporta nel file di Log l'ultima soluzione ammissibile in quel momento trovata
- `showTempBestSol` se settato a true l'algoritmo, ogni x iterazioni, riporta nel file di Log la migliore soluzione ammissibile in quel momento trovata
- `ShowTempSolEveryIt` l'algoritmo riporta nel file di Log le soluzioni (seguendo le indicazioni dei precedenti 4 par.) ogni x iterazioni, dove x è settato da questo par.
- `ShowTempFOEveryIt` l'algoritmo riporta nel file di Log tutti i valori "funzione obiettivo" ogni x iterazioni, dove x è settato da questo par.
- `ShowBetterSol` se settato a true l'algoritmo riporta nel file di Log ogni soluzione (ammissibile) che migliora la f.o.
- `PrintFileBetterSol` parametro per regolare le stampe su file formato TTD delle soluzioni ammissibili che migliorano la f.o. Se `PrintFileBetterSol` = 0 non vi sarà nessuna stampa, se `PrintFileBetterSol` = 1 il file d' output (quello dove eventualmente sarà scritta la soluzione trovata al termine dell' intera computazione) sarà via via sovrascritto, se `PrintFileBetterSol` = 2 ogni soluzione che determina un miglioramento sarà stampata su un singolo file
- `nameInput` nome del file (in formato TTD) dal quale l'algoritmo estrae i dati di input
- `nameOutput` nome del file (in formato TTD) nel quale l'algoritmo scrive la soluzione finale trovata

  - `showSolution` se settato a true alla fine dell' esecuzione l'algoritmo riporta nel file di Log la soluzione eventualmente trovata.

- `virtual double GetCost ( void ) [pure virtual]`

  Il metodo ritorna il costo della soluzione trovata.

- `virtual double GetLowerBound ( void ) [pure virtual]`

  Il metodo ritorna il costo del Lower Bound, ovvero l'ultima valutazione inferiore (quindi generalmente non ammissibile) calcolata sul valore della soluzione ottima.

- `virtual void SetDimension ( int numOfDepots, int numOfNodes, int numOfPaths, int numOfObPoints ) [pure virtual]`

  Il metodo permette di settare le "dimensioni del problema".

  Parameters:

  - `numOfDepots` il numero di depositi
  - `numOfNodes` il numero di nodi
  - `numOfPaths` il numero di percorsi
  - `numOfObPoints` il numero di punti d'osservazione riguardo agli intertempi

- `virtual void SetTimeBand ( TimeBand band, Time startDay, Time endDay ) [pure virtual]`

  Il metodo permette di settare la partizione generale dell'arco temporale in fasce orarie, nonché gli orari di inizio/fine giornata.

  Parlando di una partizione "generale", essa deve poter comprendere tutte le possibili fasce orarie utili; quindi essa sarà l'intersezione di tutte le particolari partizioni in fasce orarie presenti nell'input del problema. Ad esempio, se ci fossero due diverse fasce orarie:

  $00:00 - 12:00 - 17:00 - 23:59$

  00:00 - 08:00 - 12:00 - 18:00 - 23:59

  l'intersezione rappresentata in band sarebbe

  00:00 - 08:00 - 12:00 - 17:00 - 18:00 - 23:59.

- `virtual void GetTimeBand ( TimeBand & band, Time & startDay, Time & endDay ) [pure virtual]`

  Il metodo restituisce la partizione generale dell'arco temporale in fasce orarie, nonché gli orari di inizio/fine giornata.

- `virtual void AddDepot ( IndexNodes depotIndex, string code, string name, TimeBand minStop, TimeBand maxStop ) [pure virtual]`

  Il metodo permette di aggiungere un deposito.

Parameters:

- `depotIndex` è l'identificativo del deposito da aggiungere
- `code` è il codice del deposito
- `name` è il nome del deposito
- `minStop, maxStop` vettori che indicano, per ogni fascia oraria, rispettivamente il tempo di sosta minima/massima che ogni bus deve rispettare al deposito, per ciascuna fascia oraria (i vettori hanno la stessa dimensione del vettore delle fasce orarie, si veda `SetTimeBand()`, e l'i-esimo elemento del vettore delle fasce orarie è in relazione con gli i-esimi elementi dei vettori `minStop` e `maxStop`).

Se viene aggiunto un deposito di troppo rispetto a quanto indicato col metodo `SetDimension()`, il metodo lancia una `TTDException`.

- `virtual void AddNode ( IndexNodes nodeIndex, string code, string name ) [pure virtual]`

  Il metodo permette di aggiungere un nodo, sia un nodo capolinea che un nodo non capolinea.

  Parameters:

  - `nodeIndex` è l'identificativo del nodo da aggiungere
  - `code` è il codice del nodo
  - `name` è il nome del nodo

  Se viene aggiunto un nodo di troppo rispetto a quanto indicato col metodo `SetDimension()`, il metodo lancia una `TTDException`.

- `virtual void SetStopInTerminus ( IndexNodes terminus, TimeBand minStop, TimeBand maxStop ) [pure virtual]`

  Il metodo permette di aggiungere le soste min/max ad un capolinea.

  Parameters:

  - `terminus` è l'identificativo del capolinea al quale vengono abbinate le soste min/max
  - `minStop, maxStop` vettori che indicano rispettivamente, il tempo di sosta minima/massima che ogni bus deve rispettare al capolinea, per ciascuna fascia oraria (i vettori hanno la stessa dimensione del vettore delle fasce orarie, si veda `SetTimeBand()`, e l'i-esimo elemento del ettore delle fasce orarie è in relazione con gli i-esimi elementi dei vettori `minStop` e `maxStop`).

  Se viene indicato, tramite il parametro terminus, un capolinea che non esiste, il metodo `SetStopInTerminus` lancia un'eccezione usando la classe `TTDException` .

- `virtual void SetArcFromDepotToTerminus ( IndexNodes depotIndex, IndexNodes terminusIndex, TimeBand time ) [pure virtual]`

  Il metodo permette di settare la descrizione dettagliata del collegamento tra un particolare deposito e un particolare capolinea.

  Parameters:

  - `depotIndex, terminusIndex` indicano la coppia deposito-capolinea della quale vogliamo fornire la descrizione
  - `time`, vettore che indica, per ogni fascia oraria, il tempo di percorrenza che occorre per percorrere il tragitto dal deposito al capolinea; quindi anche il vettore time va letto in concomitanza col vettore delle fascie orarie (si veda ad es. `AddNode()`).

  Se viene settato un arco non esistente, cioè sono passati per parametro identificativi di deposito/capolinea non passati coi metodi `AddDepot()` e `AddNode()`, il metodo lancia una `TTDException`.

- `virtual void SetArcFromTerminusToDepot ( IndexNodes terminusIndex, IndexNodes depotIndex, TimeBand time ) [pure virtual]`

  Analogo a `SetArcFromDepotToTerminus()`, ma il collegamento è dal capolinea al deposito e non viceversa.

- `virtual void AddObservationPoint ( IndexNodes obPoint, bool dirDi, TimeBand lapTime, TimeBand lapTimeMin, TimeBand lapTimeMax, vector< BusChartType > chartBusProv, vector< BusChartType > chartBusDest ) [pure virtual]`

  Il metodo permette di aggiungere i "punti di osservazione", ovvero quei nodi, già precedentemente aggiunti col metodo `AddNode()`, sui quali si calcolano gli intertempi.

  Se un nodo è punto di osservazione per entrambe le direzioni, è necessario chiamare due volte il metodo `AddObservationPoint()` per aggiungerlo.

  Parameters:

  - `obPoint` è l' identificativo del nodo in questione.
  - `dirDi` se è true che il nodo è punto di osservazione per i percorsi con direzione '`Di`', se è false allora lo è per i percorsi con direzione '`As`' (si veda il commento a `AddPath()`)
  - `lapTime` è un vettore che indica, per ogni fascia oraria, l'intertempo desiderato sul nodo tra due bus consecutivi
  - `lapTimeMin` è un vettore che indica, per ogni fascia oraria, l'intertempo minimo consentito tra due bus che passano consecutivamente dal nodo
  - `lapTimeMax` analogo a `lapTimeMin` per l'intertempo massimo consentito

- `chartBusProv` è un vettore che contiene tutti gli schemi-bus, uno per ogni fascia oraria, che indicano come i sottopercorsi 'di provenienza' (ovvero quelli che finiscono col nodo d'osservazione) devono essere ordinate negli schemi delle corse. In realtà con questo metodo non vengono passati sottopercorsi bensì codici di percorsi dai quali la classe concreta dovrà ricavare il giusto sottopercorso: sarà quindi possibile indicare con il codice di un percorso specifico un particolare sottopercorso che poi, in fase di scelta delle corse, rappresenterà sia il percorso dal quale è stato ricavato sia altri percorsi formati dallo stesso primo tratto in comune.
- `chartBusDest` analogo a `chartBusProv`, riguardante però i sottopercorsi 'di destinazione' (ovvero quelli che iniziano col nodo d'osservazione)

Se viene aggiunto come punto di osservazione un nodo non esistente, ovvero è passato per parametro un identificativo di nodo non passato col metodo `AddNode()`, il metodo lancia una `TTDException`.

- `virtual void SetEvalBus (` $vector < Time >$ `times,` $vector < int >$ `numOfBus ) [pure virtual]`

  Il metodo permette di inserire, se esiste, la stima (effettuata a priori) del numero di vetture necessarie per effettuare il servizio in diversi periodi che suddividono la giornata.

  L' elemento i-esimo del vettore `times`, passato per parametro, determina l'istante temporale che conclude l'i-esimo periodo nel quale è suddivisa la giornata, mentre il corripondente elemento del vettore `numOfBus` determina la stima di vetture ritenute necessarie per quel periodo.

- `virtual void AddPath ( vector< IndexNodes > nodesIndex, string code, bool dirDi, vector< TripType > trips, vector< vector< int > > tripsNoDep ) [pure virtual]`

  Il metodo permette di aggiungere un percorso, e le sue possibili corse.

  Parameters:

  - `nodesIndex` è un vettore di indentificativi di capolinea che descrive il percorso tramite i nodi che lo compongono, dal nodo iniziale a quello finale
  - `code` è il codice identificativo del percorso
  - `dirDi` se è true il percorso ha direzione '`Di`', se è false allora il percorso ha direzione '`As`'. Si noti che sono necessarie due chiamate diverse ad `AddPath()` per inserire percorsi che toccano lo stesso insieme di nodi ma nelle due diverse direzioni. Ad esempio, se abbiamo un itinerario bidirezionale A - B - C (quindi A - B - C e C - B - A) questo sarà inserito tramite due chiamate al metodo `AddPath()`: la prima inserisce A - B - C con direzione '`Di`', la seconda chiamata inserisce C - B - A con direzione '`As`'. Al contrario dei punti di osservazione, un percorso ha una e una sola direzione. Il concetto di direzione partiziona in due sia l'insieme dei percorsi che quello dei punti di osservazione, creando un legame tra percorsi e nodi punti

di osservazione che hanno la stessa direzione. Riprendendo l' esempio di sopra, se B fosse punto di osservazione per la direzione 'Di', esso influenzerebbe la scelta, tra le altre, delle corse di A - B - C (ma non di C - B - A); se invece B fosse punto di osservazione sia per la direzione 'Di' che per quella 'As', esso influenzerebbe tutte le corse di entrambi i percorsi.

- `trips` è un vettore di elementi di tipo `TripType`; ogni elemento descrive una possibile corsa (si veda la descrizione di `TripType` per dettagli)
- `tripsNoDep` sono due elenchi di id di corse verso le quali non è consentito il collegamento al deposito in un verso o in un altro

Se viene aggiunto un percorso di troppo rispetto a quanto indicato col metodo `SetDimension()`, il metodo lancia una `TTDException`.

- `virtual void SetContTripsInFileInput ( vector< string > & contTrips ) pure virtual`

  Il metodo passa come parametro un puntatore a un vettore di stringhe che indica il contenuto testuale per ogni corsa nel file d'input.

- `virtual void ChgTempFixedTrips ( vector< int > idFixedTrips, vector< int > idNoFixedTrips ) [pure virtual]`

  Il metodo permette di considerare prefissate temporaneamente un insieme di corse indicato dal vettore di identificativi `idFixedTrips` e, di contro, temporaneamente non fissate altre corse indicate dal vettore di identificativi `idNoFixedTrips`.

  Con questo metodo è possibile fissare una o più corse non già prefissate con `AddTrips()` senza modificare (e quindi compromettere) il reale insieme di corse già prefissate.

- ```
  virtual void GetTrips ( vector< int > & idTrips,
  vector< string > & descrTrips, vector< vector< int > > & spts,
  vector< int > & timeBand, vector< vector< Time > > & timeTrips,
  vector< vector< Time > > & sptLapTimes,
  vector< vector< Time > > & sptMinLapTimes,
  vector< vector< Time > > & sptMaxLapTimes,
  vector< int > & singleInitialTrips,
  vector< int > & singleFinalTrips,
  Time & endDay, vector< vector< int > > & idTripsUsed ) [pure virtual]
  ```

  Il metodo restituisce alcune informazioni utili calcolate in precedenza.

  Il vettore `idTrips` restituisce gli id delle corse esistenti, il vettore `descrTrips` fornisce una breve descrizione delle corse, mentre la matrice `spts` indica come queste siano state partizionate per punto di osservazione. Il vettore `timeBand` indica, ad ogni corsa, la relativa fascia oraria d' appartenenza. La matrice `timeTrips` indica, per ogni corsa e per ogni nodo d'osservazione, l'eventuale orario di passaggio dal nodo d'osservazione. Le matrici `sptLapTimes`, `sptMinLapTimes`, `sptMaxLapTimes` indicano, per ogni nodo
  ```

d'osservazione e per ogni fascia oraria, l'intertempo richiesto e la relativa soglia minima/massima. I vettori `singleInitialTrips` e `singleFinalTrips` indicano rispettivamente, per ogni nodo d'osservazione, se esistono corse che sono le uniche che possono essere considerate le prime/ultime: in caso contrario viene restituito -1. `endDay` indica il passaggio al relativo nodo d'osservazione dell' ultima corsa della giornata, mentre `idTripsUsed` restituisce gli id delle corse effettivamente usate nei grafi.

- `virtual int IsNewFixFeasible (` *vector* < *int* > `idFixedTrips,` *vector* < *int* > `idNoFixedTrips ) [pure virtual]`

  Il metodo permette di stabile se fissare e defissare nuove corse siano operazioni che non violano i vincoli esistenti sui cammini minimi: il metodo restituisce -1 nel caso l'operazione non violi l'ammissibilità, altrimenti viene restituito l'identificativo del cammino minimo i cui vincoli vengono violati.

- `virtual TTDStatus Solve ( int & whatIter, int limIter = -1 ) [pure virtual]`

  Il metodo permette di risolvere il problema.

  Il metodo inoltre scrive nel parametro `whatIter` a quale iterazione il processo è stato interrotto, mentre nel parametro `limIter` viene passato, se è utile, l'iterazione alla quale si vuole interrompere temporaneamente l'esecuzione. Il metodo quindi ritorna un valore del tipo pubblico `TTDStatus` che indica lo stato dell'algoritmo al momento della terminazione:

  - `kOK` significa che è stata determinata una soluzione con l'accuratezza richiesta (si veda `SetPar()`);
  - `kStopped` significa che l'algoritmo è stato interrotto (ad es. perché è terminato il tempo massimo richiesto) avendo determinato una soluzione ammissibile, ma non con l'accuratezza desiderata;
  - `kUnSolved` significa che l'algoritmo è stato interrotto (ad es. perché è terminato il tempo massimo richiesto) senza aver determinato alcuna soluzione ammissibile;
  - `kError` significa che l'algoritmo è stato interrotto per via di un qualche errore (numerico, di memoria, ...) che non è stato possibile gestire.
  - `kInProgress` significa che l'algoritmo è stato interrotto per permettere allo strato superiore a questa interfaccia per poter gestire (temporaneamente) il controllo anche qualora l'algoritmo non abbia ancora terminato la ricerca della soluzione ottima

- ```
  virtual void GetResult ( vector< vector< int > > & busTrips,
  vector< bool > & compatOutLinea,
  double & foValue, bool makeSol = false ) [pure virtual]
  ```

  Il metodo scrive nel vettore `busTrips`, passato per riferimento, la soluzione del problema (se esiste).

  La soluzione consiste nel conoscere quanti bus saranno impegati durante l' arco temporale, quali viaggi dovranno compiere e quali corse dovranno effettuare. Il vettore
  ```

è quindi bidimensionale, per poter rappresentare tutti i viaggi fatti e, per ognuno di essi, tutti gli identificativi delle corse apartententi al viaggio. Il metodo inoltre scrive nel vettore boolean `compatOutLinea` se tra due viaggi consecutivi vi è compatibilità fuori linea (l' elemento relativo del vettore sarà settato a true) oppure se non fanno parte dello stesso turno macchina, e nel parametro `foValue` quanto costa la soluzione. L'ultimo parametro, `makeSol`, determina se la soluzione debba essere ricalcolata sul momento.

- ```
  virtual double GetCostFixedSolution (
  vector< vector< int > > tripsInJourneys,
  vector< int > idJourney,
  vector< vector< int > > turniMacchina ) [pure virtual]
  ```

  Il metodo restituisce il costo di una soluzione esterna (quindi non calcolata da TTDAlgo ma predeterminata) passata tramite: il vettore `tripsInJourneys` (che indica le corse facenti parte della soluzione divise in viaggi), il vettore `idJourney` (dove sono scritti gli id di ogni viaggio), infine il vettore `turniMacchina` (che indica come i viaggi siano distribuiti nei vari turni macchina effettuati).

- ```
  virtual void WriteFixedSolution ( vector< vector< int > > & arcs,
  string nameOutput ) [pure virtual]
  ```

  Il metodo scrive un file in formato ttd data una soluzione esterna. Il metodo passa come parametri un vettore contenente una soluzione descritta tramite tutti gli archi del sottografo BS con flusso non nullo e il nome del file sul quale deve scrivere.

11.2 TTInterface

11.2.1 Detailed Description

`TTInterface` definisce un'interfaccia astratta per il sottoproblema "Time Trips Trial".

Il sottoproblema ha come input tutte le corse (anche facenti parte di più percorsi diversi) che hanno una certa direzione e che quindi utilizzano un dato nodo come punto di osservazione.

Ad ogni corsa che soddisfa questi requisiti sarà neccessariamente legato quindi l' orario nel quale la corsa passa dal punto di osservazione; inoltre da input si può decidere se tale corsa è fissata, quindi da fare necessariamente, o meno. In input inoltre viene passata la divisione della giornata in fasce orarie, l' intertempo desiderato/minimo/massimo (per ogni fascia oraria) ai quali ogni intertempo tra due corse consecutive deve attenersi, il range d'ammissibilità iniziale (finale) che determina quali corse possono essere la prima (ultima) corsa della giornata,e il cosidetto "fattore di scala", utile per puntellare lo schema degli intertempi.

Una fascia oraria può non avere un intertempo desiderato, nel senso che gli intertempi delle corse al suo interno sono sostanzialmente liberi.

Vengono inoltre definiti i cosidetti 'schemi-bus', divisi in schemi-bus "di provenienza" e "schemi-bus di destinazione": sia per la provenienza che per la destinazione saranno previsti uno o più schemi-bus a coprire l'intera giornata. Ogni corsa buona per essere inserita in questa tabella oraria è legata al percorso che essa copre: tale percorso, formato ovviamente dai capolinea che lo compongono, sarà quindi partizionato in due sottopercorsi tramite il nodo d'osservazione che questa classe "racchiude". Ad esempio se il nodo d'osservazione è N, e il percorso è composto da i capolinea A - B - N - C - D, allora il sottopercorso "di provenienza" sarà A - B - N, mentre il sottopercorso di "destinazione" sarà N - C - D. Gli schemi-bus "di provenienza" quindi descriveranno, all'interno del periodo della giornata da loro descritto, una lista ordinata di sottopercorsi: le corse scelte dovranno quindi generare, attraverso i percorsi che coprono, una lista di sottopercorsi "di provenienza" del tutto identica a quella indicata. La stessa cosa ovviamente vale per gli "schemi-bus di destinazione".

Una soluzione ottima (o buona se non si è trovata quella ottima nel tempo massimo indicato in input) del sottoproblema è il particolare sottoinsieme delle corse che più si avvicina a soddisfare lo schema degli intertempi modificato in base al fattore di scala. La ricerca della soluzione ottima deve tener di conto che alcune delle corse possono essere "fissate" in fase d' input.

11.2.2 Classes

- `struct BusChartType`: struttura che descrive un singolo schema corse.
- `class Inf`: piccola classe per ricavarsi il valore "+ infinito" per un tipo primitivo: basta scrivere `Inf<tipo>()`.
- `class TTDException`: piccola classe per le eccezioni.

11.2.3 Enumeration Documentation

- `enum TTDStatus`

  Tipo enumerato pubblico che descrive i possibili stati del solver.

  Enumerator:

  - `kUnSolved`: nessuna soluzione disponibile
  - `kError`: nessuna soluzione trovata a causa di errore non gestibile
  - `kStopped`: trovata soluzione ammissibile ma non con la precisione richiesta
  - `kOK`: trovata soluzione con la precisione richiesta

11.2.4 Public types

`TTInterface` definisce due tipi semplici pubblici:

- `Time`, rappresenta sia istanti di tempo (hh:mm:ss, 0 codifica la mezzanotte) sia intervalli di tempo (in secondi), tramite un intero positivo
- `TimeBand`, rappresenta un vettore di `Time` utile a descrivere la divisione dell'arco temporale in fasce orarie

I tipi pubblici sono pesantemente usati nei parametri dei metodi pubblici di questa interfaccia. Questo significa che chi vuole usare l'interfaccia ha la possibilità di, ma è anche "costretto" ad, usare questi tipi.

11.2.5 Member Function Documentation

- `virtual TTInterface ()`

  Costruttore

- `virtual  TTInterface ()`

  Distruttore

- `virtual void SetDimension ( int numTrips ) [pure virtual]`

  Il metodo permette di settare le "dimensioni del problema":

  Parameters:

  - `numTrips` il numero di corse

- `virtual void SetLogPar ( bool showInput, bool showGraph, int printTempSol ) [pure virtual]`

  Il metodo determina il modo con il quale la classe interagisce con l'utente, ovvero quali dati mostrare tramite un file di log durante l'esecuzione.

Parameters:

- `showInput` se settato a true la classe `TTSolver` riporta nel file di Log l'input che riceve dalla classe `TTDDualLagrSolver`
- `showGraph` se settato a true la classe `TTSolver` riporta nel file di Log la struttura del grafo che sta creando
- `printTempSolTT`, se settato a true la classe `TTSolver` stamperà in files appositi la struttura delle soluzioni che troverà ogni x iterazioni, dove x è indicato proprio da questo parametro.

- `virtual void SetTimeBand ( TimeBand band, Time startDay, Time endDay, Time startWork, Time endWork ) [pure virtual]`

  Il metodo permette di settare la partizione generale dell'arco temporale in fasce orarie, nonché gli orari di inizio/fine giornata e gli orari di inizio/fine dei turni.

  Parlando di una partizione "generale", essa deve poter comprendere tutte le possibili fasce orarie utili; quindi essa sarà l'intersezione di tutte le particolari partizioni in fasce orarie presenti nell'input del problema. Ad esempio, se ci fossero due diverse fasce orarie:

  00:00 - 12:00 - 17:00 - 23:59

  00:00 - 08:00 - 12:00 - 18:00 - 23:59

  l'intersezione rappresentata in band sarebbe

  00:00 - 08:00 - 12:00 - 17:00 - 18:00 - 23:59.

- `virtual void SetObPointInfo ( string name, bool direction ) [pure virtual]`

  Il metodo permette di settare il nome del nodo di osservazione e la direzione sulla quale opera.

- `virtual void GetObPointInfo ( string & name, bool & direction ) [pure virtual]`

  Il metodo restituisce il nome del nodo di osservazione e la direzione sulla quale opera.

- `virtual void SetScale ( Time value ) [pure virtual]`

  Il metodo permette di settare il fattore di scala (in secondi).

  Il fattore di scala può modificare lo schema degli intertempi desiderati tra le corse nel caso in cui l'intertempo in questione non sia un multiplo del fattore di scala.

  Ad esempio se il fattore di scala è 1 minuto mentre l' intertempo desiderato su una data fascia oraria è 3 minuti, non ci sono modifiche allo schema che prevede come soluzioni ideali corse che partono a distanza di 3 minuti l'una dall' altra.

  Ipotizzando invece che l'intertempo I non sia multiplo del fattore di scala Fs, avremo che $I = Q \cdot Fs + r$. Si calcolano $n = Q \cdot Fs$, $a = r/Fs < 1$. Si sceglie quindi un nuovo numero A all'interno di 1/2, 1/3, 2/3, prendendo l'elemento dell' insieme più vicino ad a.

A seconda del valore di A viene definito un diverso schema d'intertempi.

Se A = 1/2 allora si definisce uno schema (ciclico) di intertempi desiderati del tipo n, n + Fs.

In pratica tra la prima corsa e la seconda (e fra la terza e la quarta ...) l'intertempo desiderato è $n = Q \cdot Fs$, mentre tra la seconda e la terza corsa (e fra la quarta e la quinta ...) l'intertempo desiderato ǹ + Fs.

Se A = 1/3 allora lo schema è n, n, n + Fs, mentre se A = 2/3 allora lo schema è n, n + Fs, n + Fs.

- `virtual void SetExecutionName ( string name ) [pure virtual]`

  Il metodo permette di settare il nome che identifica l'esecuzione in corso.

  Ad esempio può essere usato, per questo scopo, il nome del file d'input dal quale sono stati caricati i dati.

- `virtual void SetFunctionCost ( int what ) [pure virtual]`

  Il metodo setta quale usare tra le varie funzioni di costo implementate all'interno del solver TT.

- `virtual void SetOptimalLapTimes ( TimeBand lapTime, TimeBand lapTimeMin, TimeBand lapTimeMax ) [pure virtual]`

  Il metodo permette di settare, per ogni fascia oraria, l' intertempo desiderato a cui l' intertempo fra due corse consecutive dovrebbe tendere e l' intertempo minimo/massimo che fissa un range da rispettare.

- `virtual void GetOptimalLapTimes ( TimeBand & lapTime, TimeBand & lapTimeMin, TimeBand & lapTimeMax ) [pure virtual]`

  Il metodo restituisce, per ogni fascia oraria, l'intertempo desiderato a cui l'intertempo fra due corse consecutive dovrebbe tendere e l' intertempo minimo/massimo che fissa un range da rispettare.

- `virtual void SetCostsInLapTimes ( double alfa, double a, double b, double c, int band ) [pure virtual]`

  Il metodo permette di settare i quattro parametri (non negativi) che determinano contemporaneamente sia un vincolo che un costo sulla scelta di effettuare due corse consecutivamente all'interno della stessa fascia oraria.

  Per ogni coppia di corse quindi, tramite i quattro parametri e l' intertempo che le separa, l' algoritmo può determinare se sia possibile effettuarle consecutivamente con un intertempo non pari a quello desiderato ed eventualmente ne setta il costo; tale costo si intersecherà inevitabilmente con i costi sulle singole corse (vedi `SetTripsCosts()` e `SetTripCost()`). Il metodo permette di settare i quattro parametri singolarmente per ogni fascia oraria; dipendendo strettamente dalla divisione in fasce orarie della giornata, è obbligatorio chiamare il metodo solo dopo aver chiamato il metodo `SetTimeBand()`.

Parameters:

- `alfa` determina una soglia oltre la quale l'errore relativo fra l'intertempo effettivo della corsa t e l'intertempo desiderato d (per la specifica fascia oraria) diventa intollerabile e quindi le due corse non possono essere usate in sequenza. In particolare se $\delta = \frac{|t-f|}{f} <= \alpha$ allora l'intertempo della corsa è ammissibile.
- `a` è il coefficiente quadratico nella formula di costo
- `b` è il coefficiente lineare nella formula di costo
- `c` è il coefficiente costante nella formula di costo
- `band` è la fascia oraria alla quale associare i quattro parametri `alfa`, `a`, `b`, `c`. `band` deve essere settato in modo tale da essere compreso tra 0 e la dimensione massima del vettore delle fasce orarie settate da `SetTimeBand()`: in caso contrario viene lanciata un'eccezione con `TTDException`.

I tre parametri `a`, `b`, `c` determinano il costo (in funzione obiettivo), di scegliere una coppia di corse in sequenza (all'interno della stessa fascia oraria, per coppie di corse che appartengono a fasce diverse vedi `SetSmoothing()`).

Sono state ipotizzate tre possibili varianti di calcolo del suddetto costo.

Le prime due calcolano per prima cosa un intertempo relativo $\delta = \frac{|t-f|}{f}$.

Quindi la prima formula per il calcolo del costo dice che $costo = \begin{cases} a\delta^2 + b\delta + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

mentre la seconda dice che $costo = \begin{cases} a(\delta+1)^2 + b(\delta+1) + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

La terza ipotesi utilizza invece un intertempo assoluto $\delta = |t - f|$ per poi calcolare così il costo: $costo = \begin{cases} \frac{a\delta^2+b\delta}{f} + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

- `virtual void SetFirstTripCost ( double aI, double bI, double cI, Time rifTime, Time startTime = 0, Time endTime = Inf< Time >() ) [pure virtual]`

  Il metodo permette di settare i parametri (non negativi) che determinano il costo di effettuare la prima corsa ad una determinata distanza temporale dall'inizio della giornata.

  Parameters:

  - `aI` è il coefficiente quadratico nella formula di costo della distanza tra `rifTime` e la prima corsa
  - `bI` è il coefficiente lineare nella formula di costo della distanza tra `rifTime` e la prima corsa
  - `cI` è il coefficiente costante nella formula di costo della distanza tra `rifTime` e la prima corsa

- `rifTime` è l'istante di tempo dal quale determinare la distanza di ogni corsa
- `startTime, endTime` sono rispettivamente l' orario di inizio e l'orario di fine del periodo di tempo al quale associare i parametri `aI`, `bI`, `cI`, `rifTime`. Il parametro `startTime` è settato di default a 0, `endTime` a `Inf<Time>()`: i due valori significano rispettivamente l'inizio e la fine della giornata, qualunque siano gli orari d'inizio e fine giornata settati in precedenza da `SetTimeBand()`.

I tre parametri `aI`, `bI`, e `cI` determinano il costo (in funzione obiettivo), di scegliere una corsa come prima della giornata avente distanza /delta dall'istante passato col parametro `rifTime`.

La formula è $costo = \begin{cases} a\delta^2 + b\delta + c & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

- `virtual void SetLastTripCost ( double aI, double bI, double cI, Time rifTime, Time startTime = 0, Time endTime = Inf< Time >() ) [pure virtual]`

  Il metodo permette di settare i parametri (non negativi) che determinano il costo di effettuare l'ultima corsa ad una determinata distanza temporale dalla fine della giornata.

  Parameters:

  - `aF` è il coefficiente quadratico nella formula di costo della distanza tra `rifTime` e l'ultima corsa
  - `bF` è il coefficiente lineare nella formula di costo della distanza tra `rifTime` e l'ultima corsa
  - `cF` è il coefficiente costante nella formula di costo della distanza tra `rifTime` e l'ultima corsa
  - `rifTime` è l'istante di tempo dal quale determinare la distanza di ogni corsa; si usa per indicare la fine della giornata
  - `startTime, endTime` sono rispettivamente l' orario di inizio e l'orario di fine del periodo di tempo al quale associare i parametri `aI`, `bI`, `cI`, `rifTime`. Il parametro `startTime` è settato di default a 0, `endTime` a `Inf<Time>()`: i due valori significano rispettivamente l'inizio e la fine della giornata, qualunque siano gli orari d'inizio e fine giornata settati in precedenza da `SetTimeBand()`.

  I tre parametri `aF`, `bF`, e `cF` determinano il costo (in funzione obiettivo), di scegliere una corsa come prima della giornata avente distanza /delta dall'istante passato col parametro `rifTime`.

  La formula è $costo = \begin{cases} aF\delta^2 + bF\delta + cF & \text{se } \delta > 0 \\ 0 & \text{se } \delta = 0 \end{cases}$

- `virtual void SetTripsCosts ( vector< int > idTrips, vector< double > costs ) [pure virtual]`

  Il metodo permette di settare i costi delle corse indicate dal vettore `idTrips` degli identificativi altrimenti, di default, nulli.

  Tali costi interagiranno con i costi che riguardano le coppie di "corse consecutive" (vedi `SetCostsInLapTimes()`).

- `virtual void SetTripCost ( int idTrip, double cost ) [pure virtual]`

  Il metodo permette di settare il costo della corsa indicata da `idTrip` altrimenti di default nullo.

  Tale costo interagirà con i costi che riguardano le coppie di "corse consecutive" (`vedi SetCostsInLapTimes()`).

- `virtual bool IsNewFixFeasible ( vector< int > idFixedTrips, vector< int > idNoFixedTrips ) [pure virtual]`

  Il metodo permette di stabile se fissare e defissare nuove corse siano operazioni che non violano i vincoli esistenti.

- `virtual void SetProvChartBus ( vector< BusChartType > chartBus ) [pure virtual]`

  Il metodo permette di settare gli schemi-bus, uno per ogni fascia oraria, che indicano come i sottopercorsi 'di provenienza' (ovvero quelli che finiscono col nodo d'osservazione) devono essere ordinate negli schemi delle corse.

- `virtual void SetDestChartBus ( vector< BusChartType > chartBus ) [pure virtual]`

  Il metodo permette di settare gli schemi-bus, uno per ogni fascia oraria, che indicano come i sottopercorsi 'di destinazione' (ovvero quelli che iniziano col nodo d'osservazione) devono essere ordinate negli schemi delle corse.

- `virtual void ChgTempFixedTrips ( vector< int > idFixedTrips, vector< int > idNoFixedTrips ) [pure virtual]`

  Il metodo permette di considerare prefissate temporaneamente un insieme di corse indicato dal vettore di identificativi `idFixedTrips` e, di contro, temporaneamente non fissate altre corse indicate dal vettore di identificativi `idNoFixedTrips`.

  Con questo metodo è possibile fissare una o più corse non già prefissate con `AddTrips()` senza modificare (e quindi compromettere) il reale insieme di corse già prefissate.

- `virtual void GetWishLapTime ( vector< vector< int > > & wishLapTime ) [pure virtual]`

  Il metodo restituisce una matrice che indica, per ogni corsa, l'intertempo desiderato tra esse.

  La matrice ha 2 colonne: la prima indica l'id delle corse in questione, mentre l'ultima colonna indica l'intertempo desiderato.

- `virtual void SetSmoothing ( bool value, Time maxJump = 0 ) [pure virtual]`

  Il metodo permette di decidere se utilizzare lo smoothing durante i cambi di fascia oraria.

  Per ora è gestito solamente il caso "no smoothing".

  Nel caso "no smoothing" si controlla innanzitutto se il collegamento tra le due corse dentro diverse fasce oriarie sia ammissibile: detti rispettivamente p_d e p_c i passaggi delle due corse sul nodo d'osservazione, e detti I_{m_h}, I_{M_h}, $I_{m_{h+1}}$ e $I_{M_{h+1}}$ gli intertempi minimi e massimi rispettivamente delle fasce orarie h e $h+1$, l'intertempo è ammissibile se $min(I_{m_h}, I_{m_{h+1}}) <= p_d - p_c <= max(I_{M_h}, I_{M_{h+1}})$

  Quindi si tiene sostanzialmente conto di 3 casi possibili.

  Se le due fasce non possiedono intertempi desiderati, il costo di ogni corsa che attraversa le due fasce sarà nullo.

  Se una sola delle due fasce non ha un intertempo desiderato, il costo di ogni corsa che attraversa le due fasce sarà calcolato con la formula "normale" (vedi `SetCostsInLapTimes()`) usando come intertempo quello della fascia oraria che ne possiede uno.

  Se entrambe le fasce orarie hanno un intertempo desiderato, si deve tener conto di tre ulteriori possibilità, tre modi per calcolare δ, e quindi il costo dell'effettuare le due corse.

  Detti I_h e I_{h+1} gli intertempi desiderati nelle fasce h e $h+1$, si fissano $I_m = min(I_h, I_{h+1})$ e $I_M = max(I_h, I_{h+1})$.

  Se $p_d - p_c < I_m$ allora $\delta = |p_d - p_c - I_m|/I_m$.

  Se $p_d - p_c > I_M$ allora $\delta = |p_d - p_c - I_M|/I_M$.

  Se $I_m <= p_d - p_c <= I_M$ allora si calcola un nuovo intertempo I_{new} frutto di una media pesata degli intertempi I_h e I_{h+1} e della "precentuale di presenza" dell'arco all'interno delle due fasce.

  Detto t l'istante del cambio fascia in questione, si ha che $I_{new} = I_h * \frac{t-p_c}{p_d-p_c} + I_{h+1} * \frac{p_d-t}{p_d-p_c}$

  Quindi $\delta = |p_d - p_c - I_{new}|/I_{new}$

  Usare questo metodo dopo aver inserito le fasce orarie col metodo `SetTimeBand()` e inserito gli intertempi min/max/desiderati col metodo `SetCostsInLapTimes()`.

- `virtual void AddTrips ( vector< int > idTrips,`
  `vector< string > meansTrips, vector< vector< int > > idNodes,`
  `TimeBand startTimeTrip, TimeBand endTimeTrip,`
  `TimeBand timeOnObPoint, vector< bool > fixedTrips ) [pure virtual]`

  Il metodo aggiunge un insieme di corse al problema.

  Le corse non necessariamente devono essere aggiunte in un solo colpo. Il metodo infatti non pone limiti al numero di chiamate ad esso: l'unico vincolo è di non aggiungere un numero di corse eccessivo rispetto a quanto indicato con `SetDimension()`, altrimenti il metodo lancia una `TTDException`.

Parameters:

- `idTrips` vettore che indica gli identificativi delle corse aggiunte (l'i-esimo id appartiene alla i-esima corsa aggiunta, e ciò vale anche per gli altri vettori)
- `meansTrips` vettore che indica il significato delle corse aggiunte
- `idNodes` matrice che contiene, per ogni corsa passata, gli id dei nodi che compongono il percorso coperto dalla corsa
- `startTimeTrip, endTimeTrip` vettori che indicano, per ogni corsa, rispettivamente l' istante di partenza e d'arrivo
- `timeOnObPoint` vettore che indica, per ogni corsa, l' istante temporale nel quale il bus, all' interno della corsa, passa dal nodo delegato ad essere punto d'osservazione
- `fixedTrips` vettore che indica se le corse aggiunte sono prefissate o meno.

- `virtual void GetIdInputTrips ( vector< int > & idTrips ) [pure virtual]`

  Il metodo scrive nel vettore `idTrips` l' insieme delle corse facenti parte dell' input.

- `virtual double GetCostFixedSolution ( vector< int > idTrips, double & costStartDay, vector< double > & costTimeBands, double & costEndDay, string & exitCode, double InfValue = Inf< double >(), int numIteration = 0 ) [pure virtual]`

  Il metodo calcola il costo totale di una possibile soluzione inserita dall'esterno tramite il vettore `idTrips`, vettore degli identificatori di corse; quindi descrive tramite le variabili `costStartDay`, `costMiddleDay`, `costEndDay` come il costo globale della soluzione sia stato ottenuto, riportando nel dettaglio il costo derivato dall'inizio della giornata, dalla fine della giornata e infine da tutto il resto della giornata (per fasce orarie).

  Non è detto comunque che tale soluzione sia ammissibile; in particolare è probabile che tra alcune corse definite consecutive l' intertempo sia troppo alto o troppo basso per i limiti stabiliti inizialmente (vedi `SetOptimalLapTimes()`). Nel caso in cui tali limiti siano violati anche solo una volta, il metodo dovrà settare adeguatamente il parametro `exitCode` e dovrà restituire infinito: il secondo parametro, `InfValue`, permette di stabilire con quale valore simulare l'infinito. Il parametro `numIteration` segnala l'iterazione corrente.

- `virtual void GetUsedTrips ( vector< int > & idTrips ) [pure virtual]`

  Il metodo restituisce gli id delle corse che passano da questo nodo d'osservazione e sono collegate al grafo SPT.

- `virtual void GetTrips ( vector< int > & idTrips, vector< Time > & timeOnObPoint, Time & endDay, int & singleInitialTrip, int & singleFinalTrip ) [pure virtual]`

  Il metodo restituisce alcune informazioni utili calcolate in precedenza.

  Per ogni corsa che passa dal nodo d'osservazione in questione, il vettore `idTrips` indica l'id, mentre il vettore `timeOnObPoint` ne indica il passaggio dal suddetto nodo. `endDay`

indica il passaggio al nodo dell'ultima corsa della giornata. Infine `singleInitialTrip` e `singleFinalTrip` indicano rispettivamente, se esistono corse che sono le uniche che possono essere considerate le prime/ultime: in caso contrario viene restituito -1.

- `virtual TTStatus Solve ( int numIteration = 0 ) [pure virtual]`

  Il metodo permette di risolvere il problema.

  Il metodo ritorna un valore del tipo pubblico `TTDStatus` che indica lo stato dell'algoritmo al momento della terminazione:

  - `kOK` significa che è stata determinata una soluzione con l'accuratezza richiesta (si veda `SetPar()`);
  - `kStopped` significa che l'algoritmo è stato interrotto (es. perché è terminato il tempo massimo richiesto) avendo determinato una soluzione ammissibile, ma non con l'accuratezza desiderata;
  - `kUnSolved` significa che l'algoritmo è stato interrotto (es. perché è terminato il tempo massimo richiesto) senza aver determinato alcuna soluzione ammissibile;
  - `kError` significa che l'algoritmo è stato interrotto per via di un qualche errore (numerico, di memoria, ...) che non è stato possibile gestire.

  Può essere utile, in fase di debug della soluzione TT, avere un modo per poter osservare nel dettaglio la soluzione. Il parametro `numIteration` segnala l'iterazione corrente.

- `virtual void GetTripsResult ( vector< int > & idTrips ) [pure virtual]`

  Il metodo scrive nel vettore `idTrips` passato per riferimento l' insieme delle corse facenti parte dell' ultima soluzione trovata dal solutore.

- `virtual void GetFoResult ( double & foValue,`
  `double & costStartDay, double & costMiddleDay,`
  `double & costEndDay, bool forcedTripsWithCostsZero ) [pure virtual]`

  Il metodo scrive nella variabile `foValue` passata per riferimento il valore della funzione obiettivo dell'ultima soluzione trovata dal solutore; quindi descrive tramite le variabili `costStartDay`, `costMiddleDay`, `costEndDay` come il costo globale della soluzione sia stato ottenuto, riportando nel dettaglio il costo derivato dall' inizio della giornata, dalla fine della giornata e infine da tutto il resto della giornata.

  Il parametro `forcedTripsWithCostsZero`, se settato a true, forza il solutore a considerare nulli i costi delle singole corse nel calcolo della f.o.

11.3 BSInterface

11.3.1 Detailed Description

`BSInterface` definisce un'interfaccia astratta per il sottoproblema "Bus Scheduling Trial".

Il sottoproblema ha come input tutte le corse di tutti i percorsi esistenti. Per ogni corsa viene passato l'orario di passaggio del bus sul nodo cosidetto "punto d'osservazione".

Input aggiuntivi sono i capolinea di ogni percorso e il deposito dei bus. Per il deposito esiste una sosta minima e una sosta massima consentita che ogni bus entrante nel deposito deve necessariamente rispettare. Anche per ogni capolinea esiste un concetto analogo di sosta min/max.

Sia nel caso del deposito che nel caso dei capolinea non esiste un'unica coppia di valori sosta min/max valida per tutta la giornata, che invece è divisa in fasce orarie: per ogni fascia oraria esiste una diversa sosta min/max. I capolinea sono collegati al deposito, in una direzione e nell'altra.

Per ogni coppia capolinea-deposito, per entrambe le direzioni possibili e per ognuna delle fasce orarie esistenti è indicato un tempo di percorrenza.

Una soluzione ottima (o buona se non si è trovata quella ottima nel tempo massimo indicato in input) del sottoproblema è il particolare sottoinsieme delle corse che minimizza il numero di bus da utilizzare per coprirle e i costi da sostenere per poter effettuare la copertura delle corse. Quindi oltre al costo derivato dal mettere a disposizione un bus in più, l' algoritmo deve considerare il costo specifico di effettuare una singola corsa, ma anche il costo di effettuare, nel periodo che intercorre tra una corsa e quella seguente, una sosta al capolinea nonché il costo, in alternativa alla sosta al capolinea, di effettuare un rientro più una sosta e quindi un'uscita dal deposito. La ricerca della soluzione ottima deve tener di conto che alcune delle corse possono essere "fissate" in fase d'input.

11.3.2 Classes

- `class Inf`: piccola classe per ricavarsi il valore "+ infinito" per un tipo primitivo: basta scrivere `Inf<tipo>()`.
- `class TTDException`: piccola classe per le eccezioni.

11.3.3 Enumeration Documentation

- `enum TTDStatus`

  Tipo enumerato pubblico che descrive i possibili stati del solver.

  Enumerator:

  - `kUnSolved`: nessuna soluzione disponibile
  - `kError`: nessuna soluzione trovata a causa di errore non gestibile
  - `kStopped`: trovata soluzione ammissibile ma non con la precisione richiesta
  - `kOK`: trovata soluzione con la precisione richiesta

11.3.4 Public types

`BSInterface` definisce tre tipi semplici pubblici:

- `Time`, rappresenta sia istanti di tempo (hh:mm:ss, 0 codifica la mezzanotte) sia intervalli di tempo (in secondi), tramite un intero positivo
- `IndexNodes`, rappresenta gli indici dei capolinea o gli indici dei depositi
- `TimeBand`, rappresenta un vettore di `Time` utile a descrivere la divisione dell'arco temporale in fasce orarie

I tipi pubblici sono pesantemente usati nei parametri dei metodi pubblici di questa interfaccia. Questo significa che chi vuole usare l'interfaccia ha la possibilità di, ma è anche "costretto" ad, usare questi tipi.

11.3.5 Member Function Documentation

- `virtual BSInterface ()`

  Costruttore

- `virtual  BSInterface ()`

  Distruttore

- `virtual void SetGammaM ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso dei turni macchina all'interno della funzione obiettivo.

  Il numero di turni macchina è il numero totale di bus utilizzati da una soluzione; analogamente, il numero di volte che un bus torna al deposito di partenza per non uscirne più (nell'intervallo temporale del problema). Nella funzione obiettivo, ogni turno macchina utilizzato ha un costo `GammaM`, il cui valore è settato da questo metodo.

- `virtual void SetGammaI ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso della "compatibilità in linea" all'interno della funzione obiettivo.

  Ogni bus ha la possibilità, una volta terminata la corsa e quindi una volta arrivato al capolinea, di sostare al capolinea per un certo periodo per poi ripartirvi per effettuare un'altra corsa. Nella funzione obiettivo, questa azione (detta compatibilità in linea tra le due corse) ha un costo diviso in due parti: una parte proporzionale al tempo di attesa al capolinea in più rispetto al tempo di sosta minimo previsto moltiplicato per un opportuno fattore `GammaI`, il cui valore è settato da questo metodo, l'altra parte consiste in un costo fisso (che dipende non dalla sosta ma dalla corsa di inizio sosta) da pagare se la sosta è maggiore dell'intertempo desiderato nella fascia oraria che contiene l'istante di arrivo al capolinea dove è prevista la sosta.

Si noti che è sempre possibile evitare di fare compatibilità in linea al costo di usare un numero di bus molto alto: per ogni corsa si usa un bus diverso. Quindi, dando valori molto alti a `GammaI`, per cui il costo della compatibilità in linea sia superiore al costo di un turno macchina (si veda `SetGammaM()`), esiste una soluzione ottima che non ne usa nessuna. Pertanto, in questo caso potrebbe essere possibile semplificare il problema assumendo che nessuna compatibilità in linea venga mai eseguita. Non si ritiene che ciò abbia interesse, per cui questa condizione potrà essere ignorata.

- `virtual void SetGammaF ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso delle "compatibilità fuori linea" all'interno della funzione obiettivo.

  Ogni bus ha la possibilità, una volta terminata la corsa e quindi una volta arrivato al capolinea, di andare in un deposito, sostarci e poi ripartire da lì per effettuare una nuova corsa. Nella funzione obiettivo, il costo di tornare al deposito (detta compatibilità fuori linea tra due corse) è proporzionale al tempo di viaggio fino al deposito (ma non al tempo di attesa al deposito, in cui il mezzo "non è viaggiante") moltiplicato per un opportuno fattore `GammaF`, il cui valore è settato da questo metodo.

  Si noti che ponendo `GammaI` $\neq$ `GammaF` (si veda `SetGammaI()`) si può esprimere una preferenza tra l'attesa in linea ed il rientro al deposito.

  Si noti che è sempre possibile evitare di fare compatibilità fuori linea al costo di usare un numero di bus molto alto: per ogni corsa si usa un bus diverso. Quindi, dando valori molto alti a `GammaF`, esiste una soluzione ottima che non ne usa nessuna. Pertanto, in questo caso potrebbe essere possibile semplificare il problema assumendo che nessuna compatibilità fuori linea venga mai eseguita.

  In gergo tecnico, porre `GammaF` ad un valore molto alto, e quindi eliminare la compatibilità fuori linea, significa che nella funzione obiettivo vengono contati i "turni macchina" e non i "viaggi". I viaggi sono il numero di volte che un bus esce dal deposito, mentre i turni macchina sono il numero totale di bus utilizzati: i viaggi possono essere di più dei turni macchina perché lo stesso bus può uscire più di una volta dal deposito (proprio nel caso di compatibilità fuori linea). Usualmente minimizzare il numero di turni macchina è importante, ma può darsi ed esempio il caso che il numero di turni macchina minimo sia già noto. In questo caso, ponendo `GammaF` ad un valore molto alto si unifica il concetto di "viaggio" e "turno macchina" (che coincidono se non possono essere effettuate compatibilità fuori linea), e quindi minimizzando i secondi si minimizzano i primi.

  Si ritiene che possa essere interessante esplorare questo tipo di situazione. Pertanto, dare un costo `Inf<double>` a `GammaF` in questo metodo corrisponde ad indicare che non possono essere effettuate compatibilità fuori linea.

- `virtual void SetGammaS ( double value ) [pure virtual]`

  Il metodo permette di settare il parametro che determina il peso delle "compatibilità fuori linea iniziali/finali" all'interno della funzione obiettivo.

Per ogni singolo turno macchina il costo di uscire la prima volta dal deposito (rientrare l'ultima volta al deposito) è proporzionale al tempo che occore al bus per percorrere il tragitto deposito–capolinea moltiplicato per un opportuno fattore `gammaS`, il cui valore è settato da questo metodo.

- `virtual void SetExecutionName ( string name ) [pure virtual]`

  Il metodo permette di settare il nome che identifica l'esecuzione in corso.

  Ad esempio può essere usato, per questo scopo, il nome del file d'input dal quale sono stati caricati i dati.

- `virtual void SetLogPar ( bool showInput, bool showGraph, int printTempSol ) [pure virtual]`

  Il metodo determina il modo con il quale la classe interagisce con l'utente, ovvero quali dati mostrare tramite un file di log durante l'esecuzione.

  Parameters:

  - `showInput` se settato a true la classe `BSSolver` riporta nel file di Log l'input che riceve dalla classe `TTDDualLagrSolver`
  - `showGraph` se settato a true la classe `BSSolver` riporta nel file di Log la struttura del grafo che sta creando
  - `printTempSol` la classe `BSSolver` stamperà in files appositi la struttura delle soluzioni che troverà ogni x iterazioni, dove x è indicato proprio da questo parametro.

- `virtual void SetDimension ( int numTerminus, int numTrips ) [pure virtual]`

  Il metodo permette di settare le "dimensioni del problema":

  Parameters:

  - `numTerminus` il numero di capolinea
  - `numTrips` il numero di corse

- `virtual void SetTimeBand ( TimeBand band, Time startDay, Time endDay ) [pure virtual]`

  Il metodo permette di settare la partizione generale dell'arco temporale in fasce orarie, nonché gli orari di inizio/fine giornata.

  Parlando di una partizione "generale", essa deve poter comprendere tutte le possibili fasce orarie utili; quindi essa sarà l'intersezione di tutte le particolari partizioni in fasce orarie presenti nell'input del problema. Ad esempio, se ci fossero due diverse fasce orarie:

  00:00 - 12:00 - 17:00 - 23:59

  00:00 - 08:00 - 12:00 - 18:00 - 23:59

  l'intersezione rappresentata in band sarebbe

  00:00 - 08:00 - 12:00 - 17:00 - 18:00 - 23:59.

- `virtual void SetEvalBus ( vector < Time > times, vector < int > numOfBus ) [pure virtual]`

  Il metodo permette di inserire, se esiste, la stima (effettuata a priori) del numero di vetture necessarie per effattuare il servizio in diversi periodi che suddividono la giornata.

  L' elemento i-esimo del vettore times, passato per parametro, determina l'istante temporale che conclude l'i-esimo periodo nel quale è suddivisa la giornata, mentre il corripondente elemento del vettore `numOfBus` determina la stima di vetture ritenute necessarie per quel periodo.

- `virtual void AddTerminus ( IndexNodes node, string name, TimeBand minStop, TimeBand maxStop, TimeBand timeFromDepot, TimeBand timeToDepot ) [pure virtual]`

  Il metodo permette di aggiungere i capolinea, ovvero quei nodi che sono partenza o arrivo per un determinato sottoinsieme di corse.

  Parameters:

  - `node` è l' identificativo del nodo in questione
  - `name` è il nome del nodo in questione
  - `minStop, maxStop` vettori che indicano rispettivamente, per ogni fascia oraria, il tempo di sosta minima/massima che ogni bus deve rispettare al capolinea. I vettori hanno la stessa dimensione del vettore delle fasce orarie, si veda `SetTimeBand()`, e l'i-esimo elemento del ettore delle fasce orarie è in relazione con gli i-esimi elementi dei vettori `minStop` e `maxStop`).
  - `timeFromDepot, timeToDepot` vettori che indicano rispettivamente, per ogni fascia oraria, il tempo di percorrenza dal deposito al capolinea e viceversa. Anche questi due vettori vanno letti in concomitanza col vettore delle fasce orarie.

  Se viene aggiunto un capolinea di troppo rispetto a quanto indicato col metodo `SetDimension()`, il metodo lancia
  una `TTDException`.

- `virtual void SetDepot ( string name, TimeBand minStop, TimeBand maxStop ) [pure virtual]`

  Il metodo setta i dati del deposito.

  Parameters:

  - `name` è il nome del nodo in questione
  - `minStop, maxStop` vettori che indicano, per ogni fascia oraria, rispettivamente il tempo di sosta minima/massima che ogni bus deve rispettare al deposito, per ciascuna fascia oraria (i vettori hanno la stessa dimensione del vettore delle fasce orarie, si veda `SetTimeBand()`, e l'i-esimo elemento del vettore delle fasce orarie è in relazione con gli i-esimi elementi dei vettori `minStop` e `maxStop`).

- `virtual void AddTrips ( vector< int > idTrips,`
  `vector< IndexNodes > startTerminus, vector< IndexNodes > endTerminus,`
  `TimeBand departures, TimeBand arrivals,`
  `vector< bool > fixedTrips, vector< bool > linksDepTrips,`
  `vector< bool > linksTripsDep ) [pure virtual]`

  Il metodo aggiunge un insieme di corse al problema.

  Le corse non necessariamente devono essere aggiunte in un solo colpo. Il metodo infatti non pone limiti al numero di chiamate ad esso: l' unico vincolo è di non aggiungere un numero di corse eccessivo rispetto a quanto indicato con `SetDimension()`, altrimenti il metodo lancia una `TTDException`.

  Parameters:

  - `id` vettore che indica gli identificativi delle corse aggiunte (l'i-esimo id appartiene alla i-esima corsa aggiunta, e ciò vale anche per gli altri vettori).
  - `startTerminus, endTerminus` vettori che indicano rispettivamente i capolinea di partenza e di arrivo delle corse aggiunte.
  - `departures, arrivals` vettori che indicano rispettivamente gli orari di partenza e di arrivo per le corse aggiunte.
  - `fixedTrips` vettore che indica quali corse siano prefissate (quindi da inserire forzatamente nella soluzione) e quali no.
  - `linksDepTrips` vettore che indica quali corse abbiano collegato il proprio capolinea di partenza col deposito
  - `linksTripsDep` vettore che indica quali corse abbiamo collegato il proprio capoliena di arrivo col deposito

- `virtual void AddTrips ( vector< int > idTrips,`
  `IndexNodes startTerminus, IndexNodes endTerminus,`
  `vector< Time > departures, vector< Time > arrivals,`
  `vector< bool > fixedTrips, vector< bool > linksDepTrips,`
  `vector< bool > linksTripsDep ) [pure virtual]`

  Il metodo ha il medesimo comportamento dell'omonimo appena sopra; l'unica differenza sta nei parametri `startTerminus` e `endTerminus` dove al posto di vettori di indici abbiamo due indici solamente.

  In pratica tutte le corse aggiunte con una sola chiamata a questa variante del metodo avranno il solito capolinea di partenza e il solito capolinea d'arrivo.

- `virtual void SetMaxNumOfBus (`
  `int maxNumOfBus = Inf< int >() ) [pure virtual]`

  Il metodo permette di settare l'eventuale numero massimo di bus impiegabili durante l'arco della giornata.

- `virtual void SetWishLapTime (`
  `vector< vector< int > > wishLapTime ) [pure virtual]`

  Il metodo permette di settare una matrice che indica, per ogni corsa, l'intertempo desiderato tra esse.

  La matrice ha 2 colonne: la prima indica l'id delle corse in questione, mentre l'ultima colonna indica l'intertempo desiderato.

- `virtual void ChgTempFixedTrips (`
  `vector< int > idFixedTrips, vector< int > idNoFixedTrips ) [pure virtual]`

  Il metodo permette di considerare prefissate temporaneamente un insieme di corse indicato dal vettore di identificativi `idFixedTrips` e, di contro, temporaneamente non fissate altre corse indicate dal vettore di identificativi `idNoFixedTrips`.

  Con questo metodo è possibile fissare una o più corse non già prefissate con `AddTrips()` senza modificare (e quindi compromettere) il reale insieme di corse già prefissate. Il metodo infatti si pone, all'interno del contesto di questa classe, ad un livello intermedio per il fissaggio delle corse. Con `AddTrips()` alcune corse vengono fissate all'inizio dell'elaborazione una volta per tutte, mentre con l'omonimo metodo qui sotto è permesso un fissaggio volatile delle corse, da effettuarsi ad esempio nell'interazione tra le classi `TTSolver` e la classe `BSSolver` durante ogni iterazione. Tra questi due livelli di fissaggio vi è quello implementato da questo metodo, che permette a chi usa esternamente l' oggetto `TTDAlgoInterface` (dove vi sarà un metodo analogo che richiamerà questo) di operare un fissaggio/defissaggio delle corse dall'esterno ma in maniera temporanea.

- `virtual void ChgTempFixedTrips (`
  `vector< int > idFixedTrips ) [pure virtual]`

  Il metodo permette di considerare prefissate temporaneamente un insieme di corse indicato dal vettore di identificativi `idFixedTrips`.

  Con questo metodo è possibile fissare una o più corse non già prefissate con `AddTrips()` senza modificare (e quindi compromettere) il reale insieme di corse già prefissate: basterà infatti chiamare il metodo `ClearTempFixedTrips()` per ripristinare la situazione nel suo stato originale. L' insieme delle corse, formato da quelle prefissate già in input e quelle fissate temporaneamente con questo metodo, concorrerà alla formazione della soluzione, mentre le rimanenti corse ne saranno escluse automaticamente.

- `virtual void SetUsedTrips (`
  `vector< vector< int > > idTrips ) [pure virtual]`

  Il metodo permette di indicare quali corse tra quelle esistenti sono effettivamente usabili, basandoci sul lavoro svolto precedentemente nelle varie istanze delle classi `TTSolver`.

- `virtual void SetTripsCosts (`
  `vector< int > idTrips, vector< double > costs ) [pure virtual]`

  Il metodo permette di cambiare i costi delle corse indicate dal vettore `idTrips` degli identificativi altrimenti, di default, nulli.

- `virtual void SetTripCost ( int idTrip, double cost ) [pure virtual]`

  Il metodo permette di cambiare il costo della corsa indicata da `idTrip` altrimenti, di default, nullo.

- virtual void ClearTripsCosts (void) [pure virtual]

  Il metodo permette di settare a zero i costi di tutte le corse.

- `virtual double GetCostFixedSolution (`
  `vector< vector< int > > tripsInJourneys, vector< int > idJourney,`
  `vector< vector< int > > turniMacchina, double & costTM,`
  `double & costInLine, double & costOutLine,`
  `double & costMargTM, string & exitCode,`
  `double InfValue = Inf< double >() ) [pure virtual]`

  Il metodo restituisce il costo di una soluzione esterna passata tramite: il vettore `tripsInJourneys` (che indica le corse facenti parte della soluzione divise in viaggi), il vettore `idJourney` (dove sono scritti gli id di ogni viagigo), infine il vettore `turniMacchina` (che indica come i viaggi siano distribuiti nei vari turni macchina effettuati).

  Il costo di una soluzione esterna è la sommatoria di tre costi parziali (il costo dei turni macchina, il costo della compatibilità in linea, il costo della compatibilità fuori linea, il costo degli inizio-fine TM) che sarà restituita rispettivamente attraverso i parametri `costTM`, `costInLine`, `costOutLine`, `costMargTM`. Non è detto comunque che tale soluzione sia ammissibile; in tal caso il metodo setterà adeguatamente il parametro `exitCode` e dovrà restituire infinito: il secondo parametro, `InfValue`, permette di stabilire con quale valore simulare l'infinito.

- `virtual BSStatus Solve ( bool tripsWithCostsZero,`
  `int numIteration = 0 ) [pure virtual]`

  Il metodo permette di risolvere il problema.

  Il metodo ritorna un valore del tipo pubblico `TTDStatus` che indica lo stato dell'algoritmo al momento della terminazione:

  - `kOK` significa che è stata determinata una soluzione con l'accuratezza richiesta;
  - `kStopped` significa che l'algoritmo è stato interrotto (es. perché è terminato il tempo massimo richiesto) avendo determinato una soluzione ammissibile, ma non con l'accuratezza desiderata;
  - `kUnSolved` significa che l'algoritmo è stato interrotto (es. perché è terminato il tempo massimo richiesto) senza aver determinato alcuna soluzione ammissibile;

– `kError` significa che l'algoritmo è stato interrotto per via di un qualche errore (numerico, di memoria, ...) che non è stato possibile gestire.

Il parametro `numIteration` segnala l'iterazione corrente. Il parametro `tripsWithCostsZero`, se settato a true, forza il solutore a considerare nulli i costi delle singole corse nel calcolo della f.o.

- `virtual void GetTripsResult (`
  `vector< vector< int > > & idTrips,`
  `vector< vector< bool > > & compatInLinea ) [pure virtual]`

  Il metodo scrive nel vettore `idTrips` passato per riferimento l'insieme delle corse facenti parte dell'ultima soluzione trovata dal solutore.

  Le corse sono divise secondo il bus che le serve. Analogamente il metodo scrive nel vettore compatInlinea se, tra due corse consecutive servite dal solito bus, viene usata la compatibilità in linea o quella non in linea.

- `virtual void GetFoResult ( double & foValue,`
  `double & costTM, double & costCIn,`
  `double & costCOut, double & costMargTM,`
  `bool noTripsCosts = false ) [pure virtual]`

  Il metodo scrive nella variabile `foValue` passata per riferimento il valore della funzione obiettivo dell'ultima soluzione trovata dal solutore; quindi descrive tramite le variabili `costTM`, `costCIn`, `costCOut` come il costo globale della soluzione sia stato ottenuto, riportando nel dettaglio il costo derivato dai turni macchina, il costo derivato dalla compatibilità in linea (soste al capolinea) e dalla compatibilità fuori linea (rientri ed uscite dal deposito), il costo degli inizio-fine TM.

  Se il parametro `noTripsCosts` = true i costi legati alle corse saranno automaticamente esclusi.

- `virtual void WriteFixedSolution (`
  `vector< vector< int > > & arcs ) [pure virtual]`

  Il metodo passa come parametrio un vettore contenente una soluzione calcolata esternamente descritta tramite tutti gli archi dei sottografi interni con flusso non nullo.

12 Appendice B: il file di configurazione

12.1 Gamma Section

- *gammaM*: il costo di ogni bus utilizzato
- *gammaI*: il costo di ogni secondo passato da un bus in sosta ad un capolinea
- *gammaF*: il costo di ogni secondo passato da un bus in viaggio da/verso il deposito
- *gammaS*: il costo di ogni secondo passato da un bus in viaggio da/verso il deposito nella sua prima uscita

12.2 Cost laptime section

- *alfa*: determina una soglia oltre la quale l'errore relativo fra l'intertempo effettivo della corsa t e l'intertempo desiderato d (per la specifica fascia oraria) diventa intollerabile e quindi le due corse non possono essere usate in sequenza.

  In particolare se $\delta = \frac{|t-f|}{f} <= \alpha$ allora l'intertempo della corsa è ammissibile
- Ogni intertempo trovato avrà un costo proporzionale ai secondi di differenza rispetto all'intertempo indicato, indicati con x.

  In particolare il costo sarà $a \cdot x \cdot x + b \cdot x + c$
- Ogni secondo di differenza tra l' inizio della giornata e il primo passaggio al nodo d'osservazione, indicato con x, costerà $a_I \cdot x \cdot x + b_I \cdot x + c_I$
- Ogni secondo di differenza tra la fine della giornata e l'ultimo passaggio al nodo d'osservazione, indicato con x, costerà $a_F \cdot x \cdot x + b_F \cdot x + c_F$

12.3 Par algo section

- *maxTime*: tempo massimo d'esecuzione (in secondi); 0 secondi significa che non c'è limite di tempo
- *maxIter*: massimo numero di iterazioni; 0 significa che non c'è limite

12.4 Scale factor section

- *scaleFactor*: fattore di scala (in secondi)

12.5 Obpoint range section

La sezione seguente setta, per ogni nodo di osservazione indicato dal primo campo di ogni riga, il "range d'ammissibilità iniziale" e il "range d'ammissibilità finale" (rispettivamente secondo e terzo campo di ogni riga), ovvero i periodo temporale massimo (espresso in secondi) di distanza rispetto all'inizio o alla fine della giornata entro i quali la prima/l'ultima corsa possono essere scelti.

12.6 Heuristic section

- *heurValue*: scegli che valori usare nell'euristica: = 0 lambda positivi, = 1 primali BS, = 2 primali TT
- *firstIterBundle*: numero iterazioni del Bundle prima del primo fissaggio
- *parIter1*: parametro n. 1 per la gestione delle iterazioni del Bundle dopo il primo fissaggio. Attualmente rappresenta le iterazioni del Bundle tra un fissaggio e l'altro
- *parIter2*: parametro n. 2 per la gestione delle iterazioni del Bundle dopo il primo fissaggio. Attualmente non usato
- *parIter3*: parametro n. 3 per la gestione delle iterazioni del Bundle dopo il primo fissaggio. Attualmente non usato
- *minFixTrips*: minimo numero di corse da fissare a ogni fissaggio
- *maxFixTrips*: massimo numero di corse da fissare a ogni fissaggio
- *thresholdValue*: soglia (percentuale) sotto la quale non si scelgono le relative corse per il fissaggio
- *percFirstLaptime*: percentuale della prima fascia significativa che determina la lunghezza della prima finestra temporale
- *parWin1*: parametro n. 1 per la la crescita della finestra. Attualmente rappresenta la percentuale della fascia dell'ultima corsa fissata che determina l'allargamento della finestra temporale: in realtà è la percentuale dell'intertempo atteso all'istante di tempo del punto di osservazione per ultima corsa fissata (l'intertempo atteso al tempo di transito sul nodo pilota della corsa fissata "più a destra")
- *parWin2*: parametro n. 2 per la la crescita della finestra. Attualmente non usato
- *parWin3*: parametro n. 3 per la la crescita della finestra. Attualmente non usato
- *percTripRound*: percentuale dell'intertempo desiderato di una corsa fissata che delimita il raggio dell'intorno temporale a se stessa entro il quale si defissano le corse: ma solo se è minore dell'intertempo minimo atteso sul nodo pilota della corsa fissata a quell'istante di tempo
- *percIncrWin*: soglia percentuale per la crescita della finestra: indica quale percentuale di corse all'interno dell'attuale finestra temporale e facenti parte dell'ultima soluzione disponibile deve essere stata fissata dall'euristica per poter permettere l'allargamento della finestra temporale
- *defixZeroTrip*: dopo quanti fissaggi una corsa con valori a 0 viene defissata
- *forcedFix*: dopo quanti fissaggi "a vuoto" si decide di fissare forzatamente una corsa
- *heurLogs*: decide se creare files di log durante l'esecuzione dell'euristica

- *heurValueLogs*: decide se mostrare in un file di log, ad ogni turno di fissaggio, per ogni corsa i valori con i quali l'euristica decide come muoversi
- *heurChooseTripsLogs*: decide se mostrare in un file di log, ad ogni turno di fissaggio, le corse scelte sul momento per essere fissate o "sfissate".
- *heurSummaryLogs*: decide se mostrare in un file di log, ad ogni turno di fissaggio, il riepilogo della situazione

12.7 Screen section

- *printScreenOutput*: se settato a true saranno stampati sullo schermo dei messaggi d'output durante l'esecuzione

12.8 Log section

12.8.1 Show input

- *inputBS*: se settato a t la parte BS riporta nel file di Log l'input che riceve dal livello superiore
- *inputTT*: se settato a t le parti TT riporteranno nei files di Log gli input che riceveranno dal livello superiore

12.8.2 Show graph

- *graphBS*: se settato a t la parte BS riporta nel file di Log la struttura del grafo che sta creando
- *graphTT*: se settato a t le parti TT riporteranno nei files di Log le strutture dei grafi che stanno creando

12.8.3 Print temp solution

- *solBS*: la parte BS stamperà in files appositi la struttura delle soluzioni che troverà ogni x iterazioni, dove x è indicato proprio da questo parametro
- *solTT*: le parti TT stamperanno in files appositi la struttura delle soluzioni che troveranno ogni x iterazioni, dove x è indicato proprio da questo parametro

12.8.4 Show temp situation

- *tempSolBS*: se settato a t l'algoritmo, ogni ... iterazione, riporta nel file di Log del Rilassamento Lagrangiano la soluzione parziale in quel momento trovata dalla parte BS
- *tempSolTT*: se settato a true l'algoritmo, ogni ... iterazione, riporta nel file di Log del Rilassamento Lagrangiano le soluzioni parziali in quel momento trovate dalle parti TT

- *lastSolAmm*: se settato a true l'algoritmo, ogni ... iterazione, riporta nel file di Log del Rilassamento Lagrangiano l' ultima soluzione ammissibile in quel momento trovata
- *bestSolAmm*: se settato a true l'algoritmo, ogni ... iterazione, riporta nel file di Log del Rilassamento Lagrangiano la migliore soluzione ammissibile in quel momento trovata
- *solEvery*: l'algoritmo riporta nel file di Log le soluzioni di cui sopra ogni x iterazioni, dove x è settato da questo parametro
- *foValue*: l'algoritmo riporta nel file di Log tutti i valori "funzione obiettivo" ogni x iterazioni, dove x è settato da questo parametro

12.8.5 Better solution subsection

- *incr sol amm*: se settato a t l'algoritmo riporta nel file di Log ogni soluzione (ammissibile) che migliora la f.o.
- *PrintFileBetterSol*: parametro per regolare le stampe su file formato TTD delle soluzioni ammissibili che migliorano la f.o.:
  - se PrintFileBetterSol = 0 non vi sarà nessuna stampa
  - se PrintFileBetterSol = 1 il file d' output (quello dove eventualmente sarà scritta la soluzione trovata al termine dell'intera computazione) sarà via via sovrascritto
  - se PrintFileBetterSol = 2 ogni soluzione che determina un miglioramento sarà stampata su un singolo file
  - se PrintFileBetterSol = 3 il file d' output (quello dove eventualmente sarà scritta la soluzione trovata al termine dell'intera computazione) sarà via via sovrascritto e ogni soluzione che determina un miglioramento sarà stampata su un singolo file

12.8.6 Show final solution subsection

- *solutionInLog*: se settato a t alla fine dell'esecuzione l'algoritmo riporta nel file di Log principale la soluzione eventualmente trovata.

13 Appendice C: il formato dei files d'input/output

Nelle prossime pagine viene riportato un documento *M.A.I.O.R.* che descrive dettagliatamente la grammatica dei files *TTD*.

Formato I/O

Descrizione formato file I/O TTD4

Id.: StIOTTD-Ver4

1 Introduzione

1.1 Scopo

Questo documento descrive il formato del file di I/O di TTD4. Il formato del file con la soluzione di TTD4 è lo stesso di quello dell'importazione da ROT_DB.

1.2 Riferimenti

[1] [ReqSisMaio00-TTDMTR.Ver1] specifica dei requisiti di TTD, Sabina Pardini
[StIO] standard per i formati degli streams di input-output per i sistemi MAIOR

2 Grammatica del file

2.1 Introduzione

Prima di descrivere il formato, descriviamo brevemente le informazioni scritte nel file. Tali informazioni saranno presentate in forma aggregata, limitando duplicazione di informazioni.

1) informazioni sulla transazione
2) lista informazioni sull'importazione: (*LivServ*, *Linea*, *Validità*), dove:
   LivServ ∈ **ListaLivServ**, *Linea*∈ **ListaLinee**, *Validità*∈ **ListaValidità**,
   ListaLivServ, **ListaLinee**, **ListaValidità** parametri di importazione
3) lista validità definite nel data base
4) nodi di supporto definiti su **ListaLinee**
5) archi di supporto definiti su **ListaLinee**: archi in linea ed archi fuori linea (questi ultimi con i tempi di percorrenza per fasce a loro associati)

Per ogni linea di **ListaLinee**:

6) nodi di linea
7) archi di linea
8) nodi di percorso
9) percorsi

Per ogni linea di **ListaLinee** e livello di servizio di **ListaLivServ**:
10) tempi di percorrenza sui nodi di linea
11) tempi di frequenza per fasce sui nodi di percorso
12) tempi di sosta per fasce sui nodi di percorso di tipo capolinea
13) viaggi (a cui è associata la validità) e corse dei viaggi
14) transiti prefissati
15) note di linea

2.2 TTD4->ROT_DB

Poiché in TTD è possibile effettuare operazioni di creazione, cancellazione, modifica di dati non di rete memorizzati in ROT_DB (vedi [1] per sapere esattamente cosa si può modificare), è utile associare a tali elementi uno *stato*, che può assumere i seguenti valori:

- **DaDB**: stato iniziale associato a tutti i dati importati dal data base hanno questo stato
- **ModDB**: i dati sono stati modificati nella loro completezza.
- **ModSoloAttr:** in questo caso il dato e' stato modificato solo nei suoi attributi e non nei dati riferiti per esempio modifico il codice di una corsa ma non gli orari della corsa.
- **Nuovo**: il dato e' stato creato nel TTD.

Osserviamo che i dati cancellati nel TTD non assumono uno stato particolare, ma ne e' tenuta traccia in apposite strutture dati.

I dati che possono essere cancellati sono:
- corse
- viaggi
- note di linea

2.3 Formato del file

Convenzioni adottate:

NL	carattere speciale nuova riga
TAB	carattere speciale tabulatore
Arg1 | *Arg2*	*Arg1* oppure *Arg2*
<*argomento*>*	*argomento* può essere ripetuto 0 o più volte
<*argomento*>n	*argomento* è ripetuto esattamente n volte
<*argomento*>+	*argomento* può essere ripetuto 1 o più volte
[*argomento*]	*argomento* è un parametro opzionale
InCorsivo	simboli non terminali
InGrassetto	simboli terminali
SOTTOLINEATO	tipo di dato
(Arg1.Arg2)	NomeTabella.Nomecampo
Campi nulli	**NULL**
Separatore tra campi	tabulatore

FileI/OTTD::=
Intestazione **NL** *SezioneTransazioni* **NL** *SezioneConfigurazione* **NL** *SezioneValidità* **NL** *SezioneTipiVeicolo* ***NL*** *SezioneAziendeAppaltatrici* ***NL*** *SezioneNodiSupporto* **NL** *SezioneArchiSupporto* **NL** *SezioneServizi* **NL** *SezioneDatiPerCorse*

2.3.1 Intestazione

Intestazione::=#{ **START_FILE** **NL** *IdentificativoFile* #}

IdentificativoFile::= { *Versione* **TAB** **TTD4** **TAB** **InputTTD4** **TAB** *Azienda* **TAB** **NL**}

Versione ::= 2.0.12

Azienda ::= VARCHAR2(25) (campo Azienda della tabella Configurazione di RotDB)

2.3.2 SezioneTransazioni

SezioneTransazioni ::= #{**TRANSAZIONI** **NL** *Transazione*#}**END_TRANSAZIONI**

Transazione ::= #{**TRANSAZIONE** **NL** *IdeTransazione* **TAB** *Periodo* **TAB** *Linea* **TAB** *CodLivello* **TAB** *CodPercorrenza* **TAB** *Login* **TAB** *Anno* **TAB** *Mese* **TAB** *Giorno* **TAB** *Ore* **TAB** *Minuti* **TAB** *Secondi* **NL** #}**END_TRANSAZIONE**

IdeTransazione≥1 (campo per l'importazione da RotDB, -1 altrimenti), è l'identificativo della transazione .

Periodo::= VARCHAR2(10)

Linea::= VARCHAR2(5)

CodLivello::= CHAR(2) (codice livello di servizio)

CodPercorrenza::= CHAR(2) (codice livello di percorrenza)

Login::=VARCHAR2(20) (login dell'utente registrata in RotDB)

Anno::= ≥2001 (in cui è stata aperta la transazione)

Mese::= [1-12](in cui è stata aperta la transazione)

Giorno::= [1-31](in cui è stata aperta la transazione)

Ore::= [1-24](in cui è stata aperta la transazione)

Minuti::= [1-60](in cui è stata aperta la transazione)

Secondi::= [1-60](in cui è stata aperta la transazione)

2.3.3 SezioneConfigurazione

SezioneConfigurazione::= #{**LISTACONFIGURAZIONE** **NL** *ListaConfigurazioneRotDB* **NL** *ListaParametri* #} **END_LISTACONFIGURAZIONE**

ListaConfigurazioneRotDB ::= Azienda **TAB** *NumIntPercorsi* **TAB** *SosteTransiti* **TAB** *CheckSostaGarantita* **TAB** *SostaGarantita*

Azienda ::= VARCHAR2(25) (campo Azienda della tabella Configurazione di RotDB)

NumIntPercorsi::= CHAR(1) (S/N) (se l'Azienda usa la numerazione interna dei percorsi)

SosteTransiti ::= CHAR(1) (S/N) (se l'Azienda usa le soste sui transiti)

CheckSostaGarantita ::= CHAR(1) (S/N) (se è abilitato il controllo sulla sosta garantita = tra due corse almeno *x* minuti di sosta)

SostaGarantita::= NUMBER (valore della sosta garantita)

ListaParametri ::= <tipoParametro **TAB** *descrizioneTipo* **TAB** *parametro* **TAB***descrizione***NL***>**

tipoParametro ::= 1..n (1= numerazione viaggi, 2= Soste, 3= numerazione corse)

descrizioneTipo ::= VARCHAR2(40)

parametro ::= 1..n (il parametro del tipo parametro, ad esempio il tipo di numerazione per le corse o per i viaggi)

descrizione ::= VARCHAR2(40)

2.3.4 SezioneColorazioni

*SezioneColorazioni::= #{**LISTACOLORAZIONI NL** <Colorazione **NL**>***** #}*
END_LISTACOLORAZIONI

Colorazione ::= Nome TAB *DaControllare* TAB *Priorita* TAB *Colore*

Nome ::= VARCHAR2(50) (nome criterio colorazione)

DaControllare ::= CHAR(1) (S/N)

Priorita ::= NUMBER (priorità colorazione)

Colore ::= NUMBER (colore RGB)

2.3.5 SezioneValidità

(lista vuota in caso di importazione da MTDB)

SezioneListaValidità::= #{ **LISTAVALIDITA** *SezioneValidita* #}**END_VALIDITA**

SezioneValidita::= <*StatoValidita* **TAB** *CodValidita* **TAB** *Descrizione* **TAB** *BasataSu* **TAB** *DataInizio* **TAB** *DataFine* **NL** >*

Stato::= {**DaDB**(=importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB**(modificata nel TTD)}

CodValidita::= VARCHAR2(10)

Descrizione::= VARCHAR2(60)

BasataSu::= VARCHAR2(10)

DataInizio::= DATE

DataFine::= DATE

2.3.6 SezioneTipiVeicolo

SezioneTipiVeicolo::= #{ **TIPIVEICOLO NL** *SezioneTipoVeicolo* #}**ENDTAB TIPIVEICOLO**

SezioneTipoVeicolo::= <*CodTipoVeicolo* **TAB** *Lunghezza* **TAB** *Capacità* **TAB** *AutonomiaOre* **TAB** *AutonomiaKm* **TAB** *Descrizione* **TAB** *Tipologia* **TAB** *Controllori* **TAB** *TempoRifornimento* **NL** >*

CodTipoVeicolo::= VARCHAR2(10)

Lunghezza::= NUMBER

Capacità::= NUMBER

AutonomiaOre::= NUMBER

AutonomiaKm::= NUMBER

Descrizione::= VARCHAR2(40)

Tipologia ::= CHAR2(1) (= S/A tipo del veicolo: semplice o articolato)

Controllori ::= NUMBER(numero di controllori per il veicolo)

TempoRifornimento::= NUMBER(secondi , tempo per effettuare il rifornimento)

2.3.7 SezioneAziendeAppaltatrici

SezioneAziendeAppaltatrici ::= #{**AZIENDEAPP NL** *SezioneAziendaAppaltatrice* #}**END_NODISUPPORTO**

SezioneAziendaAppaltatrice ::= <*CodiceAziendaApp* **TAB** *Descrizione* **NL >***

CodiceAziendaApp ::= VARCHAR2(8)

Descrizione ::= VARCHAR2(8)

2.3.8 SezioneNodiSupporto

SezioneNodiSupporto::= #{**NODISUPPORTONL** *SezioneNodoSupporto* #}**END_NODISUPPORTO**

SezioneNodoSupporto::= <#{**NODOSUPPORTO NL** *CodiceNodoSupporto* **TAB** *Nome* **TAB** *TipoNodoSupporto* **TAB** *Cambio* **TAB** *Localita* **TAB** *CodiceSAF* **TAB** *CoordX* **TAB** *CoordY* **TAB** *PosX* **TAB** *PosY* **TAB** *ComuneAppartenenza* **TAB** *GruppoAppartenenza* **TAB** *ZonaApp* **TAB** *SostaMin* **TAB** *SostaMax* **TAB** *Distributore* **NL**
[#{**LISTADISPONIBILITA** *TipoVeicolo* **TAB** *ConApparato* **TAB** *Quantità* **TAB** *Capacita* **TAB** *Pernottamento* **TAB** *MinDeposito* **TAB** *MaxDeposito* **NL** #}**END_LISTADISPONIBILITA**] **NL**

#}**END_NODOSUPPORTO**>*

CodiceNodoSupporto::= VARCHAR2(8)

Localita::= VARCHAR2(15)

TipoNodoSupporto::= {'LI', 'FL', 'DE', 'PS'}

Cambio::= CHAR(1)

CodiceSAF::= VARCHAR2(7)

CoordX::= NUMBER

CoordY:= NUMBER

PosX::= NUMBER

PosY:= NUMBER

ComuneApp::= VARCHAR2(15)

GruppoApp::= VARCHAR2(10)

ZonaApp::= VARCHAR2(10)

SostaMin::= NUMBER

SostaMax::= NUMBER

Nome::= VARCHAR2(50)

Distributore::= CHAR(1) (se sul nodo è possibile eseguire rifornimento)
I seguenti campi sono significativi solo se *TipoNodoSupporto* ='DE'

TipoVeicolo::= VARCHAR2(10)

ConApparato::= NUMBER

Quantità::= NUMBER

Capacita::= NUMBER

Pernottamento::= CHAR2(1)

MinDeposito::= NUMBER

MaxDeposito::= NUMBER

2.3.9 SezioneArchiSupporto

SezioneArchiSupporto::= *SezioneArchiDiLinea* **NL***SezioneArchiFuoriLinea*

2.3.10SezioneArchiDiLinea

SezioneArchiDiLinea::= #{**ARCHIDILINEA** **NL** *SezioneArchiDiLinea* #}**END_ARCHIDILINEA**

SezioneArchiDiLinea::= <#{**ARCODILINEA** *CodiceNodoSupporto1* **TAB** *CodiceNodoSupporto2* **TAB** *Lunghezza* **TAB** *PercentualeAppComune* **TAB** *VeicoloMaxIngombro* **TAB** *Descrizione* ***TAB*** *BinarioUnico* **NL** #}**END_ARCODILINEA** >*

CodiceNodoSupporto1::= VARCHAR2(8)

CodiceNodoSupporto2::= VARCHAR2(8)

Lunghezza::= NUMBER (lunghezza in centimetri)

PercentualeAppComune::= NUMBER (campo per l'importazione da RotDB, -1 altrimenti)

VeicoloMaxIngombro::= VARCHAR2(10) (campo per l'importazione da RotDB, NULL altrimenti)

Descrizione::= VARCHAR2(50) (campo per l'importazione da RotDB, NULL altrimenti)

BinarioUnico ::= CHAR(1) (S/N)

2.3.11SezioneArchiFuoriLinea

SezioneArchiFuoriLinea::= #{**ARCHIFUORILINEANL** <*SezioneArcoFuoriLinea* **NL**>*#}**END_ARCHIFUORILINEA**

SezioneArcoFuoriLinea::= #{**ARCOFUORILINEA** *InformazioniArcoFuoriLinea* **NL** *SezioneTempiPercorrenzaArchiFuoriLineaPerFasce* **NL** #}**END_ARCOFUORILINEA**

InformazioniArcoFuoriLinea::= *CodiceNodoSupporto1* **TAB** *CodiceNodoSupporto2* ***TAB*** *Ordinale* ***TAB*** *Lunghezza* **TAB** *VeicoloMaxIngombro* **TAB** *NoteInstradamento*

CodiceNodoSupporto1::= VARCHAR2(8)

CodiceNodoSupporto2::= VARCHAR2(8)

Ordinale::= NUMBER

Lunghezza::= NUMBER (lunghezza in centimetri)

VeicoloMaxIngombro::= VARCHAR2(10) (campo per l'importazione da RotDB, NULL altrimenti)

NoteInstradamento::= VARCHAR2(255) (campo per l'importazione da RotDB, NULL altrimenti)

SezioneTempiPercorrenzaArchiFuoriLineaPerFasce::= #{ **TEMPIPERCORRENZAPERFASCE** **NL** *TempoPercorrenzaPerFasce* #}**END** **TAB** **TEMPIPERCORRENZAPERFASCE**

TempoPercorrenzaPerFasce::=< *FineFascia* **TAB** *TempoPercorrenzaMin* **TAB** *TempoPercorrenzaMax* **TAB** *LivPercorrenza* **NL** >*

FineFascia::= NUMBER (fascia oraria in secondi)

TempoPercorrenzaMin::= NUMBER

TempoPercorrenzaMax::= NUMBER

LivPercorrenza::= CHAR(2)

2.3.12SezioneStrutturaServizi

SezioneStrutturaServizi::= #{**LISTASERVIZI NL** <*SezioneStrutturaServizio* **NL** >*#}**END_LISTASERVIZI**

SezioneStrutturaServizio::= #{**SERVIZIO** *InformazioniServizio* **NL** *InformazioniLineaServizio* **NL** *SezioneFasceIntensificazione* **NL** *SezioneCoefficientiLinea* **NL** *SezioneValiditaDiLinea* **NL** *SezioneValLineaInUdp* **NL** *SezioneDepositi* **NL** *SezioneRifornimenti* **NL** *SezioneVetturePerFasce* **NL** *SezioneNodiInLinea* **NL** *SezioneArchiInLinea* **NL** *SezioneNodiInSequenzaPrimaria***NL** *SezionePercorsi* **NL** *SezioneNoteDiLinea* **NL** *SezioneDatiPerCorse* **NL** *SezioneCancelati* **NL** *SezioneTurni* #}**END_SERVIZIO**

2.3.12.1 InformazioniServizio

InformazioniServizio::= *Linea* **TAB** *CodLivello* **TAB** *CodiceOrario* **TAB** *VersioneOraria* **TAB** *NumeroCapienza* **TAB** *Descrizione* **TAB** *LivPercorrenza* **TAB** *VeicoloAFrequenza* **TAB** *InizioPasto* **TAB** *FinePasto* **TAB** *LivPercorrenzaAFL* **TAB** *Rifornimenti* **TAB** *MinSostaIntensif MaxSostaIntensif* **TAB** *MinPercentSosta* **TAB** *MaxPercentSosta* **TAB** *PercentCapPrinc* **TAB** *PercentCapSec*

Linea::= VARCHAR2(5)

CodLivello::= CHAR(2) (codice livello di servizio)

CodiceOrario::= VARCHAR2(5)

CodiceOrario::= CHAR(2) (campo per l'importazione da MTDB, NULL altrimenti)

VersioneOraria::= SHORT (campo per l'importazione da MTDB, -1 altrimenti)

NumeroCapienza::= VARCHAR2(40)

Descrizione::= VARCHAR2(40) (NULL, se importo da MTDB)

LivPercorrenza::= CHAR(2)

VeicoloAFrequenza::= VARCHAR2(10) (codice del tipo di veicolo da usare come veicolo a frequenza)

InizioPasto::= NUMBER

FinePasto::= NUMBER

LivPercorrenzaAFL::= CHAR(2) Livello di percorrenza selezionato per gli archi fuori linea

Rifornimenti ::= NUMBER (BPTC: numero di rifornimenti giornalieri richiesti dalla linea per ogni suo mezzo – turno macchina)

MinSostaIntensif ::= NUMBER (BPTC: sosta minima tra corse nelle fasce di intensificazione)

MaxSostaIntensif ::= NUMBER (BPTC: sosta massima tra corse nelle fasce di intensificazione)

MinPercentSosta ::= NUMBER (BPTC: percentuale sosta minima tra corse fuori delle fasce di intensificazione)

MaxPercentSosta ::= NUMBER (BPTC: percentuale sosta massima tra corse fuori delle fasce di intensificazione)

PercentCapPrinc ::= NUMBER (BPTC: percentuale della sosta tra corse da attribuire al capolinea principale fuori dalle fasce di intensificazione)

PercentCapSec::= NUMBER (BPTC: percentuale della sosta tra corse da attribuire al capolinea secondario fuori dalle fasce di intensificazione)

2.3.12.2 InformazioniLineaServizio

InformazioniLineaServizio::= #{**SEZIONEINFOLINEA NL** *Linea* **TAB** *NomeEsteso* **TAB** *NomeAscendente* **TAB** *NomeDiscendente* **NL** *Tipo* **TAB** *Bacino* **TAB** *Zona* **TAB** *CodiceInterno* **TAB** *Copertura* **TAB** *Disagiata* **TAB** *Isolata* **TAB** *SubConcessione* **TAB** *NumeroConcessione* **TAB** *EnteConcedente* **TAB** *Contributo* **TAB** *NormativaCEE* **TAB** *Biglietteria* **TAB** *Navetta* **TAB** *Monitorata* ***TAB*** *APrenotazione* **NL** #}**END_ SEZIONEINFOLINEA**

Linea::= VARCHAR2(5)

NomeEsteso::= VARCHAR2(120) (NULL se importazione da MTDB)

NomeAscendente::= VARCHAR2(120) (NULL se importazione da MTDB)

NomeDiscendente::= VARCHAR2(120) (NULL se importazione da MTDB)

Tipo::= NUMBER(-1 se importazione da MTDB)

Bacino::= NUMBER(-1 se importazione da MTDB)

Zona::= VARCHAR2(10) (NULL se importazione da MTDB)

CodiceInterno::= NUMBER(-1 se importazione da MTDB)

Copertura::= NUMBER(-1 se importazione da MTDB)

Disagiata::= CHAR(1) (NULL se importazione da MTDB)

Isolata::= CHAR(1) (NULL se importazione da MTDB)

SubConcessione::= CHAR(1)

NumeroConcessione::= VARCHAR2(15) (NULL se importazione da MTDB)

EnteConcedente::= NUMBER(-1 se importazione da MTDB)

Contributo::= CHAR(1) (NULL se importazione da MTDB)

NormativaCEE::= CHAR(1) (NULL se importazione da MTDB)

Biglietteria::= CHAR(1) (NULL se importazione da MTDB)

Navetta::= CHAR(1) (NULL se importazione da MTDB)

Monitorata::= CHAR(1) (NULL se importazione da MTDB)

APrenotazione ::= NUMBER (0 se non è a prenotazione, 1..9 se ha corse a prenotazione)

2.3.12.3 SezioneFasceIntensificazione

SezioneFasceIntensificazione ::=#{**LISTAFASCEINTENSIF NL** *FasciaIntensificazione* #}**END_LISTACOEFFICIENTIRIEMP NL**

*FasciaIntensificazione::=< InizioFascia **TAB** FineFascia **NL** >**

InizioFascia ::= NUMBER

FineFascia ::= NUMBER

2.3.12.4 SezioneCoefficientiLinea

SezioneCoefficientiLinea ::=#{**LISTACOEFFICIENTIRIEMP NL** *CoefficienteRiempimento* #}**END_LISTACOEFFICIENTIRIEMP NL**

*CoefficienteRiempimento::=< FineFascia **TAB** Coefficiente **NL** >**

FineFascia ::= NUMBER

Coefficiente ::= NUMBER (BPTC: percentuale riempimento autobus per calcolo frequenza da frequentazione)

2.3.12.5 SezioneValiditaDiLinea

SezioneValiditaDiLinea::= #{**LISTAVALIDITALINEA NL** *SezioneUnaValLinea* #}**END_ LISTAVALIDITALINEA**

SezioneUnaValLinea::= <*Stato* **TAB** *Codice* **TAB** *Descrizione* **TAB** *Validità* **TAB** *Linea* **TAB** *DataInizio* **TAB** *DataFine* **TAB** *Padre* **TAB** *Versione* **TAB** *VisualizzaCorse* **TAB** *IdColore* **NL** >*

Stato::= {**DaDB**(=importato dal database), **ModSoloAttr**(modificato nei suoi attributi nel TTD)}

Codice::= NUMBER

Descrizione::=VARCHAR2(120)

Validità::= VARCHAR2(10)

Linea::= VARCHAR2(5)

DataInizio::= DATE→VARCHAR2(10)→ DD/MM/YYYY (in TTD le date sono gestite come stringhe)

DataFine::= DATE→VARCHAR2(10)→ DD/MM/YYYY

Padre::= NUMBER

Versione::= NUMBER

VisualizzaCorse::= CHAR(1) (S/N)

IdColore::= VARCHAR2(11) (aaa.bbb.ccc)

2.3.12.6 SezioneValLineaInUdp

SezioneValLineaInUdp::= #{**LISTAVALLINEAUDP NL** *SezioneUnaValLineaInUdp* #}**END_LISTAVALLINEAUDP**

SezioneUnaValLineaInUdp::= <*Stato* **TAB** *CodiceVLL* **TAB** *NumUdp* **TAB** *ListaCodUdp* **NL** >*

ListaCodUdp::= <CodUdp **TAB**>*

Stato::= {**DaDB**(=importato dal database), **ModSoloAttr**(modificato nei suoi attributi nel TTD)}

CodiceVLL::= NUMBER

NumUdp::=NUMBER

ListaCodUdp::= lista codici delle udp che hanno la validità di linea

CodUdp::=NUMBER

2.3.12.7 SezioneDepositi

SezioneDepositi::= #{**LISTADEPOSITI NL** *Nodo* **TAB** *Principale* ***NL*** #}**END_LISTADEPOSITI**

Nodo::= VARCHAR2(8)

Principale::= CHAR(1) (P=Principale, S=Secondario, N=Terziario)

2.3.12.8 SezioneRifornimenti

SezioneRifornimenti := #{**LISTARIFORNIMENTI NL** *Nodo* ***NL*** #}**END_ LISTARIFORNIMENTI**

Nodo::= VARCHAR2(8)

2.3.12.9 SezioneVetturePerFasce

SezioneDepositi::= #{**LISTAVETTUREPERFASCE NL** *FineFascia* **TAB** *NumeroVetture* #}**END_ LISTAVETTUREPERFASCE**

FineFascia::= NUMBER (fascia oraria in secondi)

NumeroVetture::= NUMBER

2.3.12.10 SezioneNodiInLinea

SezioneNodiInLinea::= #{**NODIINLINEA NL** *SezioneNodoInLinea* #}**END_NODIINLINEA**

SezioneNodoInLinea::= <#{**NODODILINEA NL** *CodiceNodoSupporto* **TAB** *NumeroVisualizzazione* **TAB** *PosX* **TAB** *PosY* **TAB** *Cambio* ***TAB*** *Pilota* ***TAB*** *VersoPilota* ***TAB*** *SostaMax* ***NL*** *SezioneFrequenzePerFasce* ***NL*** *SezioneTempiSostaCapolinea* ***NL*** *#}**END_NODODILINEA***>*

CodiceNodoSupporto::= VARCHAR2(8)

Stato::= {**DaDB**(=importato dal database), **ModSoloAttr**(modificato nei suoi attributi nel TTD)}

NumeroVisualizzazione::= NUMBER(-1 se importazione da MTDB)

PosX::= NUMBER(-1 se importazione da MTDB)

PosY::= NUMBER(-1 se importazione da MTDB)

Cambio::= CHAR(1)

Pilota::= CHAR(1) (S/N, S se il nodo è pilota per la linea-livello di servizio. È ammesso un solo nodo pilota per verso)

VersoPilota::= CHAR(4) (verso del nodo pilota, è letto solo se *Pilota*=S e un solo nodo pilota per verso) : As, Di,AsDi

SostaMax ::= NUMBER (BPTC: sosta massima sul nodo di linea: per ora è gestita come “sosta ammessa” e può assumere valori 0 e 1)

SezioneFrequenzePerFasce::= #{**FREQUENZEPERFASCELINEA NL** *SezioneFrequenzaPerFasce* #}**END_FREQUENZEPERFASCELINEA**

SezioneFrequenzaPerFasce::=< *Tipo* **TAB** *Verso* **TAB** *FineFascia* **TAB** *FrequenzaMin* **TAB** *FrequenzaMedia* **TAB** *FrequenzaMax* **NL** >*

Sono le frequenze e le frequentazioni del nodo pilota o degli altri nodi della linea (spalmate dalla frequenza del nodo pilota per ogni verso e fascia)

Tipo ::= CHAR(1) (se è una frequenza o una frequentazione di passeggeri)

Verso ::= CHAR(4) (verso del nodo per le frequenze della fascia)

FineFascia::= NUMBER

FrequenzaMin::= NUMBER

FrequenzaMedia::= NUMBER

FrequenzaMax::= NUMBER

SezioneTempiSostaCapolinea ::= **#{TEMPISOSTAPERFASCELINEA NL** *SezioneTempiSosta* **#} END_TEMPISOSTAPERFASCELINEA**

Sono i tempi di sosta su capolinea definiti nella tabella tempisostacapolinea

SezioneTempiSosta::=< FineFascia **TAB** *SostaMin* **TAB** *SostaMax* **NL** *>**

2.3.12.11 SezioneArchiInLinea

SezioneArchiInLinea::=#{**ARCHIINLINEA NL** <*SezioneArcoInLinea* **NL**>* #}**ENDTAB ARCHIINLINEA**

SezioneArcoInLinea::=#{**ARCOINLINEA NL** *DefinizioneArcoInLinea* **NL** *SezioneTempiPercorrenzaPerFasce* #}**END_ARCOINLINEA**

DefinizioneArcoInLinea::= *CodiceNodoSupporto1* **TAB** *CodiceNodoSupporto2*

CodiceNodoSupporto1::= VARCHAR2(8)

CodiceNodoSupporto2::= VARCHAR2(8)

SezioneTempiPercorrenzaPerFasce::= #{**TEMPIPERCORRENZAPERFASCE NL***SezioneTempiPercorrenza* #}**END_TEMPIPERCORRENZAPERFASCE**

SezioneTempiPercorrenza::=< *FineFascia* **TAB** *TempoPercorrenzaMin* **TAB** *TempoPercorrenzaMax* ***TAB*** *LivPercorrenza* ***TAB*** *TempoSostaArrivo***NL**>*

FineFascia::= NUMBER (fascia oraria in secondi)

TempoPercorrenzaMin::= NUMBER

TempoPercorrenzaMax::= NUMBER

LivPercorrenza::= CHAR(2)

TempoSostaArrivo::= NUMBER

2.3.12.12 SezioneNodiInSequenzaPrimaria

SezioneNodiInSequenzaPrimaria ::=<#{ **NODIINSEQUENZAPRIMARIA NL** <*SezioneNodiSeq* **NL**>2 #}**END_NODIINSEQUENZAPRIMARIA**>

SezioneNodiSeq::= <*Verso* **TAB** *Stato* **NL** *CodiceNodoSupporto* **TAB** *OrdineVisualizzazione* **NL**>*

Verso::= CHAR(2)

Stato::= {**DaDB**(=importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB**(modificata nel TTD)}

CodiceNodoSupporto::= VARCHAR2(8)

OrdineVisualizzazione::= NUMBER

2.3.12.13 SezionePercorsi

SezioneStrutturaPercorsi::= #{ **PERCORSI** **NL** <*SezioneStrutturaPercorso* **NL** >*#}**END_PERCORSI**

SezioneStrutturaPercorso::= #{**PERCORSO** **NL** *InformazioniPercorso* **NL** *SezioneVeicoliCompatibili* ***NL*** *SezioneNodiDiPercorso* ***NL*** *SezioneArchiDiPercorso* **NL** #}**END_PERCORSO**

InformazioniPercorso::= *StatoPercorso* **NL** *CodicePercorso* **TAB** *Descrizione* **TAB** *Verso* **TAB** *CodiceElettronico* **TAB** *Carteggio* **TAB** *LetteraProvenienza* **TAB** *LetteraDestinazione* **TAB** *NumeroConcessione* **TAB** *EnteConcedente* **TAB** *usaNumInterna* **TAB** *NumerazioneInterna* **TAB** *Barrato* **TAB** *Base* **TAB** *IndiceProvenienza* **TAB** *Metri* **TAB** ~~*CapolineaCapolinea* **TAB**~~ *LunghVeicMaxIng* ***TAB*** *Categoria* **NL**

StatoPercorso::= **DaDB**(=importato dal database)

CodicePercorso::= VARCHAR2(6)

Descrizione::= VARCHAR2(40)

Verso::= CHAR(2)

CodiceElettronico::= NUMBER

Carteggio::= VARCHAR2(3)

LetteraProvenienza::= CHAR(1)

LetteraDestinazione::= CHAR(1)

NumeroConcessione::= VARCHAR2(15)

EnteConcedente::= NUMBER

usaNumInterna ::= CHAR(1) (S/N) (se l'Azienda usa la numerazione interna dei percorsi)

NumerazioneInterna::= NUMBER

Barrato::= CHAR(1)

Base::= CHAR(1)

IndiceProvenienza::= NUMBER

Metri:: = NUMBER

~~*CapolineaCapolinea::=CHAR(1)*~~

LunghVeicMaxIng::= NUMBER

Categoria::= NUMBER (categoria del percorso, per BPTC il tipo : Full, Partial, Express)

2.3.12.14 SezioneVeicoliCompatibili

SezioneVeicoliCompatibili::= #{**VEICOLICOMPATPERC** **NL** <*VeicoloCompatibile* **NL** >* #}**END_ VEICOLICOMPATPERC**

VeicoloCompatibile::= CHAR(10)

2.3.12.15 SezioneNodiDiPercorso

*SezioneNodiDiPercors*o::= #{**NODIDIPERCORSONL** <*SezioneNodoDiPercorso* **NL** >* #}**END_NODIDIPERCORSO**

SezioneNodoDiPercorso::= #{**NODODIPERCORSONL** *NodoDiLinea* **TAB** *NumOrdineinPercorso* **TAB** *Cambio* **TAB** *PosXAsc* **TAB** *PosYAsc* **TAB** *PosXDisc* **TAB** *PosYDisc* **TAB** *TipoNodo* **NL** *SezioneFrequenzePerFasce* **NL** *SezioneTempiSostaPerFasce* **NL** #}**END _NODODIPERCORSO**

CodicePercorso::= VARCHAR2(6)

NodoDiLinea::= VARCHAR2(8)

NumOrdineinPercorso::= NUMBER

NumVisualizzazione::=SHORT

Cambio::= CHAR(1)

PosXAsc::=NUMBER (solo se non importo da ROT_DB)

PosYAsc::= NUMBER (solo se non importo da ROT_DB)

PosXDisc::=NUMBER (solo se non importo da ROT_DB)

PosYDisc::= NUMBER (solo se non importo da ROT_DB)

TipoNodo::= VARCHAR2(8) (solo se non importo da ROT_DB: {N, Ins, Dis, L, Cap})

SezioneFrequenzePerFasce::= #{**FREQUENZEPERFASCE** **NL** *SezioneFrequenzaPerFasce* #}**END_FREQUENZEPERFASCE** (se il nodo è “pilota” e solo se RotDB)

SezioneFrequenzaPerFasce::=< *FineFascia* **TAB** *Frequenza* **NL** >*

FineFascia::= NUMBER

Frequenza::= NUMBER

SezioneTempiSostaPerFasce::= #{**TEMPISOSTAPERFASCE** **NL** *SezioneTempoSostaPerFasce* #}**END_TEMPISOSTAPERFASCE** (tempo di sosta definito sui nodi di tipo capolinea)

SezioneTempoSostaPerFasce::=<*CodLivello***TAB** *FineFascia***TAB** *SostaMin* **TAB***SostaMax* **NL**>*

CodLivello::= CHAR(2) (codice livello di servizio)

FineFascia::= NUMBER

SostaMin::= NUMBER

SostaMax::= NUMBER

2.3.12.16 SezioneArchiDiPercorso

SezioneArchiDiPercorso::= #{**ARCHIDIPERCORSONL** <*SezioneArcoDiPercorso* **NL** >* #}**END_ARCHIDIPERCORSO**

SezioneArcoDiPercorso::= #{**ARCODIPERCORSONL** *NumOrdineinPercorso* **TAB** *NodoOrigineInLinea* **TAB** *NumOrdineinPercorsoNodoOrigine* **TAB** *NodoDestinazioneInLinea* **TAB** *NumOrdineinPercorsoNodoDestinazione* ***TAB*** *Lunghezza* ***TAB*** *Direttrice* **TAB** *Pilota* **TAB** *PilotaInit* **TAB** *PilotaEnd* **NL** *SezioneFrequenzePerFasce* **NL** #}**END _ARCODIPERCORSO**

NumOrdineinPercorso::= NUMBER

NodoOrigineInLinea::= VARCHAR2(8)

NumOrdineinPercorsoNodoOrigine::= NUMBER

NodoDestinazioneInLinea::= VARCHAR2(8)

NumOrdineinPercorsoNodoDestinazione::= NUMBER

Lunghezza::= NUMBER

Direttrice::= CHAR(1) (campo per l'importazione da MTDB)

Pilota::= CHAR(1) (campo per l'importazione da MTDB)

PilotaInit::= NUMBER(solo se l'arco è pilota nel caso d'importazione da MTDB)

PilotaEnd::= NUMBER(solo se l'arco è pilota nel caso d'importazione da MTDB)

NOTA: La sezione sotto è non vuota nel caso dell'importazione da MTDB (in cui le frequenze sono associate all'arco pilota, e non ai nodi del percorso)

SezioneFrequenzePerFasce::= #{**FREQUENZEPERFASCE NL** *SezioneFrequenzaPerFasce* #}**END TAB FREQUENZEPERFASCE** (se l'arco è "pilota")

SezioneFrequenzaPerFasce::=< *CodLivello* **TAB** *FineFascia* **TAB** *FrequenzaMin* **TAB** *FrequenzaMedia* **TAB** *FrequenzaMax* **NL** >*

CodLivello::= CHAR(2) (codice livello di servizio)

NOTA: i valori seguenti coincidono nel caso dell'importazione da ROT_DB

FineFascia::= NUMBER

FrequenzaMin::= NUMBER

FrequenzaMedia::= NUMBER

FrequenzaMax::= NUMBER

2.3.12.17 SezioneDatiPerCorse

SezioneDatiPerCorse::= #{**DATICORSE** *Linea* **TAB** *CodLiv* **NL** *SezioneViaggi* **NL** *SezioneTransitiPrefissati* **NL** *SezioneNoteLinea* **NL** *SezioneCancellati* **NL** #}**END_DATICORSE**

Linea::= VARCHAR2(5)

CodLiv::= CHAR(2) (codice livello di servizio)

2.3.13 SezioneViaggi

SezioneViaggi::= #{**VIAGGI** **NL** <*SezioneViaggio* **NL**>* #}**END_VIAGGI**

2.3.13.1.1 SezioneViaggio

SezioneViaggio::= #{**VIAGGIO** *StatoViaggio* **NL** *CodViaggio* **TAB** *DepositoForzato* **TAB** *VeicoloForzato* **TAB** *Validita* **TAB** *ValiditaDiLinea* **TAB** *MetriFuoriLinea* **TAB** *NumAttivitaCorseTTD* **TAB** *MetriInLinea* **TAB** *TempoSosta* **TAB** *NomeAppaltatore* **TAB** *SubConcessione* **TAB** *Sussidio* **TAB** *SostaIniziale* **TAB** *SostaFinale* ***TAB*** *TipoServizio* ***TAB*** *IndicePerTurno* **NL** *SezioneCoperturaTMeTG* **NL** *SezioneVeicoliCompatibili* **NL** *SezioneAttività* **NL** #}**END_VIAGGIO**

StatoViaggio::= {**DaDB**viaggio importato dal database), **Nuovo** (creato nuovo nel TTD), **ModDB** (modificato nel TTD nelle sue componenti; corse e transiti), **ModSoloAttr** (modificato solo nei suoi attributi nel TTD)}

CodViaggio::= NUMBER

DepositoForzato::= VARCHAR2(8)

VeicoloForzato::= VARCHAR2(10)

Validita::= VARCHAR2(10)

MetriFuoriLinea::= NUMBER

NumAttivitaCorseTTD::= NUMBER

MetriInLinea::= NUMBER

TempoSosta::= NUMBER

NomeAppaltatore::= VARCHAR2(20)

SubConcessione::= CHAR(1)

Sussidio::= CHAR(1)

SostaIniziale::= NUMBER

SostaFinale::= NUMBER

TipoServizio ::= CHAR(1) → NUMBER (da RotDB 1.11.28.H può assumere i valori 0,1,2,3,4,5,6,7,8,9)

IndicePerTurno ::= NUMBER (indice univoco all'interno del servizio per i viaggi, per mantenere codifica di riconoscimento viaggi con la nuova sezione turni)

2.3.13.2 SezioneCoperturaTMeTG

SezioneCoperturaTMeTG ::= SezioneCoperturaTM **NL** *SezioneCoperturaTG*

SezioneCoperturaTM ::= **#{COPERTURATM NL** <*unTM* **NL**>*** #}END_COPERTURATM**

SezioneCoperturaTG ::= **#{COPERTURATG NL** <*unTG* **NL**>*** #}END_COPERTURATG**

unTM ::= StatoTurno ***TAB*** *CodAzTurno* ***TAB*** *nomeUdp* ***TAB*** *descrUdp*

unTG ::= StatoTurno ***TAB*** *CodAzTurno* ***TAB*** *nomeUdp* ***TAB*** *descrUdp*

2.3.13.3 SezioneVeicoliCompatibili

SezioneVeicoliCompatibili::= #{**VEICOLICOMPATT NL** <*VeicoloCompatibile* **NL** >* #}**END_ VEICOLICOMPATT**

VeicoloCompatibile::= CHAR(10)

2.3.13.4 SezioneAttività

SezioneAttività::= *SezioneCorsaAVuoto/ SezioneCorsa*

2.3.13.4.1 SezioneCorsaAVuoto

SezioneCorsaAVuoto::=#{**CORSAAVUOTO** *StatoCorsaAVuoto* **NL** < *SezioneCorsaAVuoto* **NL** >* #}**END TAB CORSAAVUOTO**

StatoCorsaAVuoto{**DaDB**(=importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB**(modificata nel TTD)}

SezioneCorsaAVuoto::=#{ *NodoDA* **TAB** *NodoDA* **TAB** *Ordinale* **TAB** *TempoInizio* **TAB** *TempoFine* **NL** #}

NodoDA::= VARCHAR2(8)

NodoDA::= VARCHAR2(8)

Ordinale ::= NUMBER

TempoInizio::= NUMBER

TempoFine::= NUMBER

2.3.13.4.2 SezioneCorsa

SezioneCorsa::=#{**CORSA** *StatoCorsa* **NL** *CodCorsa* **TAB** *Percorso* **TAB** *Verso* **NL** *SezioneAttributi* **NL** *SezioneTransiti* **NL** #}**END_CORSA**

StatoCorsa::= {**DaDB** corsa importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB** (modificata nel TTD nelle sue componenti: transiti), **ModSoloAttr** (modificata solo nei suoi attributi)}

CodCorsa::= NUMBER

Percorso::= VARCHAR2(6)

Verso::= CHAR(2)

SezioneAttributi::=#{ **ATTRIBUTICORSA** **NL** *CodiceAziendale* **TAB** *IdCorsa* **TAB** *Biglietteria* **TAB** *CorsaPostale* **TAB** *CorsaPacchi* **TAB** *Contributi* **TAB** *Anticipo* **TAB** *Ritardo* **TAB** *TipoCorsa* **TAB** *TipoSosta* **TAB** *KilometriIn* **TAB** *Scolastica* **TAB** *GiornoSettimanale* **TAB** *AlgFissa* **TAB** *Fissa* **NL** *GiorniEsclusi* **NL** *StatoNota* **TAB** *TipoNota* **TAB** *Nota* **TAB** *CodNota* **NL** *SezioneNoteDiLineaInCorsa* #}**END_ATTRIBUTICORSA**

CodiceAziendale::= VARCHAR2(20) (campo utilizzato soltanto in caso di importazione da ROT_DB)

IdCorsa::= VARCHAR2(6)

Biglietteria::= CHAR(1) (se la corsa è ordinaria)

CorsaPostale::= CHAR(1) (se la corsa è ordinaria)

CorsaPacchi::= CHAR(1) (se la corsa è ordinaria)

Contributi::= CHAR(1) (se la corsa è ordinaria)

Anticipo::= NUMBER (campo utilizzato soltanto in caso di importazione da ROT_DB, 0 altrimenti)

Ritardo::= NUMBER (campo utilizzato soltanto in caso di importazione da ROT_DB, 0 altrimenti)

TipoCorsa::= CHAR(1)

TipoSosta::= CHAR(1)

KilometriIn::= NUMBER (Standard, Cambio)

Scolastica::= CHAR(1) (campo per l'importazione da MTDB, 'N' altrimenti)

GiornoSettimanale::= VARCHAR2(7) (campo per l'importazione da MTDB, "NULL" altrimenti)

AlgFissa::= CHAR(1) (campo utilizzato soltanto in caso di importazione da ROT_DB, valore 'S' oppure 'N', corsa fissa per l'algoritmo di incorporazione)

Fissa::= CHAR(1) (campo utilizzato soltanto in caso di importazione da ROT_DB, valore 'S' oppure 'N', corsa fissa su una linea a prenotazione)

GiorniEsclusi::= VARCHAR2(7) (campo per l'importazione da MTDB, "NULL" altrimenti)

StatoNota::= {**DaDB**(=importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB**(modificata nel TTD)}

TipoNota::= NUMBER

Nota::= VARCHAR2(255) (nota sulla corsa)

CodNota::= VARCHAR2(5) (simbolo nota sulla corsa)

SezioneNoteDiLineaInCorsa::=<*CodiceNota* **NL** >*

SezioneTransiti:=#{**TRANSITI** <*StatoTransiti* **NL** *SezioneOrari* >* #}**END TAB TRANSITI**

StatoTransiti:= {**DaDB**viaggio importato dal database), **Nuovo** (creato nuovo nel TTD), **ModSoloAttr** (modificato nel TTD nei suoi orari)}

SezioneOrari ::=*NodoDiPercorso* **TAB** *TempoTransitoArrivo* **NL** *TempoTransitoPartenza*

NodoDiPercorso::= VARCHAR2(8)

TempoTransitoArrivo::= NUMBER

TempoTransitoPartenza::= NUMBER

2.3.13.5 SezioneTransitiPrefissati

SezioneTransitiPrefissati::= #{**TRANSITIPREFISSATI NL**
<*TransitiPrefissati***NL**>*#}**END_TRANSITIPREFISSATI**

TransitiPrefissati::=*CodicePercorso* **TAB** *Verso* **TAB** *Nodo* **TAB** *NodoDa* **TAB** *NodoA* **TAB** *Dalle* **TAB** *Alle* **TAB** *Preferibilmente* **TAB** *NumOrdineinPercorso* **TAB** *NumOrdineinPercorsoDa* **TAB** ***NumOrdineinPercorsoA*** **TAB** *NumeroCorse* **TAB** *Nota*

CodicePercorso::= VARCHAR2(6)

Verso::= CHAR(2)

Nodo::= VARCHAR2(8)

NodoDa::= VARCHAR2(8)

NodoA::= VARCHAR2(8)

Dalle::=NUMBER

Alle::=NUMBER

Preferibilmente::= NUMBER

NumOrdineinPercorso::= NUMBER

NumOrdineinPercorsoDa::= NUMBER

NumOrdineinPercorsoA::= NUMBER

NumeroCorse::= NUMBER (BPTC: numero di corse per fascia oraria sui percorsi Branch)

Nota::= VARCHAR2(255)

2.3.13.6 SezioneNoteLinea

SezioneNoteLinea::= #{**NOTELINEA NL** <*NotaLinea***NL**>*#}**END_NOTELINEA**

NotaLinea::= #{**NOTALINEA NL** *StatoNota* **TAB** *CodiceNota* **TAB** *TipoNota* **TAB** *Nota* **TAB** #}**END_NOTALINEA**

StatoNota::= {**DaDB** (importata dal database), **Nuovo** (creata nuova nel TTD), **ModDB** (modificata nel TTD)}

CodiceNota::= VARCHAR2(5)

TipoNota::= NUMBER

Nota::= VARCHAR2(255)

2.3.13.7 SezioneCancellati

La seguente sezione risulta vuota se interagiamo con MTDB.

SezioneCancellati::=#{ **CANCELLATI** <*SezioneNoteLineaCancellate* >***NL** <*SezioneViaggiCancellati*>* **NL** <*SezioneCorseCancellate*>* **NL** >* #}**END_CANCELLATI**

2.3.13.7.1 SezioneNoteLineaCancellate

SezioneNoteLineaCancellate::= **#{ NOTELINEACANC NL** <*CodiceNota* **NL>*#}** **END_NOTELINEACANC**

2.3.13.7.2 SezioneViaggiCancellati

SezioneViaggiCancellati::= **#{ VIAGGICANC NL** <*CodiceViaggio* **NL>*#}** **END_VIAGGICANC**

2.3.13.8 SezioneCorseCancellate

SezioneCorseCancellate::=#{ **CORSECANC NL** *CodiceCorsa* **TAB** ~~*CodPercorso* **TAB**~~ *Verso*CodValiditaBase **TAB** *DescrValiditaBase****NL*** #}**END_CORSECANC**

I campi sotto descritti identificano la corsa.

CodiceCorsa::= NUMBER

~~*CodPercorso*::= VARCHAR2(6)~~

~~*Verso*::= CHAR(2)~~

CodValiditaBase ::= VARCHAR(10)

DescrValiditaBase ::= VARCHAR(60)

2.3.14 SezioneTurni

SezioneTurni::=**{#TURNI NL** <*SezioneUdp*> **NL** <*SezioneFileTVB*> **NL #}END_TURNI**

2.3.14.1 SezioneUDP

SezioneUdp::= **#{UDP NL** *CodiceUdp* **TAB** *LivPercAFL* **TAB** *dataInizio* **TAB** *datafine* **NL** *DescrizioneUdp* **NL>* #}END_UDP**

SezioneFileTVB ::=**{#File_TVB NL** <*SezioneVersione*> **NL** <*SezioneTurniMacchina* > ***NL*** <*SezioneTurniGuida*> **NL** <*SezioneNormativa*> **#End}**

2.3.14.2 SezioneVersione

{#Versione NL *numVersione* **NL #End}**

numVersione ::= NUMBER (numero versione file "salva stato")

2 → GIFR : 20090219

3 → GIFR : 20090429 : gestione secondi

2.3.14.3 SezioneTurniMacchina

SezioneTurniMacchina::= **#{ TURNIMACCHINA NL** *NumeroTurniMacchina* **NL** <*SezioneTurnoMacchina* **NL>* #End}**

NumeroTurniMacchina ::= NUMBER (numero di turni macchina nella sezione *SezioneTurnoMacchina*)

SezioneTurnoMacchina::= **{TM NL** *id* **TAB** *TipoServizio* **TAB** *Stato* **TAB** *Numero* **TAB NL** *CodiceAziendale* **NL** *Veicolo* **NL** *Deposito* **NL** *SezioneAttivitaTM* **#End}**

SezioneAttivitaTM::= **#{ Attivita NL** *NumeroAttivitaTM* **NL** <*AttivitaTM*>* **#End}**

AttivitaTM ::= *id* **TAB** *Inizio* **TAB** *Fine* **NL** *Tipo* **NL** *idLuogoInizio* **TAB** *idLuogoFine* **TAB** *Ordinale* **NL** *SiglaLuogoInizio* **NL** *NomeLuogoInizio* **NL** *SiglaLuogoFine* **NL** *NomeLuogoFine* **NL** *DepositoForzato* **TAB** *VeicoloForzato* **NL** *SubConcessione* (S/N) **TAB** *AziendaSubConc* **NL**

Ordinale ::= NUMBER (ordinale arco fuori linea per le attività di tipo FuoriLinea, -1 altrimenti)

2.3.14.4 SezioneTurniGuida

SezioneTurniGuida::= **{# TURNIGUIDA NL** *NumeroTurniGuida* **NL** <*SezioneTurnoGuida* **NL>* #End}**

SezioneTurnoGuida::= **{#TG NL** *id* **TAB** *TipoServizio* **TAB** *TempoGuida* **TAB** *TempoLavoro* **NL** *CodiceAziendale* **NL** *Residenza* **NL** *Classe* **NL** *Zona* **NL** *Tipologia* **NL** *DescrizioneClassificatore* **NL** *SezioneAttivitaTG* **#End}**

SezioneAttivitaTG::= **#{ Attivita NL** *NumeroAttivitaTG* **NL** <*AttivitaTG*>* **#End}**

AttivitaTG ::= *id* **TAB** *Inizio* **TAB** *Fine* **TAB** *IdTM* **TAB** *idAttivita* **NL** *TipoAttivita* **NL** *idLuogoInizio* **TAB** *idLuogoFine* **NL** *SiglaLuogoInizio* **NL** *NomeLungoLuogoInizio* **NL** *SiglaLuogoFine* **NL** *NomeLungoLuogoFine* **NL**

2.3.14.5 SezioneNormativa

SezioneNormativa ::= **{#Normativa NL 0 NL #End}**

Parameter name	Value	Requirements Reference	Meaning	Version
{#Parameters for Planning Rules				
Version:	2		Version of Planning Rules file	2
#}				
{Rests				
Lunch:	20	2.5.4.3	Lunch rest duration	1
Dinner:	20	2.5.4.3	Dinner rest duration	1
Splitted_Shifts:	240	2.5.4.1	Minimal rest for splitted shifts	1
BusDriverChange:	10	2.5.4.4	Minimal rest for bus drivers shifts	1
MealsOnMainTerminus:	1		1 if the meals are possible only over main terminus	2
#}				
{PreparationTime:				
BusPreparation:	15	2,5,2	Accessory preparation time at the begin of a shift. Splitted shifts have a preparation time both at the shift start and after the break	1
BusLeaving:	0		Accessory "debriefing" time not required at the moment by BPTC rules	1
#}				
{Meals:				
LunchBegin:	630	2.5.4.3	Lunch Interval begin	1
LunchEnd:	870	2.5.4.3	Lunch Interval end	1
DinnerBegin:	1080	2.5.4.3	Dinner Interval begin	1
DinnerEnd:	1320	2.5.4.3	Dinner Interval end	1
#}				
{Times			Driving and working time for each driver shift typology	1
{Typology Continous				
MinimumWork:	120	2,5,6	Minimum work for continous shifts	1
MediumWork:	300	2,5,6	Medium work for continous shifts	1
MaximumWork:	480	2,5,6	Maximum work for continous shifts	1
MinimumDrive:	0	2,5,5	Minimum driving time for continous shifts	1
MediumDrive:	300	2,5,5	Medium driving time for continous shifts	1
MaximumDrive:	390	2,5,5	Maximum driving time for continous shifts	1
#}				
{Typology Splitted				

MinimumWork:	120	2,5,6	Minimum work for splitted shifts	1
MediumWork:	300	2,5,6	Medium work for splitted shifts	1
MaximumWork:	480	2,5,6	Maximum work for splitted shifts	1
MinimumDrive:	0	2,5,5	Minimum driving time for splitted shifts	1
MediumDrive:	300	2,5,5	Medium driving time for splitted shifts	1
MaximumDrive:	390	2,5,5	Maximum driving time for splitted shifts	1
#}				
#}				
{Typologies			How continous shifts are divided	1
{Class				
Morning				
EndTime:	480	2,5,1	Maximum departure time for Morning continous shifts	1
#}				
{Class				
Daily				
EndTime:	660	2,5,1	Maximum departure time for Daily continous shifts	
#}				
{Class				
Evening				
EndTime:	1800	2,5,1	Maximum departure time for Evening continous shifts	1
#}				
#}				
{Refueling				
StartRefuelingLimitation:	720	2.4	Vehicle blocks starting after "StartRefuelingLimitation" can't refuel before ending the first round (the first ascending run + the first descending run)	1
#}				

E' passato talmente tanto tempo da quando mi sono iscritto all'Università che ringraziare la Famiglia (siete in parecchi... Lorella, Paolo, Giulia, Marcello, Liliana, Rosetta...) per il sostegno ricevuto non è solo una prassi da rispettare, è un atto dovuto: un monumento probabilmente sarebbe stato più indicato... e allora Grazie, Grazie ancora una volta.

Grazie a Giulia, il mio presente e (spero) il mio futuro: a lei devo tutta la pazienza che ha dovuto tirare fuori nel sopportarmi e anche qualche esame che, senza di lei, non avrei mai pensato di dare.... dovrei strappare un pezzo di Laurea e regalartela.
Grazie anche alla sua famiglia, che sopporta da anni la mia infinita chiacchera e nonostante questo continua ad accettarmi.

Grazie a tutti i miei amici di Piombino e di Pisa: a dir l'onesta verità tra distrazioni, calcetto, birra e playstation non so quanto mi avete aiutato con lo studio, ma va bene così, non cambierei una virgola di niente.

Grazie anche a Samuela e Francesco, gli "esperti M.A.I.O.R.", senza la cui competenza e disponibilità questa tesi non sarebbe stata possibile.

Infine grazie a coloro che con il loro lavoro e il loro aiuto hanno reso possibile la mia formazione e la mia Laurea.

Grazie
Alessandro Bertolini