# Relazione: Progettazione di Algoritmi, a.a. 2013

*Di Luzio Adriano, Francati Danilo*

## Linguaggio e scelte implementative

Per la realizzazione del progetto si è deciso di utilizzare il linguaggio Java. Le strutture dati sfruttate sono, fondamentalmente, **ArrayList** ed **HashMap**, le prime poiché permettono rimozione ed aggiunta in una data posizione in tempo costante e l'iterazione in modo efficiente, le seconde invece, poiché permettono (stando alla documentazione di Java) l'accesso tramite chiave ad un preciso elemento in tempo costante, nella maggior parte dei casi. Infatti, la complessità tende a diventare lineare soltanto nel caso in cui si abbiano delle collisioni nell'algoritmo di Hashing, e la probabilità di tale evento è considerata irrisoria nelle possibili istanze che ci riguardano.
Più in dettaglio, le HashMap hanno permesso di accedere alle specifiche riguardanti una stazione o un treno, sfruttando il codice dato, per adempiere ai vari tipi di richieste e per permettere di creare in modo agevole i grafi sfruttati in seguito dagli algoritmi di risoluzione.

## Funzione e contenuto di ogni package

***io*** contiene le classi addette all'Input/Output:

- Parser: si occupa di effettuare il parsing dal file di input e creare una rete opportuna (oggetto Net) per ogni testcase ivi contenuto.
- Writer: si occupa di scrivere sul file di output le soluzioni ai problemi in input.

***net*** contiene le classi addette all'Input/Output:

- Net: contiene l'istanza di testcase creata dal Parser; in particolare due HashMap, rispettivamente per le stazioni ed i treni, una lista per le richieste ed i metodi di creazioni dei grafi.
- Station: rappresenta una stazione, con relativa tabella di orari di arrivo e partenza per ogni treno che vi transita; include anche alcuni metodi utili a calcolare i vari tempi di attesa e percorrenza dei treni.
- Train: rappresenta un treno, con annessa lista ordinata delle stazioni in cui transita; per riempire e tenere ordinata tale lista si sfrutta l'inserimento ordinato man mano che si legge dal Parser, con dovuta attenzione ai treni che terminano il viaggio il giorno successivo a quello di partenza.
    - TrainAtTime: questa classe annidata viene sfruttata soltanto per poter ordinare comodamente le fermate usando un ArrayList. Tutto ciò che ne deriva, viene convertito in HashMap alla fine del Parsing, in modo da poter accedere in tempo costante a qualsiasi fermata ed ottenere il codice della successiva.

***graph*** contiene le classi utili a rappresentare i generici nodi ed archi di un grafo e le rispettive sottoclassi:

- Edge: rappresenta un arco del grafo, non pesato.
  - WeightedEdge: rappresenta invece un arco del grafo pesato.
- Node: rappresenta un nodo del grafo (*astratto*).
  - NodeStation: rappresenta una stazione nei problemi MinTempo/MinOrario.
  - NodeTrain: viene utilizzato per calcolare agevolmente il tempo di attesa in una specifica stazione per prendere un determinato treno.
  - NodeChange: rappresenta un generico nodo, ma si è preferito estendere Node per migliore efficienza in memoria; infatti, ogni NodeChange ha un numero arbitrario di archi uscenti non pesati, mentre i NodeTrain ne hanno soltanto uno ed i NodeStation hanno un numero variabile di archi pesati.

***utils*** contiene le classi utili a rappresentare i generici nodi ed archi di un grafo e le rispettive sottoclassi:

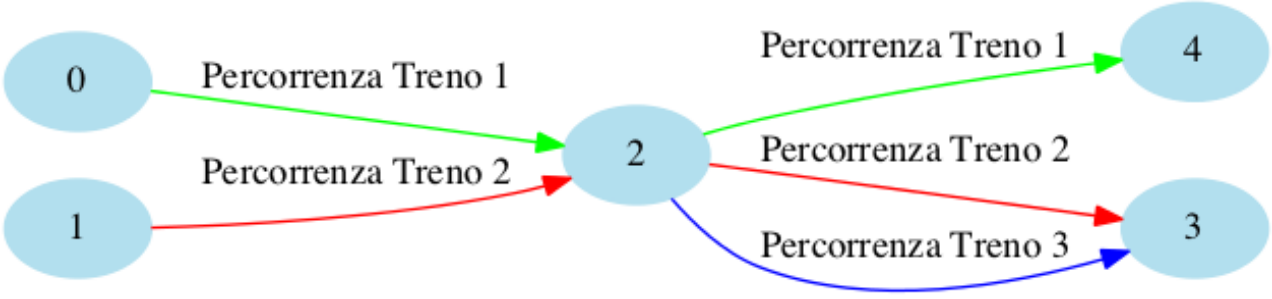
- Request: rappresenta una generica richiesta e un metodo di risoluzione (*astratta*); per i dettagli sulle singole implementazioni si rimanda a un successivo paragrafo.
  - MinTempo
  - MinOrario
  - MinCambi
- Dijkstra: in questa classe è implementato il noto algoritmo per il calcolo dei percorsi minimi, modificato per l'occorrenza con una relativa implementazione di MinHeap.
- BFS: in questa classe è implementato l'algoritmo di visita in ampiezza, senza alcuna modifica; si utilizza una LinkedList come coda.

## Algoritmi e grafi

Per ciò che riguarda gli algoritmi risolutivi, le richieste sono state divise in due categorie, quelle di tipo *MinTempo* e *MinOrario* e quelle di tipo *MinCambi,* e in seguito ad ogni categoria è stato applicato un processo risolutivo specifico.

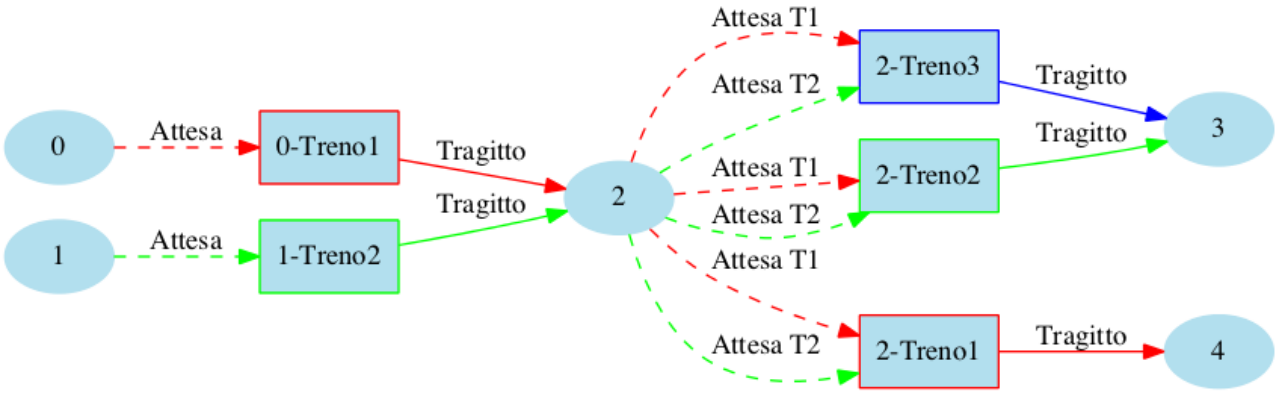
### Dijkstra e MinTempo/MinOrario

Per risolvere i problemi di tipo *MinTempo* e *MinOrario* è stato costruito un grafo in modo da tener conto sia dei tempi di percorrenza da stazione a stazione, che dei tempi di attesa.
Di conseguenza, ogni stazione è stata rappresentata tramite un nodo (*NodeStation*) ed ogni possibile tratta tramite un arco pesato, con peso pari al tempo di percorrenza:




**In figura, ogni stazione è rappresentata da un nodo ed ogni tratta relativa ad un treno da un apposito arco.**

In seguito, per tener conto dei tempi di attesa in stazione, sulla base del treno che ci ha condotto in loco, si sfruttano appositi nodi "fittizi" (*NodeTrain*):



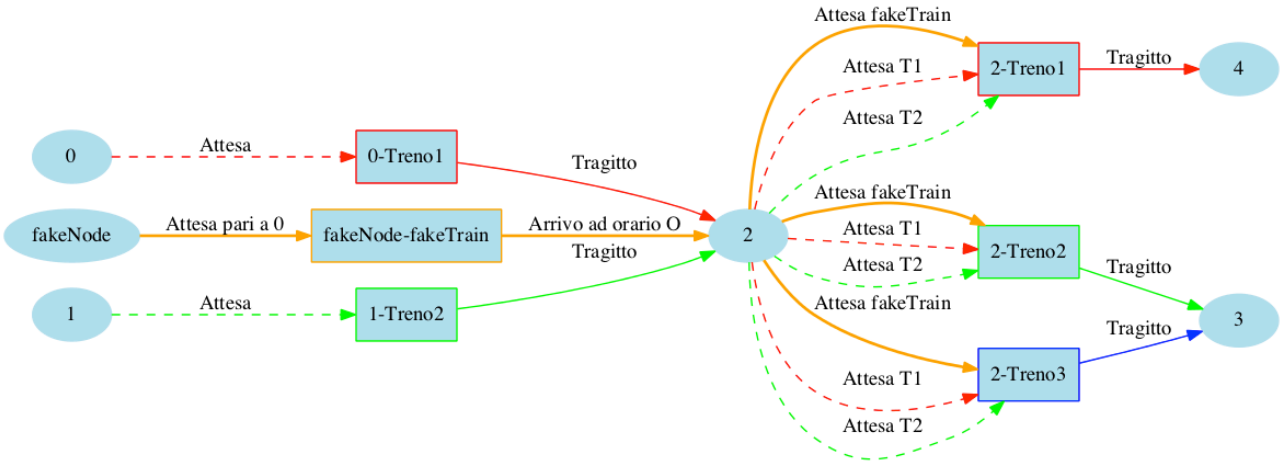

**In figura si può notare l'aggiunta dei NodeStation con un arco entrante pari all'attesa in stazione, differenziato in base al treno di provenienza.**

A questo punto si può quindi eseguire il classico algoritmo di Dijkstra, con l'accortezza di considerare a ogni iterazione un unico arco quello che comprende, nell'ordine, tragitto e attesa relativa al treno del NodeTrain precedente.


**In figura si nota ciò che viene considerato come un singolo arco ad ogni iterazione dell'algoritmo di Dijkstra.**

Ovviamente, se la stazione in esame è proprio quella di interesse, il tempo di attesa non viene calcolato.

In particolare poi, per risolvere il problema del MinOrario è bastato utilizzare la stessa strategia, con l'aggiunta di un treno fittizio, che arriva alla stazione di partenza esattamente all'orario 0, sfruttando archi dal peso nullo. In questo modo si permette all'algoritmo di Dijkstra di operare correttamente e scegliere il percorso appropriato.




**In figura, in arancione, è stato aggiunto un finto nodo che, tramite archi dal peso nullo, arriva in stazione esattamente in orario 0.**

Una volta impostato il problema come sopra, si può stimare la complessità e l'efficienza dell'esecuzione di una singola istanza di MinTempo o MinOrario.

Assumendo, come verificato sperimentalmente, che ci sia un pari numero $n$ di treni e di stazioni in ogni TestCase e supponendo che in media, in ogni stazione transiti un numero $p$ di treni, poiché per ogni stazione vengono creati tanti altri nodi quanti i treni che vi transitano, una stima ragionevole del numero di nodi del grafo è:

$$n + np$$

Invece, una stima per il numero di archi del grafo è:

$$np + np^2$$

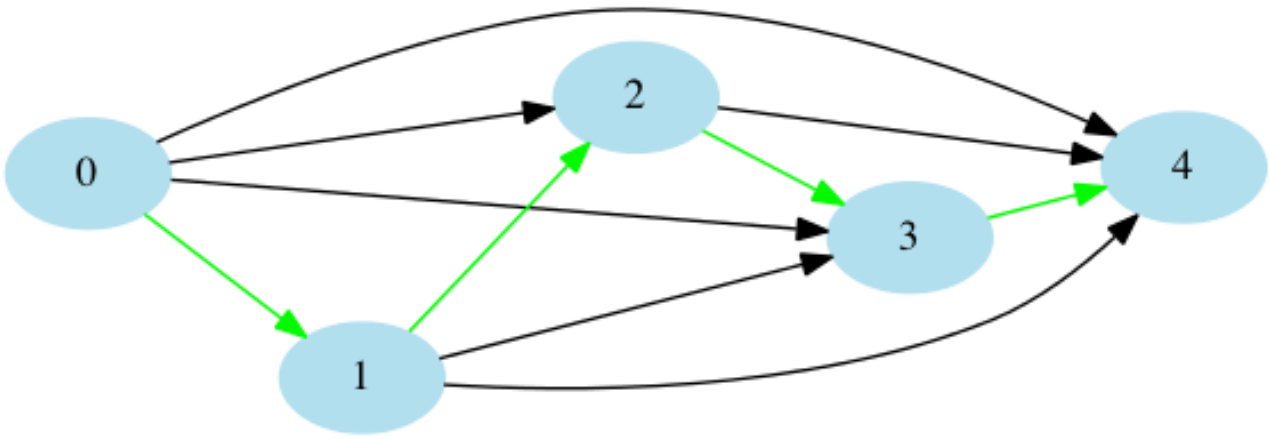
In realtà, però, queste stime riguardano soltanto la rappresentazione in memoria del grafo, perché ogni arco di arrivo in un nuovo NodeStation vincola la scelta sull'arco di uscita, e, tenendo conto di ciò che viene considerato dall'algoritmo ad ogni iterazione e della complessità originale dell'algoritmo, si ottiene una complessità di:

$$O((np + np^2)\log(np))$$

Tale complessità potrebbe essere migliorata ulteriormente con una stima meno lasca per $p$, approssimandola ad un valore del tipo $n/\delta$ .

### BFS e MinCambi

Per ciò che riguarda la seconda categoria invece, quella delle richieste di tipo MinCambi, il ragionamento che ha portato alla soluzione è stato completamente differente, influenzato dagli esempi di grafi derivati da applicazioni della Programmazione Dinamica. Più in dettaglio si è pensato di costruire un grafo in cui ogni arco unisce due nodi (*NodeChange*) raggiungibili senza alcun cambio di treno, per poi usare una semplice BFS, inizializzando le distanze a *-1,* per calcolare la distanza tra due qualsiasi nodi.




**In figura si può osservare, in verde il tragitto originario di un generico treno ed in nero gli archi aggiunti, permettendo di raggiungere il nodo 4 utilizzando un unico arco, e quindi un unico cambio.**

Anche in questo caso, è facile stimare la dimensione del grafo; infatti, sempre assumendo che $n$ sia il numero di nodi e di treni e che ogni treno in media attraversi $f$ stazioni, si ha un numero di archi per treno pari a:

$$\sum_{i=1}^{f}(f-i) = \frac{1}{2}(f-1)f$$

Da cui un numero totale di:

$$\frac{n}{2}(f-1)f$$

A questo punto, eseguendo l'algoritmo di visita in ampiezza su tali numeri di nodi e di archi, si ottiene una complessità di:

$$O\left(n+\frac{n}{2}(f-1)f\right)=O(n+nf^2)$$

Anche qui si potrebbe rendere tale complessità più stringente ottenendo una migliore stima di *f*. In particolare, se *f* non fosse comparabile, in una particolare istanza ad *n*, otterremmo una complessità pari a:

$$O(n+n)$$

## Risultati sperimentali

Al termine della realizzazione l'intero progetto è stato sottoposto ad una fase di test, su varie macchine, per valutarne sperimentalmente le prestazioni.
Ecco i risultati ottenuti lanciando il progetto tramite il tool Unix *time* usando come file di input quello fornito in esempio:

| | Intel(R) Pentium(R) 4 CPU 2.80GHz | Intel(R) Pentium(R) Dual CPU E2180 @ 2.00Ghz | Intel(R) Core(TM)2 Quad CPU Q8300 @ 2.50GHz | Intel(R) Core(TM) i7-3667U CPU @ 2.00GHz |
|---|---|---|---|---|
| **Cache** | 512KB | 1024KB | 2048KB | 4096KB |
| **Real** | 0m11.409s | 0m4.352s | 0m2.388s | 0m1.626s |
| **User** | 0m10.816s | 0m6.316s | 0m4.004s | 0m2.845s |
| **Sys** | 0m0.196s | 0m0.256s | 0m0.120s | 0m0.097s |

## Conclusioni

Durante la realizzazione di questo progetto ci siamo trovati di fronte a sfide di varia natura: dall'analisi dei requisiti e della complessità, alla realizzazione efficiente di algoritmi visti soltanto in teoria e mai sperimentati nella pratica, fino ad arrivare al debugging e al testing delle soluzioni proposte al fine di verificarne l'ottimalità e le prestazioni. Tutto ciò, a prescindere dall'ambito accademico, ha permesso di misurarci con istanze di problemi da un alto livello di astrazione, costringendoci ad acquisire un metodo risolutivo funzionale e ponderato, per poter raggiungere i risultati richiesti nel modo più accurato e meno dispendioso possibile.
È stato, in conclusione, un modo entusiasmante per mettere alla prova le conoscenze acquisite finora e farci un'idea di ciò che ci aspetterà in future esperienze lavorative.