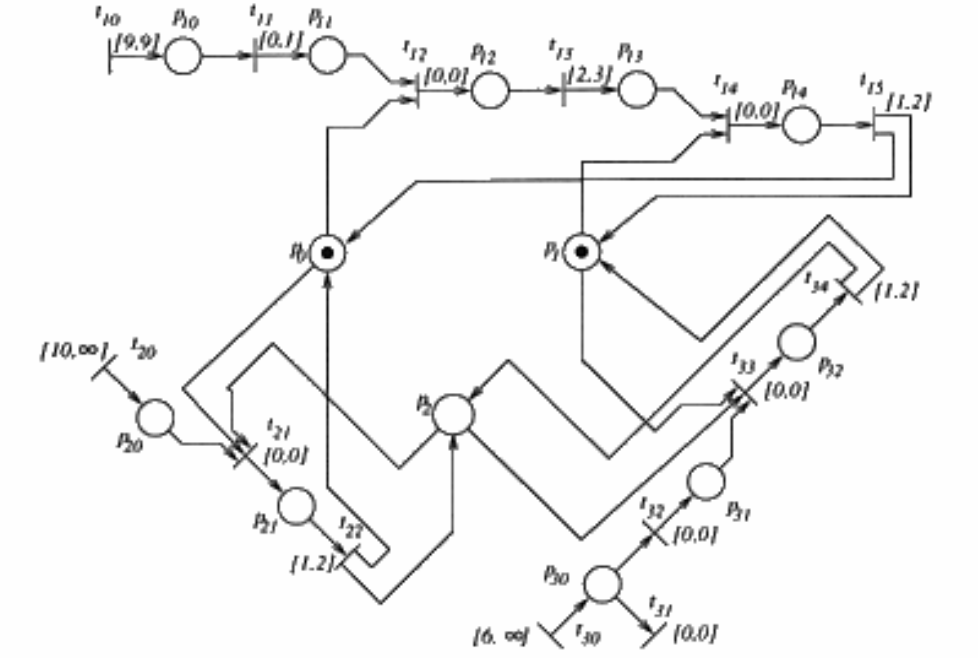
UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elaborato per l'esame di
"Ingegneria del Software" A.A.1998-99
Prof. G. Bucci

Andrea Fedeli Sauro Menchetti

Analisi di shortest e longest path in un grafo ottenuto da Time Petri Net



Sommario

INTRODUZIONE **2**

1.1 IL PROBLEMA DELL'APPROCCIO ENUMERATIVO 4

GLI ALGORITMI **5**

2.1 IL PERCORSO MINIMO E IL PERCORSO MASSIMO 5
2.2 LA RICERCA DEI CICLI ORIENTATI 7
2.2.1 La ricerca delle componenti fortemente connesse *8*
2.2.2 Controllo basato sul costo massimo possibile *9*
2.2.3 Controllo basato sul numero massimo di aggiornamenti *9*
2.2.4 Controllo basato sulla ricerca dei cicli online *9*
2.2.5 Confronto tra i metodi di ricerca dei cicli *10*
2.3 SCELTE EFFETTUATE 11

IL MANUALE DELL'UTENTE **12**

3.1 I PARAMETRI ALLA LINEA DI COMANDO 12
3.1.1 La sintassi completa della linea di comando *13*
3.1.2 Tolleranza agli errori alla linea di comando *14*
3.2 IL FORMATO DEL FILE IN INGRESSO 15
3.2.1 I margini di variazione sul formato del file *16*
3.2.2 Eventuali errori presenti sul file *17*
3.3 IL FORMATO DEL FILE DI USCITA 17
3.4 UN ESEMPIO 19

IL MANUALE DELL'AMMINISTRATORE **20**

4.1 LE DIPENDENZE TRA I FILE SORGENTI 21
4.2 IL MAKEFILE 22

IL MANUALE DEL PROGRAMMATORE **23**

5.1 I FILE E LE CLASSI 25
5.2 L'INTERPRETAZIONE DEL FILE `*.tg` 26
5.2.1 La classe `BufferFile` *27*
5.2.2 La classe `BufferArco` *27*
5.2.3 La classe `BufferTransizione` *28*
5.2.4 La classe `BufferStato` *28*
5.2.5 La classe `ListaStati` *28*
5.2.6 La classe `InformazioniGrafo` *29*
5.2.7 La classe `BufferGrafo` *29*
5.3 LE CLASSI PER LA GESTIONE DEI GRAFI SPARSI 30
5.3.1 La classe `Arco` *31*
5.3.2 La classe `Nodo` *31*
5.3.3 La classe `NodoIns` *32*
5.3.4 La classe `Grafo` *33*
5.3.5 La classe `GrafoDir` *34*
5.4 I TEMPLATE DI USO GENERALE 36
5.4.1 La classe template `ElemLista<T>` *36*
5.4.2 La classe template astratta `Lista<T>` *37*
5.4.3 La classe template `Coda<T>` *38*
5.4.4 La classe template `Pila<T>` *38*
5.4.5 La classe `CodaNodi` *38*
5.4.6 La classe `DequeueNodi` *39*
5.5 L'INTERCONNESSIONE TRA LE PRINCIPALI CLASSI 39
5.5.1 La classe `Interfaccia` *40*

INDICE DELLE FIGURE E DELLE TABELLE **41**

BIBLIOGRAFIA **42**

Introduzione

Nella modellazione di sistemi fisici, il tempo ricopre spesso un ruolo molto importante, sia per il rispetto di certi vincoli temporali, sia per il sequenziamento delle operazioni. Un'analisi degli aspetti temporali è utile per poter prevedere il comportamento del sistema. Nella fase di modellazione del sistema, i modelli più usati sono gli automi a stati finiti e le reti di Petri, che si basano sui concetti di stato e di transizione di stato. Tali modelli vengono estesi in modo opportuno per tener conto dello sviluppo temporale del sistema: le Time Petri Nets sono una estensione delle reti di Petri che tiene conto del tempo.

I due modelli precedentemente citati fanno parte dei modelli operazionali, cioè di quei modelli in cui la specifica di un sistema ha luogo tramite la descrizione degli stati in cui il sistema può venirsi a trovare durante la sua evoluzione e delle transizioni che portano il sistema da uno stato ad un altro. Formalismi di tale genere sono detti operazionali in quanto fanno riferimento alle operazioni effettuate da una macchina astratta la cui esecuzione specifica il comportamento del sistema modellato. Tale macchina indica quali sono le possibilità di trasformazione tra gli stati e qual è lo stato raggiunto durante una specifica evoluzione a partire da un determinato stato iniziale. Ciò che differenzia le reti di Petri dagli ordinari automi a stati finiti è l'introduzione di una nuova concezione di stato e di transizione di stato [6]. In una reti di Petri, lo stato di un sistema e la transizione che porta il sistema da uno stato originario ad un nuovo stato sono concetti distribuiti: ogni stato è la riunione di più stati parziali e indipendenti e una transizione in generale non riguarda lo stato globale del sistema, ma si limita a variarne solo una parte.

Una volta che il sistema è stato descritto con un modello formale, è interessante analizzare il modello del sistema per poter prevedere il suo comporta-

mento. Nell'analisi di un modello tempificato, il criterio con cui modellare il tempo ricopre un aspetto di fondamentale importanza: un modello discreto del tempo porta ad un'analisi semplificata, mentre un modello temporale denso introduce delle problematiche che richiedono l'introduzione di metodologie di analisi più sofisticate. Un approccio promettente per risolvere questo problema è quello di introdurre una relazione di equivalenza e di raggruppare gli stati che soddisfano tale relazione in delle classi di equivalenza. Con questo metodo si riescono così a trattare anche quei modelli in cui il tempo varia su un insieme denso di valori.

Il concetto di relazione di raggiungibilità tra gli stati viene quindi sostituito da una relazione di raggiungibilità tra classi di stati che raggruppano insiemi densi di stati. Una classe di stati è raggiungibile da un'altra classe di stati se raggruppa tutti e soli gli stati che sono raggiungibili da un qualunque stato dell'altra classe di stati. La precedente metodologia permette pertanto di passare da un'infinità densa di stati ad un insieme finito di classi di equivalenza: a questo punto si può pensare di eseguire un'analisi del modello del sistema. Il passaggio da un'infinità non numerabile di stati ad un insieme finito di classi di equivalenza, porta alla costruzione di un grafo in cui i nodi rappresentano le classi di equivalenza e gli archi la relazione di raggiungibilità tra le classi di stati, cioè l'enumerazione delle classi di stati raggiungibili del sistema risulta in un grafo delle classi. Ogni arco rappresenta una transizione che può eseguire a partire da un certo insieme di stati. Per ogni transizione esistono un earliest e un latest time che vincolano la sua esecuzione ad un certo intervallo temporale. Possono esistere anche dei vincoli temporali tra più transizioni, che ne vincolano l'esecuzione in un certo ordine.

Avendo a disposizione questo grafo delle classi, è possibile enumerare tutte le tracce che iniziano da una transizione start e che finiscono in una transizione target. È possibile ad esempio calcolare quanto tempo trascorre da quando ha eseguito una transizione a quando se ne è conclusa un'altra. Un approccio possibile impiega una tecnica enumerativa che supporta l'analisi di raggiungibilità e di temporalità di modelli dipendenti dal tempo. Vengono enumerate tutte le tracce che partono da una transizione iniziale ad una finale: a questo punto è possibile valutare tutti i parametri di interesse per poter eseguire un'analisi di predicibilità sul modello del sistema fisico. Una traccia rappresenta un cammino tra due nodi del grafo: si capisce perciò che l'enumerazione di tutte le tracce è un problema intrinsecamente complesso [1].

1.1 Il problema dell'approccio enumerativo

Una traccia è composta da un insieme di transizioni vincolate in modo complesso l'una all'altra. Ciascuna transizione può eseguire in un certo intervallo temporale che dipende dai vincoli dello stato da cui la transizione origina (earliest e latest firing time), ma anche dai vincoli che appaiono sugli altri stati visitati dalla traccia. Per questo l'analisi del tempo massimo e minimo con cui è possibile raggiungere uno stato destinazione a partire da uno stato sorgente, richiede che siano enumerate tutte le possibili tracce tra i due stati e che per ciascuna di queste sia valutata separatamente la tempificazione. Questo porta ad una complessità esponenziale rispetto al numero di nodi del grafo. Trascurando la dipendenza tra i tempi delle transizioni, la complessità del problema può essere ridotta ad un tempo polinomiale con indice quadratico.

L'idea è quella di considerare una traccia come una sequenza di transizioni che possono eseguire tutte in corrispondenza dei loro earliest time o dei loro latest time: quello che si ottiene è una stima in difetto e in eccesso rispettivamente del tempo minimo e massimo di esecuzione di una traccia. Così facendo il tempo associato a ciascuna transizione è indipendente dalla traccia in cui questa è inserita. Questo permette di ridurre l'analisi della tempificazione delle tracce tra due stati ad un problema di cammino minimo e massimo su un grafo etichettato sugli archi.

Per realizzare questo, si associano a ciascun arco due valori: il più piccolo rappresenta l'earliest time della transizione associata all'arco mentre l'altro non è che il minimo dei latest time delle transizioni che possono eseguire a partire da quella classe di stati. Utilizzando un algoritmo in grado valutare lo shortest e il longest path in un grafo diretto orientato, si trovano delle stime del tempo minimo e massimo di esecuzione di una traccia.

Gli algoritmi

I problemi di calcolo del cammino minimo e massimo sul grafo etichettato possono essere unificati nella forma di un problema di cammino minimo su un grafo con pesi eventualmente negativi. Per questo viene di seguito descritto un algoritmo.

2.1 Il percorso minimo e il percorso massimo

Il calcolo del percorso più lungo su di un grafo può essere ricondotto a quello del cammino minimo in un grafo analogo con pesi cambiati di segno. Questo tuttavia introduce alcuni problemi. Infatti, come si può notare dall'esempio della figura sottostante, l'usuale algoritmo di Dijkstra [2, 4] per calcolare il percorso minimo su di un grafo fallisce se ci sono pesi negativi.

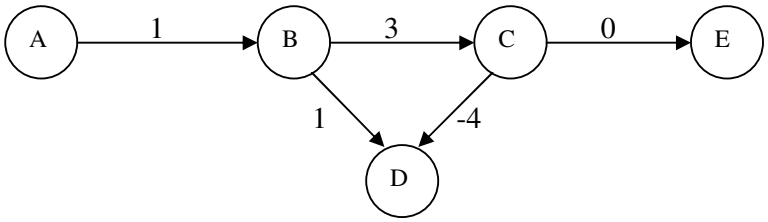


Figura 1: Esempio di grafo in cui l'algoritmo di Dijkstra fallisce.

Ad esempio, nel grafo sopra illustrato, l'algoritmo di Dijkstra avrebbe trovato che il percorso migliore da *A* a *D* è *ABD*, mentre invece è *ABCD*: cerchiamo di capire perché questo metodo non funziona. L'algoritmo di Dijkstra utilizza due liste: la prima, chiamata *OPEN*, contiene i nodi per cui non è ancora stato trovato un percorso minimo; la seconda, chiamata *CLOSED*, contiene i nodi per cui tale cammi-

no è stato trovato. Ad ogni passo l'algoritmo rimuove un nodo dalla lista *OPEN* e lo colloca nella lista *CLOSED*. Procedendo in questo modo, tuttavia, non si riesce a "*guardare in avanti*": questo perché, nel caso di archi non negativi, è inutile conoscere il costo degli archi a cui non siamo ancora giunti, in quanto nessun arco potrà diminuire il costo di un cammino. Questo invece accade quando sono presenti archi negativi. Se permettiamo invece ad un nodo della lista *CLOSED* di tornare nella *OPEN*, quando scopriamo che il costo del cammino trovato non è corretto, abbiamo ottenuto un algoritmo che è valido anche nel caso di archi negativi. Tuttavia un algoritmo così fatto risulta esponenziale [2].

Il seguente algoritmo risolve il problema in un tempo pari a $O(NM)$, dove N indica il numero dei nodi e M il numero degli archi, sotto l'ipotesi che il grafo non contenga cicli orientati di costo negativo. È possibile dimostrare che questa è la migliore complessità ottenibile per questa classe di problemi [2]. Nella rappresentazione dell'algoritmo, *S* rappresenta il nodo iniziale:

---

1. *Costo[S] = 0 e pred[S] = 0*
2. *Costo[N] = + ∞ per ogni nodo N diverso da S*
3. *Metti S nella CODA*
4. *Preleva un nodo N dalla CODA*
   - *4.1. Sia A un nodo adiacente ad N e C_{NA} il costo dell'arco $N \rightarrow A$:*
     - *4.1.1. Se Costo[A] > Costo[N] + C_{NA}*
       - *4.1.1.1. Costo[A] = Costo[N] + C_{NA}*
       - *4.1.1.2. pred[A] = N*
       - *4.1.1.3. Se A non è nella CODA, metti A nella CODA*
   - *4.2. Se N ha altri nodi adiacenti torna al punto 4.1.*
5. *Se CODA non è vuota, torna a 4.*
6. *Esci*

Figura 2: Algoritmo per i grafi con costi negativi.

L'insieme di nodi *CODA* rappresenta una struttura gestita secondo la politica *FIFO*. Si può dimostrare che la precedente complessità si ottiene ***solo*** con questa struttura dati: questo perché procedendo così, è possibile dimostrare che ogni nodo viene aggiornato al massimo *N-1* volte. Utilizzando delle strutture gestite con altre politiche (ad esempio *LIFO* o altre ancora), si mantiene comunque la corret-

tezza e la finitezza dell'algoritmo e si possono avere anche miglioramenti nel caso medio, ma tuttavia la complessità nel caso peggiore diventerà esponenziale. Utilizzando una struttura particolare si ottengono le migliori prestazioni nel caso medio su grafi sparsi: un nodo verrà inserito in coda se è già stato inserito in precedenza, altrimenti verrà inserito in testa. Tale struttura è nota come dequeue [2].

2.2 La ricerca dei cicli orientati

Il precedente algoritmo non tratta il caso di grafi che contengono un ciclo orientato negativo, come nell'esempio in Figura 3.

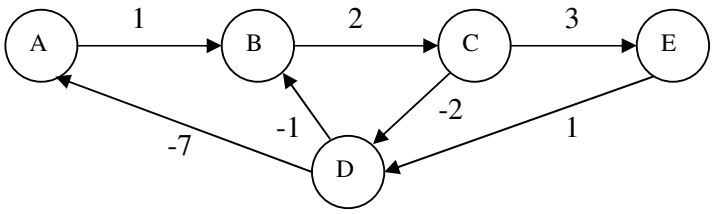


Figura 3: Esempio di grafo con dei cicli orientati.

Questo grafo ha tre cicli: il ciclo *BCEDB* che ha costo 5, il ciclo *BCDB* che ha costo -1 e il ciclo *ABCEDA* che ha costo 0. Vediamo cosa accade nel ciclo *BCD*. Supponiamo che il nodo *B* abbia come costo corrente 1 ed i nodi *C*, *D* $+\infty$:

1) si aggiorna il costo di *C* a 3 e si mette *C* nella coda;
2) si esamina *C*, si aggiorna il costo di *D* a 1 e si mette *D* nella coda; ignoriamo *E*;
3) si esamina *D*, si aggiorna *B* a 0 e si mette *B* nella coda; ignoriamo *A*;
4) si esamina *B*, si aggiorna il costo di *C* a 2 e si mette *C* nella coda;
5) si esamina *C*, si aggiorna il costo di *D* a 0 e si mette *D* nella coda; ignoriamo *E*;
6) si esamina *D*, si aggiorna *B* a -1 e si mette *B* nella coda; ignoriamo *A*;
7) si esamina *B*, si aggiorna il costo di *C* a 1 e si mette *C* nella coda;

e così via…

A questo punto è evidente che il costo dei nodi *BCD* diminuirà di 1 ad ogni ciclo senza mai terminare: risultato peraltro corretto perché non esiste alcun percorso minimo per andare da *A* ad *E,* ovvero il costo minimo sarà $-\infty$. Con un ragionamento analogo si può intuire la correttezza dell'algoritmo se i cicli orientati hanno un costo non negativo.

Possiamo a questo punto, procedere in due modi:

- controllare preliminarmente la presenza di cicli negativi ed in presenza di questi non eseguire l'algoritmo, ma limitarsi soltanto a segnalare il ciclo;
- inserire dei controlli che siano in grado di rilevare la presenza di un ciclo all'interno dell'algoritmo stesso.

Vediamo nei dettagli queste due strategie, illustrandone delle possibili realizzazioni, insieme ai vantaggi ed agli svantaggi risultanti.

2.2.1 La ricerca delle componenti fortemente connesse

Diamo innanzitutto alcune definizioni.

Un grafo si dice *fortemente connesso* se da ogni suo nodo è possibile raggiungere un qualunque altro nodo; si dice *debolmente connesso* se questo è possibile trascurando l'orientamento degli archi; altrimenti si dice *sconnesso*.

In un grafo G, si dice che un sottografo G' è una *componente fortemente connessa* se G' è un grafo fortemente connesso. Assumiamo che un nodo sia un grafo fortemente connesso.

Una componente fortemente connessa G' si dice *massimale* se non esiste nessun altra componente fortemente connessa G^* che contenga propriamente G'.

Dalle precedenti definizioni, discendono le seguenti proprietà:

- una componente fortemente connessa con più di un nodo contiene almeno un ciclo orientato;
- un grafo fortemente connesso ha una sola componente fortemente connessa e viceversa;
- un grafo con N nodi aciclico ha N componenti fortemente connesse e viceversa.

La ricerca delle componenti fortemente connesse massimali può essere realizzata in tempo $O(N + M)$ dove N rappresenta il numero dei nodi ed M il numero degli archi, ed è la soluzione senza dubbio più conveniente in termini di complessità se il grafo è sparso [4]. Infatti tale ricerca fornisce *tutti* i cicli orientati e non aggrava in alcun modo l'algoritmo di ricerca, se questo può essere eseguito. Tuttavia tale metodo è utile solo quando la presenza di cicli è indicativa di per se stessa. Questo è vero quando cerchiamo il percorso massimo in un grafo con costi non negativi o il minimo in un grafo con costi non positivi, ma non è utile se il

grafo contiene costi positivi e negativi. Chiameremo questo algoritmo appena illustrato *metodo A*.

2.2.2 Controllo basato sul costo massimo possibile

L'idea di questo metodo è quella di verificare se esiste un nodo a cui sia associato un costo maggiore di quello massimo possibile in un grafo senza cicli a costo negativo. Infatti se non ci sono cicli in un grafo con N nodi e C è il costo dell'arco negativo più piccolo, nessun nodo potrà mai avere un percorso che porta ad esso con costo associato minore di NC. Se quindi un nodo assume costo inferiore, esisterà senz'altro un ciclo orientato [2].

Tale controllo richiede un tempo $O(1)$, tuttavia potrebbe essere necessario far girare molto a lungo l'algoritmo prima che si arrivi ad un costo che soddisfi questa condizione, soprattutto se l'arco di costo C non fa parte del ciclo orientato e quelli presenti a costo negativo hanno pesi molto più piccoli in modulo di C. Inoltre questo controllo non individua il ciclo orientato ed è compatibile con qualunque struttura dati venga impiegata nell'algoritmo. Chiameremo questa strategia di ricerca dei cicli *metodo B*.

2.2.3 Controllo basato sul numero massimo di aggiornamenti

Tale controllo può essere realizzato solo quando si utilizza una struttura con politica *FIFO* per l'algoritmo. Infatti, come si è detto, è stato dimostrato che in questo caso un nodo non viene aggiornato più di $N-1$ volte se non ci sono cicli orientati negativi [2]. Se un nodo viene aggiornato N volte, esiste sicuramente un ciclo. Questa realizzazione richiede un tempo $O(1)$. La sua velocità per rintracciare il ciclo dipende dalla dimensione, in numero di nodi, del ciclo stesso: con un ciclo negativo grande quanto tutto il grafo, si avrà il caso peggiore. Neppure questo controllo individua il ciclo orientato negativo. Chiameremo questo *metodo C*.

2.2.4 Controllo basato sulla ricerca dei cicli online

Un'ultima possibilità per la ricerca online dei cicli è data dal fatto che in presenza di cicli l'algoritmo, dopo un certo numero finito di passi, mantiene nella struttura

utilizzata soltanto i nodi che fanno parte del ciclo stesso. Quindi è sufficiente scorrere l'elenco dei predecessori di un nodo ogni *TOT* passi e verificare che tale elenco non conduca al nodo considerato in quel passo. Scorrere questa lista ha una complessità $O(N)$ e se *TOT* è sufficientemente grande in modo che questa verifica venga fatta un numero di volte molto minore di M, la complessità complessiva rimane $O(NM)$ [2]. Tale algoritmo è il più dispendioso ma anche il più informativo in quanto è in grado di segnalare, oltre alla presenza del ciclo, anche il ciclo stesso. Chiameremo questo *metodo D*.

2.2.5 Confronto tra i metodi di ricerca dei cicli

La seguente tabella riporta un confronto tra i principali metodi di ricerca dei cicli.

METODO	A	B	C	D
Online / Offline	Offline	Online	Online	Online
Richiede una struttura FIFO	NO	NO	SI	NO
Dipende dalla struttura del grafo	NO	SI	SI	NO
Richiede costi con lo stesso segno	SI(NO)	NO	NO	NO
Individua il ciclo	SI	NO	NO	SI
Complessità	O(N + M)	O(1)	O(1)	O(N)

Tabella 1: Confronto tra i metodi di ricerca dei cicli orientati.

Il metodo A riporta un NO tra parentesi in quanto si può condurre un'analisi delle componenti fortemente connesse per stabilire se esiste un ciclo a costo negativo. Tale analisi ha un costo che la rende sconveniente in ogni caso. Se non si è interessati alla conoscenza dell'eventuale ciclo negativo o è del tutto improbabile che ci sia un ciclo negativo, può essere conveniente utilizzare i metodi B o C se la struttura del grafo, utilizzando la conoscenza a priori disponibile, lo consente. Infatti il metodo B richiede archi con costi più o meno uniformi (almeno quelli negativi) e il metodo C richiede che gli eventuali cicli negativi siano composti da pochi nodi. A questo punto si può utilizzare l'algoritmo di ricerca delle componenti fortemente connesse solo dopo che è stato individuato un ciclo. Se è sempre richiesta la determinazione dell'eventuale ciclo negativo o se è probabile che venga richiesta, oppure se non si hanno informazioni sulla natura del grafo o se non

sono accettabili per i metodi B e C, si dovrà scegliere tra il metodo A e il metodo D: il primo conviene senz'altro se i costi hanno lo stesso segno.

2.3 Scelte effettuate

Si è scelto di implementare un solo algoritmo sia per il percorso minimo, sia per il percorso massimo, eseguendo a priori l'algoritmo di ricerca delle componenti fortemente connesse alla luce dei risultati a cui si doveva pervenire e della conoscenza del problema. In particolare:

- si è scelto un solo algoritmo in quanto la complessità dell'algoritmo di Dijkstra è analoga quando il grafo è molto sparso che è il caso comune in questo tipo di applicazione;
- si è scelto di eseguire a priori una ricerca delle componenti fortemente connesse in quanto i costi hanno lo stesso segno. Inoltre, essendo il grafo molto sparso, la complessità è dell'ordine $O(N + aN)$ con a poco superiore ad 1.

L'algoritmo per la ricerca delle componenti fortemente connesse è inoltre usualmente presentato in forma ricorsiva come modifica della ricerca depth-first [4]. Si è scelto tuttavia, di convertire la versione ricorsiva in una iterativa per garantire una maggiore robustezza ed evitare i problemi a cui va incontro usualmente un algoritmo ricorsivo.

Il manuale dell'utente

Il manuale dell'utente è suddiviso in tre parti. La prima parte illustra i parametri che è necessario passare al programma per ottenere i risultati desiderati. Nella seconda parte viene descritto il formato del file in ingresso, mentre nella terza parte viene illustrato il formato del file in uscita.

3.1 I parametri alla linea di comando

Il programma accetta dei parametri alla linea di comando che ne guidano l'esecuzione. Dopo il nome del file eseguibile, i parametri specificati permettono di effettuare tre azioni:

- chiedere l'help della linea di comando;
- procedere all'analisi del grafo;
- eliminare alcune transizioni dal grafo.

La prima alternativa può essere eseguita digitando semplicemente `-h` o `-help` dopo il nome del file eseguibile. Per procedere all'analisi del grafo, occorre fornire il nome e il percorso del file `*.tg` contenente la descrizione del grafo, il nome e il percorso del file di uscita che conterrà l'analisi, l'opzione `-s` seguita dal numero della transizione di partenza e l'opzione `-t` seguita dal numero della transizione di arrivo. Per eliminare delle transizioni dal grafo, occorre assegnare il nome e il percorso del file `*.tg` contenente la descrizione del grafo, il nome e il percorso del file `*.tg` di uscita che non conterrà più quelle transizioni e l'opzione `-d` seguita dagli identificatori numerici delle transizioni da eliminare. Il file di uscita sovrascrive qualunque file presente con lo stesso nome.

3.1.1 La sintassi completa della linea di comando

Per completezza viene riportata la sintassi formale della linea di comando:

```
      Help della linea di comando

Sintassi:

      ([unità:][percorso]<fileSorgente>
       [unità:][percorso]<fileDestinazione>
      (-d {<numeroTransizioni>}+ |
      (-s <tStart> -t <tTarget>))) |
      -h | -help

Parametri:

[unità:][percorso]
      Specifica l'unità e la directory in cui si trova il file.

<fileSorgente>
      Nome del file da cui deve essere letto il grafo.

<fileDestinazione>
      Nome del file in cui deve essere scritto l'output.

-d
      Indica che i numeri interi non negativi che seguiranno
      sono gli identificatori delle transizioni da eliminare.

{<numeroTransizione>}+
      Indica che ci deve essere almeno una transizione da elimina-
      re.Ogni transizione è identificata da un numero non negati-
      vo.

<numeroTransizione>
      Indica il numero della transizione da eliminare.

-s
      Permette di inserire il numero della transizione di Start.

<tStart>
      Indica il numero della transizione di Start.

-t
      Permette di inserire il numero della transizione Target.

<tStart>
      Indica il numero della transizione Target.
```

3.1.2 Tolleranza agli errori alla linea di comando

In caso di errata composizione della riga di comando, l'analizzatore della linea di comando produce i seguenti errori e riporta l'help:

- ✓ *Non è presente nessun argomento alla linea di comando!* Significa che non è presente nessun argomento alla linea di comando: c'è solo il nome del file eseguibile.
- ✓ *L'unico parametro presente non è supportato! Se vi è un solo parametro, deve essere* `-h` *oppure* `-help`*!* Significa che vi è un solo parametro alla linea di comando diverso da quelli permessi.
- ✓ *Errore alla linea di comando! Non ci possono essere due argomenti!* Non sono permessi due argomenti alla linea di comando.
- ✓ *Errore alla linea di comando! Non ci possono essere tre argomenti!* Non sono permessi tre argomenti alla linea di comando.
- ✓ *Il file di origine coincide con quello di destinazione !!! Modificare il nome del file di destinazione.* Il nome e il percorso del file che contiene la descrizione del grafo coincidono con il nome e il percorso del file di uscita.
- ✓ *Dopo le opzione* `-s` *e* `-t` *ci devono essere solo numeri interi non negativi!* Gli identificatori delle transizioni sono numeri interi non negativi, quindi non è possibile fornire un numero negativo in questi campi.
- ✓ *Dopo l'opzione* `-d` *ci devono essere solo numeri interi non negativi!* Gli identificatori delle transizioni sono numeri interi non negativi, quindi non è possibile fornire un numero negativo in questo campo.
- ✓ *Uno dei numeri delle transizioni è più grande di quello della massima transizione presente nel grafo!* Indica che il numero di una delle transizioni inserite è più grande di quello massimo presente nel grafo.
- ✓ *Errore alla linea di comando!* Indica un generico errore alla linea di comando: ad esempio, ci sono delle opzioni non supportate o degli spazi tra il carattere − e la lettera successiva.

Nel caso che dopo l'opzione `-d` ci siano più transizioni da eliminare, i loro numeri identificativi devo essere spaziati. Le opzioni `-h` e `-help` possono essere o meno attaccate al nome del file eseguibile. I caratteri che seguono il simbolo − devono essere invece attaccati a tale simbolo.

3.2 Il formato del file in ingresso

Il programma accetta in ingresso un file di testo `*.tg` contenente la descrizione del grafo delle classi. Il file viene interpretato linea per linea, cercando di individuare ogni volta dei caratteri chiave per individuare il tipo linea.

All'inizio del file è presente un'intestazione che contiene il nome del file, il numero dei livelli e dei vertici del grafo. Due parentesi graffe, una aperta e l'altra chiusa, separate da uno spazio, delimitano l'intestazione dall'insieme dei nodi. Dopo l'intestazione si trovano tutti i nodi che compongono il grafo separati l'un l'altro da un new line; ogni nodo viene identificato da un numero intero a partire da zero. All'interno di un nodo si trovano il suo identificatore, il livello, il marcamento, l'insieme delle transizioni che identificano quella particolare classe di stati e l'insieme degli archi uscenti. Concluso l'insieme dei nodi, il file può terminare oppure può seguire l'analisi delle tracce che viene però ignorata.

Non tutte le informazioni presenti nel file sono utili per eseguire un'analisi approssimata di una traccia ed una parte viene quindi ignorata. Ad esempio, il simbolo `[n]` marca le transizioni newly enabled, informazione ininfluente per i nostri fini, mentre una transizione racchiusa tra parentesi tonde `()` informa l'utente che quella transizione è abilitata ma bloccata su delle risorse. Non interessano neppure i vincoli che contengono più di una transizione: tutte queste informazioni vengono quindi scartate durante la lettura del file.

Ecco un esempio di come deve essere interpretato un file `*.tg`:

```
Grafo di primaprova
322L & 1751S
{ }
L0.S0:4p4 p9 p12 3p14
      (600 < t0 < 600)
       300 < t2 < 300 [n]
   2.4e+06 < t5 < 2.4e+06
         2 < t9 - t4 < 3
        -0 < t8 < 0 [n]
t5->S5
t0->S4
```

La prima linea contiene il nome del grafo, `322` è il numero dei livelli mentre `1751` indica il numero degli stati. Le due parentesi graffe separano l'intestazione dalla lista dei nodi. Il nodo che segue va interpretato nel seguente modo: livello `0` stato `0`; marcamento: `4` gettoni sulla piazza `p4`, un gettone sulla piazza `p9` e così via; la transizione `t0` non può eseguire prima di `600` e dopo di `600` (il simbolo < minore rappresenta un minore uguale); così per `t2`, `t5` e `t8`; la linea con la differenza tra due transizioni rappresenta un vincolo che non ha interesse per i nostri fini; seguono poi le transizioni che possono eseguire: l'arco di uscita etichettato con `t5` porta dallo stato `0` allo stato `5`, mentre quello etichettato con `t0` conduce nello stato `4`. Durante la lettura del file, vengono estratte anche delle informazioni che sono inutili come il nome del file, il livello ed il marcamento. Il file può terminare con un'analisi delle tracce che viene però scartata.

I tempi minimo e massimo di ogni arco sono derivati dai vincoli sulle transizioni abilitate: il minimo di un arco è l'earliest time della transizione stessa, mentre il massimo è il minimo dei latest time tra tutte le transizioni che possono eseguire. Il nostro compito consiste nel calcolare la traccia da un evento trigger ad evento target, includendo il tempo di esecuzione dell'evento target ma non quello del trigger (da vertice ad arco), ovvero calcolare lo shortest e il longest path da ciascun vertice in cui si entra con il trigger a ciascun vertice in cui si entra con target.

3.2.1 I margini di variazione sul formato del file

Il file `*.tg` di ingresso si può discostare leggermente da quello appena descritto. La procedura di lettura tollera qualsiasi carattere di spaziatura aggiuntivo presente prima dell'intestazione, tra l'intestazione e il primo nodo e tra i vari nodi; all'interno di una linea, dove compare uno spazio, ne può comparire un numero arbitrario; un new line che spezza una riga che invece deve essere consecutiva provoca invece un messaggio di errore sul formato del file, così come un qualunque carattere alfanumerico presente in una posizione errata. Un nodo deve sempre cominciare con il livello, seguito dal suo identificatore e dal marcamento; le transizioni e gli archi si possono alternare in un ordine qualsiasi. I nodi devono essere ordinati secondo il loro identificativo in modo crescente a partire da zero. Il programma verifica anche se gli viene passato un file vuoto in ingresso.

3.2.2 Eventuali errori presenti sul file

Anche se il file `*.tg` è sintatticamente corretto, ci possono essere degli errori residui non visibili ad un primo sguardo. Durante la lettura del file, vengono segnalate le seguenti situazioni anomale:

- *Il nodo ha identificatore maggiore del numero di nodi.* Il numero del nodo letto è maggiore od uguale del numero dei nodi dichiarati nell'intestazione.
- *L'arco finisce in un nodo che ha identificatore maggiore del numero di nodi.* Il nodo di destinazione dell'arco è identificato da un numero maggiore od uguale del numero dei nodi dichiarati nell'intestazione.
- *Il formato del nodo non è corretto: un arco uscente non appartiene alla lista delle transizioni.* Non è possibile che un arco uscente da un nodo non faccia parte delle transizioni di quel nodo.
- *I nodi non sono ordinati in modo crescente secondo il loro identificatore.* Questa condizione è necessaria per poter riempire in modo corretto il grafo su cui verrà eseguita l'analisi.
- *Il numero dei nodi letti non coincide con quello riportato nell'intestazione.* Il numero dei nodi letti deve coincidere con quello riportato nell'intestazione del grafo.

La procedura di lettura del file ricerca anche l'etichetta numerica massima delle transizioni presenti in tutti i nodi del grafo: questa informazione è utile per verificare se alla linea di comando viene fornito un valore non valido superiore alla massima etichetta estrapolata dal file.

3.3 Il formato del file di uscita

Il programma produce due tipi di file in uscita: uno è semplice file `*.tg` contenente la descrizione di un grafo delle classi, mentre l'altro contiene l'analisi approssimata del grafo delle classi che gli è stato passato come parametro. Si ottiene un file `*.tg` in uscita quando viene richiesta l'eliminazione di alcune transizioni da un grafo delle classi: in output si ha lo stesso file di ingresso, senza però le transizioni eliminate.

Per ottenere in uscita un file di testo con l'analisi di una particolare insieme di tracce, è necessario fornire alla riga di comando il nome e il percorso del file contenente la descrizione del grafo delle classi, il nome e il percorso del file che conterrà l'analisi ed i numeri delle transizioni di partenza e di arrivo. Il file di testo contenente l'analisi è strutturato nel seguente modo: inizialmente è presente una sintesi che illustra in maniera concisa quali sono il percorso a costo minore e quello a costo maggiore; segue poi una sezione in cui vengono riportati i dettagli di entrambi i percorsi. Nella sezione che riporta i dettagli, per ogni nodo in cui termina la transizione di partenza, vengono elencati tutti i percorsi che partono da quel nodo e che finiscono in un qualunque nodo in cui finisce la transizione di arrivo.

Scendendo nei particolari, un percorso del tipo:

```
t1->S6[0]->t7->S9[0]->t9->S12[40000]
```

deve essere interpretato come segue: si parte dalla transizione `t1` che termina nello stato `S6`; il numero racchiuso tra parentesi quadre indica il costo del percorso dal nodo iniziale fino al nodo corrente: in questo caso si ha ovviamente `0`; andando avanti, dallo stato `S6` tramite la transizione `t7` si giunge nello stato `S9` con costo `0` (questo vuol dire che l'arco etichettato con `t7` ha costo nullo) e così via. L'ultimo numero racchiuso tra parentesi fornisce il costo totale del cammino dalla transizione di partenza a quella di arrivo; l'ultima transizione e l'ultimo nodo del percorso sono la transizione di destinazione e il nodo in cui termina.

Ovviamente il percorso più breve esiste sempre, mentre quello più lungo può entrare in un ciclo a costo positivo e divergere. In questo caso, invece di fornire in uscita il percorso più lungo, appare la scritta:

```
Il grafo contiene un ciclo orientato.
```

Tra i dettagli del percorso più lungo viene riportato l'insieme dei nodi che formano il ciclo orientato.

È anche interessante conoscere se una traccia presenta delle rientranze. Una rientranza in un percorso tra una transizione di origine e una di destinazione è la ripetizione della transizione di partenza all'interno del percorso che le congiun-

ge. Se è presente una rientranza anche non appartenente al percorso più breve o a quello più lungo, nella sintesi presente all'inizio del file compare la seguente espressione:

```
Ci sono rientranze tra i percorsi.
```

Andando ad esaminare la fine di ogni percorso, si può esaminare se è presente una rientranza in quella specifica traccia: se è effettivamente presente una rientranza, viene indicato il numero di volte che la transizione di partenza si ripete lungo il cammino, altrimenti non compare nessuna indicazione: questo vale sia per la sintesi iniziale, sia per i dettegli che seguono.

Un'ultima osservazione riguarda la possibilità di trovare nel file contenente l'analisi dei percorsi che consistono di un solo nodo, come ad esempio:

```
t1->S1173[0]
```

Questo significa che la transizione `t1` termina sul nodo `S1173` e che da quel nodo non è possibile giungere in nessun nodo in cui termina la transizione di destinazione.

3.4 Un esempio

Riportiamo di seguito un esempio di utilizzo della sintassi della riga di comando nelle due modalità previste. Il nome del file eseguibile è `analisi`, il file `*.tg` in ingresso è `modello.tg`, il file con l'analisi è `output.txt`, il file `*.tg` modificato è `modello2.tg`.

```
analisi modello.tg output.txt -s 1 -t 10
```

Questo è un esempio in cui si analizza il grafo con transizione di start 1 e transizione di target 10: se compaiono cicli la seguente linea elimina alcune transizioni.

```
analisi modello.tg modello2.tg -d 5 3 7
```

Il manuale dell'amministratore

I compiti dell'amministratore del sistema riguardano cosa deve essere fatto per poter eseguire il programma: ad esempio se occorrono dei file di configurazione ed eventualmente la loro locazione. Nel caso del nostro programma, uno dei possibili incarichi che rientra tra i compiti dell'amministratore riguarda la generare del file eseguibile a partire dal makefile e dai file sorgente; il programma, infatti, non interagisce con nessun altra applicazione: è quello che viene detto un programma "*stand alone*".

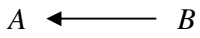
Per quanto riguarda la generazione del file di uscita, se esiste nello stesso percorso un file con il medesimo nome, quest'ultimo viene sovrascritto e quindi si perdono tutti i dati precedentemente memorizzati in quel file. Nel caso si generi un file di uscita contenente l'analisi del grafo, tale file viene costruito a partire da quattro file temporanei che poi vengono cancellati. Il contenuto di questi quattro file temporanei e i loro nomi sono i seguenti:

- `tmp002_aa.tmp`: contiene la sintesi del percorso più breve;
- `tmp002_bb.tmp`: contiene la sintesi del percorso più lungo.
- `tmp001_aa.tmp`: contiene i dettagli del percorso più breve;
- `tmp001_bb.tmp`: contiene i dettagli del percorso più lungo.

Se quindi nello stesso percorso esiste un file con uno dei precedenti nomi, tale file viene prima sovrascritto e poi cancellato. Il file contenente l'analisi viene creato copiando uno di seguito all'altro i quattro file temporanei nell'ordine precedente riportato.

4.1 Le dipendenze tra i file sorgenti

Per costruire il makefile è necessario conoscere le interdipendenze tra i file sorgenti che compongono il programma ed in particolare le loro inclusioni. La seguente figura illustra queste dipendenze, dove con la notazione



si intende che il file *B* include il file *A*:

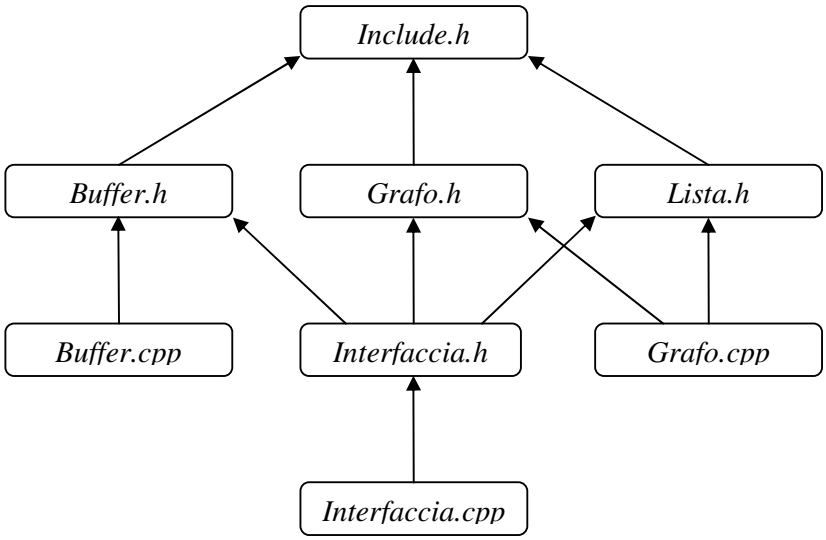


Figura 4: Illustrazione delle dipendenze tra i file sorgenti.

Il file `Include.h` contiene solo le inclusioni delle librerie standard e viene incluso da tutti gli header file; ogni file `*.cpp` include il corrispondente header file ad eccezione di `Grafo.cpp` che include anche `Lista.h`; infine l'header `Interfaccia.h` include tutti gli altri header file.

4.2 Il makefile

A partire dalle precedenti inclusioni ed esaminando le librerie standard incluse nel programma, è possibile creare il makefile per compilare e linkare i file sorgenti. Un possibile makefile è il seguente:

```
OBJECTS    = Grafo.o Buffer.o Interfaccia.o
outputfile = analisi
CC = gcc
$(outputfile) : $(OBJECTS)
  gcc -g -lstdc++ -Wall -o $(outputfile) $(OBJECTS)
Interfaccia.o : Buffer.h Lista.h Grafo.h
Grafo.o       : Lista.h Include.h
Buffer.o      : Include.h
```

Il manuale del programmatore

Questa applicazione è stata realizzata in C++ ANSI 3° standard (1995). Il progetto si compone di più parti tra loro interconnesse, tra le quali possiamo individuare tre blocchi che possono essere utilizzati separatamente:

- una sezione che contiene alcuni template per le strutture dati usate più di frequente;
- una sezione per la gestione dei grafi sparsi;
- una sezione per l'interpretazione dei file `*.tg`.

Mentre la prima è utilizzata all'occorrenza, le ultime due sono interconnesse tra loro tramite la classe `Interfaccia` per mezzo di un solo metodo, come si vedrà negli schemi riportati più avanti. Questo modo di procedere, oltre ad avvantaggiare la suddivisione dei compiti, permette una efficace riusabilità del codice in quanto la sezione sui grafi sparsi non tiene conto in alcun modo della esistenza dei file `*.tg`, tanto che non sono nemmeno previsti costruttori specifici; d'altra parte, la sezione sull'interpretazione del file `*.tg`, si occupa di creare in memoria una struttura che contenga tutte le informazioni scritte nel file, trascurando completamente che queste informazioni serviranno poi per riempire una struttura per grafi sparsi.

Questo modo di procedere permette e facilita notevolmente la fase di test del programma e garantisce un sistema molto più robusto. Ad esempio, per testare la correttezza dell'interpretazione del file `*.tg`, si è pensato di stampare su file la struttura in memoria contenente il file, assieme ad altre informazioni. Una perfetta corrispondenza tra il file in ingresso e quello prodotto in uscita, garantisce che in memoria ci sia la corretta interpretazione del file.

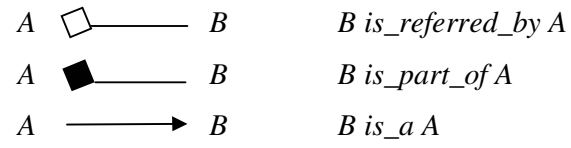
Lo sviluppo dell'applicazione ha cercato di seguire il più possibile lo spirito della programmazione orientata agli oggetti. In particolare:

- tutti gli attributi sono stati incapsulati e per alcuni particolarmente critici è stato bloccato l'accesso anche alle classi derivate. I metodi `set` non sono presenti, o non in modo banale, per tutti gli attributi. Per alcuni attributi non esiste nemmeno il metodo `get`;
- non sono state usate funzioni `friend`;
- non sono visibili tutti quei metodi il cui uso non appropriato potrebbe danneggiare l'integrità della classe;
- nella parte di interpretazione del file si è cercato di contemplare svariate situazione di errore nel formato del file, arrestando in modo sicuro il programma o risolvendo l'errore se possibile.

Illustriamo adesso le tre parti principali di cui si compone il programma, quindi la classe `Interfaccia` di collegamento. Gli schemi riportati sono in notazione UML-like (Unified Modeling Language) [5]. Ogni classe è rappresentata da un rettangolo diviso in tre parti contenenti rispettivamente il nome, gli attributi e i metodi. All'interno di una classe, il simbolo:

– indica degli attributi o dei metodi privati;
indica degli attributi o dei metodi protetti;
+ indica degli attributi o dei metodi pubblici.

I metodi sono contraddistinti da `()`, ma i parametri non sono stati specificati per problemi di spazio. Sia per i metodi che restituiscono un valore, sia per gli attributi è stato specificato il tipo. Le relazioni tra le classi sono espresse nel seguente modo:



In ogni classe sono riepilogati tutti gli attributi, mentre per quanto riguarda i metodi non sono riportati i costruttori, i distruttori, i metodi `set` e i metodi `get`.

5.1 I file e le classi

Di seguito si riportano i file in cui è suddiviso il programma ed il loro contenuto:

- *Buffer.h*: contiene le dichiarazioni delle classi descritte nel paragrafo 5.2. Contiene le funzioni `inline` di quelle classi.
- *Grafo.h*: contiene le dichiarazioni delle classi descritte nel paragrafo 5.3. Contiene le funzioni `inline` di quelle classi.
- *Lista.h*: contiene le dichiarazioni delle classi e dei template descritti nel paragrafo 5.4. Contiene le funzioni `inline` di quelle classi e di quei template.

I corpi dei metodi e delle funzioni globali sono definiti in file `*.cpp` con lo stesso nome del file `*.h` in cui è presente la dichiarazioni. Per le dipendenze e le inclusioni fare riferimento al manuale dell'amministratore [pagg. 23 e seguenti]. Un esempio di makefile può essere trovato ancora nello stesso manuale dello amministratore.

Per la sintassi dei file in ingresso ed in uscita fare riferimento al manuale dell'utente [pagg. 12 e seguenti].

5.2 L'interpretazione del file `*.tg`

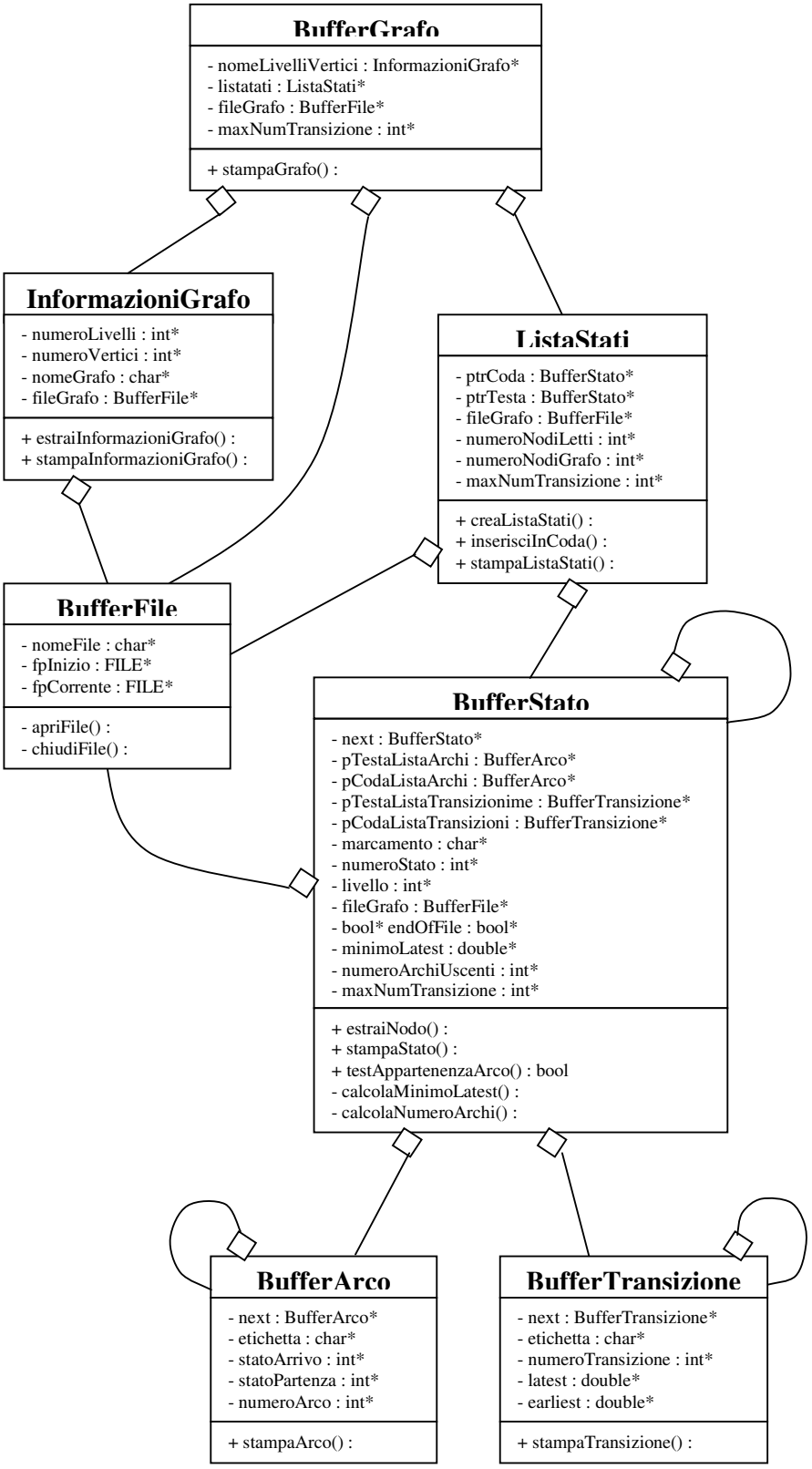


Figura 5: Albero delle classi che servono da buffer per il riempimento del grafo.

La classe fondamentale di questo gruppo di classi è la classe `BufferGrafo` che contiene tutta l'interpretazione del file `*.tg`. Al suo interno compaiono tre puntatori alle classi `BufferFile`, `InformazioniGrafo` e `ListaStati`. La classe `ListaStati` fa riferimento alla classe `BufferStato` per quanto riguarda l'insieme dei nodi che formano il grafo. Infine la classe `BufferStato` contiene due liste, una contenente gli elementi della classe `BufferArco` e l'altra gli elementi della classe `BufferTransizione`.

5.2.1 La classe `BufferFile`

Questa classe viene associata ad ogni file aperto in lettura o in scrittura. I suoi attributi identificano il nome del file, il file pointer che punta alla posizione corrente e il file pointer che punta sempre all'inizio del file. I due costruttori presenti si differenziano attraverso il parametro che rappresenta il nome del file che deve essere aperto: il costruttore che non ha questo parametro, chiede il nome del file allo utente durante l'esecuzione del programma, mentre l'altro legge il nome del file dalla riga di comando. L'altro parametro presente nei costruttori indica se il file deve essere aperto in lettura o in scrittura. Il costruttore si preoccupa di invocare la procedura di apertura del file che verifica anche la correttezza dell'operazione. Il distruttore dealloca i membri precedentemente allocati e chiude il file, verificando la presenza di eventuali errori.

5.2.2 La classe `BufferArco`

Rappresenta un arco uscente da un nodo del grafo delle classi. I suoi membri includono un identificatore dell'arco, un nodo di partenza e uno di destinazione, un puntatore ad un arco successivo utile per creare una lista di archi e un'etichetta. Questa classe non contiene nessun metodo particolare: il costruttore si limita ad assegnare agli attributi i parametri che gli vengono passati e il distruttore li dealloca. L'unico attributo modificabile dall'esterno è il puntatore `next` che serva a creare una lista di archi. Tra i metodi esiste una procedura di stampa dell'arco che permette di stampare l'arco nel solito formato `*.tg` o in un formato contenente con più informazioni.

5.2.3 La classe `BufferTransizione`

Implementa una transizione presente nei nodi del grafo. I suoi attributi includono un identificatore della transizione, due `double` che rappresentano l'earliest time e il latest time, un'etichetta e un puntatore che permette di creare una lista di transizioni. Il costruttore si limita ad assegnare i parametri che gli vengono passati agli attributi della classe, mentre il distruttore li dealloca. È presente un metodo per la stampa della transizione e come nella precedente classe l'unico attributo modificabile dall'esterno è `next` che serve per realizzare una lista di transizioni.

5.2.4 La classe `BufferStato`

È la classe più ricca di attributi e di metodi ed è quella che costituisce i blocchi fondamentali per l'interpretazione del file `*.tg`. Uno stato è composto essenzialmente da un identificatore, un livello, un marcamento, una lista di archi e una di nodi e da un puntatore ad un nodo successivo che serve a realizzare una lista. Per le due liste presenti, esistono un puntatore alla testa e alla coda che servono per poter realizzare la procedura di inserimento in coda, così da mantenere l'ordine degli archi e delle transizioni presente nel file. Per una descrizione dettagliata degli attributi si rimanda alla definizione della classe. Il costruttore si limita ad allocare la memoria necessaria per i vari attributi, inizializzandone i valori, mentre il distruttore la dealloca. Il metodo fondamentale è `estraiNodo()` che legge un nodo dal file `*.tg`. Vi sono un metodo per aggiornare il puntatore al nodo successivo, una procedura per calcolare il numero di archi uscenti ed una per individuare il minimo dei latest time. Durante la lettura di un nodo, vengono anche aggiornate tutte quelle informazioni necessarie per capire se siamo giunti alla fine del file e per eseguire tutti quei controlli di correttezza descritti nella sezione riguardante il formato del file `*.tg`. Il metodo di stampa permette di selezionare tramite il vettore `eliminare` quali transizioni devono essere eliminate dal grafo.

5.2.5 La classe `ListaStati`

Rappresenta la lista degli stati che compongono il grafo delle classi. Tra i suoi attributi troviamo un puntatore alla testa ed uno alla coda di una lista di nodi, il numero di nodi letti dal grafo, il numero di nodi dichiarati nell'intestazione del grafo

e il massimo identificatore di tutte le transizione lette. Il puntatore alla coda serve per poter realizzare il metodo di inserimento in coda, necessaria per mantenere l'ordinamento dei nodi presenti nel file, mentre gli altri attributi servono per poter eseguire i controlli di correttezza del formato del file. Il costruttore ha come parametri il file da cui leggere il grafo e il numero di nodi dichiarato nella intestazione. Il distruttore si occupa di deallocare tutta la lista dei nodi richiamando il distruttore della classe `BufferStato` e tutti gli attributi presenti. Il metodo `creaListaStati()` crea la lista degli stati leggendo un nodo alla volta dal file, inserendo tale nodo in coda alla lista degli stati ed eseguendo alcuni controlli di correttezza.

5.2.6 La classe `InformazioniGrafo`

Contiene l'intestazione presente all'inizio di ogni file `*.tg`. I suoi attributi sono il nome del file, il numero dei vertici ed il numero di livelli del grafo. Il costruttore alloca la memoria necessaria, mentre il distruttore la dealloca. Il principale metodo è `estraiInformazioniGrafo()` che legge l'intestazione dal file.

5.2.7 La classe `BufferGrafo`

Questa classe contiene tutta l'interpretazione del file `*.tg` in ingresso. Scorrendo i suoi attributi, si possono estrarre tutte le informazioni riguardanti il grafo delle classi. Tra i suoi attributi troviamo un puntatore alle informazioni del grafo, una lista di stati, un intero che indica qual è il massimo identificatore di tutte le transizioni presenti nel grafo che serve per controllare l'esattezza della linea di comando ed un puntatore alla classe `BufferFile` che indica il file che descrive il grafo. Tra i metodi troviamo due procedure di stampa che si differenziano per il fatto che una estrae le transizioni da eliminare dalla linea di comando, mentre l'altra le chiede all'utente durante l'esecuzione. Il costruttore si preoccupa di invocare il metodo per leggere l'intestazione del file e quello per costruire la lista dei nodi.

5.3 Le classi per la gestione dei grafi sparsi

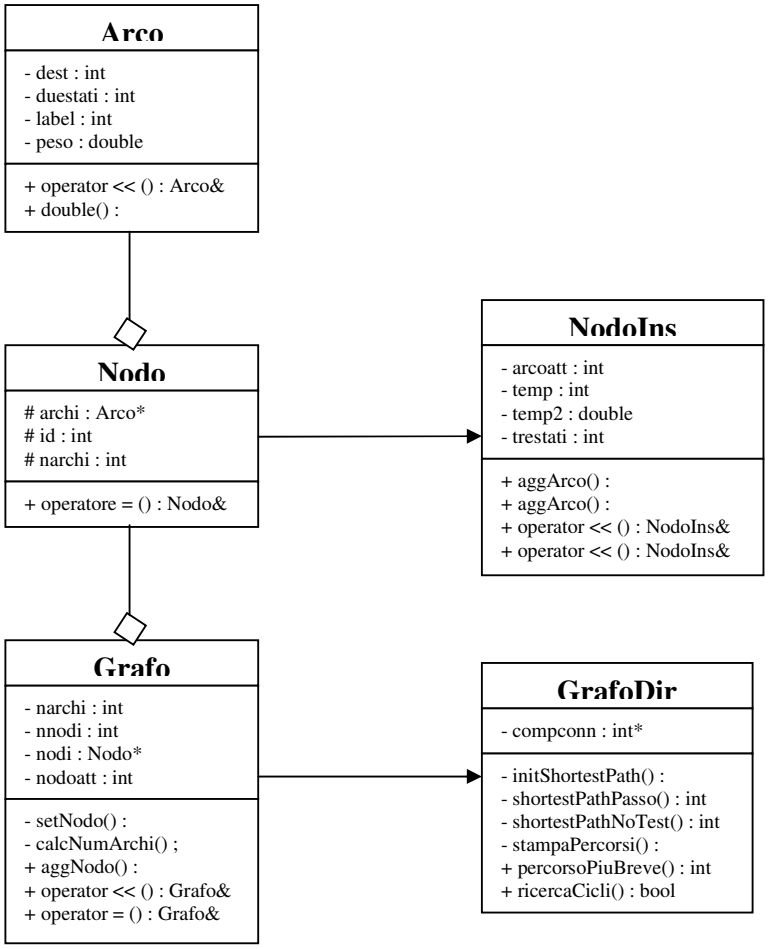


Figura 6: Albero delle classi per la gestione dei grafi sparsi.

La classe principale è la classe `GrafoDir` che contiene l'algoritmo per la ricerca del percorso più breve e quello per la ricerca delle componenti connesse. Tale classe non prevede alcun metodo per essere riempita: devono essere utilizzati i metodi della classe padre `Grafo`. Quest'ultima classe può facilmente essere riempita con dei nodi per mezzo dell'operatore << e potrebbe essere specializzata in altro modo, per esempio in `GrafoNonDir`, se volessimo utilizzare grafi non diretti.

La classe `Grafo` memorizza i nodi tramite un puntatore alla classe `Nodo`. Quest'ultima contiene i soli attributi necessari al nodo stesso, ma poiché erano utili alcuni attributi aggiuntivi per costruire più facilmente il nodo, si è specializzata la classe `Nodo` in `NodoIns`. La classe `Nodo` memorizza gli archi da esso uscenti tramite un puntatore alla classe `Arco`. Vediamo una panoramica delle varie classi.

5.3.1 La classe `Arco`

Implementa un arco utilizzando come attributi il nodo di destinazione specificato tramite un identificatore numerico, una etichetta numerica, un peso che può non essere intero. Per costruire un oggetto di questa classe, si possono seguire due vie:

- utilizzare il costruttore passando come parametri un'etichetta, un nodo di destinazione ed, opzionalmente, il peso: se questo è assente viene assunto zero;
- utilizzare il costruttore vuoto e quindi impostare gli attributi con i metodi set, oppure utilizzando l'operatore <<: tale operatore riconosce ciclicamente un'etichetta, un nodo di destinazione ed un peso.

Un esempio dei due metodi è il seguente:

```
Arco a(1, 5, -2);
Arco a; a << 1 << 5 << -2;
```

Viene anche definito l'operatore di conversione da `Arco` a `double`: restituisce il costo dell'arco.

5.3.2 La classe `Nodo`

Tale classe implementa un nodo: i suoi attributi sono il numero di archi uscenti, un identificatore del nodo ed un vettore di archi. Un oggetto di questa classe può essere costruito come elemento vuoto; opzionalmente passando l'identificatore ed il numero di archi; passando l'identificatore, il numero di archi e il vettore di archi. Tuttavia è abbastanza inutile entrare nei dettagli in quanto è stata derivata un'apposita classe solo per definire gli attributi della classe `Nodo`.

Esiste un metodo virtuale per impostare il numero degli archi: tale metodo può essere anche utilizzato per eliminare tutti gli archi di un nodo passando come parametro zero. Questo è anche l'unico utilizzo "*pubblico*" che abbia un senso in quanto la classe non fornisce alcun metodo per impostare gli archi, con la eccezione del costo di un arco specifico. Non viene fornito alcun metodo per impostare o modificare l'identificatore. Questo perché in genere l'identificatore di un nodo non cambia durante la sua vita.

Vengono forniti i metodi `get` necessari per accedere a tutti gli attributi: in particolare il metodo `getArco(int quale)` permette di accedere all'arco '*quale+1*'-esimo del vettore degli archi uscenti. Poiché la classe contiene alcuni vettori dinamici, viene sovrascritto l'operatore di assegnamento di modo che non venga copiato il valore del puntatore al vettore, ma il contenuto del vettore e viene per lo stesso motivo definito esplicitamente il costruttore di copia.

5.3.3 La classe `NodoIns`

Questa classe specializza la classe `Nodo` e non ha alcun significato se non operativo. Infatti è utilizzata solo per definire gli attributi della classe `Nodo`, tramite una efficace sovrascrittura dell'operatore << che segue la sintassi riportata sotto:

```
NodoIns n;
n << <numero di archi uscenti>;
∀ arco uscente da n con etichetta l, destinazione d e costo c:
n << d << c << l;
oppure
n << <numero di archi uscenti>;
∀ arco uscente da n con etichetta l, destinazione d e costo c:
Arco a(l, d, c);
n << a;
```

Questo semplifica notevolmente la costruzione degli archi uscenti, in quanto in genere si dispone di tutte le informazioni riguardanti gli archi in una struttura diversa da quella qui usata. Inoltre notiamo che il vettore degli archi uscenti viene riempito automaticamente e non c'è bisogno di specificare se un arco è il 2° o il 4° arco uscente. Una volta costruito un `NodoIns`, questo potrà essere inserito in un oggetto della classe `Grafo` tramite l'operatore << per cui, nella nostra applicazione, viene allocata una sola istanza della classe `NodoIns` e soltanto quando deve essere riempito un oggetto `Grafo`. Come si può vedere infatti, la classe `Grafo` possiede un vettore di oggetti della classe `Nodo`, ma accetta in ingresso soltanto oggetti della classe `NodoIns`.

Oltre ai metodi indicati, la classe `NodoIns` fornisce altri metodi per impostare in vario modo gli attributi della classe `Nodo`. Questo modo di procedere

garantisce che gli oggetti della classe `Nodo`, avendo solo metodi `set` "*innocui*", rimangano "*sicuri*" da manipolazioni improprie.

5.3.4 La classe `Grafo`

La classe `Grafo` gestisce la creazione e la distruzione di un grafo non orientato: non fornisce alcun metodo per l'analisi del grafo stesso: questo perché si vogliono proteggere i dati del grafo da possibili manipolazioni errate utilizzando gli algoritmi per l'analisi. Questa classe ha come attributi il vettore dei nodi, il numero dei nodi, il numero degli archi ed una variabile temporanea `nodoatt` utilizzata per costruire il grafo. Tutti gli attributi sono dichiarati privati in modo che non possano essere utilizzati, in scrittura, nemmeno dalle classi derivate. Il numero degli archi viene calcolato tramite un metodo privato quando il grafo viene alterato. Oltre ai metodi `get` per accedere al valore di tutti gli attributi nascosti, eccetto `nodoatt` che è ad uso interno, questa classe fornisce l'operatore `<<` per aggiungere un nodo alla classe, o equivalentemente il metodo `aggNodo()`. Per considerazioni analoghe a quelle di ogni classe contente vettori dinamici, ridefinisce l'operatore di assegnamento e il costruttore di copia. ***L'assunzione fondamentale*** per un utilizzo corretto della classe `Grafo` consiste:

- nel numerare tutti gli *N* nodi del grafo utilizzando tutti i numeri da 0 a *N-1*;
- nell'inserire i nodi nella classe `Grafo` iniziando da quello numerato con 0.

In questo modo l'identificatore di ogni nodo del grafo corrisponde alla posizione nel vettore dei nodi e può essere utilizzato facilmente come funzione hash. Per inserire un nodo nella classe `Grafo`, questo deve essere di tipo `NodoIns` per i motivi illustrati in precedenza. Il vettore dei nodi tuttavia è di tipo `Nodo`: vengono così ignorati tutti gli attributi della classe `NodoIns` che una volta costruito il nodo sono inutili. Si proceda quindi nel seguente modo:

```
Grafo grafo;
Nodoins nodo;
∀ nodo del grafo
  costruisci nodo
  grafo << nodo;
end
```

5.3.5 La classe `GrafoDir`

La classe `GrafoDir` deriva da `Grafo` ed è la classe effettivamente utilizzata in quanto contiene gli algoritmi per risolvere il problema descritto. Possiede come attributo il vettore delle componenti connesse che può essere impostato dall'esterno. Non si è scelto di impostare tale vettore dall'interno in quanto potrebbero esistere grafi con la stessa struttura e quindi con lo stesso vettore. Questa classe contiene alcuni metodi non banali che adesso illustreremo nei dettagli.

Il metodo `shortestPathPasso()`

Accetta i seguenti parametri:

- una lista di nodi *LISTA*;
- il vettore dei costi o delle etichette correnti;
- il vettore dei nodi predecessori correnti;
- il vettore delle transizioni: contiene il valore della transizione che collega il nodo nella posizione i-esima con il suo predecessore.

Questo metodo esegue il passo basse dell'algoritmo per il percorso più breve su grafi generici. La struttura *LISTA* passata come parametro è una classe astratta che definisce i metodi virtuali `aggiungi()` e `rimuovi()`: tali metodi sono effettivamente implementati in derivazioni della classe *LISTA*. Così facendo si evita di scrivere più metodi qualora sia diversa soltanto la politica di selezione ed inserimento di un elemento nella struttura *LISTA*. Produce in uscita un intero contenente la componente fortemente connessa incontrata nell'analisi del nodo prelevato da *LISTA*. Se il vettore delle componenti fortemente connesse non è stato impostato o il nodo non appartiene a nessuna componente fortemente connessa, restituisce zero.

Il metodo `shortestPathNoTest()`

Implementa l'algoritmo per il percorso più breve in grafi generici [vedi paragrafo 2.1]. Utilizza il metodo precedente. Si preoccupa di inizializzare l'algoritmo e di ripetere il passo base fino a che non si incontra un ciclo o non esiste più alcun nodo con etichetta aggiornabile. Non effettua alcun test online per la presenza di cicli orientati negativi [vedi paragrafo 2.2]. Volendo inserire gli algoritmi che effettuano i test al volo può ancora essere sfruttato il metodo precedente. Accetta gli

stessi parametri del metodo precedente con in più il nodo da cui iniziare la ricerca dei percorsi più brevi. Restituisce zero se tutte le chiamate del metodo precedente sono zero, altrimenti il primo valore diverso da zero restituito dal metodo precedente.

Il metodo `percorsoPiuBreve()`

È l'interfaccia verso l'esterno dei metodi di ricerca. Accetta in ingresso:

- uno stream di uscita su cui produrre la sintesi dei risultati trovati;
- uno stream di uscita su cui produrre i dettagli dei risultati trovati;
- una pila contente tutti i nodi da cui deve partire la ricerca dei cammini ottimi;
- un flag opzionale per indicare quale struttura deve essere usata (`false` = Dequeue, `true` = Coda);
- un intero opzionale per indicare l'etichetta comune dell'arco incidente nei nodi inseriti nella pila. Nel nostro progetto tale numero corrisponde alla transizione di start;
- un vettore di flag opzionale che indica se il nodo i-esimo fa parte dell'insieme dei nodi target come altrove definito. Nel nostro caso tale insieme sarà costituito da tutti i nodi su cui incide la transizione target.

Questo metodo richiama il precedente e quindi stampa tutti i percorsi che terminano in un nodo dell'insieme target se questo è definito, altrimenti stampa i percorsi tra tutti i nodi di partenza e tutti i nodi del grafo. Difatti il metodo `shortestPathNoTest()` e qualunque altro metodo implementato che voglia essere ancora compatibile con questa funzione, riempie un vettore, detto "*dei predessori*". Tale vettore contiene per ogni nodo quello che lo precede nel cammino ottimale, oppure –1 se è il nodo di partenza: si può quindi ricostruire il percorso dalla fine all'inizio. In fase di stampa, il metodo `stampaPercorsi()` ha cura di invertire il percorso tramite una pila e di stamparlo dall'inizio alla fine. Nell'esaminare tutti i percorsi, tiene traccia di quello con costo minimo e che viene poi stampato nella sintesi.

5.4 I template di uso generale

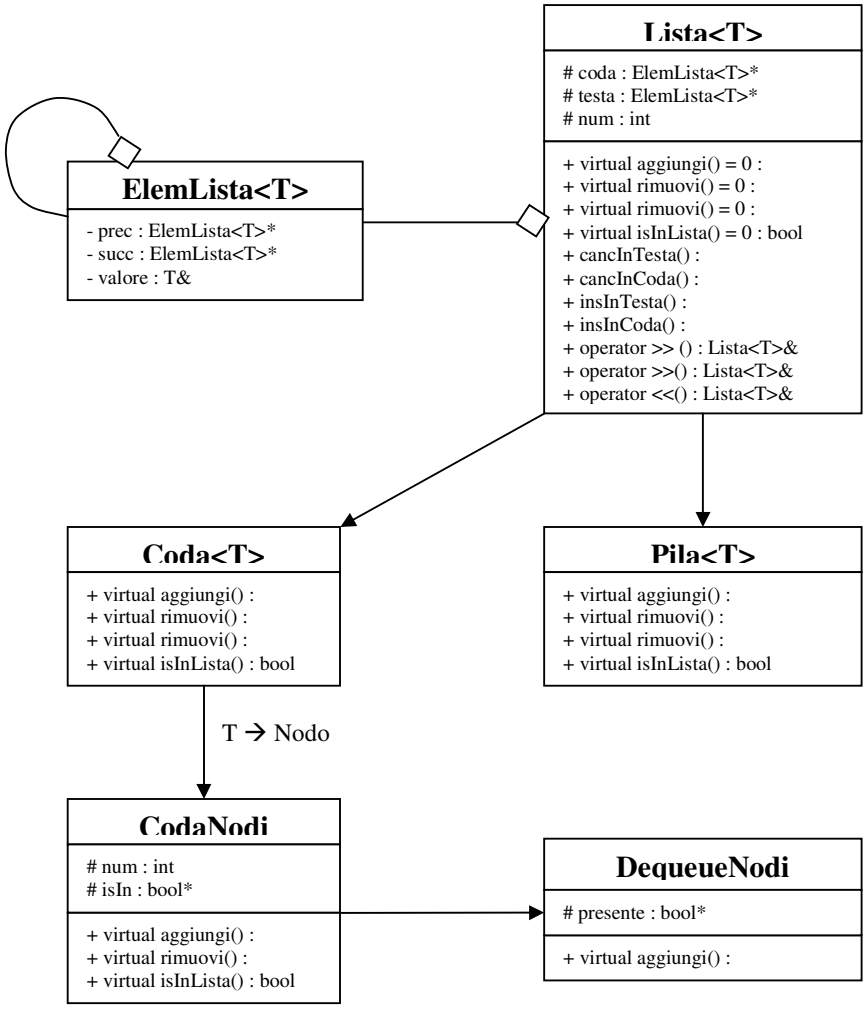


Figura 7: Albero delle classi dei template di uso generale.

Questa sezione contiene una serie di classe template per implementare una lista con differenti politiche di selezione e inserimento. In particolare la classe template `Lista<T>` viene specializzata nei due template `Pila<T>` e `Coda<T>`. Quest'ultima viene ulteriormente specializzata per implementare una coda di nodi o una dequeue.

5.4.1 La classe template `ElemLista<T>`

Tale classe è l'elemento base della classe `Lista<T>`: contiene un puntatore all'elemento successivo, un puntatore all'elemento precedente ed il valore di tipo parametrizzato `T` memorizzato tramite un riferimento: è importante notare che gli

elementi memorizzati in questa classe e quindi in tutte le liste *non* sono duplicati e quindi la distruzione della lista *non* comporta la distruzione degli elementi che la compongono a meno che non sia richiesto esplicitamente. Oltre agli attributi citati, contiene tutti i metodi per accedere a questi. Richiede per un corretto funzionamento che il tipo `T` abbia un operatore di assegnamento ed un costruttore di copia.

5.4.2 La classe template astratta `Lista<T>`

Implementa una generica lista di cui non viene specificata alcuna politica di inserimento/selezione: questi metodi vengono tuttavia definiti virtuali puri. Un altro metodo virtuale puro è quello per stabilire la presenza di un elemento dato nella lista stessa. Questi tre metodi andranno sovrascritti in un tutte le classi che si vogliono rendere utilizzabili.

Come attributi questa classe possiede due puntatori, quello di testa e quello di coda di tipo `ElemLista<T>` ed il numero `num` di elementi presenti nella lista.

Possiede i metodi per cancellare, inserire in testa o in coda ed inoltre sovrascrive l'operatore << per inserire un elemento nella lista e l'operatore >> per estrarne uno. Questi operatori sono definiti tramite le funzioni virtuali pure. L'operatore >> viene sovrascritto in due modi differenti:

1. se a destra c'è una variabile di tipo `T` o `T&`, viene restituito il valore dell'elemento selezionato;
2. se a destra c'è una variabile di tipo `T*`, viene restituito il puntatore dell'elemento selezionato.

Facciamo un esempio e sia `Coda<T>` una classe template derivata da `Lista` non virtuale pura.

```
Coda<int> coda;
int a;
a = 3;
coda << a;
a = 5;
int b;
coda >> b;
```

Poiché `ElemLista<T>` contiene solo il riferimento, e non il valore, se modifichiamo il valore di una variabile inserita nella coda, si modifica anche il valore di quella in coda e quindi in questo caso `b` vale 5.

```
Coda<int> coda;
int a = 3;
coda << a;
int b, *c;
coda >> b;
coda << *a;
coda >> c;
```

A questo punto valgono le seguenti uguaglianze e disuguaglianze:

- `&a = c; &a ≠ &b;`
- `a = *c = b;`

Questo permette di inserire elementi allocati dinamicamente al momento dell'inserimento, potendo poi recuperare e deallocare il puntatore al momento del prelievo, utilizzando la seconda forma per prelevare dalla lista.

5.4.3 La classe template `Coda<T>`

Questa classe non possiede attributi e specializza il template `Lista<T>` in modo da prelevare dalla testa ed inserire in coda, secondo una politica *FIFO*. Per un esempio di utilizzo di questa classe, si veda il paragrafo 5.4.2.

5.4.4 La classe template `Pila<T>`

Questa classe non possiede attributi e specializza il template `Lista<T>` in modo da prelevare ed inserire in testa, secondo una politica *LIFO*. Per un esempio di utilizzo di questa classe, si veda il paragrafo 5.4.2.

5.4.5 La classe `CodaNodi`

Specializza il template `Coda<T>` assumendo `T = Nodo`. Tale coda è particolare in quanto viene specificato a priori il numero massimo di elemento possibili e

viene utilizzato un vettore di flag per stabilire se un nodo appartiene o meno alla lista. In questo modo è possibile verificare l'appartenenza alla struttura di un oggetto `Nodo` in un tempo $O(1)$ invece del consueto $O(N)$ se N è il numero di elementi presenti nella lista.

5.4.6 La classe `DequeueNodi`

Specializza la classe precedente: questa struttura mantiene un vettore di flag per stabilire se un nodo ha mai fatto parte della struttura. A seconda di questo flag inserisce in testa - flag `false` - o in coda - flag `true` -.

5.5 L'interconnessione tra le principali classi

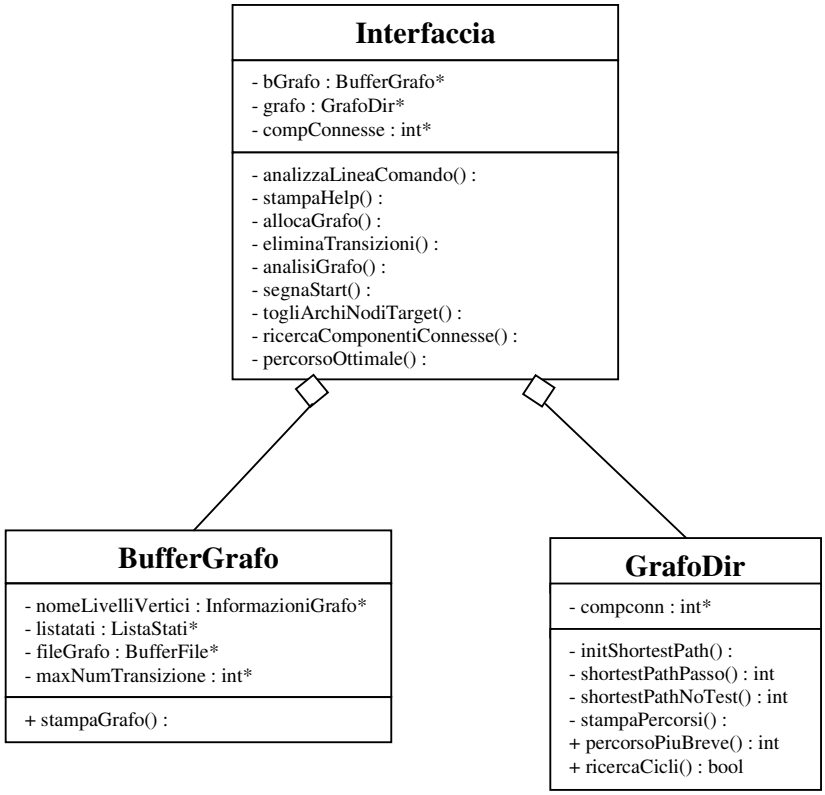


Figura 8: Interconnessione tra le principali classi.

La classe `Interfaccia` realizza l'interconnessioni tra le due principali classi dell'applicazione, `BufferGrafo` e `GrafoDir`. È a questo livello che si realizza quel meccanismo che permette di costruire un'istanza della classe `GrafoDir` a partire da un'istanza della classe `BufferGrafo`.

5.5.1 La classe `Interfaccia`

Oltre alle caratteristiche già menzionate, questa classe si occupa di analizzare la linea di comando e di eseguire le chiamate necessarie a svolgere i compiti richiesti. Tra i suoi attributi troviamo un'istanza di ognuna delle due classi `BufferGrafo` e `GrafoDir` e un vettore di interi che indica a quale componente fortemente connessa appartiene ogni nodo del grafo. Il costruttore della classe si limita a passare i parametri presenti alla linea di comando al metodo per l'analisi di quest'ultima: il `main()` dell'applicazione contiene solo la chiamata a questo costruttore. L'analizzatore verifica innanzitutto se ci sono stati degli errori nella compilazione della linea di comando, poi esegue le opportune chiamate di funzione. Il metodo `stampaHelp()` stampa la sintassi formale della linea di comando con una descrizione dei vari parametri. La procedura `eliminaTransizioni()` alloca un'istanza della classe `BufferGrafo` e poi la stampa su file eliminando le transizioni passate alla linea di comando. Il metodo `allocaGrafo()` realizza la connessione tra due parti indipendenti del programma. Conoscendo il nome del file in cui si trova la descrizione del grafo, si costruisce un'istanza della classe `BufferFile`; a questo punto è possibile allocare un'istanza della classe `BufferGrafo` da cui, tramite un'opportuna scansione, si crea una realizzazione della classe `GrafoDir`. A questo punto si può procedere con l'analisi del grafo chiamando il metodo `analisiGrafo()`, che inizia eliminando gli archi uscenti dai nodi in cui termina la transizione di destinazione. Per eseguire l'identificazione del percorso più breve, basta lanciare l'apposita procedura, mentre per individuare il percorso più lungo occorre prima ricercare le componenti connesse del grafo. Continuando con l'analisi dei metodi, `segnaStart()` trova i nodi in cui incide la transizione di partenza, mentre `ricercaComponentiConnesse()` individua a quale componente connessa appartengono i nodi del grafo. Infine il metodo `percorsoOttimale()` individua i percorsi ottimali del grafo e si occupa di costruire il file di uscita contenente l'analisi: avendo a disposizione la variabile `grafo`, non fa altro che chiamare il metodo `percorsoPiuBreve()` della classe `GrafoDir`.

Indice delle figure e delle tabelle

Figura 1: Esempio di grafo in cui l'algoritmo di Dijkstra fallisce. *5*
Figura 2: Algoritmo per i grafi con costi negativi. *6*
Figura 3: Esempio di grafo con dei cicli orientati. *7*
Figura 4: Illustrazione delle dipendenze tra i file sorgenti. *21*
Figura 5: Albero delle classi che servono da buffer per il riempimento del grafo. *26*
Figura 6: Albero delle classi per la gestione dei grafi sparsi. *30*
Figura 7: Albero delle classi dei template di uso generale. *36*
Figura 8: Interconnessione tra le principali classi. *39*

Tabella 1: Confronto tra i metodi di ricerca dei cicli orientati. *10*

Bibliografia

[1] M. Lusini, E. Vicario, “Static Analysis and Dynamic Steering of Time-Dependent Systems”, Dipartimento Sistemi e Informatica, 1998

[2] R.K. Ahuja ed altri, “Network Flows”, Prentice Hall, anno 1997

[3] B. Stroustrup, “Il Linguaggio C++”, Addison-Wesley, 1995

[4] R. Sedgewick, “Algoritmi in C++”, Addison-Wesley, 1993

[5] Booch, “Object Oriented Analysis and Design with Applications”, Addison Wesley, 1994

[6] C. Ghezzi e altri, “Ingegneria del Software”, Mondadori, 1997