UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

IMPLEMENTAZIONE DI UN SOLUTORE DI SISTEMI LINEARI BASATO SU MULTIGRID ALGEBRICO

Tutore: Prof. Antonio Frangioni

Candidato: Giorgio Fiderio

ANNO ACCADEMICO 2006–07

Ai mie genitori,
ai miei nonni,
ai miei amici
e a Katia

Indice

Prefazione

Il presente lavoro è stato realizzato come tirocinio per la laurea di primo livello, presso il dipartimento di Informatica dell'Università degli Studi di Pisa. Nonostante le difficoltà incontrate, posso ritenermi soddisfatto delle esperienze formative acquisite: ho raggiunto una buona dimistichezza nella programmazione in C++, nella gestione di progetti software di medio-grande dimensione e nell'utilizzo di sofisticate librerie esistenti. Devo a riguardo ringraziare il mio tutore Prof. Antonio Frangioni per la gentilezza e disponibilità concessami. Un ringraziamento va inoltre al Prof. Marco Donatelli per il materiale messomi a disposizione e per i validi suggerimenti.

Un grazie speciale è rivolto naturalmente ai miei genitori.

Giorgio Fiderio

Introduzione

I. Contesto

I problemi di Flusso di Costo Minimo costituiscono un'importante classe di problemi di ottimizzazione, per i quali ha interesse sviluppare approcci risolutivi molto efficienti. Una classe di approcci interessanti è quella basata su tecniche di tipo "interior point". Il punto critico di tali approcci è la necessità di risolvere efficientemente sistemi lineari con una matrice definita positiva di grandi dimensioni (pari al numero dei nodi del grafo).
Per tale compito sono stati tipicamente utilizzati algoritmi del Gradiente Coniugato Precondizionato, utilizzando diversi tipi di precondizionatori basati sull'idea di estrarre un opportuno sottografo facile (ad esempio un albero) del grafo originale [6].
Una possibilità alternativa è quella di usare una classe di tecniche note come *multigrid algebrico*. L'idea principale di queste tecniche è quella di risolvere il sistema applicando una procedura che prevede di rendere l'errore particolarmente smooth utilizzando un metodo iterativo classico, per proiettare poi, con una tecnica particolarmente semplice, il problema in uno di dimensione minore, ottenendo così un sistema più piccolo e facile da risolvere. La soluzione del sistema approssimante ottenuta viene poi riproiettata indietro sul sistema approssimato per ottenere una buona approssimazione finale della soluzione del sistema originale. Questa tecnica di proiezione può essere iterata fino a raggiungere una dimensione sufficientemente piccola in cui il sistema ottenuto possa essere risolto agevolmente anche con un metodo diretto.
Il multigrid è quindi in definitiva, la composizione di almeno due metodi iterativi, uno classico chiamato *smoother* ed uno di proiezione chiamato solitamente *coarse grid correction*.

II. Sintesi del lavoro svolto

Il tirocinio svolto può essere suddiviso in varie fasi distinte:

1. nella prima fase si sono compiuti una serie di studi preparatori, necessari per poter operare e compiere scelte implementative con cognizione di causa. In particolare è stato necessario documentarsi su:

   - i metodi multigrid per sistemi lineari strutturati [4, 2];
   - i metodi iterativi classici, con particolare attenzione al metodo di Jacobi, utilizzato nell'implementazione [1];
   - il metodo di Cholesky per la risoluzione diretta di sistemi lineari [1].

2. nella seconda fase si sono analizzate varie librerie di algebra lineare, al fine di scegliere quelle più idonee al progetto. La scelta, alla fine, è caduta sulla *Matrix Template Library* (in breve MTL) [9], una libreria a componenti generici e ad alte prestazioni che fornisce una vasta gamma di funzionalità per l'algebra lineare insieme a diverse varietà di rappresentazioni di base per matrici e vettori e, a complemento di questa, sulla *Iterative Matrix Library* (in breve ITL) [8]. Questa seconda libreria si occupa di implementare metodi iterativi per il calcolo di autovalori, autovettori e soluzioni di sistemi lineari. Il livello di astrazione di entrambe consente di combinarle in una incredibile varietà di modi, sfruttando metodi iterativi consolidati (Gauss-Seidel, Jacobi) o estendendone di nuovi per risolvere problemi rappresentati da matrici dense, sparse, o da arbitrarie strutture la cui semantica è definibile attraverso i costrutti che le rispettive interfacce mettono a disposizione.

3. nella fase successiva si sono analizzati i vari moduli in C++ facenti parte dell'ambiente dentro il quale è stato realizzato il lavoro e si è costruito una specializzazione della classe MCFLSSolver che implementa l'algoritmo del multigrid algebrico.

4. l'ultima fase è servita per testare l'algoritmo su un insieme già noto e sviluppato di istanze, e per confrontare i risultati così ottenuti con i migliori algoritmi disponibili (già implementati all'interno del framework) basati sugli algoritmi del Gradiente Coniugato Precondizionato.

III. Problematiche incontrate in corso d'opera

Nell'implementazione dell'algoritmo ci si è confrontati, nonostante l'utilizzo di un linguaggio abbastanza ad alto livello come il C++, con diverse problematiche, suddivisibili sostanzialmente in due categorie:

Problemi derivanti da codice di terze parti : avendo utilizzato, come detto, per le operazioni di algebra lineare la MTL e la ITL, si è dovuto tener conto di alcuni problemi insorti come diretta conseguenza di tale scelta. In primo luogo è stata necessaria un'analisi piuttosto dettagliata del codice sorgente, in quanto non sempre la documentazione si è rivelata soddisfacente e completa. In secondo luogo si è dovuto far fronte ad alcuni errori riscontrati nel codice di alcune funzioni che hanno creato non pochi problemi in fase di debug e a cui è stato posto rimedio laddove possibile, non senza parecchio "spreco" di tempo.

Complessità di implementazione dell'algoritmo : l'algoritmo del multigrid, piuttosto intuitivo se espresso ad alto livello, ha richiesto un'implementazione abbastanza complessa, dovuta fondamentalmente al concretizzarsi di problematiche trasparenti ad alto livello, principalmente relative alla ricerca della massima efficienza nell'implementazione delle varie parti del progetto (passaggio da una matrice di un livello ad una di livello immediatamente inferiore, operatori di restrizione e prolungamento, ecc...). Non si dimentichi, infatti, che il tutto nasce dalla necessità di risolvere sistemi lineari di grandi dimensioni.

IV. Struttura del lavoro

Il presente lavoro è strutturato in 4 capitoli:

- nel *capitolo 1* si descrive il metodo multigrid. Vengono analizzate le sue caratteristiche essenziali e le varie parti che lo compongono che, come detto, sono uno o più metodi iterativi classici, anche non particolarmente sofisticati ma convergenti e due operatori di proiezione, uno di restrizione, per il passaggio da un problema di una certa dimensione ad uno di dimensione minore ed uno di interpolazione, per il passaggio inverso. In particolare viene fatto riferimento alle componenti effettivamente utilizzate nel lavoro.

- nel *capitolo 2*, nella prima parte, si descrive brevemente l'ambito dove il multigrid è stato impiegato, in particolare si parla dei problemi di Flusso di Costo

Minimo e dei metodi Interior Point per problemi di Programmazione Lineare con particolare struttura. Nel resto del capitolo si descrive l'ambiente dove si è sviluppato il lavoro, illustrando la sua struttura complessiva e analizzando in dettaglio le varie classi che lo compongono.

- nel *capitolo 3* si illustra la struttura dell'applicazione realizzata e successivamente viene descritta l'implementazione della stessa. Il capitolo non ha lo scopo di fornire una documentazione del codice prodotto, ma piuttosto si prefigge lo scopo di spiegare il perchè delle scelte implementative fatte, scendendo nel dettaglio solo nelle parti più significative.

- il *capitolo 4* e il *capitolo 5* valutano il lavoro svolto, anche attraverso l'analisi dei risultati ottenuti dai test svolti sull'applicazione. Evidenziano quali sono i punti di forza e quelli piu' deboli dell'approccio proposto e indicano quali potrebbero essere gli sviluppi futuri possibili.

Capitolo 1

Metodi Multigrid

Analizzando la velocità di convergenza dei metodi iterativi classici (Jacobi, Gauss-Seidel), si può notare che l'errore decresce rapidamente nelle prime iterazioni, dopo di che decresce molto più lentamente. Il decremento iniziale corrisponde al rapido annullamento dell'errore nello spazio generato dalle alte frequenze. Il rallentamento successivo è dovuto, invece, alla persistenza dello stesso nello spazio delle basse frequenze. Proprio per questa caratteristica che possiedono, i metodi iterativi classici tendono a far assumere all'errore una forma particolarmente *smooth*, come mostrato anche nella figura 1.1.

Questa proprietà viene sfruttata dai metodi multigrid (MGM) per la minimizzazione dell'errore nelle alte frequenze, mentre per l'abbattimento dell'errore nelle basse frequenze si usa un metodo di proiezione detto coarse grid correction.
Il MGM è, dunque, un metodo iterativo che può essere interpretato come la composizione di almeno due distinti metodi iterativi con comportamenti spettrali di tipo complementare.

In questo capitolo si descrive brevemente il metodo iterativo classico utilizzato nel MGM implementato e si analizzano le varie parti che costituiscono la coarse grid correction (CGC).

1.1 Jacobi rilassato

Una delle prime scelte che si deve effettuare quando si vuole realizzare un MGM è quella del metodo iterativo classico da usare come smoother. Per il seguente lavoro si

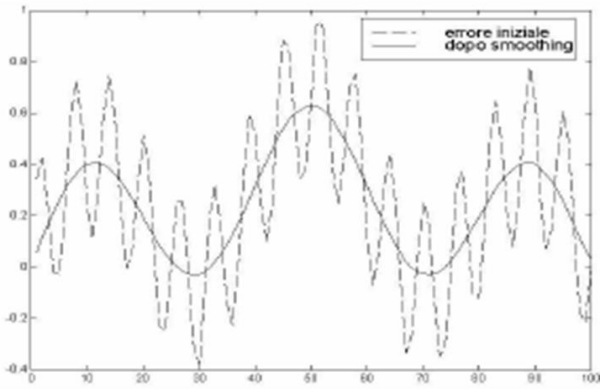


Figura 1.1: Smorzamento dell'errore iniziale dopo 15 passi del metodo di Jacobi rilassato, con parametro di rilassamento $\omega = \frac{2}{3}$, applicato all'equazione di Poisson monodimensionale

è deciso di utilizzare il metodo iterativo di Jacobi con rilassamento. Questo perchè il MGM non richiede un metodo iterativo particolarmente sofisticato, purchè convergente, e perchè esso si presta ad essere facilmente implementato.

Dato un generico sistema

$$Ax = b \tag{1.1}$$

la generica iterazione del metodo di Jacobi può essere ricavata facilmente scomponendo A nel seguente modo:

$$A = D - L - U$$

dove D è la diagonale di A, $-L$ è la triangolare inferiore in senso stretto e $-U$ è la triangolare superiore in senso stretto. A questo punto il sistema (1.1) può essere scritto come:

$$Dx = (L + U)x + b$$

ovvero

$$x = D^{-1}(L + U)x + D^{-1}b$$

e, notando che $L + U = D - A$, si ottiene che

$$x = x + D^{-1}(b - Ax).$$

La k-esima iterazione del metodo sarà quindi

$$x^k = x^{k-1} + D^{-1}r^{k-1}$$

dove si è indicato con $r^{k-1} = b - Ax^{k-1}$ il residuo alla (k-1)-esima iterazione del metodo[1].
Il metodo di Jacobi illustrato sopra è quello classico, è tuttavia semplice ricavare da esso la sua forma rilassata. È sufficiente effettuare una media pesata fra la soluzione calcolata al passo precedente e quella che si otterrebbe con il metodo classico, cioè

$$x^k = (1-\omega)x^{k-1} + \omega[x^{k-1} + D^{-1}rk - 1]$$

che diventa

$$x^k = x^{k-1} + \omega D^{-1}rk - 1$$

e, nello specifico della nostra implementazione, scegliendo come parametro di rilassamento $0 \leq \omega \leq 1$, il metodo iterativo di Jacobi con rilassamento è convergente. Il sistema lineare da risolvere tramite il MGM, infatti, è della forma

$$E\Theta E^T v = u$$

dove la matrice $E\Theta E^T$ è simmetrica, definita positiva, con elementi positivi sulla diagonale e non positivi fuori[2] (si veda il capitolo 2), ed in tal caso il metodo iterativo di Jacobi e la sua variante con rilassamento sono convergenti [1].

Osservazione 1 *Il metodo di Jacobi è una specializzazione del metodo iterativo di Richardson Precondizionato. Quest'ultimo si ottiene moltiplicando entrambi i lati del sistema (1.1) per l'inverso di una matrice invertibile P (precondizionamento)*

$$P^{-1}Ax = P^{-1}b$$

ed aggiungendo x in entrambi i lati

$$x + P^{-1}Ax = x + P^{-1}b.$$

Tramite poche e semplici operazioni si arriva a

$$x = x + P^{-1}(b - Ax).$$

[1] Una misura computazionabile di quanto bene x^{k-1} approssima b.
[2] Una matrice siffatta è detta M-matrice

La k-esima iterazione sarà quindi

$$x^k = x^{k-1} + P^{-1}r^{k-1}$$

dove, al solito, si è indicato con $r^{k-1} = b - Ax^{k-1}$ il residuo alla (k-1)-esima iterazione. È facile vedere che scegliendo come P la diagonale D di A si ottiene Jacobi.

Si fa notare ciò perchè in fase di implementazione si è realizzato, per realizzare Jacobi con rilassamento ω, il metodo di Richardson Precondizionato, usando come precondizionatore la diagonale di A, scalata di un parametro ω.

1.2 Coarse Grid Correction (CGC)

Sia

$$A_n x_n = b_n \tag{1.2}$$

il sistema (1.1) riscritto in modo da evidenziare la dimensione n dello spazio a cui appartiene. E sia $\bar{x}_n$ la soluzione approssimata restituita dal metodo iterativo smoother applicato al sistema (1.2).
L'errore introdotto da tale metodo è quindi

$$e_n = x_n - \bar{x}_n \tag{1.3}$$

e il residuo misura

$$r_n = b_n - A_n\bar{x}_n = A_n x_n - A_n\bar{x}_n. \tag{1.4}$$

Da (1.3) e (1.4) si ottiene

$$A_n e_n = r_n \tag{1.5}$$

e risolvendo, la soluzione esatta x_n del sistema (1.2) può essere ottenuta mediante

$$x_n = \bar{x}_n + e_n. \tag{1.6}$$

Il sistema (1.5) non è più facile da risolvere rispetto a quello originale (1.2), ma è possibile sfruttarne un'importante caratteristica.
All'inizio del capitolo si è detto infatti, che l'applicazione di v passi di un metodo iterativo classico ad un sistema della forma (1.1), partendo da un'approssimazione iniziale x_0, ha la proprietà di rendere l'errore smooth. Tale proprietà permette di poter proiettare l'errore in modo soddisfacente su uno spazio di dimensione minore rispetto

all'originale e risolvere in tale spazio l'equazione (1.5). Riproiettando poi nello spazio di partenza la soluzione dell'errore ottenuta nello spazio più piccolo, tramite l'equazione (1.6), è possibile determinare una nuova soluzione approssimata più "vicina" a x_n di quanto non lo fosse $\bar{x}_n$.

Per potere implementare la procedura sopra descritta, che prende il nome di *coarse grid correction* rimangono da fissare alcuni punti:

- quanto più piccola deve essere la dimensione dello spazio in cui viene proiettato l'errore, nel seguito indicato con Ω_k, rispetto a quello originale, indicato con Ω_n;
- come trasferire il residuo da Ω_n a Ω_k;
- come determinare, per ottenere un sistema analogo a (1.5), ma definito su Ω_k, la matrice dei coefficienti A_k;
- ed infine, come riproiettare l'errore stimato in Ω_k indietro su Ω_n.

1.2.1 Dimensione di Ω_n e di Ω_k

Per come verranno definiti gli operatori lineari per il passaggio da Ω_n a Ω_k e viceversa, si impone che

$$n = 2k + 1$$

e cioè, che lo spazio di dimensione maggiore deve avere sempre dimensione dispari.

Nello specifico della nostra implementazione, questa forte assunzione non crea disagi, in quanto nella maggior parte dei casi lo spazio di partenza ha come dimensione n una potenza di 2 e la matrice dei coefficienti del sistema (1.2) da risolvere ha rango $\mathbb{R}^{n-1}$.
Infatti, sia $M = E\Theta E^T$ la matrice dei coefficienti, che come detto è simmetrica e definita positiva ed inoltre, per ogni riga, il valore dell'elemento sulla diagonale è uguale all'opposto della somma degli elementi fuori dalla diagonale.
M può assumere la forma

$$M = \begin{pmatrix} M' & m \\ m^T & \delta \end{pmatrix}$$

dove $M' \in \mathbb{R}^{n-1 \times n-1}$, $m \in \mathbb{R}^{n-1}$ e $\delta \in \mathbb{R}^+$.
Allo stesso modo, si possono scrivere il vettore x e il vettore dei termini noti b come

$$x = \begin{pmatrix} x' \\ x_n \end{pmatrix} \qquad b = \begin{pmatrix} b' \\ b_n \end{pmatrix}.$$

dove, analogamente a prima, x', $b' \in \mathbb{R}^{n-1}$ e x_n, $b_n \in \mathbb{R}$. Il sistema (1.2) è pertanto equivalente al seguente

$$\begin{pmatrix} M'x' + mx_n \\ m^T x' + \delta x_n \end{pmatrix} = \begin{pmatrix} b' \\ b_n \end{pmatrix} \tag{1.7}$$

cioè

$$M'x' + mx_n = b' \tag{1.8}$$
$$m^T x' + \delta x_n = b_n \tag{1.9}$$

Indicando con $e^T = \begin{pmatrix} 11 \cdots 1 \end{pmatrix}$ il vettore di tutti 1, appartenente ad $\mathbb{R}^{n-1}$ e moltiplicando entrambi i membri di (1.8) per e^T si ottiene

$$e^T M'x' + e^T mx_n = e^T b'$$

Ora, notando che

$$e^T M' = -m^T$$
$$e^T b' = -b_n,$$

si può affermare che

$$m^T x' - e^T mx_n = b_n.$$

Confrontando quest'ultima equazione con la (1.9) si può concludere che il sistema (1.7) ha soluzione solo quando $-e^T mx_n = \delta x_n$ ed in particolare, quando $x_n = 0$.
Tirando le somme, per risolvere il sistema $Mx = b$ è sufficiente risolvere il sistema ridotto $M'x' = b'$, ottenuto togliendo a M l'ultima riga e l'ultima colonna, e ricavare x tramite

$$x = \begin{pmatrix} x' \\ 0 \end{pmatrix}.$$

1.2.2 Operatore di restrizione

Per l'implementazione della coarse grid correction, come si è detto, è necessario definire una funzione (lineare) che permetta di proiettare il residuo r_n da Ω_n ad Ω_k.

Sia $G(\Omega)$ l'insieme delle funzioni definite su una generico spazio Ω. L'operatore di

restrizione è definito come

$$R_n : G(\Omega_n) \to G(\Omega_k)$$

ed il residuo proiettato dallo spazio più grande a quello più piccolo sarà

$$r_k = R_n r_n.$$

Come operatore di restrizione si è deciso di utilizzare il cosiddetto *full weighting operator*. Per ogni valore di Ω_k si è definita una media pesata con i valori vicini di Ω_n:

$$r_{k,j} = \frac{1}{4}\left(r_{k,2j} + 2r_{k,2j+1} + r_{k,2j+2}\right), \quad j = 0, \ldots, k-1$$

che in forma matriciale diventa

$$R_n = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & & \\ & & 1 & 2 & 1 & & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & 1 & 2 & 1 \end{pmatrix}_{k \times n}.$$

1.2.3 Operatore di prolungamento

La seconda classe di funzioni di trasferimento "inter-spazio" riguarda la proiezione del vettore dell'errore da uno spazio più piccolo ad uno più grande.
Definendo l'operatore di prolungamento

$$P_n : G(\Omega_k) \to G(\Omega_n)$$

l'errore proiettato da Ω_k a Ω_n sarà

$$e_n = P_n e_k.$$

Questa procedura, abbastanza comune in analisi numerica, è generalmente chiamata *interpolazione*. Come operatore di prolungamento può essere utilizzata qualsiasi tecnica di interpolazione, ma, per la sua semplicità e per il fatto che offre risultati più che soddisfacenti, si è deciso di utilizzare l'interpolazione lineare. Tale operatore è così

definito

$$e_{n,j} = \begin{cases} \frac{1}{2}(e_{k,i-1} + e_{k,i}) & \text{per } j = 2i+1 \\ e_{k,i} & \text{per } j = 2i \end{cases} \quad , \quad i = 0, \ldots, k-1$$

Se si passa in forma matriciale, è facile notare che

$$P_n = 2R_n^T.$$

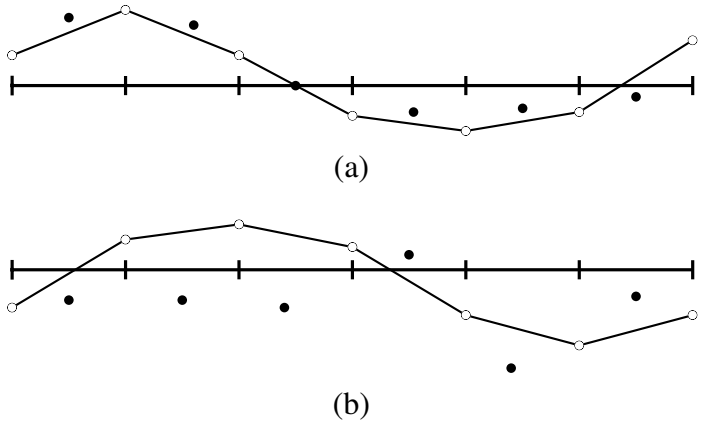
È utile credo, far notare che se l'errore in Ω_n è un vettore smooth e se si conosce una approssimazione esatta dell'errore su Ω_k, allora l'interpolazione di tale approssimazione sullo spazio Ω_n è anch'essa smooth. Quindi, ci si aspetta una buona approssimazione finale dell'errore su Ω_n, come mostrato nella figura 1.2(a). Al contrario, se l'errore in Ω_n è oscillatorio, una buona approssimazione di esso sullo spazio Ω_k ne può produrre, interpolando, una non molto accurata su Ω_n. Tale situazione è mostrata in figura 1.2(b).
Poichè l'interpolazione è necessaria nella CGC, si può concludere che tale processo è più efficace quando l'errore è smooth.
Si può apprezzare, dunque, la complementarità delle due fasi del multigrid. Il rilassamento su Ω_n (con il metodo di Jacobi, ad esempio) elimina velocemente i componenti oscillatori dell'errore, lasciando un errore relativamente smooth. L'interpolazione lineare, poichè l'errore è smooth, assumendo che l'equazione del residuo (1.5) venga risolta accuratamente su Ω_k, trasferisce accuratamente l'errore all'indietro sullo spazio Ω_n. Proprio per il fatto di essere complementari, i due metodi sopra citati, che presi separatamente richiederebbero un numero improponibile di iterazioni per raggiungere la convergenza, combinati insieme risultano particolarmente veloci.

1.2.4 Matrice dei coefficienti

Ciò che rimane da determinare per poter risolvere l'equazione del residuo (1.5) in Ω_k, è la matrice dei coefficienti A_k. La scelta di A_k deriva dalla condizione di Galerkin [2]:

$$A_k = R_n A_n P_n. \tag{1.10}$$

È facile vedere che, usando come operatore di restrizione R_n e come operatore di prolungamento P_n quelli definiti in precedenza, un generico elemento della matrice A_k sarà uguale a



(a)

(b)

Figura 1.2: (a) Se l'errore esatto e_n su Ω_n (indicato dai $\bullet$ e dai $\circ$) è smooth, un'interpolazione dell'errore e_k (la linea che connette i $\circ$) dovrebbe dare una "buona" rappresentazione di e_n.
(b) Se l'errore esatto e_n su Ω_n (indicato dai $\bullet$ e dai $\circ$) è oscillatorio, un'interpolazione dell'errore e_k (la linea che connette i $\circ$) potrebbe dare una "cattiva" rappresentazione di e_n.

$$A_{k\ i,j} = \frac{1}{8}\begin{pmatrix} A_{n\ 2i,2j} + 2A_{n\ 2i,2j+1} + A_{n\ 2i,2j+2} + \\ +2A_{n\ 2i+1,2j} + 4A_{n\ 2i+1,2j+1} + 2A_{n\ 2i+1,2j+2} + \\ +A_{n\ 2i+2,2j} + 2A_{n\ 2i+2,2j+1} + A_{n\ 2i+2,2j+2} \end{pmatrix} \quad \text{per } i,j = 0,\ldots,k-1. \tag{1.11}$$

Ora che si posseggono tutti gli strumenti si può definire formalmente quanto discorsivamente descritto in questa sezione.
Sia $\bar{x}_n$ una prima approssimazione della soluzione del sistema (1.2), è possibile "correggere" tale approssimazione tramite la seguente procedura:

CGC($\bar{x}_n$, b_n)
• Calcola il residuo $r_n = b_n - A_n\bar{x}_n$ su Ω_n.
• Restringi r_n su Ω_k, applicando $r_k = R_n r_n$.
• Ricava $A_k = R_n A_n P_n$ e risolvi $e_k = A_k^{-1} r_k$ su Ω_k.
• Risolvi $e_k = A_k^{-1} r_k$ su Ω_k.
• Interpola e_k su Ω_n, applicando $e_n = P_n e_k$.
• Correggi l'approssimazione iniziale tramite $\bar{x}_n = \bar{x}_n + e_n$.

1.3 Metodo Two-Grid

Combinando insieme le due techiche analizzate sopra, è possibile definire il *metodo two-grid* (TGM, dall'inglese Two-Grid Method). Si applica prima uno smoother per rendere l'errore smooth, poi la coarse grid correction ed infine, facoltativamente, si può riapplicare uno smoother (eventualmente diverso dal primo) per "addolcire" ulteriormente l'errore lasciato dalla CGC[3].
Quindi, dato un vettore iniziale x_0, è possibile ottenere una buona approssimazione finale x_n tramite la seguente procedura:

TGM(x_0, x_n)
• Rilassa v_1 volte $A_n x_n = b_n$ su Ω_n ottenendo una approssimazione $\bar{x}_n$. • Applica CGC($\bar{x}_n$, b_n). • Rilassa v_2 volte $A_n x_n = b_n$ su Ω_n partendo da $\bar{x}_n$.

1.4 μ-Cycle Scheme

La coarse grid correction, e di conseguenza il TGM, lascia aperta un'incombente questione procedurale: quale sia il modo migliore di risolvere il sistema

$$A_k e_k = r_k. \tag{1.12}$$

La risposta non può che trovarsi nella ricorsione. È facile intuire infatti, che è possibile applicare il TGM all'equazione del residuo su Ω_k, cioè rilassare e muoversi su Ω_p, con $p = \frac{k-1}{2}$ e così via ricorsivamente finchè non si giunge ad uno spazio di dimensioni sufficientemente piccoli, in cui il sistema possa essere risolto agevolmente con un metodo diretto. La soluzione ottenuta, una volta interpolata, viene utilizzata per correggere l'approssimazione precedente ed eventualmente si può applicare il post-smoother. Questo procedimento viene ripetuto all'indietro per ogni spazio intermedio fino a ritornare a quello di partenza Ω_n.

[3] In questo caso, il primo smoother viene detto *pre-smoother*, mentre il secondo è detto *post-smoother*.

MGM$\mu(\bar{x}_l, b_l)$
Se Ω_l *é* lo spazio dell'ultimo livello, allora $b_l = A_l^{-1} x_l$. Altrimenti • rilassa v_1 volte $A_l x_l = b_l$, partendo da $\bar{x}_l$ • calcola il residuo $r_l = b_l - A_l \bar{x}_l$ • applica $r_{l+1} = R_l r_l$ • chiama MGM$\mu(x_{l+1}, b_{l+1})$ μ volte • correggi tramite $\bar{x}_l = \bar{x}_l + e_l$ • rilassa v_2 volte $A_l x_l = b_l$ partendo da $\bar{x}_l$

Il parametro μ serve per definire diversi schemi di chiamate ricorsive. Nella pratica, vengono utilizzati solo $\mu = 1$, che dà un V-cycle e $\mu = 2$, che dà un W-cycle. In figura 1.3(a) viene mostrata la rappresentazione grafica delle operazioni di restrizione e proiezione per il V-cycle, mentre la figura 1.3(b) mostra quelle per il W-cycle.

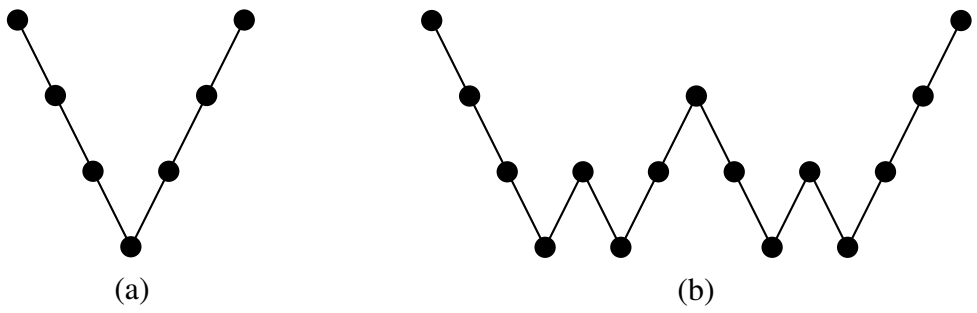


Figura 1.3: Rappresentazione grafica di un V-cycle (a) e di un W-cycle (b)

Capitolo 2

Verso l'implementazione

Nell'introduzione al presente lavoro, è stato accennato che la tecnica del multigrid, analizzata in dettaglio nel precedente capitolo, sarebbe stata utilizzata per la risoluzione di sistemi lineari di grandi dimensioni, derivanti dall'applicazione, a problemi di flusso di costo minimo, di tecniche di tipo interior point. Nel prossimo capitolo verrà mostrato come ciò è stato fatto e quali scelte implementative sono state compiute. Prima però, in questo, vengono brevemente descritti i problemi di flusso di costo minimo e la tecnica interior point, e viene inoltre analizzato il framework dentro al quale si è inserito tutto il lavoro.

2.1 Flusso di costo minimo e metodi interior point

Il *problema di flusso di costo minimo* (MCF, da *Min Cost Flow* problem) consiste nel determinare la via più economica per trasportare una determinata quantità di flusso da uno o più punti di produzione ad uno o più punti di consumo, attraverso una rete di trasporto data.
I dati del problema sono un grafo (orientato) G = (N, A) con n = |N| nodi e m = |A| archi (orientati). Ad ogni nodo $i \in N$ è associato un deficit b_i, ovvero, la quantità di flusso che è prodotto/consumato dal nodo: i nodi sorgente (i quali producono flusso) hanno deficit negativo e i nodi destinatari (che consumano flusso) hanno deficit positivo. Ad ogni arco $(i, j) \in A$ è associato un costo c_{ij} ed una capacità superiore u_{ij} (la capacità inferiore si assume pari a 0). Le variabili di flusso x_{ij} rappresentano la quantità di flusso da spedire sull'arco (i, j).

Il problema può essere formulato nel modo seguente:

$$\min \sum_{(i,j)\in A} c_{i,j}x_{i,j}$$

$$\sum_{(j,i)\in A} x_{j,i} - \sum_{(i,j)\in A} x_{i,j} = b_i \qquad i \in N$$

$$0 \leq x_{i,j} \leq u_{i,j} \qquad (i,j) \in A$$

Le n equazioni sono i vincoli di conservazione del flusso e le $2m$ disequazioni sono i vincoli di nonnegatività e di capacità del flusso.
Tale problema può essere espresso in forma vettoriale come

$$\min\{cx : Ex = b, 0 \leq x \leq u\} \tag{2.1}$$

dove E è la matrice di incidenza del grafo G, $c = [c_i]$ è il vettore dei costi, $u = [u_i]$ è il vettore delle capacità, $b = [b_i]$ è il vettore dei deficit e $x = [x_i]$ è il vettore dei flussi.
I problemi MCF hanno un ampio insieme di applicazioni, o in sé o, più spesso, come sottomodelli di problemi più complessi. Questo risulta evidente dall'enorme quantità di ricerca che è stata dedicata allo sviluppo di algoritmi efficienti per la loro soluzione, o attraverso la specializzazione di algoritmi di Programmazione Lineare, come il metodo del simplesso, al caso delle reti, o attraverso lo sviluppo di approcci ad hoc.

Rientrano in questa seconda categoria i metodi *interior point* (in breve, IP).
Ad ogni iterazione di un metodo IP, deve essere risolto un sistema lineare della forma

$$E\Theta E^T v = u \tag{2.2}$$

dove Θ è una matrice diagonale $m \times m$ ($m = |A|$) definita positiva e u un vettore di $\mathbb{R}^n$ ($n = |N|$). Entrambi dipendono dalla soluzione corrente e dal particolare tipo di algoritmo IP scelto.
La soluzione efficiente del sistema (2.2) rappresenta la principale difficoltà computazionale di tali algoritmi. Il multigrid, in questo ambito, si presenta come una valida alternativa alla risoluzione di sitemi della forma (2.2) con tecniche basate sul gradiente coniugato precondizionato[6].

2.2 Framework

2.2.1 I linguaggi

Il framework che si andrà ad analizzare è interamente scritto in C++.
Per questo motivo, tale linguaggio è stato designato come elemento fondamentale del lavoro svolto, determinando di conseguenza la scelta degli strumenti che vi sono stati interfacciati: in particolare la libreria MTL e il suo complemento, la ITL, di cui si è accennato brevemente in fase di introduzione.

> Il C++ è un linguaggio di programmazione general-purpose basato sul linguaggio C. In aggiunta fornisce estensioni come: classi ed ereditarietà multipla, funzioni inline, overloading degli operatori, overloading dei nomi di funzione, tipi costanti, references, operatori per l'allocazione di memoria, checking degli argomenti di funzione, conversione di tipi, gestione delle eccezioni, funzioni template, classi annidate e namespace [5].

Da oltre vent'anni è un punto di riferimento inossidabile nello sviluppo di tutti i sistemi commerciali di grosso calibro oltre che per l'indiscussa qualità del linguaggio anche per le innumerevoli librerie satellite che lo accompagnano e che coprono ogni possibile area tecnico-scientifica.

Oltre al C++ è comunque giusto citare, i linguaggi di script (csh, bash, make) con i quali sono stati realizzati i Makefile e, in fase di test, le batch.

2.2.2 La struttura

La fase di implementazione ha la sua parte centrale nella realizzazione di una specializzazione della classe MCFLSSolver che implementa l'algoritmo del multigrid algebrico.
La classe astratta MCFLSSolver fa parte di un ampio framework la cui struttura è rappresentata nella figura 2.1.

- La classe astratta[1] `IPClass` implementa vari algoritmi IP per problemi di programmazione lineare o di programmazione quadratica (convessa). Questa classe è da intendersi come base per lo sviluppo di algoritmi IP specializzati per problemi con una particolare struttura della matrice dei coefficienti. Essa fornisce

[1] In C++ una classe è astratta se presenta almeno un metodo con una sintassi del tipo `virtual return_type method(parameters) = 0`. Tali metodi prendono il nome di *Metodi Virtuali Puri*.

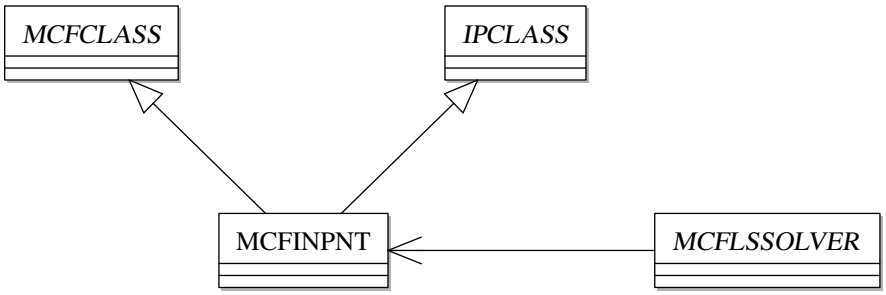


Figura 2.1: Diagramma UML delle classi del framework di lavoro.

una serie di metodi "virtuali" che definiscono l'interfaccia per le varie classi che la ereditano.

- La classe astratta *MCFClass* fornisce un'interfaccia standard per i risolutori di problemi di flusso di costo minimo, lineari (e non), i quali devono essere implementati "concretamente" come classi derivate di essa.

- La classe *MCFInPnt* implementa un solver di prolemi di MCF, usando differenti varianti di algoritmi IP, dove la soluzione del sistema (2.2) è affidata ad un generico solver di sistemi lineari, definito dalla classe MCFLSSolver.
  MCFInPnt è conforme all'interfaccia standard definita da MCFClass e utilizza gli algoritmi IP forniti da IPClass.

MCFLSSolver

La classe *MCFLSSolver* merita una descrizione più accurata, in quanto diretta superclasse della classe che implementerà il multigrid.
Essa è una classe astratta che definisce un'interfaccia, come richiesto dall'approccio IP utilizzato, per la risoluzione di un sistema nella forma[2]

$$(ADA^T)Res = RHS. \tag{2.3}$$

MCFLSSolver contiene varie strutture dati per rappresentare il problema e metodi per inizializzarlo e risolverlo. Di seguito si analizzano alcuni di essi, i più importanti:

[2]Questo sistema è identico al sistema (2.2). Il cambio di notazione è stato fatto per rimanere coerenti con i nomi dei vari campi della classe.

Strutture Dati :

- `cHpRow D`: un vettore di scalari rappresentante la diagonale della matrice D;
- `cHpRow RHS`: il termine noto del sistema (RHS, da Right Hand Side);
- `Index n, m`: rispettivamente il numero di nodi e il numero di archi del grafo;
- `cIndex_Set Sn, En`: contengono rispettivamente, i nodi di inizio degli archi e i nodi di fine;
- `cHpRow Prcsn`: il vettore di ϵ che indica la precisione richiesta alla soluzione.

Metodi :

- `virtual void SetGraph()` fornisce informazioni circa la topologia del grafo che dà la matrice di incidenza A. È giusto pensare che tale topologia cambi meno frequentemente rispetto agli altri elementi del sistema (2.3) (si vedano `SetD()` e `SetRHS()` sotto). Proprio per questo è meglio, in termini di efficienza, eseguire in `SetGraph()` tutte quelle operazioni che non coinvolgono `D` e `RHS`, ma che coinvolgono, ad esempio, solo la struttura della matrice ADA^T;
- `virtual BOOL SetD()` setta l'm-vettore `D` che contiene gli elementi (positivi) della matrice diagonale D. `SetD()` deve essere chiamato solamente dopo che è stato chiamato `SetGraph()`, e deve essere chiamato almeno una volta, prima di una chiamata di `SolveADAT()` (si veda sotto), sebbene `SolveADAT()` può essere chiamato più di una volta con lo stesso vettore `D` (ad esempio, con un differente `RHS`);
- `virtual void SetRHS()` setta l'n-vettore `RHS` che contiene il termine noto del sistema lineare. `SetRHS()` deve essere chiamato solamente dopo che è stato chiamato `SetGraph()`, e deve essere chiamato almeno una volta, prima di una chiamata di `SolveADAT()`, sebbene `SolveADAT()` può essere chiamato più di una volta con lo stesso `RHS`. Questo perchè, come spiegato meglio sotto, il sistema può essere risolto con minore precisione rispetto a quella indicata dall'ultima chiamata di `SetPrcsn()` (si veda sotto), ma il chiamante può allora richiamare `SolveADAT()` per tentare di incrementare la precisione, trovando quella attuale poco soddisfacente. Queste chiamate successive possono essere

distinte dalla prima dal fatto che `SetRHS()` non è chiamato di nuovo, e questa informazione può essere sfruttata dal solver, per esempio, per riiniziare il processo di iterazione da dove si era interrotto piuttosto che da capo.

- `virtual void SetPrcsn()` setta l'n-vettore `Prcsn`. In particolare, il sistema (2.3) può dirsi "ben risolto" se, alla fine di `SolveADAT()`, si ha
$$(ADA^T)Res = RHS + \epsilon,$$
dove
$$0 \leq \epsilon[i] \leq Prcsn[i] \quad \text{per } i = 0 \dots n-1.$$

- `virtual LSSStatus SolveADAT()` dovrebbe, dopo che `D`, `RHS` e `Prcsn` son stati settati, risolvere il sistema (2.3) con il grado di precisione richiesto.
Se questo è fatto, viene restituito `KOk` (=0), e non viene più richiamato, a meno che `SetD()` e/o `SetRHS()` non vengano riinvocati (cambiando, rispettivamente, il valore di `D` e/o `RHS`).
Se si verifica un errore numerico "fatale" che impedisce al solver di raggiungere la precisione richiesta, e se non c'è modo di incrementare la precisione, anche con chiamate successive, viene restituito `KNError` (=2) e il calcolo della soluzione verrà immediatamente interrotto.
Alternativamente, il solver può fermarsi senza aver raggiunto la precisione richiesta perché, basandosi su una condizione specifica del solver, o ha deciso che la soluzione trovata è "buona abbastanza", o ha esaurito qualche risorsa (tempo, numero di iterazioni ...). In questo caso, viene restituito `kLwPrcsn` (=1). Il chiamante allora, può "accettare" questa decisione e andare avanti, o può chiamare di nuovo `SolveADAT()` con lo stesso `D` e lo stesso `RHS`, per ottenere una soluzione "più accurata"; in questo caso, il valore iniziale dell'approssimazione con cui far (ri)partire il solver deve essere la stessa di quella ottenuta nella precedente chiamata (il metodo iterativo, cioè, può ripartire dalla sua ultima soluzione). L'MCFLSSolver deve assicurare la seguente importante condizione: dopo un numero finito di chiamate a SolveADAT(), o si trova una soluzione con la precisione richiesta, e quindi si ritorna KOk, o si decide che tale precisione è impossibile da raggiungere e si restituisce KNError.

Nel framework esistono già varie specializzazioni della classe MCFSSolver, come mostrato in figura (2.2).

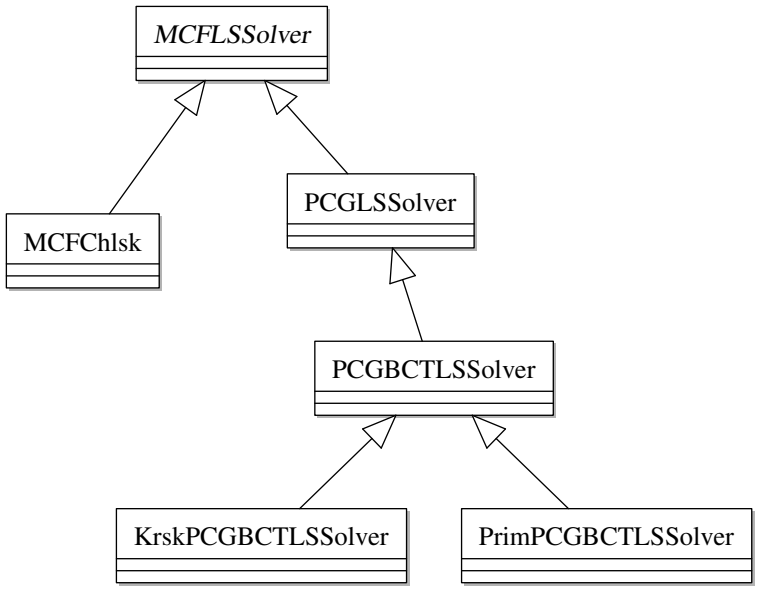


Figura 2.2: Diagramma UML raffigurante la classe MCFLSSolver e le sue varie sottoclassi.

- La classe *MCFChlsk* risolve il sistema (2.3) attraverso la fattorizzazione (incompleta) di Cholesky, la quale è computazionalmente efficiente e numericamente stabile. Il metodo consiste, brevemente, nel trovare una fattorizzazione $M = LSL^T$ della matrice del sistema (2.3), tale che L (detta fattore di Cholesky) sia una matrice unità triangolare inferiore[3] ed S una matrice diagonale con elementi positivi. Una volta calcolata tale fattorizzazione, il sistema riguardante M può essere risolto con due sostituzioni in avanti su L. Se, l'efficienza e la stabilità rappresentano un punto di forza di tale tecnica, il fenomeno del fill-in ne rappresenta sicuramente uno svantaggio: una matrice sparsa M può avere, infatti, un fattore di Cholesky L denso. In generale, questo fenomeno non può essere evitato e quindi sono stati proposti metodi alternativi.

- Molti di questi metodi usano, per risolvere il sistema, un metodo del gradiente coniugato precondizionato (PCG, dall'inglese Preconditioned Conjugate Gradient) [6].

[3]Matrice triangolare inferiore con tutti 1 sulla diagonale.

La classe *PCGLSSolver* implementa, appunto, un algoritmo "generico" del PCG per risolvere il sistema (2.3). La scelta del precondizionatore è importante: esso deve essere poco costoso da calcolare e da invertire, ma deve apportare una consistente riduzione del numero di iterazioni del gradiente coniugato richieste per approssimare la soluzione di (2.3).
Un generico PCGLSSolver implementa un semplice precondizionatore diagonale e lascia alle sue sottoclassi la possibilità di implementarne degli altri.

- La classe *PCGBCTLSSolver* deriva da PCGLSSolver e implementa un "generico" precondizionatore tree/BCT+diagonale per algoritmi PCG.
  Data una matrice $K = ADA^T$, dove A è la matrice di incidenza di un grafo G, si vuole calcolare un precondizionatore M tale che $M^{-1} \approx K^{-1}$ ma sia facile e veloce da costruire. I precondizionatori implementati in questa classe si basano, principalmente, su due idee:

  I) selezionare un grafo parziale G' di G (ovvero, un sottoinsieme dei suoi archi) così che la matrice

  $$K' = A'D'A'^T$$

  dove A' è la matrice di incidenza del grafo G' e D' è D ristretta agli archi di G', sia facile da invertire ma contenga molto dell'"informazione" di K.

  II) Dal momento che I) potenzialmente tralascia una parte dell'informazione in K, trovare un modo di aggiungere a K' un'altra matrice W tale che $K' + W$ sia ancora facile da invertire e W tenga conto delle informazioni in $K - K'$.

  Differenti implementazioni sono fornite per ognuna di queste idee di base.

  I.1) Si può selezionare il grafo vuoto G'.

  I.2) Si può scegliere G' albero di copertura di G.

  I.3) Come I.2), con l'aggiunta a G' di altri archi per farlo diventare un "brother connected tree (BCT) di profondità 2": gli archi aggiunti possono unire solamente nodi fratelli (figli dello stesso nodo) nell'albero originale, e la rimozione di tutti i nodi dell'albero di copertura originale deve lasciare una foresta.

  II.1) Si può selezionare $W = 0$;

II.2) Si può prendere $W = \rho diag(K - K')$, dove ρ è un parametro e $diag(X)$ la matrice diagonale avente come elementi diagonali quelli di X. La classe PCGBCTLSSolver è comunque astratta, in quanto non implementa il modo di trovare un "buon" tree/BCT, lasciando questo compito alle sottoclassi che la implementano.

- La classe *KrskPCGBCTLSSolver* deriva da PCGBCTLSSolver e implementa un'euristica basata su Kruskal che usa un ordinamento degli archi o una visita BF parametrizzata.

- Infine, la classe *PrimPCGBCTLSSolver*, sottoclasse di PCGBCTLSSolver, implementa un'euristica basata sul metodo di Prim [7].

Capitolo 3

L'implementazione

Si è giunti alla descrizione dell'implementazione del solver multigrid.
Per coerenza con i nomi dei solver già presenti nel framework, si è deciso di chiamare il nuovo solver *MCFMGLSSolver*.In seguito, ne verrà descritta l'architettura e dopo si scenderà più nel dettaglio per descriverne le componenti più significative.

3.1 Architettura dell'MCFMGLSSolver

Il solver implementato ha la struttura mostrata in figura 3.1.

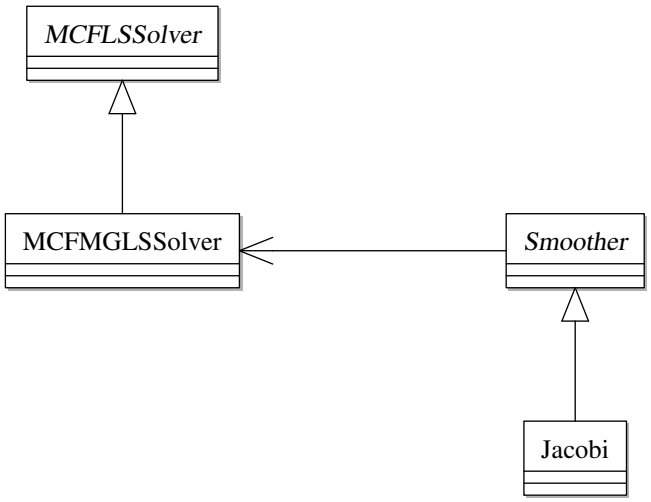


Figura 3.1: Diagramma UML della classe MCFMGLSSolver e delle classi ad essa collegate.

L'interfaccia *Smoother* non fa altro che fornire due metodi: uno per rilassare sistemi con matrici sparse, il secondo per rilassare matrici dense. Essa è utilizzata dal solver, che ne invoca uno dei due metodi quando gli serve applicare il pre-smoother e il post-smoother. Smoother, comunque, non implementa da sé i propri metodi ma lascia questo compito alle sue sottoclassi. Ciò è stato fatto per rendere il lavoro più modulare e, in particolare, per rendere il solver indipendente dal particolare smoother: infatti, ad esempio, se si vuole cambiare pre-smoother e/o post-smoother, basta implementare una nuova sottoclasse concreta di Smoother, che ne rispetti l'interfaccia e non serve modificare il solver, che rimane quindi, all'"oscuro" dei cambiamenti apportati.

In questo lavoro è stata realizzata una sola implementazione "concreta" di smoother: la classe *Jacobi*. Tale classe implementa il metodo di Jacobi rilassato, analizzato a pagina 5 nel capitolo 1. Per fare ciò, utilizza la libreria ITL ed in particolare la procedura `richardson()`, che risolve un qualunque sistema lineare $Ax = b$ usando il metodo iterativo di richardson precondizionato, con precondizionatore una generica matrice M. Scegliendo infatti, come M la matrice diagonale P avente come elementi non nulli gli elementi sulla diagonale di A si ottiene proprio Jacobi (si veda a tal riguardo l'osservazione a pagina 7).

3.1.1 La classe MCFMGLSSolver

L'implementazione della classe *MCFMGLSSolver* ha rappresentato, senza dubbio, la parte più consistente del lavoro. Essa realizza una specializzazione della classe MCFLSSolver e implementa un risolutore per sistemi lineari, basato sulla tecnica del multigrid vista nel capitolo 1.
In particolare,

- ridefinisce quei metodi la cui implementazione base, presente nella classe "padre", non è sufficiente;
- dà un'implementazione concreta del metodo (puramente virtuale) `SolveADAT()()` presente in MCFLSSolver;
- implementa l'operatore full weighting e l' interpolazione lineare, rispettivamente per l'operazione di restrizione e interpolazione.

Di seguito vengono descritti, più dettagliatamente, le strutture dati ed i metodi della classe, sia quelli ridefiniti che quelli implementati "ex novo", soffermadosi sugli aspetti più importanti ed evidenziandone le caratteristiche principali.

Strutture dati

Ad ogni livello del multigrid è necessario conoscere, fra le altre cose, l'approssimazione corrente della soluzione, il vettore con il RHS, il vettore del residuo, l'errore corrente e la particolare struttura della matrice dei coefficienti associata a quel livello. Proprio per questo si sono previste le seguenti srutture dati

- `MyVector *listaX;`
- `MyVector *listaRHS;`
- `MyVector *listaResidual;`
- `MySparseMatrix *listaSMatrix;`
- `MyDenseMatrix *listaDMatrix;`

È importante far notare due cose. La prima, che si sono utilizzati due diversi array di matrici. Uno per le matrici sparse, utilizzato nei primi livelli (quelli con spazio di dimensione maggiore), dove "quasi tutti" gli elementi della matrice dei coefficienti sono zero. Per "quasi tutti" si intende, in questo caso, che in una matrice $A \in \mathbb{R}^{n \times n}$, indicando con nnz il numero di non-zeri presenti, la quantità $\frac{nnz}{n*n}$ sia minore di 0.30, ovvero che il 70% almeno degli elementi in A siano zeri. L'altro array è utilizzato, invece, nei restanti livelli, per le matrici dense. La differenziazione si è resa necessaria per motivi legati soprattutto all'efficienza, come si vedrà sotto quando si parlerà del metodo `SetGraph()` e di `SetD()`.
La seconda cosa da far notare, è l'assenza, fra le strutture dati precedenti, di un array degli errori ai vari livelli. Ciò non è dovuto ad una dimenticanza, ma scaturisce da una semplice osservazione. L'equazione del residuo $A_k e_k = r_k$, da risolvere durante il passo di CGC, può essere benissimo risolta utilizzando come termine noto e vettore dell'approssimazione, rispettivamente, l'elemento di `MyVector *listaRHS` e l'elemento di `MyVector *listaX` specifici del particolare livello in cui ci si trova. Infatti, ad esempio, se si è al livello l, alla fine del pre-smoother, si avrà in `listaX[l]` l'approssimazione corrente e in `listaResidual[l]` il residuo corrente, mentre il contenuto di `listaX[l+1]` e `listaRHS[l+1]` sarà non significativo. Quindi si può restringere, senza problemi, `listaResidual[l]` in `listaRHS[l+1]` e risolvere l'equazione del residuo usando `listaX[l+1]` e `listaRHS[l+1]`.

Naturalmente, fra le strutture dati della classe, se ne devono prevedere alcune per la memorizzazione delle informazioni circa gli smoother. Queste sono tenute da

- `Smoother **ExtPreSmoothers;`

- `Smoother **ExtPostSmoothers;`
- `Smoother **IntPreSmoothers;`
- `Smoother **IntPostSmoothers;`

Il programma principale (contenente il `main()`) può non sapere, o non volere sapere, quanti sono i livelli del multigrid. In questo caso, passa al solver un certo numero di pre-smoother e post-smoother, che salva in `ExtPreSmoother` e in `ExtPostSmoother`. Se tale numero è maggiore di quello necessario, il solver copia in `IntPreSmoother` e in `IntPostSmoother` gli smoother che gli servono e semplicemente scarta gli altri. Altrimenti, se il numero di pre-smoother e post-smoother passati non è sufficiente, lo integra riusando l'ultimo elemento in `ExtPreSmoother` e l'ultimo elemento in `ExtPostSmoother`.

void SetGraph()

Il metodo `void SetGraph()` estende l'omonimo metodo della superclasse. Esso principalmente fa le seguenti cose:

- calcola il numero di livelli, in base al numero di nodi del grafo (la dimensione della matrice nello spazio iniziale), e in base alla dimensione dello spazio più piccolo, dato come parametro al solver. Sia n il numero di nodi del grafo e p la dimensione dello spazio più piccolo, il numero di livelli sarà dato da

$$1 + \log_2(\frac{n}{p}).$$

  Una volta conosciuto il numero di livelli, si può facilmente sapere quanti di questi saranno sparsi. Prima però, è necessario sapere quanto rapidamente si densificano le matrici scendendo di livello. Per fare ciò, si è fatto un ragionamento molto grossolano: considerando che nel passaggio da un livello a quello immediatamente inferiore il numero di elementi della matrice dei coefficienti diminuisce di un fattore 4 e notando, si veda a tal proposito la (1.11), che all'incirca ogni nove elementi della matrice più grande, se c'è un non-zero, allora si ha un non-zero anche in quella più piccola, si è fissato a 30 (≈ 36) il fattore di densificazione;

- crea le strutture dati precedenti;

- se il primo livello è sparso, crea la struttura dei non-zero della matrice di tale livello. E' conveniente fare ciò in SetGraph() in quanto tale metodo viene eseguito una volta sola per ogni grafo, e quindi tale lavoro viene fatto una volta solamente e se ne può benificiare più volte rendendo più efficiente, in SetD(), la creazione delle matrici sparse.

void SetD()

Il metodo `void SetD()` è quello deputato all'inizializzazione delle matrici dei vari livelli. Tale inizializzazione differisce a seconda che

- la matrice di primo livello sia sparsa;
- la matrice di primo livello sia densa;
- si passi da un livello con matrice sparsa ad uno con matrice sparsa;
- si passi da un livello con matrice sparsa ad uno con matrice densa[1];
- si passi da un livello con matrice densa ad uno con matrice densa;

La matrice di primo livello K è il risultato del prodotto $K = ADA^T$, dove A è la matrice di incidenza del grafo corrente $G = (V, E)$ e D è una matrice diagonale con elementi positivi. Le matrici dei livelli inferiori al primo invece, sono ottenuti, dalla condizione di Galerkin [2], attraverso

$$K_{i+1} = R_i K_i P_i \quad \text{per } i = 0, \ldots, l-1$$

dove R_i e P_i rappresentano rispettivamente, il restrittore e il prolungatore del livello i e l indica il numero di livelli[2].
K può essere facilmente calcolata notando che,

$$k_{i,i} = d_{i,i}(|FS(i)| + |BS(i)|) \quad \text{per } i = 0, \ldots, n-1$$

e

$$k_{i,j} = \begin{cases} -d_{i,j} & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases} \qquad \text{per } i, j = 0, \ldots, n-1 \text{ e } i \neq j$$

[1] Si noti che il passaggio da densa a sparsa è ovviamente impossibile
[2] Si noti che $K = K_0$

dove con $FS(i)$ e $BS(i)$ si indicano rispettivamente l'insieme degli archi uscenti da i, o stella uscente di i, e l'insieme degli archi entranti in i, o stella entrante di i.
Le matrici dei livelli inferiori possono essere ottenute tramite (1.11).
Come detto, il metodo si comporta in maniera differente a seconda che si trovi a dover inizializzare una matrice densa o una matrice sparsa. Questo perchè, in quest'ultima circostanza, non è conveniente immagazzinare la matrice come un vettore bidimensionale, ma è più efficiente utilizzare uno dei metodi standard di immagazzimento di matrici sparse. In questo caso si è scelto il formato CSR (Compressed Sparse Rows).

Vale la pena fare un'ultima considerazione. Le matrici dei coefficienti sono utilizzati dai metodi iterativi per il calcolo delle approssimazioni della soluzione ai vari livelli. Nella fase preliminare dell'implementazione, si era pensato che potesse essere più conveniente non calcolare esplicitamente le varie matrici, ma ricavare il prodotto $K_i x_i$, per un qualche livello i, attraverso una serie di moltiplicazioni, ovvero

$$K_i x_i = R_i R_{i-1} \dots R_0 K_0 P_0 P_1 \dots P_i x_i \quad \text{per un qualche livello i.}$$

L'idea era quella di trovare, sperimentalmente, un parametro α che facesse da discriminante fra il calcolo esplicito o meno delle matrici. Il tutto è stato poi accantonato, decidendo per il solo calcolo esplicito, quando in fase di implementazione ci si è accorti della necessità di conoscere, in ogni caso, ad ogni livello, perchè necessario a Jacobi, la diagonale della matrice dei coefficienti.

`SetD()`, una volta inizializzate le matrici dei vari livelli, compie un'altra importante operazione: la fattorizzazione LL^T, associata al metodo diretto di Cholesky, della matrice di ultimo livello. Infatti è in tale livello che il sistema (1.12) si reputa sufficientemente piccolo da poter essere risolto facilmente con un solver diretto.

void full_weighting() e void linear_interpolate()

Questi due metodi implementano la tecnica di restrizione e la tecnica di prolungamento analizzate in dettaglio nel capitolo 1. Si è deciso di calcolare, visto anche la semplicità delle due tecniche, il vettore risultato applicando una computazione elemento per elemento. Evitando, quindi, il calcolo come prodotto matrice-vettore. Quest'ultima tecnica, è stata comunque utilizzata, in fase di debug, per appurare l'effettiva correttezza della prima.

LSSStatus SolveADAT()

Questo metodo rappresenta, senza dubbio, il cuore del solver. Esso è implementato di modo che sia possibile risolvere il sistema (2.3) o mediante un V-cycle, o tramite un W-cycle[3]. La sua parte centrale è rappresentata da un ciclo in cui

- applica un'iterazione di multigrid;
- controlla se l'approssimazione corrente della soluzione ha raggiunto la precisione desiderata;
- e, se ciò non è accaduto e non ha esaurito la risorsa tempo a lui assegnata, ricicla.

L'iterazione di multigrid viene eseguita, attraverso il metodo `MGM()` analizzato più sotto, a partire da un'approssimazione iniziale fatta coincidere con il vettore nullo.
La precisione desiderata, invece, viene raggiunta se e quando

$$|\texttt{listaResidual[0][i]}| \leq \texttt{Prcsn[i]} \quad \text{per } i = 0, \ldots, n-1$$

in questo caso il metodo, che deve rispettare le specifiche dell'omonimo della superclasse, ritorna `kOK`. Se tale precisione non viene raggiunta, il metodo riapplica una nuova iterazione di multigrid, usando questa volta, come approssimazione iniziale, l'approssimazione corrente. Può accadere che il solver non possa ri-iterare perchè ha esaurito la risorsa tempo assegnatagli[4], se così fosse viene ritornato `kLwPrcsn`. Ci si potrebbe trovare nella situazione in cui la chiamata a `SolveADAT()` sia avvenuta perchè il chiamante, in questo caso l'algoritmo IP, non ha giudicato sufficientemente precisa l'approssimazione restituita da una chiamata precedente. Questo può voler dire che il chiamante ha bisogno necessariamente di una soluzione migliore, in questo caso, se dopo la chiamata la precisione non è ancora quella desiderata, il solver ritorna `kNError`.

void MGM()

Qui si implementa lo schema μ-cycle visto a pagina 14. Il metodo è un classico algoritmo ricorsivo in cui, se non si è giunti al livello più basso, si applica il pre-smoother, si restringe il residuo nel livello subito inferiore e si applica la ricorsione. Una volta

[3] In teoria può implementare anche altri schemi diversi, come il VW-cycle o il WV-cycle, ma in pratica si utilizzano solo questi due.

[4] Nel caso dei test effettuati sull'applicazione, questa risorsa tempo è stata fissata in 10 minuti.

giunti all'ultimo livello, si risolve direttamente e si risale, interpolando l'errore, correggendo l'approssimazione della soluzione che si ha in quel livello e rilassando con il post-smoother, fino a giungere al livello di partenza ottenendo l'approssimazione finale.
La soluzione (diretta) del sistema all'ultimo livello avviene utilizzando il metodo di Cholesky sulla matrice precedentemente fattorizzata in `SetD()`.

Capitolo 4

Test

In questo capitolo vengono presentati i risultati dei test compiuti sull'applicazione, al fine di evidenziarne la correttezza implementativa, ma anche i limiti che possiede.

Per effettuare i test si sono usati tre noti generatori casuali di problemi di MCF: *goto* (GridOnTOrus), *gridgen* e *netgen* [3]. Per ogni generatore, si sono create un totale di 4 istanze di grafo, con numeri di nodi $n = 2^k$ per $k = 12, 14$ e con due differenti densità di archi. In particolare, per $k = 12$ si sono generate istanze con densità 64 e 256, per $k = 14$ si sono generate istanze con densità 8 e 64. Nel seguito, si userà la forma genX_Y per riferirsi alle istanze generate dal generatore gen (net, goto o grid), con k =X e densità uguale a Y.

Per tutte le istanze precedenti, si è lanciato il KrskPCGBCTLSSolver per ottenere i dati per poter riprodurre i sistemi lineari di ogni iterazione IP (la matrice D e il vettore dei termini noti). Successivamente, si è testata l'implementazione del multigrid su questi sistemi, variando alcuni dei parametri fondamentali che lo caratterizzano (il tipo di ciclo utilizzato, il numero di iterazioni degli smoother, il numero di livelli). In questo modo, si è sicuri che il test avvenga esattamente sugli stessi sistemi lineari, così che i risultati possano essere confrontati fra di loro in modo consistente. Si è indicato con ADATi, il sistema lineare all'iterazione i-esima dell'IP.

4.1 Two-grid

I test effettuati sulle istanze sopra descritte, quando la dimensione del livello più basso è metà rispetto alla dimensione del sistema di partenza, mostrano che l'applicazione implementata è convergente.

Era possibile concludere ciò anche solo notando, che il metodo di Jacobi rilassato è convergente, quando applicato alle M-matrici [1] e che una CGC "fatta bene" non può che migliorare l'approssimazione della soluzione del sistema. Dai risultati ottenuti, quindi, si può concludere, con un certo grado di ragionevolezza, che l'implementazione realizzata è, dal punto di vista algoritmico, corretta. La tabella 4.1 mostra il grado di accuratezza ottenuto, applicando alle istanze definite in precedenza il Two-grid ed effettuando 5 iterazioni del metodo di Jacobi.

4.2 Multigrid

Aumentando il numero di livelli del metodo e effettuando i test, ci si è accorti che alcune scelte effettuate, circa i vari componenti del multigrid, si sono rivelate infelici. I risultati, infatti, mostrano che non sempre l'applicazione risulta convergente (tabelle 4.2 e **??**). I motivi della non convergenza sono da ricercarsi nel fatto che lo smoother scelto non è "compatibile" con le tecniche di proiezione utilizzate. Infatti, calcolando la matrice di livello inferiore attraverso l'equazione (1.10) e scegliendo come operatore di restrizione il full weighting operator e come operatore di prolungamento l'interpolazione lineare, succede che la matrice ottenuta non risulta più una matrice di grafo e che quindi il metodo iterativo di Jacobi non è garantito essere convergente [1, 6].
Per porre rimedio a tale problema si possono, ad esempio, seguire due strade:

- cambiare pre-smoother e post-smoother. Sostituendo, ad esempio, il metodo di Jacobi con quello di Gauss-Seidel e mantenendo gli stessi proiettori, infatti, è possibile affermare che l'applicazione diventa convergente anche con un numero di livelli maggiori di due. Per dimostrare ciò è sufficiente far notare che le matrici ai vari livelli continuano ad essere definite positive[1] e che una condizione necessaria e sufficiente per la convergenza del metodo di Gauss-Seidel è data, per l'appunto, dal fatto che la matrice a cui è applicato sia definita positiva [1];
- definire un proiettore o una tecnica di proiezione più robusta, prendendo spunto ad esempio da quelle che stanno alla base del multigrid algebrico [2, 10].

[1] Se $x^T A_n x > 0 \forall x \neq 0$, allora anche $x^T (R_n A_n P_n) x > 0$.

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_64	1	2.66	$5.7839e^{-9}$
	2	2.61	$1.50781e^{-9}$
	16	7.75	$1.41923e^{-7}$
	29	83.46	$4.96386e^{-8}$
	43	-	$2.02958e^{-4}$
	57	-	$7.54476e^{-4}$
	33	-	$7.38277e^{-4}$
goto12_64	1	1.77	$2.59094e^{-10}$
	2	-	$5.13344e^{-3}$
	35	-	$2.28314e^{-2}$
	68	-	$8.47913e^{-3}$
	101	-	$3.71472e^{-3}$
	134	-	$2.25789e^{-3}$
	135	-	$2.03166e^{-3}$
grid12_64	1	0.65	$7.29039e^{-10}$
	2	2.14	$5.55327e^{-10}$
	11	9.61	$7.0397e^{-9}$
	20	34.58	$6.88987e^{-9}$
	27	-	$4.20201e^{-4}$
	38	-	$4.41726e^{-4}$
	39	-	$4.45042e^{-4}$

(a)

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_256	1	6.7	$1.42917e^{-10}$
	2	8.76	$9.67959e^{-10}$
	26	27.68	$4.06461e^{-8}$
	50	153.39	$1.59783e^{-8}$
	74	-	$2.86437e^{-4}$
	97	-	$5.03135e^{-4}$
	98	-	$4.9369e^{-4}$
goto12_256	1	1.44	$6.03264e^{-12}$
	2	-	$4.26652e^{-3}$
	110	-	$3.80021e^{-2}$
	218	-	$2.61684e^{-3}$
	326	-	$7.20086e^{-3}$
	436	-	$1.70608e^{-3}$
	437	-	$1.67122e^{-3}$
grid12_256	1	2.4	$4.07013e^{-9}$
	2	2.91	$6.37872e^{-10}$
	11	22.47	$9.85046e^{-9}$
	19	-	$3.77749e^{-7}$
	27	-	$2.30225e^{-4}$
	35	-	$1.15765e^{-3}$
	36	-	$1.13036e^{-3}$

(b)

Tabella 4.1: Accuratezza del Two-grid con V-cycle e 5 iterazioni di Jacobi.
(-) indica che è stato raggiunto il tempo limite concesso per l'esecuzione, senza raggiungere la precisione sperata.

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_64	1	18.92	$3.41427e^{-9}$
	2	17.01	$5.62336e^{-9}$
	10	68.81	$4.88242e^{-8}$
	17		*
	25		*
	32		*
	33		*
goto12_64	1	24.9	$4.4219e^{-10}$
	2		*
	35		*
	68		*
	101		*
	134		*
	135		*
grid12_64	1	3.64	$3.07552e^{-9}$
	2	9.82	$8.16601e^{-10}$
	11	57.73	$7.47697e^{-9}$
	20	174.05	$7.07177e^{-9}$
	29	-	$2.05729e^{-3}$
	38	-	$1.34936e^{-3}$
	39	-	$1.38697e^{-3}$

(a)

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_256	1	26.11	$1.40228e^{-10}$
	2	31.03	$5.77497e^{-10}$
	26		*
	50		*
	74		*
	97		*
	98		*
goto12_256	1	8.75	$9.79582e^{-12}$
	2		*
	110		*
	218		*
	326		*
	436		*
	437		*
grid12_256	1	9.74	$2.39424e^{-9}$
	2	48.76	$7.81392e^{-10}$
	11	42.9	$1.27442e^{-8}$
	19	-	$2.25867e^{-6}$
	27	-	$2.22638e^{-3}$
	35	-	$2.59222e^{-3}$
	36	-	$2.6248e^{-3}$

(b)

Tabella 4.2: Comportamento del Multigrid (W-cycle, 5 iterazioni di Jacobi, livello più basso di dimensione 512).
(*) indica che il metodo diverge.

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_64	1	16.76	$5.77607e^{-9}$
	2	18.12	$1.62179e^{-9}$
	10	67.88	$4.88225e^{-8}$
	17		*
	25		*
	32		*
	33		*
goto12_64	1	26.61	$3.62926e^{-10}$
	2		*
	35		*
	68		*
	101		*
	134		*
	135		*
grid12_64	1	5.44	$4.79723e^{-10}$
	2	10.96	$8.16599e^{-10}$
	11	64.44	$7.72797e^{-9}$
	20	200.42	$7.07203e^{-9}$
	29	-	$2.23752e^{-3}$
	38	-	$1.47711e^{-3}$
	39	-	$1.47323e^{-3}$

(a)

rete	ADAT	tempo(sec.)	accuratezza($\frac{\|r\|}{\|b\|}$)
net12_64	1	26.77	$1.79609e^{-9}$
	2	30.74	$1.23653e^{-10}$
	26	52.43	$5.11938e^{-8}$
	50		*
	74		*
	97		*
	98		*
goto12_64	1	32.39	$3.71297e^{-11}$
	2		*
	110		*
	218		*
	326		*
	436		*
	437		*
grid12_64	1	8.75	$1.02704e^{-9}$
	2	11.24	$9.40645e^{-10}$
	11	85.95	$1.03773e^{-8}$
	19	193.39	$5.96512e^{-9}$
	27		*
	35		*
	36		*

(b)

Tabella 4.3: (a) Comportamento del Multigrid con W-cycle, 5 iterazioni di Jacobi e livello più basso di dimensione 256. (b) Comportamento del Multigrid con W-cycle, 10 iterazioni di Jacobi e livello più basso di dimensione 512.

Capitolo 5

Conclusioni

Si è giunti alla fine di questo lavoro ed è tempo di analizzare tutto quello che si è fino ad ora trattato per capire se gli obiettivi che il tirocnio si prefiggeva sono stati raggiunti e per identificarne, eventualmente, di nuovi.

Di seguito, verrà fatta una valutazione critica del lavoro svolto e delle tecnologie utilizzate e, successivamente, si elencheranno quali potrebbero essere gli sviluppi futuri possibili.

5.1 Valutazione critica del lavoro svolto e degli strumenti utilizzati

Gli obiettivi che il presente lavoro si poneva di raggiungere erano essenzialmente questi:

1. costruire una specializzazione della classe astratta MCFLSSolver che implementasse l'algoritmo del multigrid algebrico utilizzando componenti generiche (eventualmente esse stesse dei MCFLSSolver) per le diverse parti dell'approccio;

2. individuare una o più implementazioni interessanti per ciascuna e testare gli approcci così ottenuti su un insieme (già noto e sviluppato) di istanze, utilizzando i moduli già disponibili e/o implementandone di nuovi, eventualmente appoggiandosi a librerie di algebra lineare esistenti;

3. confrontare gli approcci così ottenuti con i migliori algoritmi disponibili (già implementati all'interno del framework) basati sul GCP.

I primi due obiettivi possono considerarsi parzialmente raggiunti. Per quanto riguarda l'efficienza dell'applicazione, essa non è tuttora in grado di reggere il confronto con gli algoritmi del gradiente coniugato precondizionato presenti nel framework. Ciò è essenzialmente dovuto al fatto che la ricorsione fa perdere molto tempo e che le tecniche utilizzate per le varie parti del multigrid, sono state scelte più per la loro semplicità, che per la loro reale efficienza (si pensi ad esempio a Jacobi).

È significativo spendere qualche parola sugli strumenti esterni utilizzati in fase di realizzazione, ossia la libreria MTL e ITL. Le due librerie, inizialmente scelte perchè sembravano adattarsi benissimo al compito che dovevano svolgere, si sono rivelate una fonte considerevole di problemi. Innanzitutto perchè contenevano parecchi errori nel codice[1], che hanno reso parecchio difficoltosa la fase di debug e ,in secondo luogo, perchè non sempre sono risultate ben documentate e per questo piuttosto ostiche da sfruttare. Sostanzialmente, essendo abbastanza scarsi anche i commenti interni al codice, è stato necessario, in alcuni frangenti, adottare un procedimento di tipo "try & fail", per comprendere appieno alcuni aspetti dell'esecuzione del codice.

5.2 Possibili sviluppi futuri

In questa sezione si elencano brevemente quelle che possono essere le migliorie future, apportabili al software realizzato.
Come già detto nel capitolo precedente, è necesssario per ottenere un metodo almeno convergente, anche nel caso si abbiano più di due livelli, utilizzare uno smoother che assicuri la convergenza quando la matrice su cui itera è definita positiva.
In realtà, per avere un'applicazione che, in termini di prestazioni, possa reggere il confronto con le tecniche PCG è necessario definire una tecnica di proiezione che prenda spunto da quella del multigrid algebrico. L'idea di quest'ultimo è quella di definire l'operazione di proiezione sfruttando le proprietà spettrali della matrice dei coefficienti del sistema, che quindi dovrà avere proprietà strutturali tali da permettere di determinare le informazioni necessarie per definire tale operazione. Essendo il MGM un metodo ricorsivo, è di fondamentale importanza che queste proprietà strutturali siano mantenute anche dalle matrici dei sistemi dei livelli inferiori, altrimenti risulterebbe complesso definire una strategia uniforme di proiezione del problema [4].

[1] A tal proposito si spera che la nuova versione, in fase di testing, risulti, almeno da questo punto di vista, migliore.

Comunque il codice dell'applicazione è strutturato in maniera tale da rendere tali modifiche possibili, senza dover apportare grandi rivoluzioni all'assetto generale.

Per finire, è giusto accennare che esiste un'ulteriore tecnica che potrebbe portare ad un miglioramento dell'applicazione. L'idea, molto brevemente, consiste nell'utilizzare un metodo di proiezione che, proiettando le matrici nei vari livelli, riesca a far loro mantenere la struttura di matrice di grafo. Un riferimento a ciò può essere trovato in [11], che però parla di matrici di grafo diverse da quelle su cui si è basato questo lavoro. Quindi, qualora si volesse seguire questa strada, è essenziale un approfondito studio teorico, per poter capire se e come tale approccio può essere sfruttato ai fini del presente lavoro.

Bibliografia

[1] D. Bini, M. Capovani e O. Menchi. *Metodi numerici per l'algebra lineare*. Zanichelli, Bologna, Italy, 1988.

[2] W. L. Briggs, V. E. Henson e S. F. McCormick. *A Multigrid Tutorial*. siam, Philadelphia, seconda ed., 2000.

[3] Codice sorgente dei generatori casuali di problemi MCF goto, gridgen e netgen. http://www.di.unipi.it/optimize/Data/MMCF.html.

[4] M. Donatelli. *Metodi multigrid per sistemi lineari strutturati e applicazioni*. Tesi di laurea, Università degli Studi di Firenze - Facoltà di Scienze Matematiche, Fisiche e Naturali - Corso di Laurea in Informatica, 2002.

[5] M. A. Ellis e B. Stroustrup. *The Annotated C++ Reference Manual.*. Addison-Wesley, 1990.

[6] A. Frangioni e C. Gentile. New Preconditioners for KKT Systems of Network Flow Problems. *SIAM Journal on Optimization*, vol. 14:pp. 894–913, mag. 2004.

[7] A. Frangioni e C. Gentile. Prim-based BCT preconditioners for Min-Cost Flow Problems. pp. 271–287, 2007.

[8] Iterative Template Library. http://www.osl.iu.edu/research/itl/.

[9] Matrix Template Library. http://osl.iu.edu/research/mtl/.

[10] J. W. Ruge e K. Stüben. Algebraic multigrid. In *Multigrid methods*, vol. 3 di *Frontiers Appl. Math.*, pp. 73–130. SIAM, Philadelphia, PA, 1987.

[11] H. D. STERCK, T. A. MANTEUFFEL, S. F. MCCORMICK, Q. NGUYEN e J. RUGE. Multilevel Adaptive Aggregation for Markov Chains, with Application to WEB Ranking. *SIAM Journal on Scientific Computing*, 2007.