UNIVERSITÀ DEGLI STUDI DI PARMA

Facoltà di Scienze

Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Tesi di Laurea

Un'implementazione incrementale e su aritmetica esatta del simplesso primale

Candidato:
Andrea Cimino

Relatore:
Dott. Enea Zaffanella

Correlatore:
Prof. Roberto Bagnara

Anno Accademico 2004/2005

Indice

1 Introduzione **3**

1.1 La programmazione lineare 4

1.1.1 Il problema 5

1.1.2 Forma standard 6

1.1.3 Base di un problema di programmazione lineare 9

1.2 L'algoritmo del simplesso primale 11

1.2.1 Controllo di ottimalità e scelta della variabile entrante in base 13

1.2.2 Controllo di illimitatezza e scelta della variabile uscente dalla base 14

1.2.3 Operazione di cambio di base (pivoting) 15

1.2.4 Trovare una base ammissibile per l'algoritmo 16

1.2.5 Algoritmo *steepest-edge* per la variabile entrante 18

1.3 Aritmetica esatta ed Incrementalità 19

1.3.1 Aritmetica esatta 19

1.3.2 Incrementalità 21

2 Implementazione nella PPL **27**

2.1 Interfaccia Utente 27

2.1.1 Esempio d'uso della classe LP_Problem 32

2.2 Implementazione 33

3 Valutazioni sperimentali **42**

3.1 Test non incrementali 44

3.2 Test incrementali 45

4 Conclusioni **46**

Bibliografia **46**

Capitolo 1

Introduzione

Il presente lavoro di tesi ha come obiettivo l'implementazione di un risolutore di problemi di programmazione lineare nel contesto dello sviluppo della *Parma Polyhedra Library* [PPL06, BRZH02], che nel seguito, per brevità, si indicherà con l'acronimo PPL. Come dice il nome, la PPL è una libreria software nata per fornire funzionalità per la manipolazione di poliedri convessi (si noti comunque che la libreria è stata in seguito estesa ad altri domini numerici); in particolare, la libreria fornisce un supporto pressoché completo per lo sviluppo di applicazioni nel campo dell'analisi statica di sistemi software e hardware. In tale contesto, il presente lavoro di tesi si prefigge di sviluppare un risolutore di problemi di programmazione lineare caratterizzato dalle seguenti proprietà:

- **Aritmetica esatta.** La maggior parte delle implementazioni dell'algoritmo del simplesso è basata sull'aritmetica *floating-point* (ovvero, in virgola mobile) standard del processore. Tale scelta, effettuata per motivi di efficienza, rende tali implementazioni soggette ad errori di arrotondamento più o meno significativi; cosa ancora peggiore, tali errori di arrotondamento possono tipicamente propagarsi in modo incontrollato: in mancanza di opportuni meccanismi di validazione delle soluzioni calcolate, essi possono

portare a risultati totalmente inaffidabili (ad esempio, al calcolo di soluzioni "ottime" quando il problema di partenza non ammette soluzione alcuna). Questa situazione non è accettabile in contesti, quali quello dell'analisi statica, nei quali la correttezza della computazione è un requisito irrinunciabile, per ottenere il quale è lecito sacrificare l'efficienza del calcolo. Per questo motivo, l'implementazione studiata utilizza tecniche che garantiscono l'esattezza dei calcoli eseguiti.

- **Incrementalità.** In alcune applicazioni capita di dover determinare la soluzione di un numero elevato di problemi "simili". Ad esempio, può capitare di dovere ottimizzare diverse funzioni obiettivo definite sullo stesso spazio delle soluzioni; in altri casi, può essere necessario effettuare una ricerca euristica all'interno di uno spazio di soluzioni molto complesso, partendo da una soluzione parziale e raffinandola passo passo, mediante l'aggiunta incrementale dei vincoli del problema. In entrambi i casi, ai fini dell'efficienza, è importante evitare di ripetere un numero elevato di volte porzioni significative della computazione. Le tecniche di calcolo incrementali consentono di fattorizzare buona parte di questi calcoli ripetuti, consentendo di determinare le soluzioni delle varianti del problema a partire dalla soluzione precedentemente calcolata per il problema originale.

1.1 La programmazione lineare

Per poter comprendere a fondo il tema trattato nel lavoro di tesi, introduciamo i concetti definiti qui di seguito, aiutandoci, per maggiore chiarezza, con un semplice esempio di problema di programmazione lineare.

1.1.1 Il problema

Definizione 1.1 *Un generico problema di* programmazione lineare *consiste nel calcolare*

$$\max \sum_{j=1}^{n} c_j x_j \tag{1.1}$$

in modo da soddisfare gli m vincoli

$$\begin{cases} \left\{\sum_{j=1}^{n} a_{ij}x_j \leq b_i \;\middle|\; i \in I_1\right\} \\ \left\{\sum_{j=1}^{n} a_{ij}x_j \geq b_i \;\middle|\; i \in I_2\right\} \\ \left\{\sum_{j=1}^{n} a_{ij}x_j = b_i \;\middle|\; i \in I_3\right\} \end{cases} \tag{1.2}$$

La (1.1) è detta funzione obiettivo *mentre la (1.2) è detta* regione ammissibile*. Devono inoltre valere le due seguenti proprietà:*

$$I_1 \cup I_2 \cup I_3 = \{1, \ldots, m\},$$

$$I_1 \cap I_2 = I_1 \cap I_3 = I_2 \cap I_3 = \emptyset.$$

La regione ammissibile, quindi, è descritta da equazioni e disequazioni non strette. Questa è la forma più generale per definire un problema di programmazione lineare. Individuiamo tre classi di problemi che dipendono dalla regione ammissibile e dalla funzione obiettivo da ottimizzare:

1. **Problema insoddisfacibile**: il caso si presenta se la regione di spazio, individuata dal sistema di vincoli, è vuota.

2. **Problema con ottimo finito**: esiste almeno un punto appartenente alla regione ammissibile che ottimizza il problema. Da sottolineare il fatto che, siccome i vincoli e la funzione obiettivo sono lineari e definiti su di uno spazio denso, se esiste più di un punto ottimale, allora questi sono infiniti. Geometricamente questa situazione può essere descritta da due vertici di un

poliedro che sono soluzioni ottime: in questo caso il segmento che congiunge i due vertici contiene infinite soluzioni ottime.

3. **Problema con ottimo illimitato**: la regione ammissibile non è vuota, ma non esiste alcun punto in grado di ottimizzare il problema. In questo caso la funzione obiettivo è non limitata superiormente.

Abbiamo parlato di massimizzazione di una funzione obiettivo: ma se volessimo invece minimizzarla? Possiamo sfruttare la seguente relazione:

$$\min \sum_{j=1}^{n} c_j x_j \;=\; -\max - \sum_{j=1}^{n} c_j x_j .$$

Siamo in grado quindi di esprimere indistintamente problemi di massimizzazione e minimizzazione.

Un esempio concreto di problema di programmazione lineare è il seguente

$$\max \quad x_1 + 2x_2 + 2x_3 \tag{1.3}$$

$$\begin{cases} x_1 + x_2 + x_3 \leq 4 \\ x_1 \leq 2 \\ x_3 \leq 3 \\ 3x_2 + x_3 \leq 6 \\ x_1, x_2, x_3 \geq 0 \end{cases} \tag{1.4}$$

Quale è il punto (sempre che esista) appartenente al sistema di vincoli definito dalle (1.4) in grado di massimizzare la (1.3)?

1.1.2 Forma standard

Preliminari

Per ogni $i \in \{1, \ldots, n\}$, v_i denota la i-esima componente a valore reale del vettore colonna $\boldsymbol{v} = \langle v_1, \ldots, v_n \rangle \in \mathbb{R}^n$. Un vettore $\boldsymbol{v} \in \mathbb{R}^n$ può anche essere interpretato

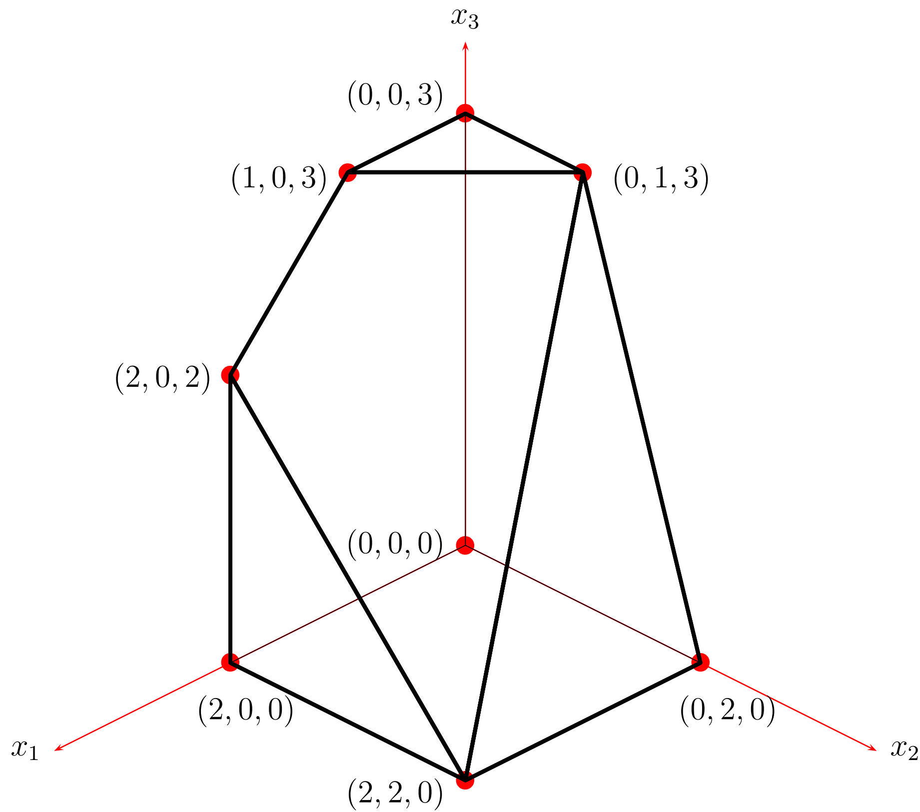


Figura 1.1: Il poliedro definito dalle (1.4)

come una matrice $\mathbb{R}^{n\times 1}$ e manipolato in modo appropriato con le usuali definizioni per l'addizione, la moltiplicazione (sia per uno scalare che per un'altra matrice), la trasposizione, che è denotata da $\boldsymbol{v}^{\mathrm{T}}$, cosicché $\langle v_1, \ldots, v_n\rangle = (v_1, \ldots, v_n)^{\mathrm{T}}$. Il *prodotto scalare* di $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ è il numero reale $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{w} = \sum_{i=1}^{n} v_i w_i$. Scriveremo $\mathbf{0}$ per denotare un vettore $\mathbb{R}^n$ che ha tutte le sue componenti uguali a zero; la dimensione n sarà chiara dal contesto.

Definizione 1.2 *Siano dati*

$$\langle c_1, \ldots, c_n\rangle = \boldsymbol{c} \in \mathbb{R}^n$$

$$\langle x_1, \ldots, x_n\rangle = \boldsymbol{x} \in \mathbb{R}^n$$

$$\langle b_1, \ldots, b_m\rangle = \boldsymbol{b} \in \mathbb{R}^m$$

$$A \in \mathbb{R}^{m\times n}.$$

Un problema di programmazione lineare si dice in forma standard *se definito da*

$$\max \quad \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \tag{1.5}$$

$$\begin{cases} A\boldsymbol{x} = \boldsymbol{b} \\ \boldsymbol{x} \geq \boldsymbol{0} \end{cases} \tag{1.6}$$

Questa è la forma del problema di programmazione lineare gestita dall'algoritmo dal simplesso primale. Si dimostra che ogni problema di programmazione lineare ammette una *forma standard*. Vediamo come avviene la trasformazione: questa si divide in due fasi. Nella prima si rendono non negative le variabili del problema; nella seconda si trasformano le disequazioni in equazioni.

1. Per trasformare una variabile libera in segno (non limitata né superiormente né inferiormente) della forma

   $$x_j \lesseqgtr 0$$

   creiamo due variabili x_j^+ e x_j^- e imponiamo

   $$x_j = x_j^+ - x_j^- \qquad \text{con} \qquad x_j^+ \geq 0 \quad \text{e} \quad x_j^- \geq 0.$$

   In ogni vincolo e nella funzione obiettivo del problema di programmazione lineare dove è presente la variabile x_j, eseguiamo la trasformazione appena introdotta ed inseriamo i vincoli di non negatività corrispondenti.

2. Data una generica disequazione della forma

   $$\sum_{j=1}^{n} a_{ij}x_j \geq b_i$$

   introduciamo una nuova variabile s_i, chiamata anche variabile di *slack*. Possiamo quindi scrivere

   $$\sum_{j=1}^{n} a_{ij}x_j - s_i = b_i, \qquad s_i \geq 0.$$

Nel caso invece di disequazioni della forma

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i,$$

queste vengono trasformate in

$$\sum_{j=1}^{n} a_{ij}x_j + s_i = b_i, \qquad s_i \geq 0.$$

Ad esempio l'equazione

$$x_1 \leq 2$$

appartenente al sistema definito dalle (1.4), secondo i criteri appena descritti, viene trasformata in

$$x_1 + s_2 = 2, \qquad s_2 \geq 0.$$

Da notare che non è necessario introdurre una nuova variabile x_1^-, perché il vincolo

$$x_1 \geq 0$$

ci assicura la non negatività di x_1.

La corrispondente forma standard della (1.3) e delle (1.4) è

$$\max \quad x_1 + 2x_2 + 2x_3 \tag{1.7}$$

$$\begin{cases} x_1 + x_2 + x_3 + s_1 = 4 \\ x_1 + s_2 = 2 \\ x_3 + s_3 = 3 \\ 3x_2 + x_3 + s_4 = 6 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 \geq 0 \end{cases} \tag{1.8}$$

1.1.3 Base di un problema di programmazione lineare

Introduciamo adesso un nuovo concetto: la base di un problema di programmazione lineare in forma standard.

Definizione 1.3 (Base) *Si dice* base *di un problema di programmazione lineare in forma standard, un sottoinsieme:*

$$B = \{x_{i_1}, \dots, x_{i_m}\}$$

di m delle n variabili del problema di programmazione lineare con la proprietà che la matrice $A_B \in \mathbb{R}^{m \times m}$, ottenuta considerando le sole colonne di A di indice i_k tali che $x_{i_k} \in B$ per $k = 1, \dots, m$, sia invertibile.

Le variabili dell'insieme B sono dette *variabili in base*, quelle al di fuori di B sono raggruppate nell'insieme

$$N = \{x_{i_{m+1}}, \dots, x_{i_n}\}.$$

Queste vengono dette *variabili fuori base.* Chiameremo inoltre $A_N \in \mathbb{R}^{(n-m)\times m}$ la matrice ottenuta considerando le sole colonne di A relative alle variabili $x_i \in N$.

Definizione 1.4 (Soluzione di base) *Si dice soluzione di base associata alla base B, il vettore $\boldsymbol{x} \in \mathbb{R}^n$ con*

$$x_{i_j} = \begin{cases} 0 & se \quad x_{i_j} \in N \\ \left(A_B^{-1}\boldsymbol{b}\right)_j & se \quad x_{i_j} \in B \end{cases}$$

Se $A_B^{-1}\boldsymbol{b} \geq \mathbf{0}$, la soluzione si dice ammissibile. *Se inoltre si ha $A_B^{-1}\boldsymbol{b} > \mathbf{0}$, questa si dice* soluzione non degenere, *altrimenti si dice* degenere.

La differenza tra una soluzione di base *degenere* ed una *non degenere* è che la prima può essere rappresentata da più basi, la seconda esclusivamente da una sola base.

Nell'esempio da noi considerato, una base del problema è data dall'insieme di variabili

$$B = \{s_1, s_2, s_3, s_4\}.$$

Sia

$$\langle x_1, x_2, x_3, s_1, s_2, s_3, s_4 \rangle = \boldsymbol{x} \in \mathbb{R}^7,$$

la soluzione di base associata a B è

$$\boldsymbol{x} = \langle 0, 0, 0, 4, 2, 3, 6 \rangle.$$

Questa base è ammissibile e non degenere per il problema di programmazione lineare in quanto $A_B^{-1}\boldsymbol{b} > \boldsymbol{0}$.

1.2 L'algoritmo del simplesso primale

Fatta una panoramica generale sui concetti di base della programmazione lineare, abbiamo le nozioni fondamentali per poter descrivere l'algoritmo del simplesso primale. È necessario però prima trasformare il problema in forma "esplicita" in funzione delle variabili in base. Vediamo come si arriva a questa descrizione. Sfruttiamo la notazione introdotta nella sezione 1.1.3, e introduciamo quattro nuovi vettori:

$$\langle x_1, \ldots, x_m \rangle = \boldsymbol{x}_B \in \mathbb{R}^m \quad \text{dove} \quad x_i \in \boldsymbol{x}_B \Longleftrightarrow x_i \in B$$

$$\langle x_{m+1}, \ldots, x_n \rangle = \boldsymbol{x}_N \in \mathbb{R}^{n-m} \quad \text{dove} \quad x_i \in \boldsymbol{x}_N \Longleftrightarrow x_i \in N$$

$$\langle c_1, \ldots, c_m \rangle = \boldsymbol{c}_B \in \mathbb{R}^m \quad \text{dove} \quad c_i \in \boldsymbol{c}_B \Longleftrightarrow x_i \in \boldsymbol{x}_B$$

$$\langle c_{m+1}, \ldots, c_n \rangle = \boldsymbol{c}_N \in \mathbb{R}^{n-m} \quad \text{dove} \quad c_i \in \boldsymbol{c}_N \Longleftrightarrow x_i \in \boldsymbol{x}_N.$$

Possiamo quindi riscrivere la (1.5) e le (1.6) nella seguente forma

$$\max \quad \boldsymbol{c}_B^{\mathrm{T}}\boldsymbol{x}_B + \boldsymbol{c}_N^{\mathrm{T}}\boldsymbol{x}_N$$

$$\begin{cases} A_B\boldsymbol{x}_B + A_N\boldsymbol{x}_N = \boldsymbol{b} \\ \boldsymbol{x}_B \geq \boldsymbol{0} \\ \boldsymbol{x}_N \geq \boldsymbol{0} \end{cases}$$

o ancora

$$\max \quad \boldsymbol{c}_B^{\mathrm{T}}\boldsymbol{x}_B + \boldsymbol{c}_N^{\mathrm{T}}\boldsymbol{x}_N$$

$$\begin{cases} A_B\boldsymbol{x}_B = \boldsymbol{b} - A_N\boldsymbol{x}_N \\ \boldsymbol{x}_B \geq \boldsymbol{0} \\ \boldsymbol{x}_N \geq \boldsymbol{0} \end{cases}$$

Moltiplicando dove necessario per A_B^{-1} otteniamo

$$\max \quad \boldsymbol{c}_B A_B^{-1} + (\boldsymbol{c}_N - \boldsymbol{c}_B A_B^{-1} A_N)\boldsymbol{x}_N$$

$$\begin{cases} \boldsymbol{x}_B = A_B^{-1}\boldsymbol{b} - A_B^{-1}A_N\boldsymbol{x}_N \\ \boldsymbol{x}_B \geq \boldsymbol{0} \\ \boldsymbol{x}_N \geq \boldsymbol{0} \end{cases}$$

Indicando con

- γ_0 il valore di $\boldsymbol{c}_B A_B^{-1}$ (il valore corrente della funzione obiettivo)
- γ_j, per $j = 1, \ldots, n-m$, le componenti del vettore $\boldsymbol{c}_N - \boldsymbol{c}_B A_B^{-1} A_N$
- β_{rj}, per $r = 1, \ldots, m$, le componenti del vettore $A_B^{-1}\boldsymbol{b}$
- α_{rj}, per $r = 1, \ldots, m$, $j = 1, \ldots, n-m$, le componenti della matrice $-A_B^{-1}A_N$

possiamo riscrivere tutto come

$$\max \quad \gamma_0 + \sum_{j=1}^{n-m} \gamma_j x_{m+j} \tag{1.9}$$

$$\begin{cases} x_1 = \beta_1 + \sum_{j=1}^{n-m} \alpha_{1j} x_{m+j} \\ \ldots \\ x_k = \beta_k + \sum_{j=1}^{n-m} \alpha_{kj} x_{m+j} \\ \ldots \\ x_m = \beta_m + \sum_{j=1}^{n-m} \alpha_{mj} x_{m+j} \end{cases} \tag{1.10}$$

Questa è detta riformulazione esplicita del problema di programmazione lineare rispetto alla base B.

Nella sezione 1.1.3 abbiamo trovato, per il nostro esempio, una base ammissibile. La riformulazione esplicita del problema in funzione di questa base è

$$\max \quad x_1 + 2x_2 + 2x_3 \tag{1.11}$$

$$\begin{cases} s_1 = 4 - x_1 - x_2 - x_3 \\ s_2 = 2 - x_1 \\ s_3 = 3 - x_3 \\ s_4 = 6 - 3x_2 - x_3 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 \geq 0 \end{cases} \tag{1.12}$$

L'algoritmo del simplesso primale si basa su tre passaggi ripetuti fino al raggiungimento di una condizione d'uscita. Questi sono:

1. Controllo di ottimalità e scelta della variabile entrante in base.

2. Controllo di illimitatezza e scelta della variabile uscente dalla base.

3. Cambio di base (pivoting).

1.2.1 Controllo di ottimalità e scelta della variabile entrante in base

Per controllare che non sia possibile migliorare ulteriormente il valore della funzione obiettivo, terminando con successo l'algoritmo, basta osservare il segno dei coefficienti delle variabili fuori base della stessa funzione obiettivo: se sono tutti strettamente negativi, cioè

$$\forall x_j \in N : \; \gamma_j < 0$$

l'algoritmo del simplesso ha ottimizzato il problema. Nel caso in cui questa condizione non sia soddisfatta, è necessario scegliere una variabile da fare entrare

in una nuova base: un possibile criterio, che garantisce di non peggiorare il valore della funzione obiettivo, è quello di scegliere una variabile x_{m+h} avente coefficiente non negativo nella funzione obiettivo. Questo criterio ha lo svantaggio di soffrire del problema della "ciclicità". Potrebbe capitare infatti, dopo una serie di passi, di ritornare ad una base già calcolata in precedenza: in questo caso l'algoritmo non avrebbe speranza di terminare. Un criterio alternativo, in grado di ovviare al problema appena descritto, è quello di *Bland per le variabili entranti in base.* In questo caso si sceglie la variabile fuori base x_{m+h}, con coefficiente non negativo nella funzione obiettivo, avente indice più piccolo.
Ritorniamo al nostro esempio: al primo passo, la funzione obiettivo è definita da

$$\max \quad x_1 + 2x_2 + 2x_3$$

e l'insieme di variabili fuori base è

$$N = \{x_1, x_2, x_3, x_4\}.$$

I corrispondenti valori γ_j sono

$$\gamma_1 = 1$$
$$\gamma_2 = 2$$
$$\gamma_3 = 2$$
$$\gamma_4 = 0$$

Possiamo quindi concludere che la variabile entrante è x_1.

1.2.2 Controllo di illimitatezza e scelta della variabile uscente dalla base

Supponiamo che non sia stata soddisfatta la condizione di ottimalità, quindi sia stata scelta x_{m+h} come variabile entrante. Resta da vedere quale è la variabile che

le farà posto. Per decidere quale variabile dovrà uscire dalla base, utilizzeremo il seguente criterio: sceglieremo la variabile $x_k \in B$ tale che

$$-\frac{\beta_k}{\alpha_{kh}} = \min\left\{ -\frac{\beta_r}{\alpha_{rh}} \;\middle|\; \alpha_{rh} < 0 \right\}.$$

Nel caso in cui ci siano più variabili candidate, sceglieremo quella di indice inferiore, questo sempre per evitare il ciclo (*Regola di Bland per la variabile uscente dalla base*). Nel caso in cui non ci sia alcuna variabile candidata, si può concludere che l'obiettivo da ottimizzare è *illimitato*, facendo terminare l'algoritmo.
Ritornando al nostro esempio, notiamo che sono due la variabili candidate ad uscire: s_1, s_2. Sceglieremo però come variabile uscente s_2 in quanto

$$-\frac{2}{-1} < -\frac{4}{-1}$$

1.2.3 Operazione di cambio di base (pivoting)

Una volta determinata la nuova base $B' = \{x_1, ..., x_{k-1}, x_{m+h}, x_{k+1}, ..., x_m\}$, dovremo esprimere il problema di programmazione lineare tramite B'. Si otterrà, mediante semplici sostituzioni in (1.9) e (1.10)

$$\max \quad \gamma_0 - \gamma_h \frac{\beta_k}{\alpha_{kh}} + \sum_{j=1, j\neq h}^{n-m} \gamma_j x_{m+j}$$

$$\begin{cases} x_1 = \beta_1 - \alpha_{1h}\frac{\beta}{\alpha_{kh}} + \frac{\alpha_{1h}}{\alpha_{kh}} x_k + \sum_{j=1,j\neq h}^{n-m} \left(\alpha_{1j} - \alpha_{1h}\frac{\alpha_{kj}}{\alpha_{kh}}\right) \\ \dots \\ x_{m+h} = -\frac{\beta_k}{\alpha_{kh}} + \frac{1}{\alpha_{kh}} x_k - \sum_{j=1,j\neq h}^{n-m} \frac{\alpha_{kj}}{\alpha_{kh}} x_{m+j} \\ \dots \\ x_m = \beta_m - \alpha_{mh}\frac{\beta}{\alpha_{kh}} + \frac{\alpha_{mh}}{\alpha_{kh}} x_k + \sum_{j=1,j\neq h}^{n-m} \left(\alpha_{mj} - \alpha_{mh}\frac{\alpha_{kj}}{\alpha_{kh}}\right) \end{cases}$$

da qui in poi si ripeteranno in sequenza i passi 1), 2) e 3) sopra descritti.
Riferendoci ancora al nostro esempio, otteniamo, riformulando il problema in

funzione della nuova base $B = \{x_1, s_1, s_3, s_4\}$

$$\max \quad 2 - s_2 + 2x_2 + 2x_3 \tag{1.13}$$

$$\begin{cases} s_1 = 2x_2 - x_3 + s_2 \\ x_1 = 2 - s_2 \\ s_3 = 3 - x_3 \\ s_4 = 6 - 3x_2 - x_3 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 \geq 0 \end{cases} \tag{1.14}$$

L'algoritmo del simplesso primale riuscirà a stabilire, dopo una serie di iterazioni, che la base $B_{\text{opt}} = \{x_2, x_3, s_2, s_3\}$ è non solo ammissibile, ma anche ottimale. La riformulazione del nostro esempio rispetto a quest'ultima base è

$$\max \quad 8 - x_1 - 2s_1 \tag{1.15}$$

$$\begin{cases} x_2 = 1 + \frac{1}{2}x_1 + \frac{1}{2}s_1 - \frac{1}{2}s_4 \\ x_3 = 3 - \frac{3}{2}x_1 - \frac{3}{2}s_1 + \frac{1}{2}s_4 \\ s_3 = 0 + \frac{3}{2}x_1 + \frac{3}{2}s_1 - \frac{1}{2}s_4 \\ s_2 = 2 - x_1 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4 \geq 0 \end{cases} \tag{1.16}$$

B_{opt} è ottimale perché la (1.15) soddisfa il criterio di ottimalità, quindi possiamo concludere che il punto $\boldsymbol{x} = \langle 0, 1, 3 \rangle \in \mathbb{R}^3$ è soluzione ottima.

1.2.4 Trovare una base ammissibile per l'algoritmo

Abbiamo tralasciato, fino ad adesso, un importante dettaglio: come troviamo una base ammissibile per far partire l'algoritmo del simplesso? Sfrutteremo la cosiddetta *prima fase* dell'algoritmo. La *prima fase* riuscirà a stabilire se il poliedro che si vuole ottimizzare è vuoto: se non lo è, ci fornirà un vertice ammissibile per il problema di programmazione lineare originale. In questo metodo andiamo

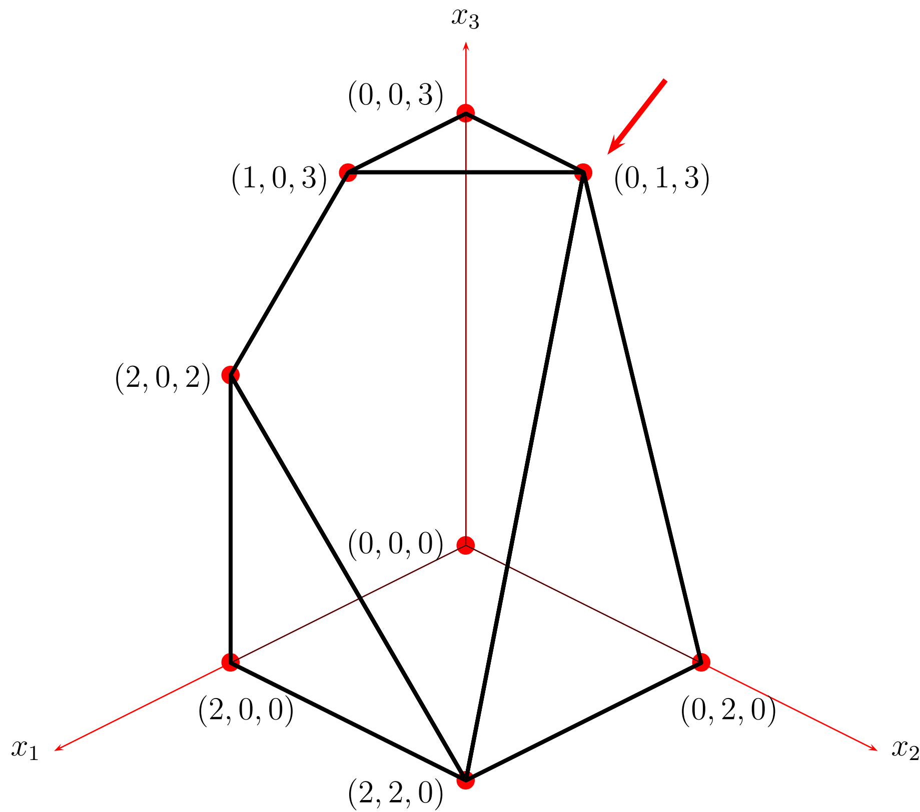


Figura 1.2: Il poliedro definito dalle (1.4) e la relativa soluzione al problema di programmazione lineare, indicata dalla freccia

ad aggiungere delle variabili *artificiali* che chiameremo x_i^a, una per ogni vincolo, in modo che ciascuna variabile artificiale abbia lo stesso segno del termine noto. Avremo quindi una base ammissibile di partenza in cui $x_i^a = b_i$. Nella prima fase andremo a minimizzare la seguente funzione obiettivo

$$\xi(\boldsymbol{x}) = \sum_{i=1}^{m} x_i^a.$$

In seguito alla risoluzione abbiamo tre possibili casi:

1. $\xi(\boldsymbol{x}) = 0$ e tutte le x_i^a sono fuori base. In questo caso abbiamo una base ammissibile per il problema di programmazione lineare originale.

2. $\xi(\boldsymbol{x}) > 0$ dopo la risoluzione. In questo caso il poliedro è vuoto: il problema è non ammissibile.

3. $\xi(\boldsymbol{x}) = 0$ e c'è qualche x_i^a in base dopo la risoluzione. Dovrà accadere che per qualche x_i^a,

$$x_i^a = 0.$$

In questo caso basterà esprimere il vincolo mediante una variabile non artificiale. Non si avrà nessuna perdita di ammissibilità proprio perché il valore del termine noto è zero. Se non dovesse esistere una variabile non artificiale in grado di esprimere il vincolo, si conclude che il vincolo è ridondante e può essere eliminato.

Ottenuta una base ammissibile per il problema originale di programmazione lineare, si può lanciare la *seconda fase* che tenterà di ottimizzare il problema. Possiamo quindi rappresentare, mediante il grafo nella figura 1.3, quali sono gli stati dell'algoritmo del simplesso primale. Utilizzeremo la seguente notazione

- `INIT` per lo stato iniziale.
- `UNS` per lo stato di non ammissibilità.
- `SAT` per lo stato di ammissibilità.
- `OPT` per lo stato di ottimalità.
- `UNB` per lo stato di illimitatezza.

1.2.5 Algoritmo *steepest-edge* per la variabile entrante

L'algoritmo del simplesso appena descritto utilizza come criterio di selezione della variabile entrante, il criterio *textbook*. Questo però tende a raggiungere la soluzione ottima, specialmente in problemi di larga scala, in un numero elevato di iterazioni. Proprio per questo John Reid e Donald Goldfarb [GR77] proposero,

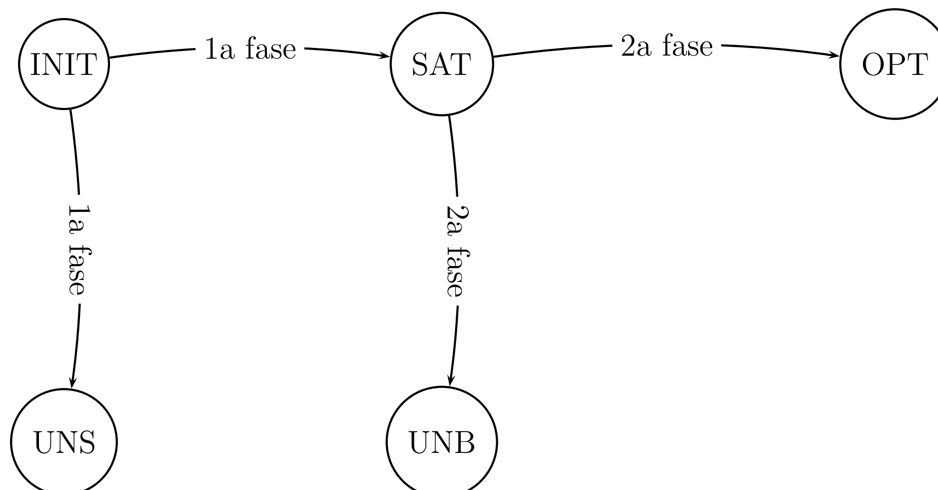


Figura 1.3: Gli stati dell'algoritmo del simplesso primale

a metà degli anni '70, il criterio *steepest-edge*. Questo criterio prevede che la variabile entrante in base sia x_k tale che

$$\gamma_k > 0 \quad \text{e} \quad \forall j \in N : \gamma_j > 0 \quad \Longrightarrow \quad \frac{\gamma_k}{\sqrt{1 + \sum_{i=1}^{m} \alpha_{ik}^2}} \geq \frac{\gamma_j}{\sqrt{1 + \sum_{i=1}^{m} \alpha_{ij}^2}}.$$

Anche qui, se non c'è alcuna variabile candidata, abbiamo raggiunto l'ottimo. L'efficienza è data dal fatto che si evitano delle scelte di vertici che possono formare dei movimenti a forma di spirale lungo il poliedro. Si procede sempre direttamente verso l'ottimo, sempre che quest'ultimo esista.

1.3 Aritmetica esatta ed Incrementalità

1.3.1 Aritmetica esatta

Una delle caratteristiche chiave del presente lavoro di tesi, è che l'algoritmo del simplesso primale implementato nella libreria PPL è basato su *aritmetica esat-*

ta. Come mai questa scelta? Esistono numerose implementazioni dell'algoritmo del simplesso primale con licenza *GNU General Public License* tra cui la più importante è GLPK [Mak01, GLP06], ma poche possono garantire un risultato corretto, non soggetto ad errori di arrotondamento dovuti alla rappresentazione macchina dei numeri in virgola mobile. In alcuni campi, come quello dell'analisi del software, ciò non è tollerabile: si può correre il rischio di ottenere delle soluzioni che sono tutt'altro che ottimali, non appartenenti ad esempio alla regione ammissibile del problema. Usando i coefficienti interi a precisione arbitraria forniti dalla libreria GMP [GNU04], la PPL è in grado di risolvere qualsiasi tipo di problema di programmazione lineare che abbia vincoli e funzione obiettivo espressi da coefficienti interi. Da notare che la soluzione finale, sia per quanto riguarda il punto ottimale che per il valore della funzione obiettivo, è di tipo razionale. Un'implementazione analoga a quella fornita dalla PPL è quella della libreria Wallaroo [Wal05], che utilizza coefficienti razionali a precisione arbitraria della libreria GMP. L'aritmetica esatta però ha un costo e, a volte, può essere molto elevato: sono infatti coinvolti un maggior numero di cicli di clock per gestire anche una singola addizione. Questo implica che i tempi di calcolo, specialmente in problemi molto complessi, possono risultare significativamente più lunghi rispetto alle implementazioni che si basano sull'aritmetica floating point: la scelta di usare meno l'aritmetica esatta o meno dipende dal campo d'applicazione.

Vediamo, con un esempio, come è stato possibile rappresentare nella PPL un vincolo a coefficienti razionali per mezzo di interi. Prendiamo dal sistema di vincoli (1.16) l'equazione

$$x_3 = 3 - \frac{3}{2}x_1 - \frac{3}{2}s_1 + \frac{1}{2}s_4.$$

Questo vincolo è rappresentato internamente da una riga appartenente ad una matrice, detta anche *tableau*, che definisce l'intero sistema di vincoli. L'equazione

sopra citata viene trasformata in

$$3x_1 + 2x_3 + 3s_1 - s_4 = 6.$$

Questa rappresentazione è ottenuta "normalizzando" i coefficienti dell'equazione, dove per normalizzazione si intende che tutti i coefficienti sono coprimi. Tutte le operazioni aritmetiche quindi sono state adattate tenendo conto di questa rappresentazione interna dei vincoli.

1.3.2 Incrementalità

Il lavoro di tesi, oltre ad essere stato incentrato su un miglioramento generale dell'efficienza dell'algoritmo del simplesso primale grazie all'implementazione della tecnica *steepest-edge*, ha avuto come intento quello di realizzare, possibilmente in modo efficiente, un'implementazione incrementale dell'algoritmo. Come mai questa scelta? Immaginiamo di voler testare la soddisfacibilità di un poliedro aggiungendo, volta per volta, nuovi vincoli. Questo può essere realizzato usando la *prima fase* dell'algoritmo del simplesso primale. Un'implementazione non incrementale può arrivare ad avere un peggioramento dell'efficienza quadratico nel numero di vincoli del problema rispetto ad un'implementazione incrementale. Vediamo quali criteri sono stati utilizzati per implementare l'incrementalità.

Lo stato iniziale

Immaginiamo di avere risolto con successo un problema di programmazione lineare. In questo caso per "con successo" intendiamo il fatto che il poliedro non sia vuoto, altrimenti ha poco senso parlare di incrementalità. Supponiamo di essere nella seguente condizione

$$\max \quad \gamma_0 + \sum_{j=1}^{n-m} \gamma_j x_{i_{m+j}}$$

$$\begin{cases} x_1 = \beta_1 + \sum_{j=1}^{n-m} \alpha_{1j} x_{m+j} \\ \dots \\ x_k = \beta_k + \sum_{j=1}^{n-m} \alpha_{kj} x_{m+j} \\ \dots \\ x_m = \beta_m + \sum_{j=1}^{n-m} \alpha_{mj} x_{m+j} \end{cases}$$

Abbiamo quindi individuato una base

$$B = \{x_1, \dots, x_m\}$$

che ci garantisce che il poliedro non sia vuoto. Inoltre sappiamo che questa base individua un vertice ben preciso del poliedro. Chiamiamo, per nostra comodità, questo vertice g. Supponiamo ancora di dover aggiungere al nostro problema di programmazione lineare dei nuovi vincoli della forma

$$\begin{cases} \sum_{j=0}^{n} a_{ij} x_j \leq b_i & i \in I_1 \\ \sum_{j=0}^{n} a_{ij} x_j \geq b_i & i \in I_2 \\ \sum_{j=0}^{n} a_{ij} x_j = b_i & i \in I_3 \end{cases} \tag{1.17}$$

Come tratteremo questi vincoli?

Analisi dei nuovi vincoli

La prima cosa da chiederci in una risoluzione incrementale del problema è: questi nuovi vincoli, che chiameremo pendenti, sono già soddisfatti dal vertice g? È particolarmente vantaggioso controllare la soddisfacibilità di g, in quanto si può evitare l'inserimento di ulteriori variabili artificiali. Basta infatti sostituire nelle incognite x_j il valore delle coordinate di g in modo da ottenere, per le disequazioni

$$\begin{cases} k_i \leq b_i & i \in I_1 \\ k_i \geq b_i & i \in I_2 \end{cases}$$

Queste sono o banalmente vere, oppure banalmente false. Nel caso in cui la disequazione sia soddisfatta, la variabile di slack, inserita come al punto 2 della

sezione 1.1.2, entrerà in base evitando l'inserimento di una variabile artificiale. Andremo quindi ad eseguire tutte le operazioni definite in 1.1.2 in maniera da trasformare tutti i vincoli in *forma standard*.

Inserimento dei vincoli pendenti nel sistema

Arrivati a questo punto, manca un solo passo per poter inserire i nuovi vincoli nel sistema ed avere uno stato consistente: dobbiamo esprimere tutti i vincoli pendenti in funzione della base B. Fatto questo, possiamo modificare B inserendo le nuove variabili in base, quelle che esprimono i vincoli pendenti. Supponendo di avere adesso p variabili e q vincoli, otteniamo una base del tipo

$$B' = \{x_1, \ldots, x_m, x_{m+1}, \ldots, x_q\}$$

ed il problema sarà riformulato da

$$\max \quad \gamma_0 + \sum_{j=1}^{p-q} \gamma_j x_{q+j}$$

$$\begin{cases} x_1 = \beta_1 + \sum_{j=1}^{p-q} \alpha_{1j} x_{q+j} \\ \ldots \\ x_m = \beta_m + \sum_{j=1}^{p-q} \alpha_{mj} x_{q+j} \\ \ldots \\ x_q = \beta_q + \sum_{j=1}^{p-q} \alpha_{qj} x_{q+j} \end{cases}$$

A questo punto siamo pronti a lanciare la prima fase ed un'eventuale seconda fase dell'algoritmo del simplesso primale. Da notare che, se tutte le equazioni fossero soddisfatte, il simplesso primale terminerebbe al primo ciclo in quanto il numero di variabili artificiali inserite sarebbe nullo.

Prendiamo ancora l'esempio definito dalla funzione obiettivo (1.3) e dai vincoli (1.4): la risoluzione del problema, svolta in precedenza, ci ha portato alla (1.15)

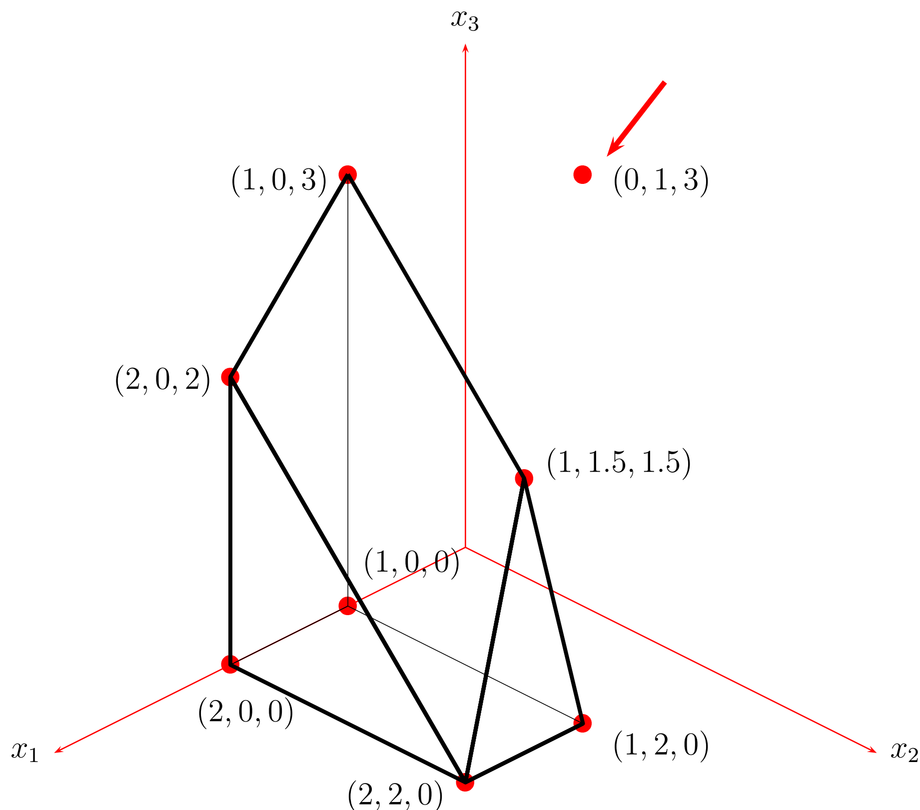


Figura 1.4: Il vertice ottimale calcolato in precedenza non è più ammissibile

ed alle (1.16). Vogliamo aggiungere adesso il vincolo

$$x_1 \geq 1.$$

L'aggiunta di questo vincolo implica che il vertice $\boldsymbol{x} = \langle 0, 1, 3\rangle \in \mathbb{R}^3$ non è più ammissibile. Per quanto detto sopra, il sistema di vincoli viene modificato aggiungendo una nuova equazione, cosicché diventa, tenendo conto dell'aggiunta di una nuova variabile di slack

$$\begin{cases} x_2 = 1 + \frac{1}{2}x_1 + \frac{1}{2}s_1 - \frac{1}{2}s_4 \\ x_3 = 3 - \frac{3}{2}x_1 - \frac{3}{2}s_1 + \frac{1}{2}s_4 \\ s_3 = 0 + \frac{3}{2}x_1 + \frac{3}{2}s_1 - \frac{1}{2}s_4 \\ s_2 = 2 - x_1 \\ s_5 = -1 + x_1 \\ x_1, x_2, x_3, s_1, s_2, s_3, s_4, s_5 \geq 0 \end{cases} \tag{1.18}$$

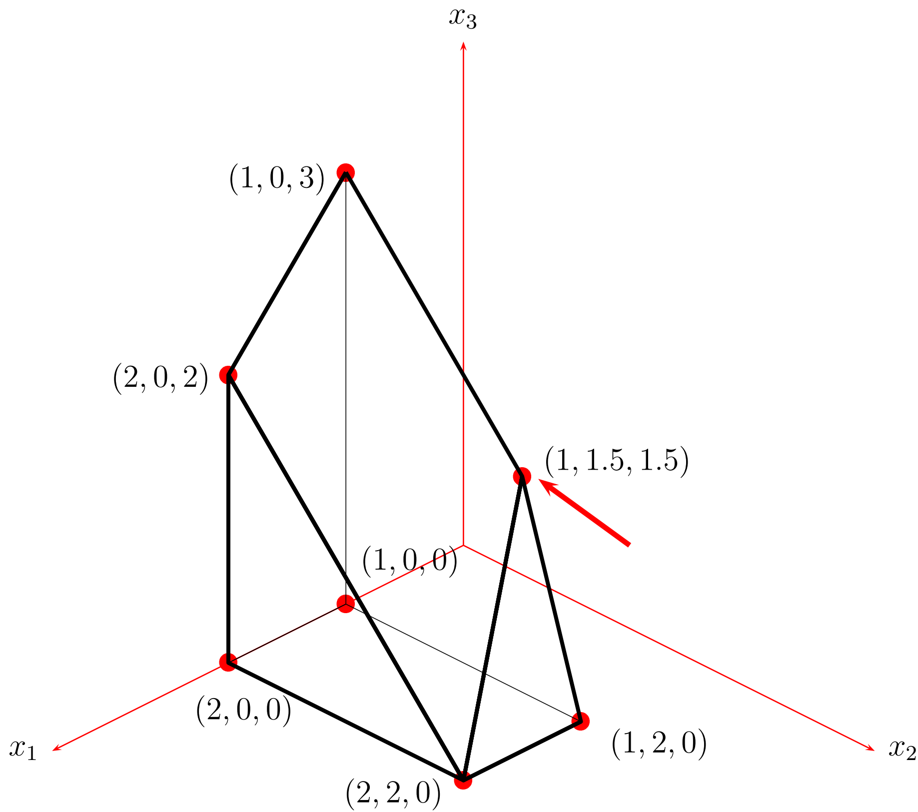


Figura 1.5: Il nuovo vertice ottimale evidenziato dalla freccia

La base associata $B' = \{x_2, x_3, s_4, s_3, s_1\}$ non è ammissibile: è necessario lanciare la prima fase dell'algoritmo del simplesso primale per trovare un vertice ammissibile. Questa terminerà individuando la base $B' = \{x_1, x_2, x_3, s_4, s_3, s_2\}$ alla quale è associato il vertice $\boldsymbol{x} = \langle 1, \frac{3}{2}, \frac{3}{2} \rangle \in \mathbb{R}^3$. La seconda fase dimostrerà anche che questo vertice è ottimale ed il nuovo valore della funzione obiettivo calcolato in $\boldsymbol{x}$ sarà 7. Come era ragionevole aspettarsi, il valore della funzione obiettivo è diminuito in quanto la regione ammissibile, in seguito all'introduzione del nuovo vincolo, si è ristretta.

Possiamo adesso completare il grafo della figura 1.3 aggiungendo un nuovo stato `PSAT` e rimuovendo `INIT`.

- `PSAT` indica la soddisfacibilità parziale, stato che si ottiene dopo l'aggiunta di un vincolo che compromette la soddisfacibilità del vertice calcolato in seguito al lancio della prima o della seconda fase. Anche lo stato inizia-

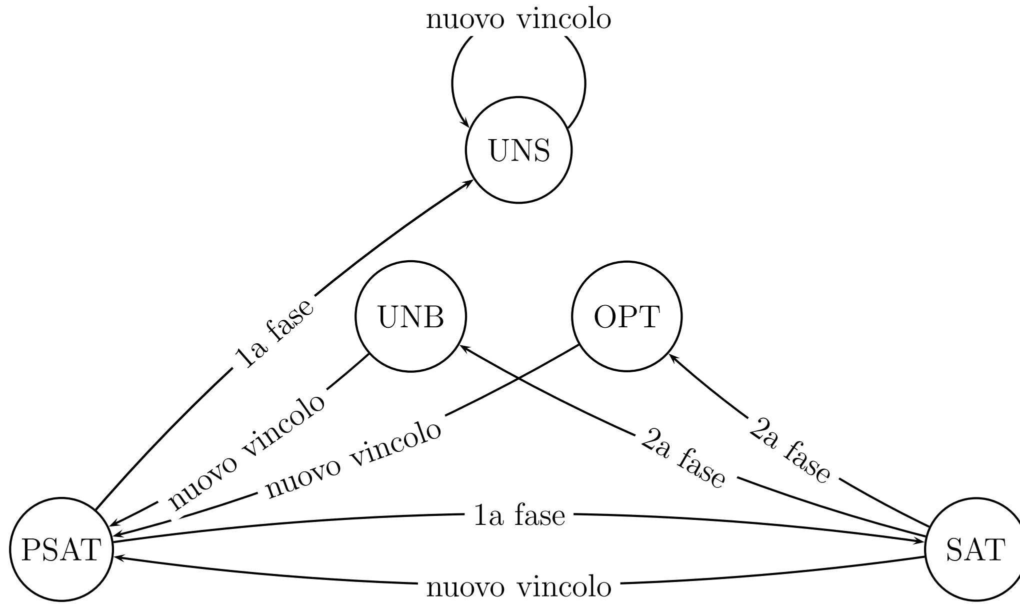


Figura 1.6: Gli stati dell'algoritmo del simplesso primale nella versione incrementale

le dell'algoritmo (`INIT`) è un caso particolare di soddisfacibilità parziale, proprio per questo diventa ridondante.

Il nuovo grafo è nella figura 1.6.

Capitolo 2

Implementazione nella PPL

2.1 Interfaccia Utente

In questa sezione verrà descritto l'utilizzo dei metodi pubblici forniti dalla classe `LP_Problem` per comprenderne il suo utilizzo lato utente. Verranno inoltre descritte le enumerazioni `LP_Problem_Status` e `Optimization_Mode`.
`LP_Problem_Status` descrive lo stato in cui un oggetto `LP_Problem` si trova dopo la sua risoluzione e può assumere uno dei seguenti tre valori:

- `UNFEASIBLE_LP_PROBLEM` indica che la regione di spazio, descritta dai vincoli del problema, è vuota.
- `UNBOUNDED_LP_PROBLEM` indica che il problema è illimitato: non esiste quindi un punto ottimale.
- `OPTIMIZED_LP_PROBLEM` indica che il problema ammette (almeno) un punto ottimale.

`Optmization_Mode` descrive invece una modalità di ottimizzazione e può assumere uno dei seguenti due valori:

- `MINIMIZATION` nel caso della minimizzazione.

- `MAXIMIZATION` nel caso della massimizzazione.

Di seguito sono elencati e descritti i metodi pubblici offerti.

Costruttori, distruttore, copia, assegnamento, scambio

- `LP_Problem()`

  Costruisce un oggetto di tipo `LP_Problem` che descrive un problema di programmazione lineare banale. Il sistema di vincoli interno è vuoto, la funzione obiettivo da ottimizzare è `Linear_Expression::zero()` e viene impostata la massimizzazione di default.

- `LP_Problem(const Constraint_System &cs,`
  `const Linear_Expression &obj=Linear_Expression::zero(),`
  `Optimization_Mode mode=MAXIMIZATION)`

  Costruisce un oggetto di tipo `LP_Problem` con sistema di vincoli definito da `cs`, funzione obiettivo `obj` e ottimizzazione impostata a `MODE`. Nel caso in cui `obj` o `mode` non siano passati al costruttore, questi assumeranno i valori di default definiti nel prototipo, ovvero `Linear_Expression::zero()` e `MAXIMIZATION`.

- `LP_Problem(const LP_Problem &y)`

  Costruttore di copia. Chiamato per ottenere delle copie di un oggetto `LP_Problem`.

- `~LP_Problem()`

  Distruttore. Garantisce la corretta deallocazione delle risorse di un oggetto `LP_Problem`.

- `LP_Problem& operator=(const LP_Problem &y)`

  Operatore di assegnamento. Assegna una copia del problema di programmazione lineare a `y`.

- `void swap (LP_Problem &y) const`

  Scambia in modo efficiente il problema di programmazione lineare con `y`.

Metodi di interrogazione

Sono metodi che non modificano l'oggetto, istanza di `LP_Problem`, e permettono di avere informazioni sul suo stato, proprio per questo dichiarati costanti.

- `dimension_type space_dimension() const`

  Ritorna la dimensione dello spazio vettoriale sul quale è definito il problema di programmazione lineare.

- `const Constraint_System& constraints() const`

  Ritorna un riferimento costante al sistema di vincoli del problema di programmazione lineare.

- `const Linear_Expression& objective_function() const`

  Ritorna un riferimento costante alla funzione obiettivo del problema di programmazione lineare.

- `Optimization_Mode optimization_mode() const`

  Ritorna un `Optimization_Mode` che indica se il problema di programmazione lineare è impostato sulla massimizzazione o sulla minimizzazione.

- `void`
  `evaluate_objective_function(const Generator &evaluating_point,`

`Coefficient &num,`

`Coefficient &den) const`

Imposta `num` e `den` in modo che $\frac{num}{den}$ sia il risultato della valutazione di `evaluating_point` nella funzione obiettivo.

Metodi di modifica

I seguenti metodi permettono di modificare l'oggetto `LP_Problem`, permettendo l'aggiunta di vincoli, la modifica della funzione obiettivo, e l'impostazione della modalità di ottimizzazione.

- `void clear()`

  Resetta il problema di programmazione lineare assegnandogli un oggetto che definisce un problema banale.

- `void add_constraint(const Constraint &c)`

  Aggiunge il vincolo `c` al problema di programmazione lineare ridimensionando le strutture dati di quest'ultimo, se necessario.

- `void add_constraints(const Constraint_System &cs)`

  Aggiunge il sistema di vincoli `cs` al problema di programmazione lineare ridimensionando le strutture dati di quest'ultimo, se necessario.

- `void set_objective_function(const Linear_Expression &obj)`

  Imposta la funzione obiettivo del problema di programmazione lineare a `obj`.

- `void set_optimization_mode(Optimization_Mode mode)`

  Imposta la modalità di ottimizzazione del problema di programmazione lineare a `mode`.

Metodi di risoluzione

Sono metodi che dal punto di vista logico non modificano il problema, per questo dichiarati costanti, ma internamente comportano una modifica dell'oggetto.

- `bool is_satisfiable() const`

  Ritorna `false` se la regione ammissibile del problema di programmazione lineare è vuota, `true` altrimenti.

- `LP_Problem_Status solve() const`

  Tenta di ottimizzare il problema di programmazione lineare usando l'algoritmo del simplesso primale. Ritorna un `LP_Problem_Status` che indica uno dei tre possibili valori d'uscita sopra descritti.

- `const Generator& feasible_point() const`

  Ritorna un punto ammissibile del problema di programmazione lineare, se esiste. In caso contrario viene lanciata un'eccezione di tipo `std::domain_error`.

- `const Generator& optimizing_point() const`

  Ritorna un punto ottimale del problema di programmazione lineare, se esiste. In caso contrario viene lanciata un'eccezione di tipo `std::domain_error`.

Metodi di servizio

Metodi ad uso interno.

- `void ascii_dump(std::ostream &s)`

  Scrive su `s` una rappresentazione ASCII del problema di programmazione lineare. Utile per il debugging.

2.1.1 Esempio d'uso della classe LP_Problem

Vediamo come è facile, grazie anche alla gradevole sintassi fornita dalla PPL, costruire un problema di programmazione lineare. Prendiamo come esempio il problema definito da (1.3) e dalle (1.4).

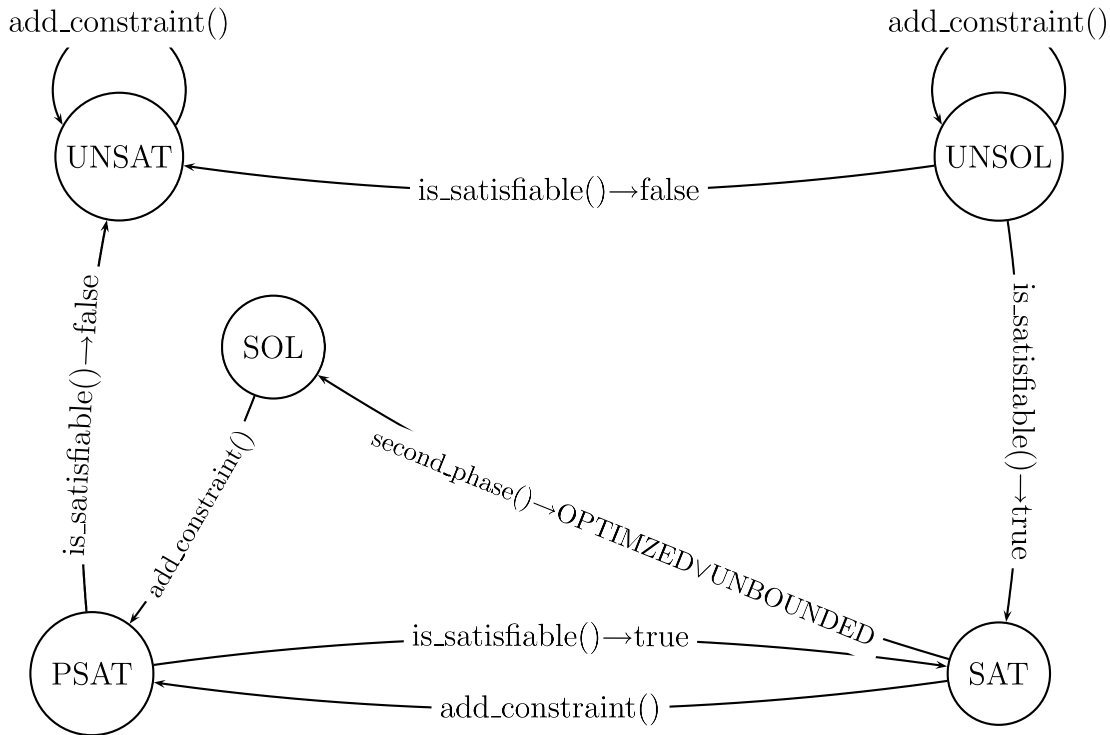
```
#include <ppl.hh>
using namespace Parma_Polyhedra_Library;
int
main() {
  // Definizione delle variabili.
  Variable X1(0);
  Variable X2(1);
  Variable X3(2);

  // Definizione della funzione di costo.
  Linear_Expression cost(X1 + 2*X2 + 2*X3);

  // Definizione dei vincoli.
  Constraint_System cs;
  cs.insert(X1 + X2 + X3 <= 4);
  cs.insert(X1 <= 2);
  cs.insert(X3 <= 3);
  cs.insert(3*X2 + X3 <= 6);
  cs.insert(X1 >= 0);
  cs.insert(X2 >= 0);
  cs.insert(X3 >= 0);

  // Costruzione del problema.
  LP_Problem lp = LP_Problem(cs, cost, MAXIMIZATION);

  // Risoluzione del problema.
  LP_Problem_Status lp_status = lp.solve();

  // Verifica dello stato del problema.
  switch (lp_status) {
    case UNFEASIBLE_LP_PROBLEM:
      std::cout << "Regione ammissibile vuota";
      break;
    case UNBOUNDED_LP_PROBLEM:
      std::cout << "Problema illimitato";
      break;
```

```
    case OPTIMIZED_LP_PROBLEM:
    {
      Coefficient num;
      Coefficient den;

      // Calcolo di un punto ottimale.
      Generator opt_point = lp.optimizing_point();

      // Valutazione della funzione obiettivo.
      lp.evaluate_objective_function(opt_point, num, den);
      std::cout << "Valore ottimo " << num << "/" << den;
      break;
    }
  }
  return 0;
}
```

2.2 Implementazione

Le linee guida scelte per la realizzazione delle strutture dati interne di un oggetto di tipo `LP_Problem` sono state quelle proposte da Christos H. Papadimitriou e Kenneth Steiglitz [PS98], poi opportunamente modificate per la gestione delle righe a coefficienti interi. Per rappresentare lo stato interno di un oggetto `LP_Problem` viene utilizzata l'enumerazione Status (da non confondere con l'altra enumerazione `LP_Problem_Status`, utilizzata invece per comunicare all'esterno l'esito di un tentativo di ottimizzazione). Per comodità di notazione, nel diagramma gli stati `OPTIMIZED` e `UNBOUNDED` sono stati aggregati in un unico stato, chiamato `SOLVED`, in quanto essi sono caratterizzati da archi entranti ed uscenti analoghi.

- `UNSOLVED` indica che non si ha alcuna informazione relativa allo stato del problema e che non sono state ancora allocate le strutture dati necessarie alla sua risoluzione.



- `UNSATISFIABLE` indica che la prima fase dell'algoritmo ha calcolato che la regione ammissibile del problema è vuota. In questo caso, a meno di non resettare l'oggetto, questo sarà lo stato finale.

- `SATISFIABLE` indica che la regione di spazio descritta dal sistema di vincoli del problema non è vuota e si ha un vertice valido per far partire la seconda fase dell'algoritmo del simplesso. Utile nel caso si voglia ottimizzare un problema cambiando ogni volta funzione obiettivo.

- `UNBOUNDED` indica che il problema ammette soluzione illimitata; non esiste quindi un punto ottimale.

- `OPTIMIZED` indica che è stato trovato un punto ottimale.

- `PARTIALLY_SATISFIABLE` indica il caso di un problema soddisfacibile al qua-

le siano stati in seguito aggiunti ulteriori vincoli con il metodo `add_constraints()`; tali vincoli, che devono ancora essere analizzati, sono detti essere vincoli "pendenti". La loro analisi comporterà l'esecuzione della parte dell'algoritmo del simplesso che si occupa dell'incrementalità del calcolo.

Andiamo adesso ad analizzare le strutture dati interne alla classe, usate per implementare `LP_Problem`.

- `Constraint_System pending_input_cs`

  Contiene tutti i vincoli pendenti, cioè quelli non ancora risolti dalla precedente chiamata di `is_satisfiable()` o `solve()`. La presenza di questi vincoli comporterà l'attivazione di quella parte dell'implementazione che si occupa dell'incrementalità del calcolo.

- `Constraint_System input_cs`

  Contiene tutti i vincoli del problema passati al problema di programmazione lineare durante la sua fase di inizializzazione. Inoltre tutti i vincoli pendenti, una volta risolto il problema, vengono aggiunti a questo `Constraint_System`.

- `Linear_Expression input_obj_function`

  Memorizza la funzione obiettivo corrente da ottimizzare.

- `Matrix tableau`

  È la matrice contenente tutti i vincoli del problema, tranne i vincoli di non negatività della forma $x_i \geq 0$, in quanto la forma standard di un problema di programmazione lineare intrinsecamente sottointende questa condizione. `tableau` viene manipolata durante l'esecuzione dell'algoritmo per provare l'ammissibilità del problema e trovare un eventuale punto ottimale.

- `Optimization_Mode working_cost`

  È la funzione obiettivo manipolata durante le iterazioni dell'algoritmo. Viene usata per verificare il criterio di ottimalità e, nel caso questo non sia soddisfatto, per scegliere una variabile entrante in base.

- `Optimization_Mode opt_mode`

  Permette di stabilire se la funzione obiettivo deve essere massimizzata o minimizzata.

- `Generator last_generator`

  L'ultimo punto ammissibile oppure ottimale calcolato durante la risoluzione del problema.

- `std::vector<std::pair<dimension_type, dimension_type> > mapping;`

  Permette di mettere in corrispondenza le variabili originali del problema con quelle utilizzate internamente per la soluzione del problema. In particolare alla variabile `i` del problema originale è associata la variabile `mapping[i].first`. Se la variabile originale del problema è libera in segno, l'indice contenente la variabile che esprime la parte negativa nella rappresentazione interna di `LP_Problem` è dato da `mapping[i].second`, altrimenti si ha che `mapping[i].second=0`.

- `std::vector<bool> is_artificial`

  `is_artificial[i]` è `true` se `i` è una variabile artificiale, `false` altrimenti. Queste variabili vengono introdotte per ricavare una base ammissibile per la prima fase dell'algoritmo del simplesso. Vengono poi eliminate prima dell'esecuzione della seconda fase.

- `std::vector<dimension_type> base`

È un vettore contenente tutte le variabili che sono in base. In particolare `base[i]` contiene l'indice della variabile tramite la quale è espressa la i-esima equazione del tableau.

- `Status status`

  Descrive lo stato interno del problema di programmazione lineare. Questo è fondamentale per poter decidere quale tecnica di risoluzione usare. Permette inoltre di evitare inutili calcoli nel caso in cui lo stato interno sia `UNSATISFIABLE`. In questo caso l'algoritmo non verrà più eseguito.

Qui invece sono elencati i metodi privati usati per implementare l'algoritmo del simplesso.

- `LP_Problem_Status compute_tableau()`

  Calcola la forma standard del problema di programmazione lineare assegnato. Vengono aggiunte la variabili di slack nel caso di disequazioni, in maniera da renderle equivalenti a delle equazioni. Inoltre ogni variabile libera in segno viene espressa tramite due variabili non negative. Gestisce inoltre le situazioni banali, come un tableau vuoto: in questo caso il problema è illimitato.

- ```
  bool parse_constraints(const Constraint_System& cs,
  dimension_type& new_num_rows, dimension_type& num_slack_variables,
  std::deque<bool>& is_tableau_constraint,
  std::deque<bool>& nonnegative_variable,
  std::vector<dimension_type>& unfeasible_tableau_rows,
  std::deque<bool>& satisfied_ineqs,
  const bool bootstrap)
  ```
  ```

Analizza i vincoli contenuti in `cs`, in modo da informare la funzione chiamante su come costruire il corrispondente tableau. Gli unici parametri in input sono il primo (cs) e l'ultimo (bootstrap); tutti gli altri sono parametri in output e quindi passati per riferimento non costante.

- `new_num_rows`: il numero di righe da aggiungere.
- `new_slack_variables`: il numero di variabili di slack da aggiungere.
- `is_tableau_constraint[i]`: se `true` , allora l'i-esimo vincolo di `cs`, sarà aggiunto al tableau, `false` altrimenti.
- `nonnegative_variable[i]`: se `true`, la i-esima variabile di `cs` è non negativa, non sarà quindi espressa da due nuove variabili, `false` altrimenti.
- `unfeasible_tableau_rows`: il numero di variabili di slack.
- `satisfied_ineqs[i]`: `true` se l'i-esima riga di `cs` è già soddisfatta da `last_generator`, `false` altrimenti. Questo permette di risparmiare l'inserimento di inutili variabili artificiali, per accelerare quindi le performance dell'algoritmo.
- `bootstrap`: `true` se si ha a disposizione un vertice valido, `false` altrimenti.

```
dimension_type textbook_entering_index() const
```

Controlla che si sia raggiunta la condizione di ottimalità. Se questa sussiste, il valore di ritorno è `0`, altrimenti ritorna l'indice della variabile che dovrà entrare in base al passo successivo dell'algoritmo. La scelta viene fatta usando la regola *textbook*. Inoltre, per evitare un possibile ciclo, è stata implementata la *regola di Bland* per le variabili entranti in base.

- `dimension_type steepest_edge_entering_index() const;`

  Svolge la stessa funzione di `textbook_entering_index()`, con la differenza che la tecnica utilizzata è quella dello *steepest-edge*.

- `dimension_type`

  `get_exiting_base_index(dimension_type entering_var_index) const`

  Ritorna l'indice di riga che è espressa dalla variabile che dovrà uscire dalla base. Anche in questo caso, per evitare il ciclo, è stata implementata la *regola di Bland* per le variabili uscenti dalla base.

- `static void`

  `linear_combine(Row& x, const Row& y, const dimension_type k)`

  Esegue la combinazione lineare fra le righe `x` e `y` in maniera che, all'uscita della procedura, si abbia `x[k]=0`. Una volta eseguita la combinazione lineare, tutti i coefficienti di `x` vengono normalizzati.

- ```
  void pivot(const dimension_type entering_var_index,
             const dimension_type exiting_base_index);
  ```

  Esegue l'operazione di pivoting.

- `bool compute_simplex()`

  Una volta calcolato il tableau, questo metodo ritorna `true` se è stato calcolato un vertice valido dall'algoritmo del simplesso, `false` altrimenti.

- `void prepare_first_phase()`

  Aggiunge le variabili artificiali al tableau per ottenere una base ammissibile per la prima fase ed inserisce il segno nella funzione di costo.

- `void erase_artificials()`
  ```

Elimina le variabili artificiali inserite nella prima fase ed i possibili vincoli ridondanti, preparando il tableau alla seconda fase.

- `bool is_in_base(const dimension_type var_index,`
  `dimension_type& row_index) const;`

  Permette di stabilire se la variabile `var_index` è nella base attuale e, se ciò sussiste, in `row_index` memorizza l'indice della del tableau espressa da tale variabile.

- `bool process_pending_constraints()`

  Elabora i vincoli pendenti durante la fase incrementale dell'algoritmo. Vengono inserite le necessarie variabili artificiali e di slack.

- `void second_phase()`

  Lancia la seconda fase dell'algoritmo del simplesso, preparando inizialmente le strutture dati interne.

- `void unsplit(dimension_type var_index,`
  `std::vector<dimension_type>& nonfeasible_cs)`

  Ricongiunge due variabili (create al momento della inizializzazione del tableau in seguito alla verifica della non negatività di una variabile) nel caso in cui venga riconosciuta una condizione esplicita per cui una variabile del problema originale è sicuramente non negativa. In `nonfeasible_cs` verranno memorizzati gli indici dei vincoli non più soddisfatti dall'ultimo vertice calcolato in seguito all'aggiunta di un vincolo di non negatività.

Implementazione dell'algoritmo *steepest-edge*

Come visto nella sezione 1.2.5, l'algoritmo *steepest-edge* richiede, ad ogni passo del simplesso primale, il calcolo delle quantità

$$\forall j \in N : \gamma_j > 0 \quad \Longrightarrow \quad \frac{\gamma_j}{\sqrt{1 + \sum_{i=1}^{m} \alpha_{ij}^2}}.$$

Il denominatore è sotto radice: questo è un problema se si vuole implementare l'algoritmo *steepest-edge* usando l'aritmetica esatta. Questo può essere aggirato elevando al quadrato numeratore e denominatore, ma tale operazione può risultare particolarmente inefficiente sia per la memoria utilizzata che per i cicli di clock impiegati. Per questo si è deciso di fornire un'implementazione basata sull'aritmetica *floating point*, oltre a quella a coefficienti interi. In questo modo si evita di dover elevare al quadrato numeratore e denominatore. Rimane da garantire la correttezza dell'algoritmo nel caso in cui si utilizzi l'aritmetica floating point, ma ciò non rappresenta un problema in quanto, anche nel caso in cui si dovesse scegliere una variabile entrante non corretta, non si diminuisce il valore della funzione obiettivo calcolata nel nuovo vertice. Le valutazioni sperimentali svolte hanno dimostrato, specialmente nel caso incrementale, che un'implementazione dell'algoritmo *steepest-edge* basata sull'aritmetica floating point può risultare determinante per le performance.

Capitolo 3

Valutazioni sperimentali

Le seguenti valutazioni sperimentali sono state eseguite in modo da testare sia la performance del metodo di risoluzione non incrementale, nel quale tutto l'input del problema viene fornito prima dell'inizio della risoluzione, sia la performance del metodo incrementale. In questo secondo caso è stato considerato un approccio per certi versi "estremo" nei confronti dell'incrementalità: il metodo di risoluzione incrementale viene invocato dopo ogni singola aggiunta di un vincolo per verificare se il problema è soddisfacibile. Si noti che, avendo a che fare con problemi soddisfacibili, questo modo di procedere comporta un notevole peggioramento del tempo totale di esecuzione rispetto all'approccio non incrementale. I problemi risolti sono descritti da file in formato MPS (Mathematical Programming System): questo formato è stato creato in seguito all'introduzione di un prodotto della *IBM* in grado di risolvere problemi di programmazione lineare, diventando in breve tempo uno standard per tutti i prodotti commerciali. Un esempio di file MPS è il seguente

```
NAME          TESTPROB
ROWS
 N  COST
 L  LIM1
 G  LIM2
```

```
 E  MYEQN
COLUMNS
    XONE      COST                 1   LIM1                 1
    XONE      LIM2                 1
    YTWO      COST                 4   LIM1                 1
    YTWO      MYEQN               -1
    ZTHREE    COST                 9   LIM2                 1
    ZTHREE    MYEQN                1
RHS
    RHS1      LIM1                 5   LIM2                10
    RHS1      MYEQN                7
BOUNDS
 UP BND1      XONE                 4
 LO BND1      YTWO                -1
 UP BND1      YTWO                 1
ENDATA
```

Utilizzando un modello basato su equazioni, il problema appena descritto è equivalente a

```
Funzione obiettivo:
 COST:    XONE + 4 YTWO + 9 ZTHREE
Vincoli:
 LIM1:    XONE + YTWO < = 5
 LIM2:    XONE + ZTHREE > = 10
 MYEQN:   - YTWO + ZTHREE  = 7
Limitazioni:
 0 < = XONE < = 4
-1 < = YTWO < = 1
```

I test sono stati eseguiti su una macchina x86 con processore AMD Sempron™ 2400+, 512 Mb di RAM. Il sistema operativo installato sulla macchina è Gentoo Linux, con kernel 2.6.16. La versione della libreria GMP usata è la 4.2.

3.1 Test non incrementali

Problema	GLPK	steepest-GMP	steepest-float	textbook
adlittle.mps	0.00	0.10	0.12	0.76
afiro.mps	0.00	0.01	0.00	0.00
bgprtr.mps	0.00	0.01	0.00	0.00
blend.mps	0.00	2.02	1.48	15.84
boeing1.mps	0.18	23.03	24.37	69.21
boeing2.mps	0.02	1.95	2.32	3.13
ex1.mps	0.00	0.00	0.00	0.00
kb2.mps	0.00	0.28	0.26	0.51
mip.mps	0.00	0.00	0.00	0.00
sample.mps	0.00	0.00	0.00	0.00
sc105.mps	0.00	0.08	0.09	0.05
sc50a.mps	0.00	0.01	0.02	0.00
sc50b.mps	0.00	0.03	0.02	0.02
unboundedmin.mps	0.00	0.00	0.00	0.00

La seconda colonna rappresenta i tempi di risoluzione di *GLPK*; la terza, la quarta e la quinta colonna rappresentano i tempi di risoluzione dei problemi da parte della PPL: *steepest-GMP* indica che è stata usata l'implementazione dell'algoritmo *steepest-edge* a coefficienti interi, *steepest-float* indica i tempi con una implementazione dello *steepest-edge* che fa uso dell'aritmetica floating point. *Textbook* indica i tempi usando tale regola per la variabile entrante. Le differenze tra *GLPK* e la PPL sono essenzialmente dovute al tipo di coefficienti usati per il calcolo: in particolare *GLPK* usa l'aritmetica *floating point*, con conseguenti approssimazioni, la PPL invece usa *GMP* (interi a precisione arbitraria) senza alcuna approssimazione.

3.2 Test incrementali

Problema	steepest-GMP	steepest-float	Wallaroo	Cassowary-GMP
adlittle.mps	0.54	0.52	0.52	1.55
afiro.mps	0.03	0.02	0.10	0.11
blend.mps	17.20	19.22	30.53	6.24
boeing1.mps	395.64	70.80	257.28	115.82
boeing2.mps	19.16	4.25	15.43	22.36
kb2.mps	0.44	0.17	0.36	0.58
sc105.mps	0.80	0.78	5.01	18.60
sc50a.mps	0.09	0.09	0.80	0.77
sc50b.mps	0.11	0.10	0.78	0.84

In questo caso è stata testata la soddisfacibilità dei poliedri definiti dai problemi in formato MPS, aggiungendo i vincoli uno alla volta. La seconda e la terza colonna rappresentano i tempi della PPL: *steepest-GMP* indica che è stata usata l'implementazione dell'algoritmo *steepest-edge* a coefficienti interi, *steepest-float* indica i tempi con una implementazione dello *steepest-edge* che fa uso dell'aritmetica floating point. Le ultime due colonne indicano i tempi delle librerie *Wallaroo* e *Cassowary*, le quali usano la libreria *GMP* per avere coefficienti a precisione arbitraria. In questo caso la PPL sembra essere particolarmente competitiva, specialmente nel caso in cui venga presa in considerazione l'implementazione *floating point* dell'algoritmo *steepest-edge* per la variabile entrante in base.

Capitolo 4

Conclusioni

Il presente lavoro di tesi ha permesso di integrare un'importante funzionalità nella libreria PPL, rendendo liberamente disponibile un risolutore di problemi di programmazione lineare con due importanti caratteristiche: l'incrementalità e l'aritmetica esatta. I test svolti hanno dimostrato che il risolutore implementato è uno dei più veloci, confrontato a risolutori analoghi che fanno uso di aritmetica esatta. Pur essendo arrivati ad una implementazione efficiente, il presente lavoro ha la possibilità di essere ulteriormente perfezionato. Ad esempio, notevoli miglioramenti potrebbero essere apportati all'implementazione dell'algoritmo *steepest-edge*: al momento, ad ogni passo del simplesso primale, vengono ricalcolate tutte le quantità descritte nella sezione 1.2.5, mentre sarebbe possibile mantenere tali coefficienti aggiornati sfruttando meglio l'incrementalità del calcolo. Un'altra possibile, ma sostanziale, modifica, può essere quella di utilizzare delle strutture dati interne che manipolino meglio matrici "sparse", ossia con un gran numero di zeri: questo permetterebbe di ridurre la dimensione occupata dal problema e, probabilmente, di avere delle riduzioni dei tempi di risoluzione.

Bibliografia

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill, *Possibly not closed convex polyhedra and the Parma Polyhedra Library*, Static Analysis: Proceedings of the 9th International Symposium (Madrid, Spain) (M. V. Hermenegildo and G. Puebla, eds.), Lecture Notes in Computer Science, vol. 2477, Springer-Verlag, Berlin, 2002, pp. 213–229.

[GLP06] *GLPK: The GNU linear programming kit*, release 4.9 ed., January 2006, Available at `http://www.gnu.org/software/glpk/glpk.html`.

[GNU04] *GMP: The GNU multiple precision arithmetic library*, release 4.1.4 ed., September 2004, Available at `http://www.swox.com/gmp/`.

[GR77] D. Goldfarb and J.K. Reid, *A practibable steepest-edge simplex algorithm*, Mathematical Proramming (1977), no. 12, 361–371.

[Mak01] Andrew Makhorin, *Implementation of the revised simplex method*, February 2001, Aviable at `http://ftp.gnu.org/old-gnu/glpk/glpk_sm.ps.gz`.

[PPL06] *PPL: The Parma Polyhedra Library*, release 0.9 ed., March 2006, Available at `http://www.cs.unipr.it/ppl/`.

[PS98] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial optimization*, Dover, 1998.

[Wal05] *Wallaroo linear constraint solving library*, release 0.1 ed., July 2005, Available at `http://wallaroo.sourceforge.net/`.