

Indice

1	Nozioni Fondamentali	1
1.1	Introduzione	1
1.2	Problemi di ottimizzazione	2
1.2.1	Restrizione	3
1.2.2	Rilassamento	3
1.2.3	Ricerca	4
1.3	Programmazione Lineare	5
1.3.1	Formulazioni equivalenti	6
1.3.2	Interpretazione geometrica	6
1.3.3	Algoritmo del simplesso	7
1.3.4	Dualità	8
1.3.5	Rilassamento Lagrangiano	10
1.4	Programmazione Lineare Intera	11
1.4.1	Rilassamento lineare	12
1.5	Frank-Wolfe	12
2	Set-Covering	15
2.1	Formulazione del problema	15
2.2	Rilassamento lineare	16
2.2.1	Problema duale	17
2.3	Rilassamento Lagrangiano	17
2.4	Applicazione dell'algoritmo di Frank-Wolfe	18
2.5	Matrice di riferimento	19
2.5.1	Rappresentazione CSR	20
3	Implementazione	21
3.1	Generazione delle istanze	21
3.1.1	Gestione dataset	22
3.2	Algoritmo del simplesso	22
3.2.1	Costruzione del modello	22
3.2.2	Soluzione del modello	23
3.3	Algoritmo di Frank-Wolfe	24
3.3.1	Scelta del punto di partenza	25

3.3.2	Soluzione del rilassamento Lagrangiano	25
3.3.3	Calcolo del subgradiente	27
3.3.4	Calcolo della soluzione che massimizza l'approssimazione lineare	28
3.3.5	Calcolo del punto successivo	29
3.3.6	Calcolo delle limitazioni al valore della funzione obiettivo	29
3.3.7	Composizione dell'algoritmo risolutivo	30
3.4	Esecuzione	32
3.4.1	Output	32

4 Risultati 33

4.1	Prove sperimentali	33
4.2	Qualità delle soluzioni	34
4.2.1	Dimensione della matrice di riferimento	34
4.2.2	Forma della matrice di riferimento	36

1

Nozioni Fondamentali

L'obiettivo di questo capitolo è quello di presentare i concetti di base che vengono utilizzati nel seguito di questo lavoro e, più in generale, nell'ambito dell'ottimizzazione matematica.

1.1 Introduzione

Nel corso della sua esistenza, l'uomo ha sempre dovuto affrontare e risolvere una grande varietà di problemi. Con il passare del tempo, le nostre capacità si sono evolute e gli strumenti a nostra disposizione sono migliorati, permettendoci di gestire problemi di complessità sempre maggiore. Di conseguenza, oggi non ci accontentiamo più di risolvere un problema trovando una soluzione qualsiasi, ma aspiriamo ad ottimizzare, cioè a identificare la soluzione migliore possibile, sulla base di criteri specifici.

Risolvere un problema di ottimizzazione significa assegnare valori alle variabili che lo caratterizzano, soddisfacendo un insieme di vincoli, con l'obiettivo di ottimizzare una grandezza specifica. La grandezza da ottimizzare e i vincoli da rispettare possono spesso essere rappresentati come funzioni delle variabili coinvolte, permettendoci di definire il problema utilizzando un modello matematico. La formulazione di un modello dovrebbe essere sufficientemente complessa da rappresentare accuratamente il problema cui si riferisce e, allo stesso tempo, abbastanza semplice da renderlo trattabile con gli strumenti risolutivi disponibili.

L'ottimizzazione spesso è un processo iterativo in cui il modello di riferimento di un problema viene continuamente raffinato, con l'obiettivo di ottenere soluzioni sempre più accurate. Con gli strumenti che abbiamo a disposizione, oggi siamo in grado di risolvere in modo efficiente problemi caratterizzati da un elevato numero di variabili e vincoli. Tuttavia, esistono classi di problemi per i quali non si può calcolare una soluzione ottima in un tempo ragionevole e in questi casi si ricorre ad algoritmi euristici, con l'obiettivo di ottenere soluzioni accettabili in tempi contenuti.

L'obiettivo di questo lavoro è quello di sviluppare un algoritmo risolutivo per il rilassamento lineare del set-covering che sia in grado di risolvere istanze del problema in modo efficiente. L'implementazione dell'algoritmo si basa sul metodo di Frank-Wolfe e l'idea è quella di operare un confronto con l'algoritmo del simplesso, relativamente ai tempi di esecuzione e alla qualità delle soluzioni trovate.

1.2 Problemi di ottimizzazione

Un problema di ottimizzazione \mathcal{P} può essere definito con la formulazione generale

$$\mathcal{P}: \begin{cases} \min \text{ (or max)} & f(\mathbf{x}) \\ & \mathcal{S} \\ & \mathbf{x} \in \mathcal{D} \end{cases} \quad (1.1)$$

dove $f: \mathcal{D} \rightarrow \mathbb{R}$, con $\mathbf{x} = [x_1, \dots, x_n]^T \in \mathcal{D} = \mathcal{D}_1 \times \dots \times \mathcal{D}_n$ tale che $x_j \in \mathcal{D}_j$ per ogni $1 \leq j \leq n$ e \mathcal{S} è un insieme finito di vincoli. Formalmente un vincolo è una funzione che coinvolge un sottoinsieme delle variabili del problema e che può assumere i valori vero o falso, corrispondenti alle condizioni di vincolo soddisfatto o violato, rispettivamente.

Calcolare il massimo di una funzione $f(\mathbf{x})$ è equivalente a calcolare il minimo della funzione $-f(\mathbf{x})$. Infatti, i due valori coincidono, a meno del segno, e si ottengono nello stesso punto. Di conseguenza, nel seguito possiamo limitarci a studiare i problemi di minimo senza perdere di generalità. Tutte le considerazioni che faremo saranno valide, eventualmente con opportune modifiche, anche per i problemi di massimo.

Definizione 1.1

Sia \mathcal{P} un problema di ottimizzazione come in (1.1). Ogni $\mathbf{x} \in \mathcal{D}$ si dice soluzione di \mathcal{P} . Una soluzione che soddisfi tutti i vincoli in \mathcal{S} si dice ammissibile per \mathcal{P} .

Per riferirci all'insieme di tutte le soluzioni ammissibili di \mathcal{P} , utilizzeremo la notazione $F(\mathcal{P})$.

Il dominio \mathcal{D} fornisce una caratterizzazione immediata dei problemi di ottimizzazione. Se \mathcal{D} è un insieme discreto, allora il problema è detto di ottimizzazione discreta. Se invece \mathcal{D} è un insieme continuo, allora il problema si dice di ottimizzazione continua. Nel caso particolare di un dominio \mathcal{D} che sia un insieme discreto e finito, si parla di ottimizzazione combinatoria.

Definizione 1.2. Soluzione ottima

Sia \mathcal{P} un problema di ottimizzazione di minimo come in (1.1). Una soluzione ammissibile $\mathbf{x}^* \in F(\mathcal{P})$ si dice ottima per \mathcal{P} se

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in F(\mathcal{P}).$$

Un problema di ottimizzazione \mathcal{P} si dice impossibile (*infeasible*) quando $F(\mathcal{P}) = \emptyset$. Diciamo invece che \mathcal{P} è illimitato (*unbounded*) quando non esiste alcun limite inferiore a $f(\mathbf{x})$, per $\mathbf{x} \in F(\mathcal{P})$. Se esiste una soluzione $\mathbf{x}^* \in F(\mathcal{P})$ ottima, allora diciamo che \mathcal{P} ammette ottimo finito.

La funzione $f(\mathbf{x})$ è chiamata funzione obiettivo e il valore $f(\bar{\mathbf{x}})$ è tipicamente noto come costo associato alla soluzione $\bar{\mathbf{x}} \in F(\mathcal{P})$.

Infine, un problema di ottimizzazione si dice risolto quando si trova una soluzione ottima, e si dimostra che è tale, oppure quando si dimostra che il problema è impossibile o illimitato.

1.2.1 Restrizione

Definizione 1.3. Restrizione

Sia \mathcal{P} un problema di ottimizzazione. Si definisce restrizione di \mathcal{P} un problema di ottimizzazione \mathcal{P}' ottenuto da \mathcal{P} aggiungendo vincoli.

Intuitivamente, aggiungere vincoli ad un problema significa ridurre lo spazio delle sue soluzioni ammissibili. Formalmente, se \mathcal{P}' è una restrizione di \mathcal{P} , allora $F(\mathcal{P}') \subseteq F(\mathcal{P})$. Di conseguenza, se $\bar{\mathbf{x}}$ è una generica soluzione ammissibile per \mathcal{P}' , ossia $\bar{\mathbf{x}} \in F(\mathcal{P}')$, allora $\bar{\mathbf{x}}$ è ammissibile per \mathcal{P} , cioè $\bar{\mathbf{x}} \in F(\mathcal{P})$. Inoltre, è facile verificare che il costo associato a $\bar{\mathbf{x}} \in F(\mathcal{P}')$ fornisce un limite superiore (*upper bound*) al valore ottimo di \mathcal{P} , ossia $f(\mathbf{x}^*) \leq f(\bar{\mathbf{x}})$, dove $\mathbf{x}^* \in F(\mathcal{P})$ rappresenta una soluzione ottima per \mathcal{P} .

Infine, poiché $F(\mathcal{P}') \subseteq F(\mathcal{P})$, si dimostra immediatamente che se \mathcal{P} è impossibile, cioè $F(\mathcal{P}) = \emptyset$, allora anche ogni sua restrizione \mathcal{P}' è impossibile, ossia $F(\mathcal{P}') = \emptyset$. Non vale il viceversa.

1.2.2 Rilassamento

Definizione 1.4. Rilassamento

Sia \mathcal{P} un problema di ottimizzazione. Si definisce rilassamento di \mathcal{P} un problema di ottimizzazione \mathcal{R} ottenuto da \mathcal{P} rimuovendo vincoli e/o sostituendo la funzione obiettivo $f(\mathbf{x})$ di \mathcal{P} con una sua approssimazione inferiore $g(\mathbf{x})$ per ogni $\mathbf{x} \in F(\mathcal{P})$. Formalmente, \mathcal{R} è un rilassamento di \mathcal{P} se

- $F(\mathcal{P}) \subseteq F(\mathcal{R})$,
- $g(\mathbf{x}) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in F(\mathcal{P})$.

Si verifica immediatamente che se \mathcal{R} è impossibile, ossia $F(\mathcal{R}) = \emptyset$, allora anche \mathcal{P} è impossibile, cioè $F(\mathcal{P}) = \emptyset$. Non vale il viceversa. Inoltre, è facile dimostrare che se $\bar{\mathbf{x}} \in F(\mathcal{R})$ è soluzione ottima per \mathcal{R} , allora $g(\bar{\mathbf{x}})$ fornisce un limite inferiore (*lower bound*) al valore ottimo di \mathcal{P} . Formalmente risulta $f(\mathbf{x}^*) \geq g(\bar{\mathbf{x}})$, dove $\mathbf{x}^* \in F(\mathcal{P})$ rappresenta una soluzione ottima per \mathcal{P} .

Infine, se la soluzione ottima $\mathbf{x}^* \in F(\mathcal{R})$ di \mathcal{R} è ammissibile per \mathcal{P} , con $g(\mathbf{x}^*) = f(\mathbf{x}^*)$, allora \mathbf{x}^* è soluzione ottima per \mathcal{P} .

1.2.3 Ricerca

Dato un problema di ottimizzazione \mathcal{P} , la ricerca è il processo che consiste nel risolvere una sequenza finita $\mathcal{P}_1, \dots, \mathcal{P}_m$ di restrizioni di \mathcal{P} . L'idea alla base della ricerca è quella di aggiungere vincoli al problema di partenza per ottenere delle restrizioni che siano più semplici da risolvere. Le soluzioni delle restrizioni possono poi essere utilizzate come informazioni aggiuntive nel processo di risoluzione di \mathcal{P} .

Definizione 1.5. Ricerca esaustiva

Siano \mathcal{P} un problema di ottimizzazione e $\mathcal{P}_1, \dots, \mathcal{P}_m$ una sequenza di sue restrizioni. Si definisce ricerca esaustiva il processo di ricerca che esplora tutto lo spazio delle soluzioni ammissibili di \mathcal{P} . Formalmente, deve valere

$$\bigcup_{i=1}^m F(\mathcal{P}_i) = F(\mathcal{P}).$$

Una ricerca non esaustiva si dice euristica. Una ricerca esaustiva permette di risolvere un problema di ottimizzazione \mathcal{P} trovando le soluzioni di tutte le restrizioni \mathcal{P}_i e scegliendo quella migliore.

La forma più semplice di ricerca esaustiva prende il nome di *generate-and-test* e consiste nel generare esplicitamente tutte le soluzioni $\mathbf{x} \in \mathcal{D}$ di \mathcal{P} , verificare quali soddisfano i vincoli del problema e scegliere tra queste quella migliore. Questa strategia è applicabile in pratica solo a una classe ristretta di problemi di ottimizzazione e in generale non è molto efficiente.

Una tipologia di ricerca migliore è quella che viene chiamata *tree-search* e che sfrutta una struttura ad albero per esplorare lo spazio delle soluzioni ammissibili di un problema. Questo tipo di ricerca è alla base di molti algoritmi che vengono utilizzati nella pratica per risolvere problemi di ottimizzazione. L'idea è quella di dividere ricorsivamente lo spazio di ricerca delle soluzioni ammissibili di \mathcal{P} , creando delle restrizioni in modo da formare una struttura ad albero in cui i nodi foglia corrispondono a restrizioni sufficientemente facili da risolvere direttamente. Una ricerca di questo tipo è spesso combinata con il concetto di rilassamento e l'obiettivo è quello di semplificare ulteriormente la risoluzione delle restrizioni associate ai vari nodi dell'albero.

Infine, è importante precisare che una ricerca esaustiva non è necessariamente l'approccio risolutivo migliore in tutte le situazioni. Esistono classi di problemi per cui ha poco senso provare a trovare una soluzione ottima, ad esempio perché esplorare per intero lo spazio delle soluzioni ammissibili richiederebbe un tempo di computazione troppo elevato. Di conseguenza, in queste situazioni è molto più utile utilizzare algoritmi euristici, il cui obiettivo è quello di fornire una soluzione accettabile in tempi ragionevoli.

1.3 Programmazione Lineare

La programmazione lineare costituisce uno dei paradigmi fondamentali nell'ambito dell'ottimizzazione, poichè si applica in modo naturale a molti problemi del mondo reale, che risultano quindi facili da modellare. Nel corso del tempo sono stati sviluppati molteplici algoritmi, con l'obiettivo di risolvere in maniera efficiente problemi di programmazione lineare che coinvolgono un numero elevato di variabili e vincoli.

Un problema di programmazione lineare (*Linear Program*, LP), è un problema di ottimizzazione in cui la funzione obiettivo e i vincoli sono funzioni lineari. Per un problema di programmazione lineare con n variabili e m vincoli si può utilizzare la formulazione generale

$$\mathcal{P}: \begin{cases} \min & z = c_1 x_1 + \cdots + c_n x_n \\ & a_{11} x_1 + \cdots + a_{1n} x_n \sim b_1 \\ & \quad \vdots \quad \ddots \quad \vdots \quad \vdots \\ & a_{m1} x_1 + \cdots + a_{mn} x_n \sim b_m \\ & \ell_j \leq x_j \leq u_j \quad \forall j: 1 \leq j \leq n \end{cases} \quad (1.2)$$

dove $\sim \in \{\leq, =, \geq\}$, $\ell_j \in \mathbb{R} \cup \{-\infty\}$ e $u_j \in \mathbb{R} \cup \{+\infty\}$, con $c_k \in \mathbb{R} \quad \forall k: 1 \leq k \leq n$, $a_{ij} \in \mathbb{R} \quad \forall i, j: 1 \leq i \leq m, 1 \leq j \leq n$ e $b_i \in \mathbb{R} \quad \forall i: 1 \leq i \leq m$. Le variabili del problema hanno come dominio intervalli (eventualmente illimitati) di \mathbb{R} e la funzione obiettivo può essere scritta nella forma compatta

$$z = \sum_{k=1}^n c_k x_k. \quad (1.3)$$

Similmente, il generico vincolo può essere scritto come

$$\sum_{j=1}^n a_{ij} x_j \sim b_i \quad \forall i: 1 \leq i \leq m. \quad (1.4)$$

La proprietà di linearità di funzione obiettivo e vincoli permette di utilizzare la teoria dell'analisi convessa come base nello sviluppo di algoritmi risolutivi per problemi di programmazione lineare. In aggiunta, questa proprietà semplifica significativamente il processo attraverso cui è possibile ottenere formulazioni alternative, tutte equivalenti tra loro, relativamente ad uno stesso problema di programmazione lineare.

1.3.1 Formulazioni equivalenti

Uno stesso problema di programmazione lineare può essere espresso attraverso molteplici formulazioni differenti, tutte equivalenti tra loro. Inoltre, con le opportune trasformazioni, è sempre possibile passare da una formulazione all'altra. Per il generico problema di programmazione lineare \mathcal{P} con n variabili e m vincoli, le due forme maggiormente utilizzate sono la forma standard e quella canonica, riportate di seguito.

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Forma Standard (1.5a)

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Forma Canonica (1.5b)

dove $\mathbf{x} \in \mathbb{R}^n$, con $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ e $\mathbf{b} \in \mathbb{R}^m$. Le trasformazioni necessarie per passare da una forma all'altra consistono nell'introduzione di variabili ausiliarie e la convenienza nell'utilizzare una forma piuttosto che un'altra dipende dallo specifico contesto applicativo.

1.3.2 Interpretazione geometrica

Prima di analizzare un problema di programmazione lineare dal punto di vista geometrico è necessario introdurre alcuni concetti, che vengono presentati di seguito.

Definizione 1.6. Insieme convesso

Un insieme \mathcal{C} si dice convesso se $\lambda x_1 + (1 - \lambda)x_2 \in \mathcal{C} \quad \forall x_1, x_2 \in \mathcal{C}, \text{ con } \lambda \in [0, 1]$.

La definizione può essere generalizzata per un numero finito di punti x_1, \dots, x_k in \mathcal{C} , utilizzando la combinazione lineare convessa

$$\sum_{i=1}^k \lambda_i x_i, \quad \sum_{i=1}^k \lambda_i = 1. \quad (1.6)$$

Esempi di insiemi convessi sono gli iperpiani $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^T \mathbf{x} = a_0\}$ e i semispazi chiusi $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^T \mathbf{x} \leq a_0\}$, dove $\mathbf{a} \in \mathbb{R}^n$ e $a_0 \in \mathbb{R}$. Si può dimostrare che l'intersezione di un numero finito di insiemi convessi è ancora un insieme convesso. Di conseguenza, l'intersezione di un numero finito di iperpiani e semispazi chiusi è ancora un insieme convesso che viene chiamato poliedro. Un poliedro limitato è chiamato politopo.

Le definizioni presentate fino a questo punto ci permettono di osservare che lo spazio delle soluzioni ammissibili di un problema di programmazione lineare è un insieme convesso, poichè intersezione di vincoli lineari, e in particolare un poliedro.

La definizione di un politopo come intersezione di iperpiani e semispazi chiusi è detta descrizione esterna. Esiste una modalità alternativa di definire un politopo, chiamata descrizione interna, che si basa sul concetto di vertice.

Definizione 1.7. Vertice

Si dice vertice di un poliedro un punto che non può essere espresso come combinazione lineare convessa stretta di altri due punti del poliedro.

Un politopo ha un numero finito di vertici e vale il teorema riportato di seguito.

Teorema 1.1. Descrizione interna di un politopo

Siano \mathcal{P} un politopo di \mathbb{R}^n e $\mathcal{V} = \{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ l'insieme dei suoi vertici. Allora ogni punto di \mathcal{P} può essere espresso come combinazione lineare convessa dei punti in \mathcal{V} :

$$\mathbf{x} \in \mathcal{P} \iff \mathbf{x} = \sum_{i=1}^k \lambda_i \mathbf{x}_i, \quad \sum_{i=1}^k \lambda_i = 1, \quad \lambda_i \geq 0 \quad \forall i: 1 \leq i \leq k.$$

La descrizione interna di un politopo è uno strumento importante perché permette di dimostrare il teorema fondamentale della programmazione lineare.

Teorema 1.2. Teorema fondamentale della programmazione lineare

Consideriamo il problema di programmazione lineare $\min\{\mathbf{c}^T \mathbf{x} \mid \mathbf{x} \in \mathcal{P}\}$, dove \mathcal{P} è il politopo che rappresenta la regione ammissibile. Allora, se il problema ammette ottimo finito, esiste almeno un vertice di \mathcal{P} che è soluzione ottima.

Questo teorema ci fornisce un modo concreto per risolvere un problema di programmazione lineare, che consiste nel limitare la ricerca della soluzione ottima all'insieme dei vertici del politopo che definisce la regione ammissibile.

Il teorema può essere esteso al caso generale di problemi di programmazione lineare in cui la regione ammissibile è un poliedro (con almeno un vertice), e non necessariamente un politopo. L'ipotesi sull'esistenza della soluzione ottima deve comunque valere, per evitare che il problema possa risultare illimitato.

1.3.3 Algoritmo del simplesso

L'algoritmo del simplesso è uno degli algoritmi maggiormente impiegati nell'ambito dell'ottimizzazione per risolvere problemi di programmazione lineare. È un algoritmo iterativo che cerca di sfruttare il teorema fondamentale della programmazione lineare in modo intelligente, con l'obiettivo ridurre il numero dei vertici da ispezionare per trovare la soluzione ottima. L'idea è quella di partire da un vertice arbitrario del poliedro che definisce la regione ammissibile e di muoversi ad ogni iterazione in un vertice adiacente non peggiore, relativamente al valore della funzione obiettivo.

È importante precisare che nonostante l'algoritmo del simplesso scelga di spostarsi tra i vertici in modo intelligente, non c'è alcuna garanzia che il vertice corrispondente alla soluzione ottima venga trovato prima di aver ispezionato tutti gli altri vertici del politopo. Per questo motivo, si può dimostrare che la complessità computazionale al caso peggiore è esponenziale.

1.3.4 Dualità

All'inizio della sezione 1.2 abbiamo argomentato che per dichiarare risolto un problema di ottimizzazione, non è sufficiente trovare una soluzione ottima, ma è necessario fornire una dimostrazione che attesti l'ottimalità di tale soluzione, relativamente al problema considerato.

Per i problemi di programmazione lineare esiste un modo elegante e rigoroso per certificare l'ottimalità di una soluzione, che utilizza un problema di ottimizzazione di supporto chiamato problema duale. Il problema duale è ottenuto a partire dal problema iniziale che, in questo contesto, prende il nome di problema primale. L'idea di base è quella di combinare i vincoli del problema primale per costruire una limitazione al valore della sua funzione obiettivo. Questo concetto può essere compreso meglio attraverso un esempio di carattere generale, che viene riportato di seguito.

Sia \mathcal{P} un problema di programmazione lineare con n variabili e m vincoli. Senza perdita di generalità, possiamo assumere che \mathcal{P} sia espresso nella forma standard (1.5a) oppure in quella canonica (1.5b). Ad esempio, sia \mathcal{P} della forma

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{1.7}$$

dove $\mathbf{x} \in \mathbb{R}^n$, con $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ e $\mathbf{b} \in \mathbb{R}^m$. Supponiamo che \mathbf{x}^* sia la soluzione ottima candidata. Per certificare l'ottimalità dobbiamo dimostrare che non esistono soluzioni migliori. Nel caso di un problema di minimo, questo significa verificare che \mathbf{x}^* è la soluzione di costo minimo, ossia che $f(\mathbf{x}^*) \leq f(\mathbf{x})$ per ogni $\mathbf{x} \in F(\mathcal{P})$.

Per la generica soluzione ammissibile $\mathbf{x} \in F(\mathcal{P})$ vale $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. Allora, introducendo un vettore $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{u} \geq 0$, chiamato vettore di moltiplicatori, risulta che

$$\mathbf{u}^T \mathbf{A}\mathbf{x} \geq \mathbf{u}^T \mathbf{b}. \tag{1.8}$$

Inoltre, possiamo costruire \mathbf{u} in modo che valga $\mathbf{c}^T \geq \mathbf{u}^T \mathbf{A}$, da cui, usando la (1.8), si trova

$$\mathbf{c}^T \mathbf{x} \geq \mathbf{u}^T \mathbf{A}\mathbf{x} \geq \mathbf{u}^T \mathbf{b} \quad \forall \mathbf{x} \in F(\mathcal{P}), \tag{1.9}$$

essendo $\mathbf{x} \geq 0$. In altre parole, possiamo utilizzare \mathbf{u} per creare una combinazione lineare dei vincoli di \mathcal{P} che costituisca una limitazione inferiore al valore della funzione obiettivo. A questo punto, dobbiamo scegliere \mathbf{u} in modo da ottenere la limitazione inferiore migliore, cioè quella più alta possibile. In effetti, questo significa risolvere il problema di programmazione lineare

$$\mathcal{D}: \begin{cases} \max & \mathbf{u}^T \mathbf{b} \\ & \mathbf{u}^T \mathbf{A} \leq \mathbf{c}^T \\ & \mathbf{u} \geq 0 \end{cases} \quad (1.10)$$

nelle variabili $\mathbf{u} = [u_1 \dots u_m]^T \in \mathbb{R}^m$, con \mathcal{D} che prende il nome di problema duale di \mathcal{P} . Con qualche modifica, il ragionamento si applica identicamente a problemi espressi nella forma standard e, in maniera del tutto simmetrica, può essere applicato a problemi di massimo.

Le disuguaglianze (1.8) e (1.9) hanno una conseguenza immediata, riassunta nel teorema della dualità debole, che viene presentato di seguito.

Teorema 1.3. Dualità debole

Siano \mathcal{P} un problema di programmazione lineare come in (1.7) e \mathcal{D} il suo problema duale come in (1.10). Allora, per le generiche soluzioni ammissibili \mathbf{x} e \mathbf{u} di \mathcal{P} e \mathcal{D} , rispettivamente, risulta

$$\mathbf{c}^T \mathbf{x} \geq \mathbf{b}^T \mathbf{u}.$$

Inoltre, se vale $\mathbf{c}^T \mathbf{x} = \mathbf{b}^T \mathbf{u}$, allora \mathbf{x} e \mathbf{u} sono soluzioni ottime di \mathcal{P} e \mathcal{D} , rispettivamente.

Naturalmente, il teorema è valido simmetricamente per problemi di massimo.

Infine, enunciamo il teorema della dualità forte, che fornisce un modo concreto di certificare l'ottimalità della soluzione di un problema di programmazione lineare.

Teorema 1.4. Dualità forte

Consideriamo una coppia primale - duale di problemi di programmazione lineare. Allora, se uno dei due ammette ottimo finito, anche l'altro ammette ottimo finito e il valore ottimo della funzione obiettivo è lo stesso per entrambi.

Di conseguenza, per verificare l'ottimalità della soluzione \mathbf{x}^* del problema primale \mathcal{P} , è sufficiente determinare una soluzione ammissibile \mathbf{u}^* del problema duale \mathcal{D} , in modo che risulti

$$\mathbf{c}^T \mathbf{x}^* = \mathbf{b}^T \mathbf{u}^*. \quad (1.11)$$

Se una tale soluzione \mathbf{u}^* esiste, allora \mathbf{x}^* è ottima per \mathcal{P} e \mathbf{u}^* è ottima per \mathcal{D} . Al contrario, l'inesistenza di \mathbf{u}^* ammissibile per \mathcal{D} che soddisfi la (1.11) dimostra che \mathbf{x}^* non è ottima per \mathcal{P} .

1.3.5 Rilassamento Lagrangiano

In alcune situazioni la formulazione di un problema di programmazione lineare può essere abbastanza complessa da impedire l'applicazione diretta di un algoritmo risolutivo. In questi casi può essere utile provare a trasformarla in una formulazione differente, con l'obiettivo di ottenere un problema più facile da risolvere. Alcune volte il problema associato alla formulazione alternativa è equivalente a quello iniziale e ha lo stesso valore ottimo. Altre volte non siamo così fortunati, e le due formulazioni non sono equivalenti. Di conseguenza, la formulazione alternativa non è sempre sufficiente per risolvere direttamente il problema di partenza, ma risulta comunque utile per ottenere delle limitazioni alla sua funzione obiettivo.

L'intuizione alla base del rilassamento Lagrangiano è quella di semplificare un problema rimuovendo alcuni dei suoi vincoli e aggiungendo un contributo alla funzione obiettivo. Questo contributo, che prende il nome di penalizzazione Lagrangiana, è ottenuto a partire dai vincoli eliminati e serve a peggiorare il valore della funzione obiettivo per soluzioni che non li soddisfano. A questo punto possiamo formalizzare questo concetto, limitandoci ad analizzare gli aspetti fondamentali che riflettono gli obiettivi di questo lavoro.

Sia \mathcal{P} un generico problema di programmazione lineare, espresso nella forma canonica

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ & \mathbf{Ax} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{1.12}$$

dove $\mathbf{x} \in \mathbb{R}^n$, con $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ e $\mathbf{b} \in \mathbb{R}^m$. Introduciamo un vettore $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{u} \geq 0$ di moltiplicatori, chiamati moltiplicatori di Lagrange, e definiamo la funzione

$$L(\mathbf{x}, \mathbf{u}) = \mathbf{c}^T \mathbf{x} + \mathbf{u}^T (\mathbf{b} - \mathbf{Ax}), \tag{1.13}$$

che prende il nome di funzione Lagrangiana di \mathcal{P} . Questa funzione ci permette di definire la famiglia di problemi Lagrangiani della forma

$$\mathcal{L}(\mathbf{u}): \begin{cases} \min \quad & \mathbf{c}^T \mathbf{x} + \mathbf{u}^T (\mathbf{b} - \mathbf{Ax}) = L(\mathbf{x}, \mathbf{u}) \\ & \mathbf{x} \geq 0 \end{cases} \tag{1.14}$$

che sono sempre un rilassamento di \mathcal{P} e quindi forniscono un limite inferiore al valore ottimo della sua funzione obiettivo. Utilizzando una forma più compatta, possiamo scrivere

$$\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x} \geq 0} L(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{x} \geq 0} \{\mathbf{c}^T \mathbf{x} + \mathbf{u}^T (\mathbf{b} - \mathbf{Ax})\}, \tag{1.15}$$

con $\mathcal{L}(\mathbf{u})$ che viene chiamata funzione duale di \mathcal{P} . Per trovare la migliore limitazione inferiore al valore ottimo di \mathcal{P} , dobbiamo risolvere all'ottimo il problema duale Lagrangiano

$$\mathcal{D}: \begin{cases} \max \quad & \mathcal{L}(\mathbf{u}) \\ & \mathbf{u} \geq 0 \end{cases} \tag{1.16}$$

Infine, si può dimostrare che se \mathcal{P} ammette ottimo finito, allora il valore ottimo della sua funzione obiettivo coincide con il costo di una soluzione ottima del problema duale Lagrangiano \mathcal{D} . Inoltre, tale soluzione può essere ottenuta anche con i moltiplicatori che risolvono all'ottimo il problema duale di \mathcal{P} , definito come in (1.10). Questo risultato costituisce la base teorica dell'algoritmo risolutivo che implementeremo nel capitolo 3.

1.4 Programmazione Lineare Intera

Il paradigma della programmazione lineare intera è un'estensione della programmazione lineare che aggiunge la possibilità di vincolare le variabili ad assumere valori interi. Il generico problema di programmazione lineare intera con n variabili e m vincoli può essere definito con la formulazione generale

$$\mathcal{P}: \begin{cases} \min & z = c_1 x_1 + \cdots + c_n x_n \\ & a_{11} x_1 + \cdots + a_{1n} x_n \sim b_1 \\ & \vdots \quad \ddots \quad \vdots \quad \vdots \\ & a_{m1} x_1 + \cdots + a_{mn} x_n \sim b_m \\ & \ell_j \leq x_j \leq u_j \quad \forall j: 1 \leq j \leq n \\ & x_j \in \mathbb{Z} \quad \forall j \in J \subseteq N = \{1, \dots, n\} \end{cases} \quad (1.17)$$

dove $\sim \in \{\leq, =, \geq\}$, $\ell_j \in \mathbb{R} \cup \{-\infty\}$ e $u_j \in \mathbb{R} \cup \{+\infty\}$, con $c_k \in \mathbb{R} \quad \forall k: 1 \leq k \leq n$, $a_{ij} \in \mathbb{R} \quad \forall i, j: 1 \leq i \leq m, 1 \leq j \leq n$ e $b_i \in \mathbb{R} \quad \forall i: 1 \leq i \leq m$. Quando tutte le variabili sono vincolate ad assumere valori interi, ossia $J = N$, il problema è detto di programmazione lineare intera pura. Se invece il vincolo di interezza è applicato solo ad alcune variabili, cioè $J \subset N$, il problema è detto di programmazione lineare intera mista. Naturalmente quando $J = \emptyset$ il problema diventa un problema di programmazione lineare.

La possibilità di vincolare le variabili ad assumere valori interi permette di modellare molti più problemi del mondo reale di quanto non si riesca a fare con la semplice programmazione lineare. Il prezzo da pagare è una difficoltà maggiore nel processo risolutivo. Infatti, per un problema di programmazione lineare intera non valgono molte delle considerazioni che abbiamo fatto per la programmazione lineare. L'aver introdotto vincoli di interezza non garantisce più la convessità della regione ammissibile, e neanche le proprietà che costituiscono il fondamento per l'algoritmo del simplesso. Per questo motivo, nel tempo sono stati sviluppati ulteriori algoritmi, specificatamente ideati per gestire i vincoli di interezza e risolvere i problemi di programmazione lineare intera.

1.4.1 Rilassamento lineare

Nel risolvere problemi di programmazione lineare intera, è molto importante utilizzare il concetto di rilassamento. Il rilassamento più intuitivo e naturale per un problema di programmazione lineare intera è il rilassamento lineare. L'idea è quella di considerare il problema iniziale, ma senza i vincoli di interezza per le variabili che lo richiedono. Questa semplificazione trasforma il problema iniziale in un problema di programmazione lineare che quindi può essere risolto con tutti gli strumenti di cui abbiamo già discusso. L'utilità del rilassamento lineare dipende dallo specifico problema. In alcuni casi il rilassamento è abbastanza forte da fornire una limitazione utile per la funzione obiettivo del problema iniziale. Nelle situazioni in cui è invece molto debole, viene spesso combinato con i piani di taglio, con l'obiettivo di ottenere un rilassamento più forte, in grado di fornire una limitazione migliore. Un piano di taglio è un vincolo che è violato dalla soluzione ottima corrente del rilassamento lineare, ma soddisfatto dalle soluzioni ammissibili del problema di partenza. Geometricamente si può visualizzare proprio come un taglio all'interno della regione ammissibile del rilassamento, che però non interseca la regione ammissibile del problema di partenza, ottenuta considerando i vincoli di interezza. Questo vincolo viene aggiunto al rilassamento lineare per ottenere una limitazione inferiore migliore. Il processo di aggiunta di piani di taglio può essere iterato per migliorare la soluzione di un rilassamento lineare e ottenere una limitazione migliore al valore ottimo del problema di partenza. In effetti, questo è proprio il meccanismo con cui i rilassamenti lineari vengono utilizzati nell'algoritmo Branch & Cut (B&C), specificatamente ideato per risolvere problemi di programmazione lineare intera.

Sebbene il rilassamento lineare sia uno strumento utile nella pratica, è fondamentale precisare che non ha alcun significato in relazione al problema di partenza. La possibilità di assegnare valori non interi per variabili che hanno il vincolo di interezza è una grande semplificazione, che però cancella la semantica di queste variabili. Ad esempio, la programmazione lineare intera può essere utilizzata per modellare problemi caratterizzati da decisioni sì/no, che quindi possono essere rappresentate da variabili binarie. In questi casi una soluzione del rilassamento lineare in cui tali variabili assumono valori frazionari non ha nessun significato nel contesto del problema iniziale.

1.5 Frank-Wolfe

L'algoritmo di Frank-Wolfe è un'algoritmo iterativo utilizzato per risolvere problemi di ottimizzazione. In questo lavoro l'algoritmo di Frank-Wolfe verrà applicato per risolvere il problema duale Lagrangiano della forma presentata in (1.16) e, di conseguenza, le

considerazioni che faremo saranno limitate a problemi di ottimizzazione della forma

$$\mathcal{P}: \begin{cases} \max & f(\mathbf{x}) \\ & \mathbf{x} \in \mathcal{D} \end{cases} \quad (1.18)$$

dove \mathcal{D} è un insieme convesso e compatto e $f: \mathcal{D} \rightarrow \mathbb{R}$ è una funzione concava, ma non necessariamente differenziabile in tutto \mathcal{D} .

Indicheremo con \mathbf{s}_k il subgradiente di f in un punto $\mathbf{x}_k \in \mathcal{D}$. Naturalmente, se f è differenziabile in \mathbf{x}_k , allora $\mathbf{s}_k = \nabla f(\mathbf{x}_k)$.

L'algoritmo di Frank-Wolfe si sviluppa come segue.

- **Inizializzazione:** sia $k \leftarrow 0$ e $\mathbf{x}_0 \in \mathcal{D}$ un punto arbitrario che utilizzeremo come punto di partenza.
- **Passo 1:** trovare $\mathbf{d} \in \mathcal{D}$ che massimizza l'approssimazione lineare di f in \mathbf{x}_k , definita dal subgradiente \mathbf{s}_k . Formalmente, dobbiamo calcolare

$$\mathbf{d}_k = \arg \max_{\mathbf{d} \in \mathcal{D}} \{\mathbf{s}_k \mathbf{d}\}.$$

- **Passo 2:** decidere come muoversi sul segmento che unisce i punti \mathbf{x}_k e \mathbf{d}_k , per determinare il punto \mathbf{x}_{k+1} da utilizzare nell'iterazione successiva. Ad esempio, in questo lavoro useremo $\gamma = \frac{2}{k+2}$ per calcolare

$$\mathbf{x}_{k+1} = (1 - \gamma)\mathbf{x}_k + \gamma\mathbf{d}_k.$$

Notiamo che per come è definito, il generico punto \mathbf{x}_{k+1} calcolato al termine dell'iterazione k è sempre contenuto in \mathcal{D} , poichè è ottenuto come combinazione lineare di due punti in \mathcal{D} , che è un insieme convesso.

I passi dell'algoritmo di Frank-Wolfe sono riassunti nella specifica ad alto livello riportata di seguito.

Algoritmo di Frank-Wolfe

```

1  $\mathbf{x}_0 \in \mathcal{D}$  punto di partenza                                /* Inizializzazione */

2 while true do

3    $\mathbf{d}_k = \arg \max_{\mathbf{d} \in \mathcal{D}} \{\mathbf{s}_k \mathbf{d}\}$                                 /* Passo 1 */

4    $\mathbf{x}_{k+1} = (1 - \gamma)\mathbf{x}_k + \gamma\mathbf{d}_k$  con  $\gamma = \frac{2}{k+2}$                 /* Passo 2 */
```

Naturalmente un'implementazione dell'algoritmo deve impostare una condizione di terminazione, che impedisca all'algoritmo di iterare in maniera indefinita.

L'obiettivo di questo capitolo è quello di presentare il problema del set-covering e di definire il suo rilassamento lineare, che viene utilizzato per testare l'algoritmo implementato nel prossimo capitolo.

2.1 Formulazione del problema

Siano $\mathcal{I} = \{1, \dots, m\}$ un insieme e $\mathcal{F} = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ una famiglia di sottoinsiemi di \mathcal{I} . Il problema del set-covering richiede di trovare la più piccola collezione $\mathcal{F}^* \subseteq \mathcal{F}$ tale che

$$\bigcup_{\mathcal{F}_j \in \mathcal{F}^*} \mathcal{F}_j = \mathcal{I}. \quad (2.1)$$

In altre parole, si richiede che ciascun elemento di \mathcal{I} sia coperto da almeno uno dei sottoinsiemi in \mathcal{F}^* . Il problema può essere formulato come problema di programmazione lineare intera, introducendo, per ciascun elemento in \mathcal{F} , una variabile binaria che rappresenta l'appartenenza a \mathcal{F}^* . In particolare, definiamo

$$x_j = \begin{cases} 1 & \text{se } \mathcal{F}_j \in \mathcal{F}^* \\ 0 & \text{altrimenti} \end{cases} \quad \forall j: 1 \leq j \leq n. \quad (2.2)$$

A questo punto, le variabili del problema permettono di specificare la funzione obiettivo

$$\min \sum_{j=1}^n x_j \quad (2.3)$$

che minimizza il numero dei sottoinsiemi scelti, ossia la cardinalità di \mathcal{F}^* . Infine, dobbiamo specificare la condizione (2.1) come vincolo lineare, e quindi richiediamo che

$$\sum_{j: i \in \mathcal{F}_j} x_j \geq 1 \quad \forall i \in \mathcal{I}. \quad (2.4)$$

Combinando tutte le considerazioni fatte, otteniamo la formulazione matematica

$$\begin{aligned} \min \quad & \sum_{j=1}^n x_j \\ & \sum_{j: i \in \mathcal{F}_j} x_j \geq 1 \quad \forall i \in \mathcal{I} \\ & x_j \in \{0, 1\} \quad \forall j: 1 \leq j \leq n \end{aligned} \quad (2.5)$$

che descrive il problema del set-covering come problema di programmazione lineare intera. Inoltre, notiamo che l'espressione della funzione obiettivo stabilisce implicitamente il limite superiore per le variabili, poiché non c'è mai convenienza nel selezionare lo stesso elemento $\mathcal{F}_j \in \mathcal{F}$ più di una volta. Questa considerazione ci permette di modificare i vincoli di dominio, richiedendo che le variabili siano semplicemente intere non negative.

2.2 Rilassamento lineare

Nel seguito di questo lavoro ci limiteremo a considerare il rilassamento lineare del set-covering, che rimuove il vincolo di interezza per le variabili e richiede solamente che siano non negative. Inoltre, nell'ottica di quanto presentato nel prossimo capitolo, è conveniente considerare una formulazione alternativa, del tutto equivalente a quella presentata in (2.5), ottenuta trasformando il vincolo (2.4) in

$$\sum_{j=1}^n a_{ij} x_j \geq 1 \quad \forall i \in \mathcal{I}, \quad (2.6)$$

dove a_{ij} è un parametro binario che assume il valore 1 se e solo se $i \in \mathcal{F}_j$, per ogni elemento $i \in \mathcal{I}$, con $1 \leq j \leq n$.

Di conseguenza, Il rilassamento lineare del set-covering può essere definito con la formulazione

$$\mathcal{P}: \begin{cases} \min & \sum_{j=1}^n x_j \\ & \sum_{j=1}^n a_{ij} x_j \geq 1 \quad \forall i \in \mathcal{I} \\ & x_j \geq 0 \quad \forall j: 1 \leq j \leq n \end{cases} \quad (2.7)$$

che è quella di un problema di programmazione lineare nella forma canonica

$$\begin{aligned} \min & \quad \mathbf{c}^T \mathbf{x} \\ & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \quad (2.8)$$

ottenuta ponendo $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{m \times n}$, con $\mathbf{x} = [x_1, \dots, x_n]^T \in \mathbb{R}^n$, $\mathbf{c} = [1, 1, \dots, 1]^T \in \mathbb{R}^n$ e $\mathbf{b} = [1, 1, \dots, 1]^T \in \mathbb{R}^m$.

Naturalmente una soluzione del rilassamento lineare non è necessariamente ammissibile per il problema di partenza, ma fornisce un limite inferiore al suo valore ottimo.

Nel seguito utilizzeremo \mathcal{P} per riferirci al rilassamento lineare del set-covering.

2.2.1 Problema duale

Procediamo ora con la definizione del problema duale di \mathcal{P} , seguendo le considerazioni che abbiamo fatto in 1.3.4. Introduciamo un vettore $\mathbf{u} = [u_1, \dots, u_m]^T \in \mathbb{R}^m$ di moltiplicatori duali per combinare linearmente i vincoli e identificare una limitazione inferiore al valore della funzione obiettivo. In particolare, dobbiamo trovare \mathbf{u} tale che

$$\sum_{j=1}^n x_j \geq \sum_{i=1}^m \left(u_i \sum_{j=1}^n a_{ij} x_j \right) \geq \sum_{i=1}^m u_i \quad (2.9)$$

A questo punto, per trovare la limitazione migliore possibile, cioè quella più alta, dobbiamo risolvere il problema duale

$$\begin{aligned} \max \quad & \sum_{i=1}^m u_i \\ & \sum_{i=1}^m a_{ij} u_i \leq 1 \quad \forall j: 1 \leq j \leq n \\ & u_i \geq 0 \quad \forall i: 1 \leq i \leq m \end{aligned} \quad (2.10)$$

che si ottiene dalle disuguaglianze nella (2.9). Notiamo che per come è definito, il vincolo del problema duale impone una limitazione superiore ai moltiplicatori, ossia $u_i \in [0, 1]$ per ogni $1 \leq i \leq m$, visto che a_{ij} è un parametro binario che può assumere solo i valori 0 e 1, per $1 \leq i \leq m$ e $1 \leq j \leq n$. Questa limitazione sarà importante più avanti, quando presenteremo la specifica applicazione dell'algoritmo di Frank-Wolfe al rilassamento lineare del set-covering.

2.3 Rilassamento Lagrangiano

Nel corso del primo capitolo abbiamo affrontato l'argomento del rilassamento Lagrangiano, relativamente ai problemi di programmazione lineare. In questa sezione possiamo sfruttare i ragionamenti che abbiamo fatto per applicare questo concetto alla formulazione (2.7) del rilassamento lineare del set-covering. Rimuoviamo i vincoli di copertura per gli elementi in \mathcal{I} e aggiungiamo la penalizzazione Lagrangiana alla funzione obiettivo: introduciamo un vettore $\mathbf{u} = [u_1, \dots, u_m]^T \in \mathbb{R}^m$ di moltiplicatori di Lagrange da utilizzare, in combinazione con i vincoli che abbiamo rimosso, per costruire un contributo da aggiungere alla funzione obiettivo che ne peggiora il valore per tutte le soluzioni che non soddisfano i vincoli che abbiamo eliminato. In questo modo, otteniamo la nuova funzione obiettivo

$$\min \sum_{j=1}^n x_j + \sum_{i=1}^m \left[u_i \left(1 - \sum_{j=1}^n a_{ij} x_j \right) \right] = \min \sum_{j=1}^n \left[x_j \left(1 - \sum_{i=1}^m a_{ij} u_i \right) \right] + \sum_{i=1}^m u_i \quad (2.11)$$

e possiamo definire il rilassamento Lagrangiano

$$\mathcal{L}(\mathbf{u}): \begin{cases} \min \sum_{j=1}^n \left[x_j \left(1 - \sum_{i=1}^m a_{ij} u_i \right) \right] + \sum_{i=1}^m u_i \\ x_j \in [0, 1] \quad \forall j: 1 \leq j \leq n \end{cases} \quad (2.12)$$

che può essere risolto scegliendo $x_j = 1$ se e solo se

$$\sum_{i=1}^m a_{ij} u_i > 1 \quad (2.13)$$

per ogni $1 \leq j \leq n$ e che fornisce, al variare di $\mathbf{u} \in \mathbb{R}^n$, un limite inferiore al valore della funzione obiettivo. A questo punto, possiamo definire il problema duale Lagrangiano

$$\max_{\mathbf{u} \geq 0} \mathcal{L}(\mathbf{u}) \quad (2.14)$$

con l'obiettivo di trovare la limitazione inferiore migliore, cioè quella più alta possibile. Inoltre, si può dimostrare che il problema duale lagrangiano è un problema di ottimizzazione convessa, poiché la regione ammissibile è un insieme convesso e la funzione obiettivo è concava, anche se non necessariamente differenziabile in tutti i punti in cui è definita.

Utilizzando la notazione con matrici e vettori, possiamo riscrivere la formulazione (2.12) nella forma compatta

$$\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x} \in [0,1]^n} \{ \mathbf{c}^T \mathbf{x} + \mathbf{u}^T (\mathbf{b} - \mathbf{A} \mathbf{x}) \} \quad (2.15)$$

ricordando che $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{m \times n}$, con $\mathbf{x} = [x_1, \dots, x_n]^T \in [0, 1]^n$, $\mathbf{c} = [1, 1, \dots, 1]^T \in \mathbb{R}^n$ e $\mathbf{b} = [1, 1, \dots, 1]^T \in \mathbb{R}^m$.

Le considerazioni fatte in 1.3.5, insieme a quelle presentate dopo aver definito il duale di \mathcal{P} , ci permettono di concludere che una soluzione ottima $\mathbf{u}^* \in \mathbb{R}^m$ del problema duale Lagrangiano si può costruire vincolando i moltiplicatori ad assumere valori nell'intervallo $[0, 1]$, ossia $\mathbf{u}^* \in [0, 1]^m$, che è un insieme convesso e compatto.

Infine, presentiamo una proprietà che sarà utile per l'implementazione dell'algoritmo risolutivo. Supponiamo che \mathbf{x}^* sia una soluzione ottima di $\mathcal{L}(\hat{\mathbf{u}})$, per qualche $\hat{\mathbf{u}} \in \mathbb{R}^m$. Allora il subgradiente di $\mathcal{L}(\mathbf{u})$ in $\hat{\mathbf{u}}$ vale $\mathbf{s}_{\hat{\mathbf{u}}} = (\mathbf{b} - \mathbf{A} \mathbf{x}^*) \in \mathbb{R}^m$.

2.4 Applicazione dell'algoritmo di Frank-Wolfe

A questo punto possiamo sfruttare quanto osservato fino a questo punto per applicare l'algoritmo di Frank-Wolfe al problema duale Lagrangiano,

$$\max_{\mathbf{u} \in [0,1]^m} \mathcal{L}(\mathbf{u}) \quad (2.16)$$

di \mathcal{P} , visto che il dominio $[0, 1]^m$ è compatto e convesso e la funzione obiettivo $\mathcal{L}(\mathbf{u})$ è concava, anche se non necessariamente differenziabile in ogni punto. I passaggi sono gli stessi che abbiamo presentato in 1.5, da cui si deriva la specifica dell'algoritmo riportata di seguito.

Algoritmo di Frank-Wolfe per il problema duale Lagrangiano di \mathcal{P}

```

1  $\mathbf{u}_0 \in [0, 1]^m$  punto di partenza                                /* Inizializzazione */
2 while true do
3    $\mathbf{x}_k = \arg \min_{\mathbf{x} \in [0, 1]^n} \{\mathbf{c}^T \mathbf{x} + \mathbf{u}_k^T (\mathbf{b} - \mathbf{A} \mathbf{x})\}$           /* Soluzione di  $\mathcal{L}(\mathbf{u}_k)$  */
4    $\mathbf{s}_k = \mathbf{b} - \mathbf{A} \mathbf{x}_k$                                           /* Subgradiente in  $\mathbf{u}_k$  */
5    $\mathbf{d}_k = \arg \max_{\mathbf{d} \in [0, 1]^m} \{\mathbf{s}_k^T \mathbf{d}\}$                                 /* Passo 1 */
6    $\mathbf{u}_{k+1} = (1 - \gamma) \mathbf{u}_k + \gamma \mathbf{d}_k$  con  $\gamma = \frac{2}{k+2}$           /* Passo 2 */
```

Osserviamo che, ad ogni iterazione, il valore

$$\alpha_k = \mathbf{c}^T \mathbf{x}_k + \mathbf{u}^T (\mathbf{b} - \mathbf{A} \mathbf{x}_k) = \mathcal{L}(\mathbf{u}_k), \quad \mathbf{x}_k = \arg \min_{\mathbf{x} \in [0, 1]^n} \mathcal{L}(\mathbf{u}_k), \quad (2.17)$$

fornisce una limitazione inferiore al valore ottimo e il valore

$$\omega_k = \mathcal{L}(\mathbf{u}_k) + \mathbf{s}_k^T (\mathbf{d}_k - \mathbf{u}_k) \quad (2.18)$$

rappresenta un limite superiore al valore ottimo. Non c'è nessuna garanzia che le due limitazioni migliorino monotonicamente ad ogni iterazione, ma è sufficiente tenere traccia delle limitazioni migliori, α^* e ω^* , e terminare l'algoritmo quando la differenza è sufficientemente piccola. A quel punto avremo che

$$\alpha^* \leq \mathcal{L}(\mathbf{u}^*) \leq \omega^* \quad (2.19)$$

con \mathbf{u}^* che è una soluzione ottima per il problema duale Lagrangiano di \mathcal{P} . In questo modo otteniamo un'approssimazione al valore ottimo di \mathcal{P} .

In alcuni casi potrebbe capitare che la differenza desiderata tra le due limitazioni non viene mai raggiunta e quindi, per evitare che l'algoritmo continui indefinitamente, si imposta un limite al numero delle iterazioni da eseguire.

2.5 Matrice di riferimento

In questo capitolo siamo partiti dalla specifica del problema del set-covering per ottenere la sua formulazione matematica come problema di programmazione lineare intera. Successivamente abbiamo definito il rilassamento lineare e ci siamo soffermati ad analizzare

alcuni aspetti che lo riguardano, concludendo con la definizione del problema duale e del rilassamento lagrangiano. L'ottenimento della formulazione (2.7), a partire da quella presentata in (2.5), ci ha permesso di ottenere una formulazione in forma canonica per il rilassamento lineare del set-covering. A questo punto, siamo finalmente pronti a capire la convenienza pratica di tale formulazione. È infatti immediato osservare che descrivere il problema come in (2.7) ci permette di caratterizzare una specifica istanza semplicemente utilizzando la matrice binaria $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{m \times n}$, evitando di dover specificare gli insiemi \mathcal{I} e \mathcal{F} . Di conseguenza, per la creazione delle istanze da utilizzare nelle prove sperimentali, ci limiteremo a generare matrici binarie di varie dimensioni e con diversi valori di sparsità.

2.5.1 Rappresentazione CSR

Solitamente le istanze per il problema del set-covering hanno matrici di riferimento molto sparse, cioè matrici con molti più zeri che uni. Per questo motivo, può essere conveniente cercare un modo intelligente di rappresentare la matrice di riferimento, che non richieda di salvare in memoria i valori di tutti gli elementi. In particolare, visto che la matrice è binaria, è sufficiente salvare solamente le informazioni relative agli elementi non nulli. In questo modo possiamo risparmiare molta memoria e provare a sfruttare una rappresentazione compatta per ridurre il numero delle operazioni da eseguire, evitando quelle che coinvolgono elementi nulli.

La rappresentazione CSR (*Compressed Sparse Row*) permette di rappresentare una matrice sparsa $\mathbf{A} \in \mathbb{R}^{m \times n}$, con `nnz` elementi non nulli, utilizzando tre vettori che chiameremo `values`, `indices` e `pointers`. I vettori `values` e `indices`, di lunghezza `nnz`, memorizzano i valori degli elementi non nulli e i loro indici di colonna, rispettivamente. Il vettore `pointers`, di lunghezza $m + 1$, memorizza in ogni posizione l'indice che segnala l'inizio della riga corrispondente nei vettori `values` e `pointers`. In altre parole, il valore dell'elemento `pointers[i]` è l'indice dei vettori `values` e `indices` che rappresenta l'inizio della riga `i` della matrice. Di conseguenza, per riferirci agli elementi della riga `i` è sufficiente considerare i vettori `values` e `indices` nell'intervallo di indici delimitato da `pointers[i]` e `pointers[i + 1] - 1`.

Questa rappresentazione permette di memorizzare una matrice binaria di m righe e n colonne in uno spazio molto ridotto. Inoltre, visto che nel nostro caso la matrice da memorizzare è binaria, non è nemmeno necessario utilizzare il vettore `values`, poiché sappiamo che i valori non nulli sono tutti 1.

Per l'implementazione dell'algoritmo risolutivo, che segue i passaggi presentati in 2.4, memorizzeremo le rappresentazioni CSR della matrice binaria di riferimento e della sua trasposta, con l'obiettivo di sfruttare la cache per ottimizzare l'accesso in memoria.

3

Implementazione

L'obiettivo di questo capitolo è sviluppare l'implementazione di un algoritmo risolutivo per il rilassamento lineare del set-covering, basato sul metodo di Frank-Wolfe. Inoltre, presenteremo anche il codice necessario a risolvere le istanze del problema con l'algoritmo del simplesso, che verrà usato per operare un confronto nel prossimo capitolo.

3.1 Generazione delle istanze

Come anticipato in 2.5, la formulazione (2.7) ci permette di identificare un'istanza per il rilassamento lineare del set-covering semplicemente utilizzando una matrice binaria di riferimento. Di seguito è riportato il codice che genera matrici binarie di riferimento.

 datagen.py

```
1 import numpy as np
2
3 def generate_matrix(rows, cols, sparsity):
4     nnz = int(round((1 - sparsity) * rows * cols))
5     matrix = np.zeros((rows, cols), dtype=int)
6
7     for r in range(rows):
8         matrix[r, np.random.choice(cols)] = 1
9         nc = np.where(matrix.sum(axis=0) == 0)[0]
10        for c in nc:
11            matrix[np.random.choice(rows), c] = 1
12
13        remaining_nnz = nnz - np.sum(matrix)
14        if remaining_nnz > 0:
15            zeros = np.argwhere(matrix == 0)
16            selected_indices = zeros[
17                np.random.choice(zeros.shape[0], remaining_nnz, False)
18            ]
19            for r, c in selected_indices:
20                matrix[r, c] = 1
21
22        return csr_matrix(matrix)
```

Listing 3.1: Generazione delle matrici binarie di riferimento.

La funzione appena presentata sfrutta gli strumenti della libreria numpy per generare una matrice binaria con un numero di righe e di colonne che è specificato dai parametri `rows` e `cols`, rispettivamente, in cui sparsità è determinata dal parametro `sparsity`.

Nel processo di generazione della matrice, vengono eseguite delle operazioni per garantire che non ci siano righe o colonne interamente popolate da valori nulli. Non vogliamo che ci siano righe nulle, poiché significherebbe avere elementi in \mathcal{I} che non sono coperti da nessuno dei sottoinsiemi in \mathcal{F} . Non vogliamo nemmeno che ci siano colonne nulle, poiché significherebbe avere sottoinsiemi in \mathcal{F} che non contengono nessun elemento di \mathcal{I} .

3.1.1 Gestione dataset

Adesso che abbiamo un modo di generare le istanze del problema, possiamo usarlo per creare dei dataset. L'idea è quella di creare molteplici istanze dello stesso tipo, relativamente a dimensione e sparsità della matrice di riferimento, per ottenere dei risultati più accurati.

Per ciascuna istanza di un dataset viene generato il file `csr_matrix.dat` che memorizza la rappresentazione CSR della matrice di riferimento e della sua trasposta, insieme alle informazioni riguardo il numero di righe, il numero di colonne e il numero degli elementi non nulli. Tale file verrà utilizzato come parametro di input per l'algoritmo del simplesso e l'algoritmo di Frank-Wolfe. Per ottenere la rappresentazione CSR delle matrici binarie di riferimento delle istanze abbiamo utilizzato il modulo sparse della libreria scipy.

3.2 Algoritmo del simplesso

Iniziamo a presentare il codice necessario a risolvere le istanze con l'algoritmo del simplesso. Utilizzeremo l'interfaccia pycipopt per accedere all'implementazione SCIP dell'algoritmo del simplesso. Le componenti necessarie sono quelle importate di seguito.

```
from pycipopt import Model, quicksum
```

3.2.1 Costruzione del modello

Per costruire il modello associato al rilassamento lineare del set-covering identificato dalla matrice A con m righe e n colonne, definiamo

```
model = Model("set-covering linear relaxation")
```

che ci permette di introdurre le variabili

```
x = [model.addVar(name=f"x_{j + 1}", vtype="C", lb=0) for j in range(n)]
```

e la funzione obiettivo


```
model.setObjective(quicksum(x[j] for j in range(n)), sense="minimize")
```

da minimizzare. A questo punto possiamo specificare i vincoli di copertura

```
for i in range(m):
    model.addCons(quicksum(A[i][j] * x[j] for j in range(n)) >= 1)
```

in accordo con (2.6).

Mettendo insieme tutti i pezzi, otteniamo la funzione riportata di seguito.

 solver.py


```
1 def build_model(A, m, n):
2     model = Model("set-covering linear relaxation")
3     x = [model.addVar(name=f"x_{j + 1}", vtype="C", lb=0) for j in range(n)]
4     model.setObjective(quicksum(x[j] for j in range(n)))
5     for i in range(m):
6         model.addCons(quicksum(A[i][j] * x[j] for j in range(n)) >= 1)
7     return model
```

Listing 3.2: Costruzione del modello per l'algoritmo del simplesso.

Osserviamo che, in accordo con la formulazione (2.7) e con le considerazioni che abbiamo fatto, le variabili del problema sono vincolate ad essere semplicemente continue non negative. Non c'è necessità di specificare i limiti superiori per le variabili, poiché la funzione obiettivo e i vincoli li impongono implicitamente.

3.2.2 Soluzione del modello

Per risolvere il modello è sufficiente utilizzare la funzione `optimize()` sul modello che abbiamo ottenuto in 3.2.1 con la funzione `build_model()`. Possiamo allora definire la funzione

 solver.py

```
1 def solve(model):
2     model.optimize()
```

Listing 3.3: Soluzione del modello con l'algoritmo del simplesso.

che ottimizza il modello ricevuto come parametro. Dopo la risoluzione, possiamo ottenere il valore ottimo e la soluzione che lo realizza. A questo punto abbiamo tutti gli strumenti necessari per risolvere il rilassamento lineare del set-covering utilizzando l'algoritmo del simplesso.

3.3 Algoritmo di Frank-Wolfe

Procediamo ora con l'implementazione dell'algoritmo risolutivo basato su Frank-Wolfe. Utilizzeremo il linguaggio C e, con l'obiettivo di creare un algoritmo efficiente, eviteremo l'allocazione dinamica della memoria. Di conseguenza, tutti i dati saranno memorizzati nello stack. Inoltre, per semplificare l'implementazione, utilizzeremo un unico file sorgente che raggruppa le funzioni necessarie per la realizzazione dell'algoritmo. In questo modo possiamo anche utilizzare delle variabili globali che siano accessibili in tutte le funzioni, senza il bisogno di includerle o di specificarle ogni volta come parametri. Infine, vista la necessità di lavorare con matrici e vettori e la scelta di utilizzare solo lo stack, dovremo specificare i valori di ritorno delle funzioni come puntatori passati in ingresso.

Iniziamo definendo la rappresentazione CSR della matrice di riferimento

```
#define MAX_ROWS 10000
#define MAX_COLS 10000
#define MAX_SIZE MAX_ROWS * MAX_COLS

typedef struct {
    int indices[MAX_SIZE];
    int pointers[MAX_ROWS + 1];
} csr_matrix;

csr_matrix A, T;
```

dove `indices` e `pointers` assumono i significati che gli abbiamo dato in 2.5.1 quando abbiamo introdotto la rappresentazione CSR, con `A` e `T` che si riferiscono alla rappresentazione CSR della matrice di riferimento e della sua trasposta, rispettivamente. La scelta di utilizzare solamente lo stack in combinazione con le variabili globali ci ha obbligato a definire le dimensioni dei vettori a tempo di compilazione. Lo svantaggio naturalmente è il fatto di dover allocare una quantità fissata di memoria massima, che in tante situazioni sarà di molto superiore a quella effettivamente necessaria. Successivamente, introduciamo le variabili globali

```
int m = MAX_ROWS, n = MAX_COLS;
int nnz = MAX_SIZE;
```

che rappresentano rispettivamente il numero delle righe, il numero delle colonne e il numero degli elementi non nulli della matrice di riferimento. Infine, definiamo le funzioni

```
int row_start(const csr_matrix * const matrix, int row) {
    return matrix->pointers[row];
}

int row_end(const csr_matrix * const matrix, int row) {
    return matrix->pointers[row + 1];
}
```

che ci permettono di navigare all'interno delle righe della matrice utilizzando la sua rappresentazione CSR.

A questo punto siamo pronti per presentare l'implementazione della specifica dell'algoritmo di Frank-Wolfe, in accordo con quanto discusso in 2.4.

3.3.1 Scelta del punto di partenza

Riprendendo la specifica presentata in 2.4, possiamo ora implementare la funzione che si occupa di calcolare il punto di partenza, che può essere un punto $\mathbf{u}_0 \in [0, 1]^m$ arbitrario.

 solver.c

```
1 void starting_u(double * const u) {
2     for (int i = 0; i < m; i++) {
3         u[i] = 1;
4     }
5 }
```

Listing 3.4: Calcolo del punto di partenza.

In questo caso abbiamo scelto per semplicità di inizializzare tutte le componenti di \mathbf{u}_0 a 1, ma ogni altra scelta sarebbe stata ugualmente valida.

3.3.2 Soluzione del rilassamento Lagrangiano

Riprendiamo la specifica del rilassamento Lagrangiano

$$\mathcal{L}(\mathbf{u}): \begin{cases} \min & \sum_{j=1}^n \left[x_j \left(1 - \sum_{i=1}^m a_{ij} u_i \right) \right] + \sum_{i=1}^m u_i \\ & x_j \in [0, 1] \quad \forall j: 1 \leq j \leq n \end{cases} \quad (2.12)$$

associato al problema \mathcal{P} definito in (2.7), con $\mathbf{u} = [u_1, \dots, u_m] \in \mathbb{R}^m$. Abbiamo già argomentato che per risolverlo è sufficiente assegnare $x_j = 1$ se e solo se

$$\sum_{i=1}^m a_{ij} u_i > 1 \quad \forall j: 1 \leq j \leq n, \quad (3.1)$$

dove la sommatoria rappresenta il prodotto di \mathbf{u} con la colonna j -esima della matrice di riferimento. Equivalentemente, lo stesso prodotto si può ottenere moltiplicando \mathbf{u} con la j -esima riga della trasposta della matrice di riferimento e può essere calcolato con il codice

```
double result = 0;
for (int i = row_start(&T, j); j < row_end(&T, j); i++) {
    result += u[T.indices[i]];
}
```

che permette di assegnare il valore a x_j , in accordo con quanto detto, ponendo

```
x[j] = (result > 1)
```

In particolare, abbiamo sfruttato la rappresentazione CSR della matrice T (trasposta della matrice A di riferimento) per calcolare il prodotto considerando solamente le componenti di \mathbf{u} che vengono moltiplicate per valori non nulli della riga j -esima di T . In questo modo tutti gli elementi nulli della riga j -esima non vengono mai acceduti e non contribuiscono alla computazione del prodotto. Naturalmente, la convenienza di questo procedimento è tanto maggiore tanto più la matrice è sparsa.

Il valore della funzione obiettivo è ottenuto a partire da due contributi. Il primo dipende dalla scelta dei valori per le variabili x_j e il secondo è la somma delle componenti di \mathbf{u} . Quest'ultimo può essere semplicemente calcolato utilizzando il codice

```
double objective = 0;
for (int i = 0; i < m; i++) {
    objective += u[i];
}
```

mentre per il primo è sufficiente sommare i coefficienti delle variabili x_j non nulle, come mostrato di seguito.

```
if (x[j]) {
    objective += (1 - result);
}
```

Mettendo insieme tutti i pezzi, otteniamo la funzione riportata sotto, che calcola e restituisce il valore della funzione obiettivo del rilassamento Lagrangiano $\mathcal{L}(\mathbf{u})$, nel punto \mathbf{u} passato come parametro.

 solver.c

```

1  double L(const double * const u, uint8_t * const x) {
2      double objective = 0;
3
4      for (int j = 0; j < n; j++) {
5          double result = 0;
6          for (int i = row_start(&T, j); i < row_end(&T, j); i++) {
7              result += u[T.indices[i]];
8          }
9          if (x[j] = (result > 1)) {
10             objective += (1 - result);
11         }
12     }
13
14     for (int i = 0; i < m; i++) {
15         objective += u[i];
16     }
17
18     return objective;
19 }
```

Listing 3.5: Soluzione del rilassamento Lagrangiano.

In questo caso abbiamo aggregato l'assegnazione del valore alla variabile x_j e il controllo che determina se il suo coefficiente va aggiunto come contributo a **objective**. Infine, il parametro \mathbf{x} in ingresso che rappresenta le variabili è in realtà un valore di ritorno che viene calcolato nella funzione e utilizzato dal chiamante per calcolare il subgradiente.

3.3.3 Calcolo del subgradiente

Consideriamo il problema duale Lagrangiano

$$\max_{\mathbf{u} \geq 0} \mathcal{L}(\mathbf{u}) \quad (2.14)$$

definito in 2.3. Procediamo ora a calcolare il subgradiente di $\mathcal{L}(\mathbf{u})$ in un punto $\hat{\mathbf{u}} \in \mathbb{R}^m$ per ottenere un'approssimazione lineare della funzione obiettivo $\mathcal{L}(\mathbf{u})$, che in questo caso è conveniente assumere nella forma

$$\mathcal{L}(\mathbf{u}) = \min_{\mathbf{x} \in [0,1]^n} \{ \mathbf{c}^T \mathbf{x} + \mathbf{u}^T (\mathbf{b} - \mathbf{A} \mathbf{x}) \}, \quad (2.15)$$

dove $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{m \times n}$, con $\mathbf{x} = [x_1, \dots, x_n]^T \in [0, 1]^n$, $\mathbf{c} = [1, 1, \dots, 1]^T \in \mathbb{R}^n$ e $\mathbf{b} = [1, 1, \dots, 1]^T \in \mathbb{R}^m$. Sappiamo che se \mathbf{x}^* è una soluzione ottima di $\mathcal{L}(\hat{\mathbf{u}})$, per qualche $\hat{\mathbf{u}} \in \mathbb{R}^m$, allora il subgradiente di $\mathcal{L}(\mathbf{u})$ in $\hat{\mathbf{u}}$ vale $\mathbf{s}_{\hat{\mathbf{u}}} = (\mathbf{b} - \mathbf{A}\mathbf{x}^*) \in \mathbb{R}^m$. Ed allora, la componente i -esima di $\mathbf{s}_{\hat{\mathbf{u}}}$ risulta

$$\mathbf{s}_{\hat{\mathbf{u}},i} = 1 - \sum_{j=1}^n a_{ij} x_j^*, \quad (3.2)$$

con la sommatoria che rappresenta il prodotto della riga i -esima della matrice di riferimento con il vettore \mathbf{x}^* . Di conseguenza, come abbiamo fatto in precedenza, possiamo calcolare tale prodotto con il codice

```
double result = 0;
for (int j = row_start(&A, i); j < row_end(&A, i); j++) {
    result += x[A.indices[j]];
}
```

da cui, utilizzando la (3.2), deriva immediatamente la funzione seguente

```
1 void subgradient(const uint8_t * const x, int * const s) {
2     for (int i = 0; i < m; i++) {
3         s[i] = 1;
4         for (int j = row_start(&A, i); j < row_end(&A, i); j++) {
5             s[i] -= x[A.indices[j]];
6         }
7     }
8 }
```

 solver.c

Listing 3.6: Calcolo del subgradiente \mathbf{s}_k di $\mathcal{L}(\mathbf{u})$ in \mathbf{u}_k .

che calcola le componenti del subgradiente \mathbf{s} di $\mathcal{L}(\mathbf{u})$ in \mathbf{u} . Anche in questo caso, il parametro \mathbf{s} è in realtà un valore di ritorno che viene utilizzato dal chiamante per il calcolo della soluzione che massimizza l'approssimazione lineare di $\mathcal{L}(\mathbf{u})$ ottenuta da \mathbf{s} .

3.3.4 Calcolo della soluzione che massimizza l'approssimazione lineare

A questo punto possiamo utilizzare il subgradiente appena calcolato per ottenere un'approssimazione lineare della funzione obiettivo $\mathcal{L}(\mathbf{u})$ del duale Lagrangiano di \mathcal{P} .

Sia $f: \mathcal{D} \subset \mathbb{R}^n \rightarrow \mathbb{R}$ una generica funzione, che assumiamo differenziabile nei punti interni di \mathcal{D} . Allora rimane definita l'approssimazione lineare del primo ordine

$$f(\mathbf{x}) \simeq f(\hat{\mathbf{x}}) + \nabla f(\hat{\mathbf{x}})(\mathbf{x} - \hat{\mathbf{x}}) \quad (3.3)$$

nelle vicinanze di un punto $\hat{\mathbf{x}}$. Nel caso di una funzione concava, tale approssimazione limita f dall'alto. Di conseguenza, per il nostro problema duale Lagrangiano, il calcolo del subgradiente ci permette di generalizzare questo concetto e ottenere l'approssimazione lineare $\mathcal{L}(\mathbf{u}) \simeq \mathcal{L}(\hat{\mathbf{u}}) + \mathbf{s}_{\hat{\mathbf{u}}}(\mathbf{u} - \hat{\mathbf{u}})$ intorno a $\hat{\mathbf{u}}$, che assume il suo valore massimo in

$$\mathbf{d}^* = \arg \max_{\mathbf{d} \in [0,1]^m} \{\mathbf{s}_{\hat{\mathbf{u}}} \mathbf{d}\}, \quad (3.4)$$

con \mathbf{d}^* che può essere calcolato ponendo $d_i^* = 1$ se e solo se $\mathbf{s}_{\hat{\mathbf{u}},i} > 0$, per ogni $1 \leq i \leq m$, da cui segue il codice della funzione seguente.



```

1 void argmax(const int * const s, uint8_t * const d) {
2     for (int i = 0; i < m; i++) {
3         d[i] = (s[i] > 0);
4     }
5 }
```

Listing 3.7: Calcolo di $\mathbf{d}_k = \arg \max_{\mathbf{d} \in [0,1]^m} \{\mathbf{s}_k \mathbf{d}\}$.

Naturalmente, il parametro \mathbf{s} rappresenta il subgradiente di $\mathcal{L}(\mathbf{u})$ nel punto \mathbf{u}_k corrente.

3.3.5 Calcolo del punto successivo

Procediamo ora a calcolare il punto successivo utilizzando quanto discusso in 2.4, che poneva $\mathbf{u}_{k+1} = (1 - \gamma)\mathbf{u}_k + \gamma\mathbf{d}_k = \mathbf{u}_k + \gamma(\mathbf{d}_k - \mathbf{u}_k)$, da cui segue il codice seguente.



```

1 void next_u(double * const u, const uint8_t * const d, double gamma) {
2     for (int i = 0; i < m; i++) {
3         u[i] += gamma * (d[i] - u[i]);
4     }
5 }
```

Listing 3.8: Calcolo di \mathbf{u}_{k+1} .

3.3.6 Calcolo delle limitazioni al valore della funzione obiettivo

Rimane ora da implementare il codice per tenere traccia delle migliori limitazioni al valore della funzione obiettivo.

Limitazione Inferiore (*Lower Bound*)

Dalle considerazioni fatte nella parte finale di 2.4 si è concluso che, ad ogni iterazione, il valore $\mathcal{L}(\mathbf{u}_k)$ è una limitazione inferiore al valore della funzione obiettivo. Di conseguenza, ad ogni iterazione è sufficiente confrontare il valore ottenuto dalla risoluzione del rilassamento Lagrangiano con la limitazione inferiore corrente, ed eventualmente aggiornare quest'ultima nel caso in cui si sia trovata una limitazione migliore.

 solver.c

```

1 void update_lower_bound(double * const lower_bound, double candidate) {
2     if (candidate > *lower_bound) {
3         *lower_bound = candidate;
4     }
5 }

```

Listing 3.9: Aggiornamento della limitazione inferiore.

Limitazione Superiore (*Upper Bound*)

Dalle considerazioni fatte in 2.4 si è concluso che il valore $\mathcal{L}(\mathbf{u}_k) + \mathbf{s}_k(\mathbf{d}_k - \mathbf{u}_k)$ costituisce una limitazione superiore al valore della funzione obiettivo. Il codice della funzione che aggiorna la limitazione superiore segue immediatamente ed è riportato di seguito.

 solver.c

```

1 void update_upper_bound(double * const upper_bound, double objective,
2     const int * const s, const uint8_t * const d, const double * const u
3 ) {
4     double candidate = objective;
5     for (int i = 0; i < m; i++) {
6         candidate += s[i] * (d[i] - u[i]);
7     }
8
9     if (candidate < *upper_bound) {
10         *upper_bound = candidate;
11     }
12 }

```

Listing 3.10: Aggiornamento della limitazione superiore.

3.3.7 Composizione dell'algoritmo risolutivo

Finalmente possiamo mettere insieme tutti i pezzi e comporre l'algoritmo risolutivo.



```
1 Pls don't give up now
```

Listing 3.11: Implementazione dell'algoritmo risolutivo.

3.4 Esecuzione

3.4.1 Output

Questo capitolo ha l'obiettivo di discutere i risultati sperimentali ottenuti con l'esecuzione dell'algoritmo del simplesso e dell'algoritmo di Frank-Wolfe su istanze del rilassamento lineare del set-covering.

4.1 Prove sperimentali

Per valutare l'efficienza e l'applicabilità dell'algoritmo di Frank-Wolfe abbiamo effettuato varie prove sperimentali su numerose istanze, variando la dimensione e la sparsità della matrice di riferimento. Inoltre, per ogni coppia di valori che identificano dimensione e sparsità abbiamo ripetuto gli esperimenti su molteplici istanze, con l'obiettivo di ottenere risultati più accurati. Infine abbiamo calcolato la media e la deviazione standard di questi risultati, in modo da poterli riassumere utilizzando opportuni grafici.

Inizialmente abbiamo condotto degli esperimenti per studiare la qualità della soluzione prodotta dall'algoritmo di Frank-Wolfe, confrontandola con il valore ottimo ottenuto attraverso l'algoritmo del simplesso.

Successivamente abbiamo svolto delle prove per confrontare i tempi di esecuzione dei due algoritmi, con l'obiettivo di capire se è possibile utilizzare l'algoritmo di Frank-Wolfe per trovare una limitazione ragionevole al valore ottimo in un tempo molto ridotto. In questo caso la difficoltà è trovare il compromesso tra la qualità della soluzione e il tempo impiegato per determinarla. In particolare, dobbiamo evitare che il tempo impiegato dall'algoritmo di Frank-Wolfe per determinare una limitazione ragionevole superi quello impiegato dall'algoritmo del simplesso per trovare il valore ottimo. In altre parole, dobbiamo impostare un limite al numero delle iterazioni che ci permetta di eseguire l'algoritmo in un tempo molto ridotto e contemporaneamente di ottenere una buona limitazione al valore ottimo. Tuttavia, potrebbero esserci casi in cui l'algoritmo di Frank-Wolfe è completamente dominato dall'algoritmo del simplesso.

Nello svolgimento degli esperimenti abbiamo scelto valori di sparsità particolarmente alti, in accordo con il fatto che i problemi di set-covering sono caratterizzati da matrici generalmente molto sparse.

4.2 Qualità delle soluzioni

Procediamo ora con l'esecuzione dei due algoritmi per confrontare la soluzione prodotta dall'algoritmo di Frank-Wolfe con il valore ottimo trovato dall'algoritmo del semplice. In questo esperimento non ci preoccupiamo di limitare il numero delle iterazioni, e quindi il tempo di esecuzione di Frank-Wolfe, poiché l'obiettivo è solamente quello di determinare la qualità della soluzione prodotta dall'algoritmo di Frank-Wolfe.

4.2.1 Dimensione della matrice di riferimento

Iniziamo considerando istanze di dimensioni differenti, in quanto a numero di elementi della matrice di riferimento. In questo caso ci limiteremo ad analizzare gli aspetti che riguardano il numero di elementi della matrice di riferimento, ignorando tutti gli altri parametri, che verranno analizzati singolarmente nel seguito. I risultati sono riassunti dai grafici della Figura 4.1.

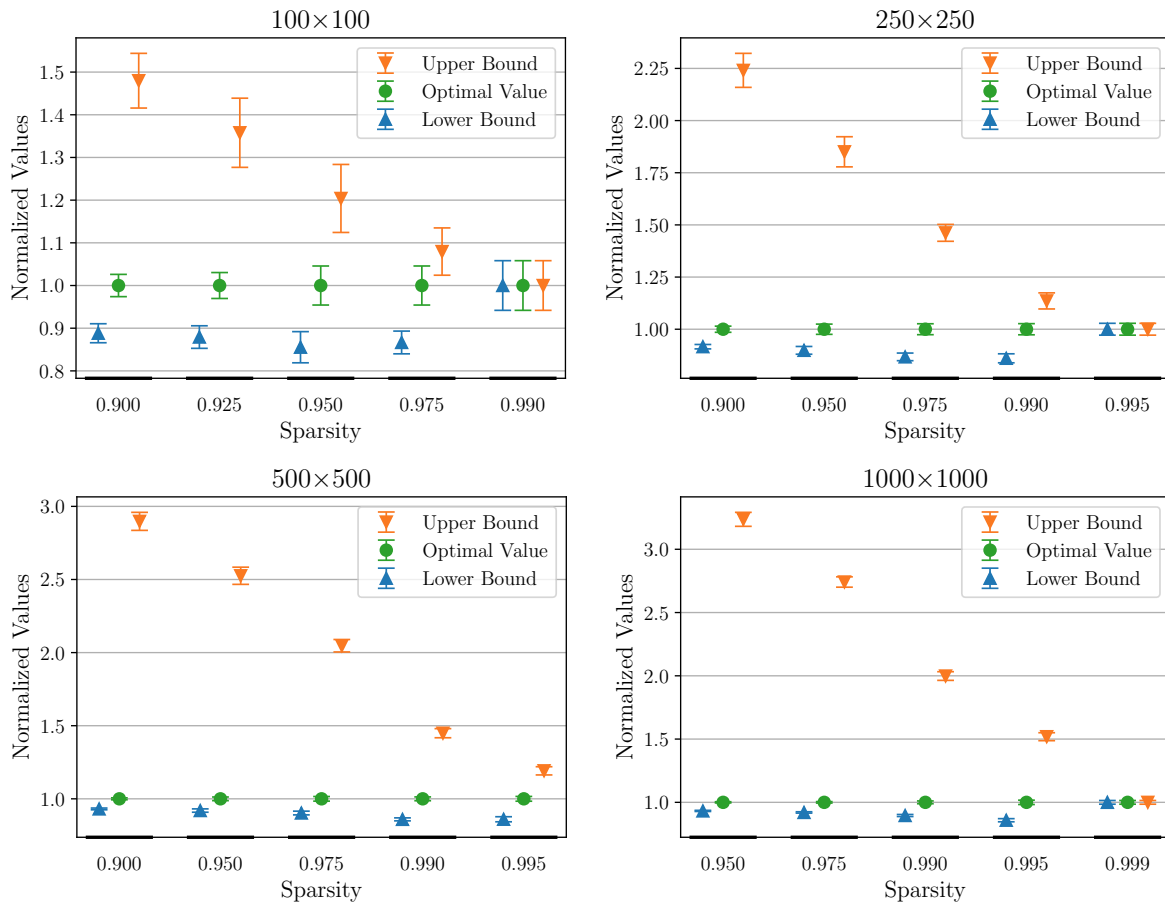


Figura 4.1: qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e del numero di elementi della matrice di riferimento delle istanze.

Osserviamo che per tutte le istanze che abbiamo scelto, avere un valore di sparsità sufficientemente elevato permette all'algoritmo di Frank-Wolfe di produrre delle buone limitazioni e, in alcuni casi, anche di chiudere sul valore ottimo fornito dall'algoritmo del simplesso. Quando i valori di sparsità sono inferiori, la situazione è molto diversa. In particolare, a parità di sparsità, l'aumentare del numero di elementi della matrice di riferimento è indicatore di un peggioramento nella qualità della limitazione superiore e di un miglioramento nella limitazione inferiore. Il peggioramento della limitazione superiore è più marcato rispetto al miglioramento della limitazione inferiore, che rimane invece molto più stabile. Per capire meglio quanto l'aumentare del numero di elementi della matrice di riferimento influenza le limitazioni prodotte da Frank-Wolfe, abbiamo riportato i valori precisi, ottenuti dallo svolgimento degli esperimenti, nelle tabelle che seguono.

Dimensione	Limitazione Inferiore	Limitazione Superiore
100×100	0.888	1.480
250×250	0.916	2.241
500×500	0.931	2.898

Tabella 4.1: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della dimensione delle matrici di riferimento di istanze con sparsità pari a 0.9.

Dimensione	Limitazione Inferiore	Limitazione Superiore
100×100	0.855	1.204
250×250	0.899	1.850
500×500	0.920	2.525
1000×1000	0.932	3.237

Tabella 4.2: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della dimensione delle matrici di riferimento di istanze con sparsità pari a 0.95.

Dimensione	Limitazione Inferiore	Limitazione Superiore
100×100	0.867	1.080
250×250	0.867	1.462
500×500	0.903	2.047
1000×1000	0.920	2.741

Tabella 4.3: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della dimensione delle matrici di riferimento di istanze con sparsità pari a 0.975.

Il comportamento della limitazione inferiore si inverte quando i valori di sparsità sono molto elevati. In questi casi, avere matrici più piccole permette di ottenere limitazioni inferiori di qualità migliore e talvolta di chiudere sul valore ottimo, come evidenziato in precedenza.

4.2.2 Forma della matrice di riferimento

Adesso che abbiamo capito l'influenza della dimensione delle matrici di riferimento sulla qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe, possiamo ripetere gli esperimenti per capire l'impatto della forma della matrice di riferimento. In particolare, vogliamo capire cosa succede quando il numero delle righe e delle colonne è sbilanciato da una parte o dall'altra, come accade nella maggior parte delle istanze di problemi di set-covering.

In primo luogo consideriamo matrici di forma differente ma caratterizzate dallo stesso numero di elementi.

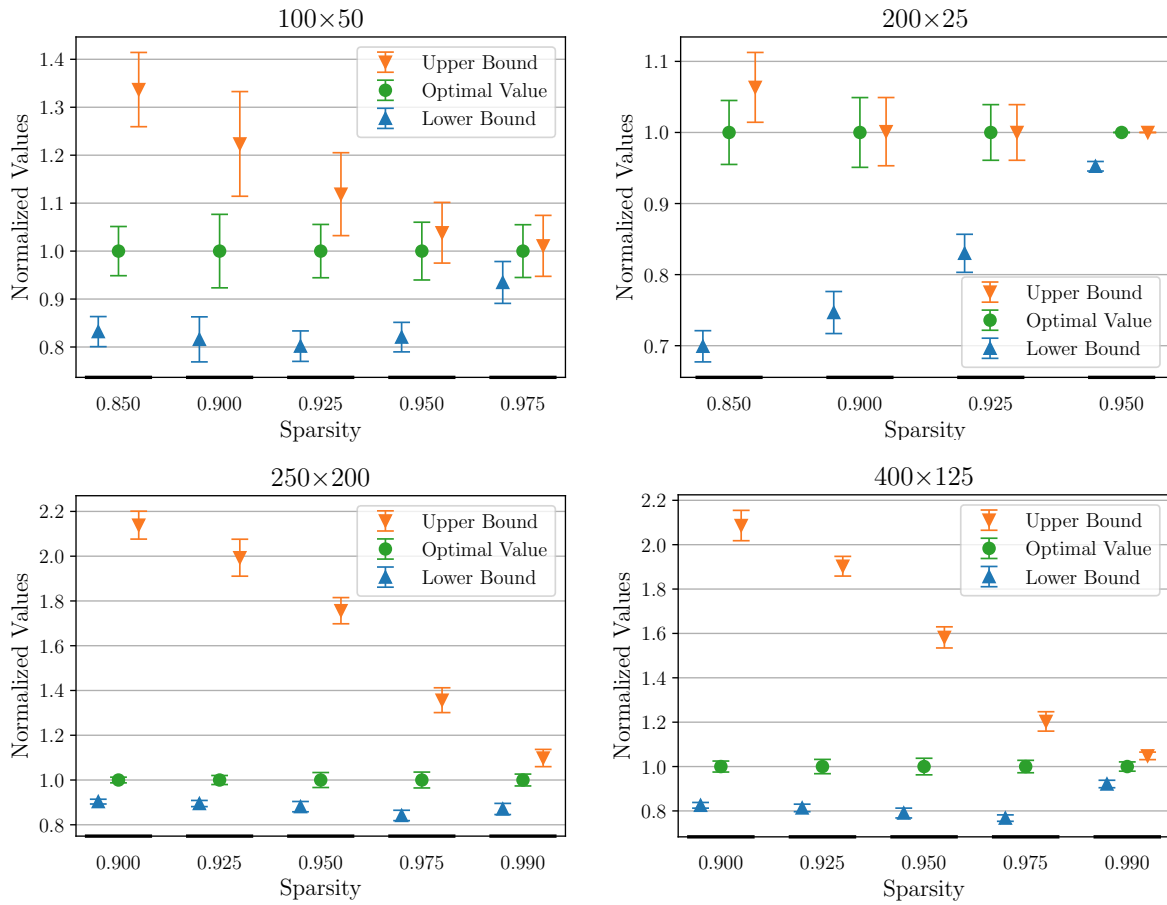


Figura 4.2: qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze piccole e medie.

I grafici della figura 4.2 mostrano, in ciascuna riga, i risultati ottenuti sperimentalmente dall'esecuzione dell'algoritmo di Frank-Wolfe su istanze con matrici che hanno lo stesso numero di elementi ma forma differente. Prese singolarmente, le istanze si comportano come discusso in precedenza: l'aumento della sparsità è indicatore di un miglioramento della limitazione superiore ma non di quella inferiore, che ha invece un comportamento più irregolare.

Osserviamo che, a parità di sparsità, la forma della matrice influenza i risultati ottenuti. In particolare, quando il numero degli elementi rimane costante si ottengono le limitazioni superiori migliori per le istanze che hanno un numero di variabili inferiore. Questo risultato è tanto più marcato tanto più sono piccole le istanze che si considerano, come evidenziato dai risultati della figura 4.3, relativi a matrici di dimensione maggiore.

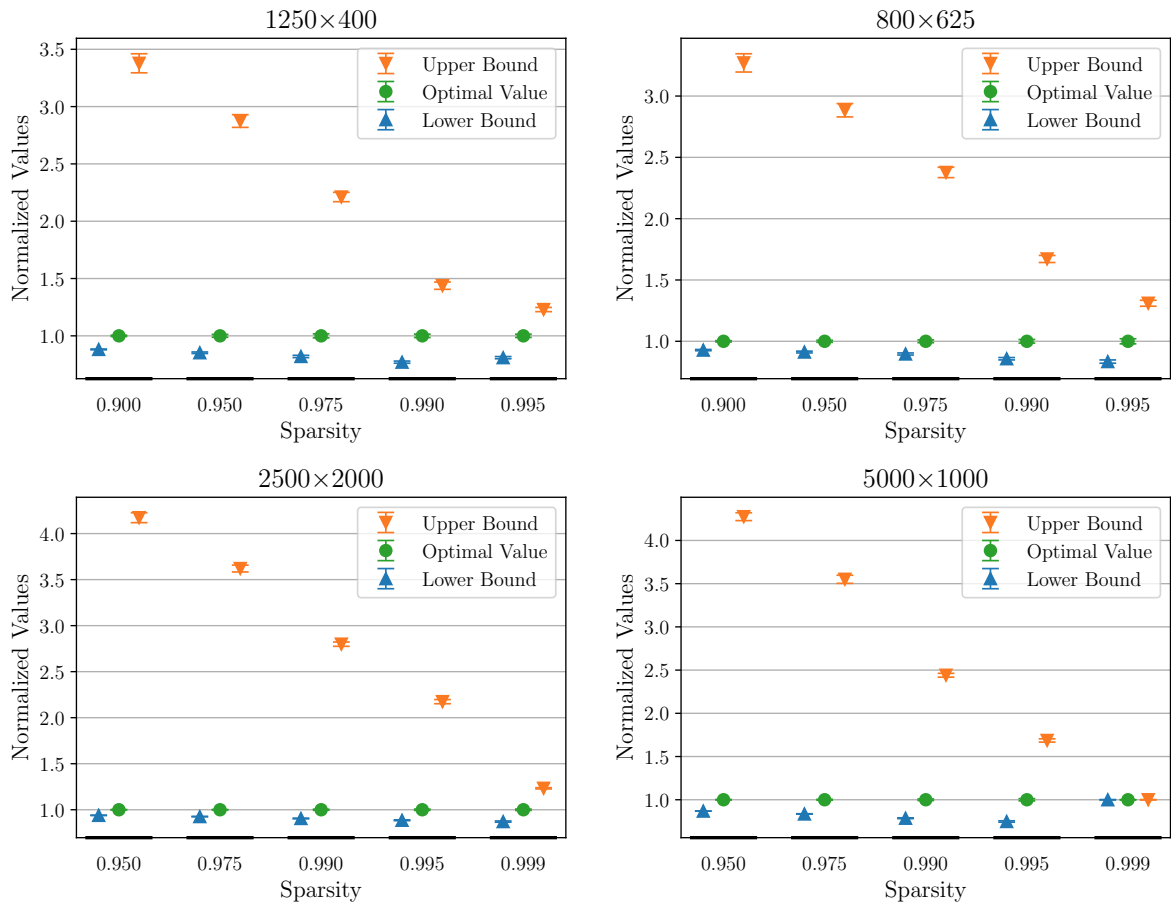


Figura 4.3: qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze piccole e medie.

Notiamo in questo caso che il comportamento è sempre lo stesso, ma solo per valori di sparsità più elevati. In effetti, per le matrici 1250×400 e 800×625 si verifica per sparsità maggiori uguali a 0.950, ma non per valori inferiori, che individuano un comportamento

opposto. Allo stesso modo, per matrici 2500×2000 e 5000×1000 , che sono ancora più grandi, è necessario avere una sparsità di 0.975 o superiore.

Le limitazioni inferiori sembrano comportarsi in modo opposto. In particolare, si ottengono le limitazioni migliori per matrici 800×625 rispetto a matrici 1250×400 , a prescindere dalla sparsità. Lo stesso vale per le matrici 2500×2000 , che producono limitazioni inferiori migliori rispetto a quanto ottenuto con matrici 5000×1000 , ma solo quando la sparsità è inferiore o uguale a 0.995.

In realtà, il comportamento delle limitazioni inferiori sembra dipendere da un altro fattore e non dal numero degli elementi della matrice. Le limitazioni inferiori non sembrano dipendere tanto dal numero di elementi o di variabili, ma piuttosto dal bilanciamento delle matrici. Non è tanto avere un minor numero di variabili che migliora la limitazione inferiore, ma quanto più avere un numero di variabili che sia bilanciato rispetto al numero dei vincoli. In altre parole, sperimentalmente si osserva che la qualità delle limitazioni inferiori prodotte dall'algoritmo di Frank-Wolfe è migliore per le istanze caratterizzate da matrici di riferimento bilanciate, relativamente al numero delle righe e delle colonne. In effetti, sperimentando ulteriormente si ottengono i risultati della figura 4.4.

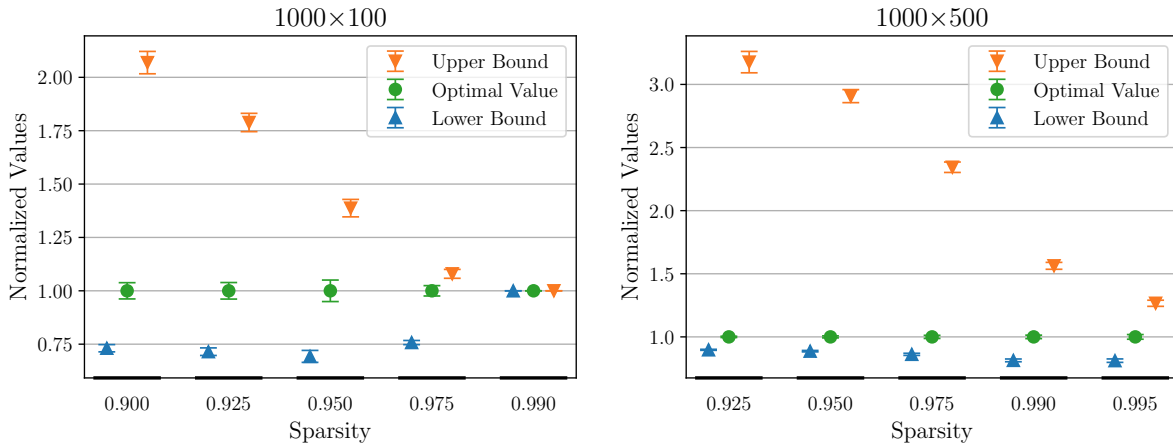


Figura 4.4: qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze piccole e medie.

Osserviamo che la limitazione superiore si comporta come argomentato in precedenza, migliorando, a parità di sparsità, per matrici con un numero minore di variabili. La limitazione inferiore invece si comporta diversamente e conferma quanto ipotizzato, tanto che la qualità è maggiore per le matrici 1000×500 quando la sparsità è inferiore o uguale a 0.975. In aggiunta, se confrontiamo i risultati appena ottenuti con quelli della figura 4.1, notiamo che per matrici 1000×1000 le limitazioni inferiori sono ancora migliori, a conferma di quanto detto, come evidenziato anche dai dati riportati per completezza nelle tabelle che seguono.

Dimensione	Limitazione Inferiore	Limitazione Superiore
1000×100	0.715	1.788
1000×500	0.898	3.177

Tabella 4.4: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della forma delle matrici di riferimento di istanze con sparsità pari a 0.925.

Dimensione	Limitazione Inferiore	Limitazione Superiore
1250×400	0.853	2.874
800×625	0.912	2.884
1000×100	0.693	1.387
1000×500	0.887	2.907
5000×1000	0.869	4.275
2500×2000	0.939	4.173

Tabella 4.5: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della forma delle matrici di riferimento di istanze con sparsità pari a 0.95.

Dimensione	Limitazione Inferiore	Limitazione Superiore
1250×400	0.820	2.211
800×625	0.897	2.377
1000×100	0.758	1.079
1000×500	0.862	2.344
5000×1000	0.836	3.551
2500×2000	0.926	3.620

Tabella 4.6: valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della forma delle matrici di riferimento di istanze con sparsità pari a 0.975.

Fino a questo momento abbiamo considerato matrici di forme differenti ma in tutti i casi il numero delle colonne era maggiore uguale al numero delle righe. Questa scelta deriva dal fatto che questa è la condizione più comune per le istanze del problema del set-covering, in cui il numero dei vincoli è solitamente dominante rispetto al numero delle variabili. Tuttavia ci sono dei contesti e delle applicazioni in cui potrebbe valere il contrario e quindi abbiamo effettuato degli esperimenti su matrici in cui il numero delle colonne, ossia il numero di variabili, è maggiore del numero delle righe, che rappresentano invece i vincoli. I risultati sono riportati nella figura 4.5.

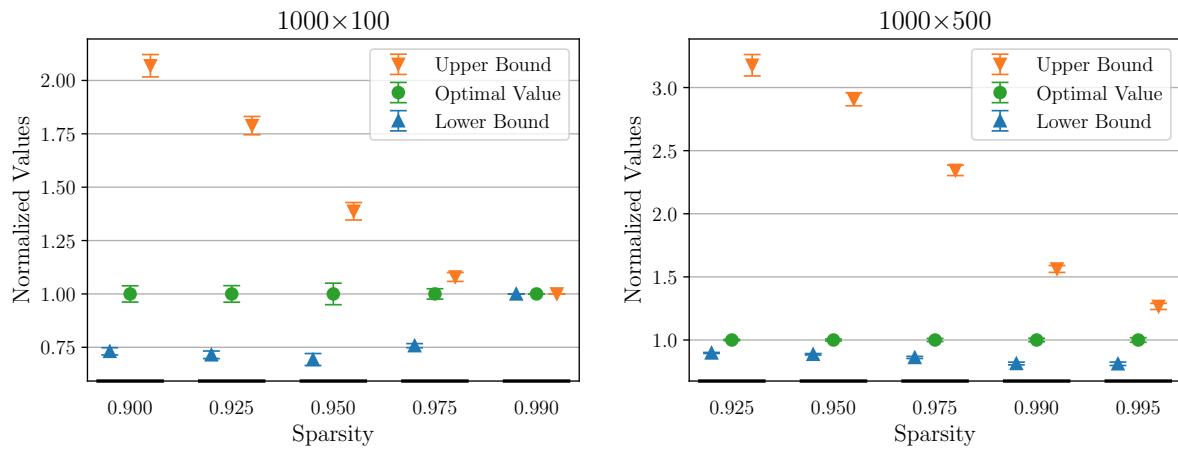


Figura 4.5: qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze piccole e medie.