Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

UNIVERSITÀ DEGLI STUDI DI PADOVA

Corso di Laurea Triennale in Ingegneria Informatica

Set-Covering: un'implementazione dell'algoritmo di Frank-Wolfe per risolvere il rilassamento lineare

---

Relatore

Prof. Domenico Salvagnin

Laureando

Leonardo Callegari

Anno Accademico 2023 – 2024

Data di Laurea

26 settembre 2024

Abstract

Il problema del Set-Covering (Set-Covering Problem, SCP) è un problema di ottimizzazione combinatoria che consiste nel determinare la più piccola collezione di elementi in una famiglia di sottoinsiemi di un insieme, in modo che ciascun elemento di quest'ultimo sia coperto dal almeno un sottoinsieme all'interno di tale collezione.

Questo problema, insieme alle varianti che ne derivano, ha numerose applicazioni pratiche in molteplici ambiti. Viene ad esempio impiegato nell'ambito della logistica per ottimizzare la gestione e la distribuzione delle risorse. Trova spazio anche all'interno delle compagnie aeree, dove viene utilizzato per semplificare e ottimizzare la gestione degli aerei e dei turni di lavoro dei piloti.

Questo lavoro propone l'implementazione di un algoritmo basato sul metodo di Frank-Wolfe, con l'obiettivo di ricercare un modo efficiente di risolvere il rilassamento lineare del SCP.

Nella prima parte vengono introdotte nozioni di carattere generale riguardanti l'ottimizzazione matematica e la programmazione lineare [1]. Inoltre, vengono brevemente descritte l'intuizione alla base dell'algoritmo del simplesso e la modalità con cui questo algoritmo viene impiegato per risolvere problemi di programmazione lineare. Successivamente vengono presentate l'idea generale alla base dell'algoritmo di Frank-Wolfe e una formulazione matematica per il problema del Set-Covering, con l'obiettivo di analizzare l'applicazione di tale algoritmo al rilassamento lineare. Infine, viene sviluppata un'implementazione dell'algoritmo per risolvere diverse istanze del problema, accompagnata da un confronto con l'algoritmo del simplesso applicato alle stesse istanze.

Indice

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, soddisfando 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 [2] 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 può essere definito con la formulazione generale

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

dove $f: \mathcal{D} \to \mathbb{R}$, con $\boldsymbol{x} = [x_1, \ldots, x_n]^{\mathrm{T}} \in \mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ 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 $f(\boldsymbol{x})$ è equivalente a calcolare il minimo della funzione $-f(\boldsymbol{x})$. 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 $\boldsymbol{x} \in \mathcal{D}$ 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 $\boldsymbol{x}^\star \in F(\mathcal{P})$ si dice ottima per $\mathcal{P}$ se

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

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

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

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 $\bar{\boldsymbol{x}}$ è una generica soluzione ammissibile per $\mathcal{P}'$, ossia $\bar{\boldsymbol{x}} \in F(\mathcal{P}')$, allora $\bar{\boldsymbol{x}}$ è ammissibile per $\mathcal{P}$, cioè $\bar{\boldsymbol{x}} \in F(\mathcal{P})$. Inoltre, è facile verificare che il costo associato a $\bar{\boldsymbol{x}} \in F(\mathcal{P}')$ fornisce un limite superiore (*upper bound*) al valore ottimo di $\mathcal{P}$, ossia $f(\boldsymbol{x}^\star) \leq f(\bar{\boldsymbol{x}})$, dove $\boldsymbol{x}^\star \in F(\mathcal{P})$ 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 $F(\mathcal{P}) = \varnothing$, allora anche ogni sua restrizione $\mathcal{P}'$ è impossibile, ossia $F(\mathcal{P}') = \varnothing$. 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 $f(\boldsymbol{x})$ di $\mathcal{P}$ con una sua approssimazione inferiore $g(\boldsymbol{x})$ per ogni $\boldsymbol{x} \in F(\mathcal{P})$. Formalmente, $\mathcal{R}$ è un rilassamento di $\mathcal{P}$ se

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

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

Infine, se la soluzione ottima $\boldsymbol{x}^\star \in F(\mathcal{R})$ di $\mathcal{R}$ è ammissibile per $\mathcal{P}$, con $g(\boldsymbol{x}^\star) = f(\boldsymbol{x}^\star)$, allora $\boldsymbol{x}^\star$ 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 $\mathcal{P}_1, \ldots, \mathcal{P}_m$ 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 $\mathcal{P}_1, \ldots, \mathcal{P}_m$ 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 $\boldsymbol{x} \in \mathcal{D}$ 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

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

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

$$z = \sum_{k=1}^{n} c_k x_k \,. \tag{1.3}$$

Similmente, il generico vincolo può essere scritto come

$$\sum_{j=1}^{n} a_{ij} x_j \sim b_i \qquad \forall i\colon 1 \leq i \leq m. \tag{1.4}$$

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.

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

Forma Standard (1.5a) Forma Canonica (1.5b)

dove $\boldsymbol{x} \in \mathbb{R}^n$, con $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ e $\boldsymbol{b} \in \mathbb{R}^m$. 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 $\lambda x_1 + (1-\lambda)x_2 \in \mathcal{C} \quad \forall x_1, x_2 \in \mathcal{C}$, con $\lambda \in [0,1]$.

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

$$\sum_{i=1}^{k} \lambda_i \, x_i \,, \quad \sum_{i=1}^{k} \lambda_i = 1 \,, \quad \lambda_i \geq 0 \quad \forall i \colon 1 \leq i \leq k. \tag{1.6}$$

Esempi di insiemi convessi sono gli iperpiani $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} = a_0\}$ e i semispazi chiusi $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} \leq a_0\}$, dove $\boldsymbol{a} \in \mathbb{R}^n$ 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 $\mathcal{V} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\}$ l'insieme dei suoi vertici. Allora ogni punto di $\mathcal{P}$ può essere espresso come combinazione lineare convessa dei punti in $\mathcal{V}$:

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

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 $\min\{\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{P}\}$, 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 di 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

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

dove $\boldsymbol{x} \in \mathbb{R}^n$, con $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ e $\boldsymbol{b} \in \mathbb{R}^m$. Supponiamo che $\boldsymbol{x}^\star$ 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 $\boldsymbol{x}^\star$ è la soluzione di costo minimo, ossia che $f(\boldsymbol{x}^\star) \leq f(\boldsymbol{x})$ per ogni $\boldsymbol{x} \in F(\mathcal{P})$.

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

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

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

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

essendo $\boldsymbol{x} \geq 0$. In altre parole, possiamo utilizzare $\boldsymbol{u}$ 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 $\boldsymbol{u}$ in modo da ottenere la limitazione inferiore migliore, cioè quella più alta possibile. In effetti, questo significa risolvere il problema di programmazione lineare

$$\mathcal{D}: \begin{cases} \max & \boldsymbol{u}^{\mathrm{T}}\boldsymbol{b} \\ & \boldsymbol{u}^{\mathrm{T}}\boldsymbol{A} \leq \boldsymbol{c}^{\mathrm{T}} \\ & \boldsymbol{u} \geq 0 \end{cases} \tag{1.10}$$

nelle variabili $\boldsymbol{u} = [u_1, \ldots, u_m]^{\mathrm{T}} \in \mathbb{R}^m$, 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 $\boldsymbol{x}$ e $\boldsymbol{u}$ di $\mathcal{P}$ e $\mathcal{D}$, rispettivamente, risulta

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

Inoltre, se vale $\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{u}$, allora $\boldsymbol{x}$ e $\boldsymbol{u}$ 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 $\boldsymbol{x}^\star$ del problema primale $\mathcal{P}$, è sufficiente determinare una soluzione ammissibile $\boldsymbol{u}^\star$ del problema duale $\mathcal{D}$, in modo che risulti

$$\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^\star = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{u}^\star. \tag{1.11}$$

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

$$\begin{aligned} \min \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned} \tag{1.12}$$

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

$$L(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}), \tag{1.13}$$

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

$$\mathcal{L}(\boldsymbol{u}) \colon \begin{cases} \min & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}) = L(\boldsymbol{x}, \boldsymbol{u}) \\ & \boldsymbol{x} \geq 0 \end{cases} \tag{1.14}$$

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

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

con $\mathcal{L}(\boldsymbol{u})$ 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

$$\mathcal{D} \colon \begin{cases} \max & \mathcal{L}(\boldsymbol{u}) \\ & \boldsymbol{u} \geq 0 \end{cases} \tag{1.16}$$

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

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

dove $\sim \in \{\leq, =, \geq\}$, $\ell_j \in \mathbb{R} \cup \{-\infty\}$ e $u_j \in \mathbb{R} \cup \{+\infty\}$, con $c_k \in \mathbb{R}$ $\forall k \colon 1 \leq k \leq n$, $a_{ij} \in \mathbb{R}$ $\forall i, j \colon 1 \leq i \leq m$, $1 \leq j \leq n$ e $b_i \in \mathbb{R}$ $\forall i \colon 1 \leq i \leq m$. 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 $J = \varnothing$ 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

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

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

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

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

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

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

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

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

Notiamo che per come è definito, il generico punto $\boldsymbol{x}_{k+1}$ 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 $\boldsymbol{x}_0 \in \mathcal{D}$ punto di partenza /* Inizializzazione */

2 **while** *true* **do**

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

4 $\quad \boldsymbol{x}_{k+1} = (1-\gamma)\boldsymbol{x}_k + \gamma \boldsymbol{d}_k$ 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.

2 Set-Covering

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 $\mathcal{F}^\star \subseteq \mathcal{F}$ tale che

$$\bigcup_{\mathcal{F}_j \in \mathcal{F}^\star} \mathcal{F}_j = \mathcal{I}\,. \tag{2.1}$$

In altre parole, si richiede che ciascun elemento di $\mathcal{I}$ sia coperto da almeno uno dei sottoinsiemi in $\mathcal{F}^\star$. 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 $\mathcal{F}^\star$. In particolare, definiamo

$$x_j = \begin{cases} 1 & \text{se } \mathcal{F}_j \in \mathcal{F}^\star \\ 0 & \text{altrimenti} \end{cases} \qquad \forall j \colon 1 \leq j \leq n. \tag{2.2}$$

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

$$\min \sum_{j=1}^{n} x_j \tag{2.3}$$

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

$$\sum_{j \colon i \in \mathcal{F}_j} x_j \geq 1 \qquad \forall i \in \mathcal{I}. \tag{2.4}$$

Combinando tutte le considerazioni fatte, otteniamo la formulazione matematica

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

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

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

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

$$\mathcal{P}\colon \begin{cases} \min & \displaystyle\sum_{j=1}^{n} x_j & \\ & \displaystyle\sum_{j=1}^{n} a_{ij}\, x_j \geq 1 & \forall i \in \mathcal{I} \\ & x_j \geq 0 & \forall j\colon 1 \leq j \leq n \end{cases} \tag{2.7}$$

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

$$\begin{aligned} \min \quad & \boldsymbol{c}^{\mathrm{T}} \boldsymbol{x} \\ & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned} \tag{2.8}$$

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

Naturalmente una soluzione del rilassamento lineare non è neccessariamente 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 $\boldsymbol{u} = [u_1, \dots, u_m]^{\mathrm{T}} \in \mathbb{R}^m$ di moltiplicatori duali per combinare linearmente i vincoli e identificare una limitazione inferiore al valore della funzione obiettivo. In particolare, dobbiamo trovare $\boldsymbol{u}$ tale che

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

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

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} u_i \\ & \sum_{i=1}^{m} a_{ij}\, u_i \leq 1 \quad \forall j \colon 1 \leq j \leq n \\ & u_i \geq 0 \qquad\qquad \forall i \colon 1 \leq i \leq m \end{aligned} \tag{2.10}$$

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 $\boldsymbol{u} = [u_1, \dots, u_m]^{\mathrm{T}} \in \mathbb{R}^m$ 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

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

e possiamo definire il rilassamento Lagrangiano

$$\mathcal{L}(\boldsymbol{u}): \begin{cases} \min & \displaystyle\sum_{j=1}^{n}\Big[x_j\Big(1-\sum_{i=1}^{m} a_{ij}\,u_i\Big)\Big]+\sum_{i=1}^{m} u_i \\ & x_j \in [0,1] \qquad \forall j: 1 \le j \le n \end{cases} \tag{2.12}$$

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

$$\sum_{i=1}^{m} a_{ij}\,u_i > 1 \tag{2.13}$$

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

$$\max_{\boldsymbol{u} \ge 0} \mathcal{L}(\boldsymbol{u}) \tag{2.14}$$

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

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

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

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 $\boldsymbol{u}^\star \in \mathbb{R}^m$ del problema duale Lagrangiano si può costruire vincolando i moltiplicatori ad assumere valori nell'intervallo $[0,1]$, ossia $\boldsymbol{u}^\star \in [0,1]^m$, che è un insieme convesso e compatto.

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

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,

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

di $\mathcal{P}$, visto che il dominio $[0,1]^m$ è compatto e convesso e la funzione obiettivo $\mathcal{L}(\boldsymbol{u})$ 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 $\boldsymbol{u}_0 \in [0,1]^m$ punto di partenza /* Inizializzazione */
2 **while** *true* **do**
3 $\quad \boldsymbol{x}_k = \arg\min_{\boldsymbol{x} \in [0,1]^n} \{\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{u}_k^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})\}$ /* Soluzione di $\mathcal{L}(\boldsymbol{u}_k)$ */
4 $\quad \boldsymbol{s}_k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k$ /* Subgradiente in $\boldsymbol{u}_k$ */
5 $\quad \boldsymbol{d}_k = \arg\max_{\boldsymbol{d} \in [0,1]^m} \{\boldsymbol{s}_k^{\mathrm{T}}\,\boldsymbol{d}\}$ /* Passo 1 */
6 $\quad \boldsymbol{u}_{k+1} = (1-\gamma)\boldsymbol{u}_k + \gamma \boldsymbol{d}_k$ con $\gamma = \frac{2}{k+2}$ /* Passo 2 */

Osserviamo che, ad ogni iterazione, il valore

$$\alpha_k = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}_k + \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k) = \mathcal{L}(\boldsymbol{u}_k), \tag{2.17}$$

fornisce una limitazione inferiore al valore ottimo e il valore

$$\omega_k = \mathcal{L}(\boldsymbol{u}_k) + \boldsymbol{s}_k^{\mathrm{T}}(\boldsymbol{d}_k - \boldsymbol{u}_k) \tag{2.18}$$

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, $\alpha^\star$ e $\omega^\star$, e terminare l'algoritmo quando la differenza è sufficientemente piccola. A quel punto avremo che

$$\alpha^\star \leq \mathcal{L}(\boldsymbol{u}^\star) \leq \omega^\star \tag{2.19}$$

con $\boldsymbol{u}^\star$ 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 $\boldsymbol{A} = [a_{ij}] \in \mathbb{R}^{m \times n}$, 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 $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, 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 quello di sviluppare l'implementazione di un algoritmo risolutivo per il rilassamento lineare del set-covering, basato sul metodo di Frank-Wolfe.

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

```python
import numpy as np
from scipy.sparse import csr_matrix

def generate_matrix(rows, cols, sparsity):
    nnz = int(round((1 - sparsity) * rows * cols))
    matrix = np.zeros((rows, cols), dtype=int)

    for r in range(rows):
        matrix[r, np.random.choice(cols)] = 1

    nc = np.where(matrix.sum(axis=0) == 0)[0]
    for c in nc:
        matrix[np.random.choice(rows), c] = 1

    remaining_nnz = nnz - np.sum(matrix)
    if remaining_nnz > 0:
        zeros = np.argwhere(matrix == 0)
        selected_indices = zeros[
            np.random.choice(zeros.shape[0], remaining_nnz, False)
        ]
        for r, c in selected_indices:
            matrix[r, c] = 1

    return csr_matrix(matrix)
```

Listing 3.1: Generazione delle matrici binarie di riferimento.

La funzione appena presentata sfrutta gli strumenti della libreria numpy [4] 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, poichè 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 [5].

3.2 Algoritmo del simplesso

Iniziamo a presentare il codice necessario a risolvere le istanze con l'algoritmo del simplesso. Utilizzeremo l'interfaccia pyscipopt [7] per accedere all'implementazione SCIP dell'algoritmo del simplesso. Le componenti necessarie sono quelle importate di seguito.

```
from pyscipopt 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)), sense="minimize")
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 $\boldsymbol{u}_0 \in [0,1]^m$ arbitrario.

solver.c

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

Listing 3.4: Calcolo del punto di partenza.

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

3.3.2 Soluzione del rilassamento Lagrangiano

Riprendiamo la specifica del rilassamento Lagrangiano

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

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

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

dove la sommatoria rappresenta il prodotto di $\boldsymbol{u}$ con la colonna j-esima della matrice di riferimento. Equivalentemente, lo stesso prodotto si può ottenere moltiplicando $\boldsymbol{u}$ 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 $\boldsymbol{u}$ 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 $\boldsymbol{u}$. 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 $\mathcal{L}(\boldsymbol{u})$, nel punto `u` passato come parametro.

solver.c

```c
double L(const double * const u, uint8_t * const x) {
    double objective = 0;

    for (int j = 0; j < n; j++) {
        double result = 0;
        for (int i = row_start(&T, j); i < row_end(&T, j); i++) {
            result += u[T.indices[i]];
        }
        if (x[j] = (result > 1)) {
            objective += (1 - result);
        }
    }

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

    return objective;
}
```

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 `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

$$\max_{\boldsymbol{u} \geq 0} \mathcal{L}(\boldsymbol{u}) \tag{2.14}$$

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

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

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

$$\boldsymbol{s}_{\hat{\boldsymbol{u}},i} = 1 - \sum_{j=1}^{n} a_{ij}\, x_j^\star\,, \tag{3.2}$$

con la sommatoria che rappresenta il prodotto della riga i-esima della matrice di riferimento con il vettore $\boldsymbol{x}^\star$. 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

solver.c

```
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  }
```

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

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

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

$$f(\boldsymbol{x}) \simeq f(\hat{\boldsymbol{x}}) + \nabla f(\hat{\boldsymbol{x}})^{\mathrm{T}}(\boldsymbol{x} - \hat{\boldsymbol{x}}) \tag{3.3}$$

nelle vicinanze di un punto $\hat{\boldsymbol{x}}$. 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 $\mathcal{L}(\boldsymbol{u}) \simeq \mathcal{L}(\hat{\boldsymbol{u}}) + \boldsymbol{s}_{\hat{\boldsymbol{u}}}^{\mathrm{T}}(\boldsymbol{u} - \hat{\boldsymbol{u}})$ intorno a $\hat{\boldsymbol{u}}$, che assume il suo valore massimo in

$$\boldsymbol{d}^{\star} = \arg\max_{\boldsymbol{d} \in [0,1]^m} \{\boldsymbol{s}_{\hat{\boldsymbol{u}}}^{\mathrm{T}} \boldsymbol{d}\}, \tag{3.4}$$

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

solver.c

```
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 $\boldsymbol{d}_k = \arg\max_{\boldsymbol{d} \in [0,1]^m} \{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{d}\}$.

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

3.3.5 Calcolo del punto successivo

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

solver.c

```
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 $\boldsymbol{u}_{k+1}$.

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 $\mathcal{L}(\boldsymbol{u}_k)$ 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

```c
void update_lower_bound(double * const lower_bound, double candidate) {
    if (candidate > *lower_bound) {
        *lower_bound = candidate;
    }
}
```

Listing 3.9: Aggiornamento della limitazione inferiore.

Limitazione Superiore *(Upper Bound)*

Dalle considerazioni fatte in 2.4 si è concluso che il valore $\mathcal{L}(\boldsymbol{u}_k) + \boldsymbol{s}_k^{\mathrm{T}}(\boldsymbol{d}_k - \boldsymbol{u}_k)$ 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

```c
void update_upper_bound(double * const upper_bound, double objective,
    const int * const s, const uint8_t * const d, const double * const u
) {
    double candidate = objective;
    for (int i = 0; i < m; i++) {
        candidate += s[i] * (d[i] - u[i]);
    }

    if (candidate < *upper_bound) {
        *upper_bound = candidate;
    }
}
```

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 [9], che viene riportato nel frammento di codice che segue.

solver.c

```c
void solve(int K) {
    double u[m], objective, lower_bound = 0, upper_bound = 1e9;
    int s[m]; uint8_t d[m], x[n];

    starting_u(u);
    for (int k = 0; k < K; k++) {
        objective = L(u, x);
        subgradient(x, s);
        argmax(s, d);
        update_lower_bound(&lower_bound, objective);
        update_upper_bound(&upper_bound, objective, s, d, u);
        next_u(u, d, 2 / ((double) k + 2));
    }
}
```

Listing 3.11: Implementazione dell'algoritmo risolutivo.

La funzione `solve` esegue l'algoritmo di Frank-Wolfe per il numero `K` di iterazioni che viene specificato come parametro di ingresso.

3.4 Esecuzione

In questa sezione riportiamo i dettagli tecnici e le informazioni necessarie per eseguire il codice che abbiamo presentato fino a questo punto.

3.4.1 Generazione delle istanze

Per la generazione delle istanze, utilizziamo `datagen.py`. Per generare `count` istanze caratterizzate da matrici di riferimento con `m` righe e `n` colonne e una sparsità pari a `sparsity`, utilizziamo il comando riportato di seguito.

```
$ python3 datagen.py <count> <m> <n> <sparsity>
```

3.4.2 Esecuzione algoritmo del simplesso

Per risolvere un'istanza con l'algoritmo del simplesso, è sufficiente specificare il percorso che identifica il file `csr_matrix.dat` associato. In particolare, utilizziamo il comando riportato di seguito.

```
$ python3 solver.py <path/to/csr_matrix.dat>
```

Il percorso specificato è relativo alla cartella che contiene il sorgente `solver.py`.

3.4.3 Esecuzione algoritmo di Frank-Wolfe

Per risolvere un'istanza con l'algoritmo di Frank-Wolfe, è sufficiente specificare il percorso che identifica il file `csr_matrix.dat` associato e il numero di iterazioni da effettuare. In particolare, utilizziamo il comando

```
$ gcc solver.c -o solver -Ofast
```

per compilare il codice sorgente dell'algoritmo e il comando

```
$ ./solver <path/to/csr_matrix.dat> <K>
```

per eseguire l'algoritmo. Il percorso specificato è relativo alla cartella che contiene il sorgente `solver.c`.

4 Risultati

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. In particolare, abbiamo utilizzato 20 istanze differenti dello stesso tipo, in quanto a dimensione e sparsità della matrice di riferimento, e abbiamo calcolato media e deviazione standard dei risultati ottenuti, in modo da poterli riassumere utilizzando opportuni grafici [6].

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.

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.

4.1.1 Specifiche tecniche

Per compilare il codice sorgente in linguaggio C abbiamo utilizzato gcc 13.2.0 [8] e per eseguire il codice Python abbiamo utilizzato Python 3.11.7 [3].

Tutti gli esperimenti sono stati eseguiti sulla stessa macchina, utilizzando un processore Intel(R) Core(TM) i7-1165G7@2.80GHz e 16 GB di memoria RAM sul sistema operativo Ubuntu 23.10.

4.2 Qualità delle soluzioni

Procediamo ora con l'esecuzione dei due algoritmi per confrontare le limitazioni prodotte dall'algoritmo di Frank-Wolfe con il valore ottimo trovato dall'algoritmo del simplesso. 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 delle limitazioni prodotte.

4.2.1 Dimensione della matrice di riferimento

Iniziamo considerando istanze di dimensioni differenti. In questo contesto 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 nei grafici della Figura 4.1 e riportati nella Tabella 4.1.

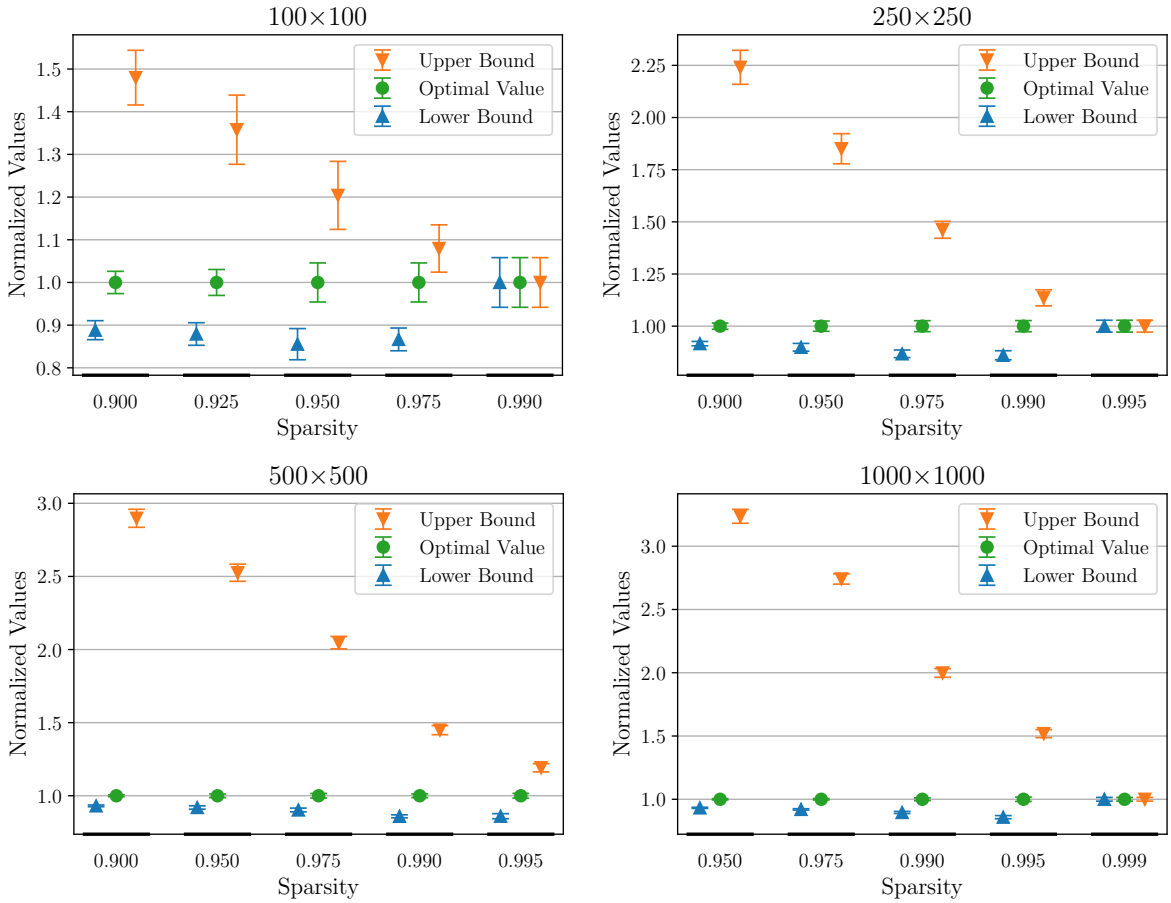


Figura 4.1: Qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e del numero di elementi della matrice di riferimento delle istanze.

Matrice	Sparsità	Limitazione Inferiore	Limitazione Superiore
100×100	0.900	0.888	1.480
	0.925	0.879	1.358
	0.950	0.855	1.204
	0.975	0.867	1.080
	0.990	1.000	1.000
250×250	0.900	0.916	2.241
	0.950	0.899	1.850
	0.975	0.867	1.462
	0.990	0.861	1.136
	0.995	1.000	1.000
500×500	0.900	0.931	2.898
	0.950	0.920	2.525
	0.975	0.903	2.047
	0.990	0.859	1.449
	0.995	0.860	1.191
1000×1000	0.950	0.932	3.237
	0.975	0.920	2.741
	0.990	0.896	1.998
	0.995	0.859	1.519
	0.999	1.000	1.000

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.

Osserviamo che, a parità di sparsità, l'aumentare del numero di elementi della matrice di riferimento è indicatore di un peggioramento nella qualità della limitazione superiore per tutte le istanze che abbiamo considerato. Il comportamento della limitazione inferiore è invece più irregolare. In particolare, quando il valore di sparsità è abbastanza basso le limitazioni inferiori migliori si ottengono per matrici di dimensione maggiore. Al contrario, per valori di sparsità sufficientemente elevati avere matrici di dimensione minore permette di ottenere limitazioni inferiori di qualità maggiore e talvolta di chiudere sul valore ottimo.

Per le matrici di dimensione fissata, l'aumento di sparsità determina un miglioramento della limitazione superiore ma non della limitazione inferiore, che ha un comportamento più irregolare.

In tutti i casi, le variazioni delle limitazioni superiori sono più marcate rispetto alle variazioni delle limitazioni inferiori, che rimangono invece molto più stabili.

4.2.2 Forma della matrice di riferimento

Procediamo ora a considerare istanze caratterizzate da matrici di riferimento di forme differenti, con l'obiettivo di capire come la forma della matrice influenza la qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe. In particolare, faremo riferimento a istanze caratterizzate da matrici di riferimento in cui il numero delle righe e delle colonne è sbilanciato da un parte oppure dall'altra, come accade nella maggior parte delle istanze relative a problemi di set-covering.

I risultati per istanze di taglia piccola e media, caratterizzate da matrici di riferimento con lo stesso numero di elementi ma forma differente, sono riassunti nei grafici della Figura 4.2 e riportati nella Tabella 4.2.

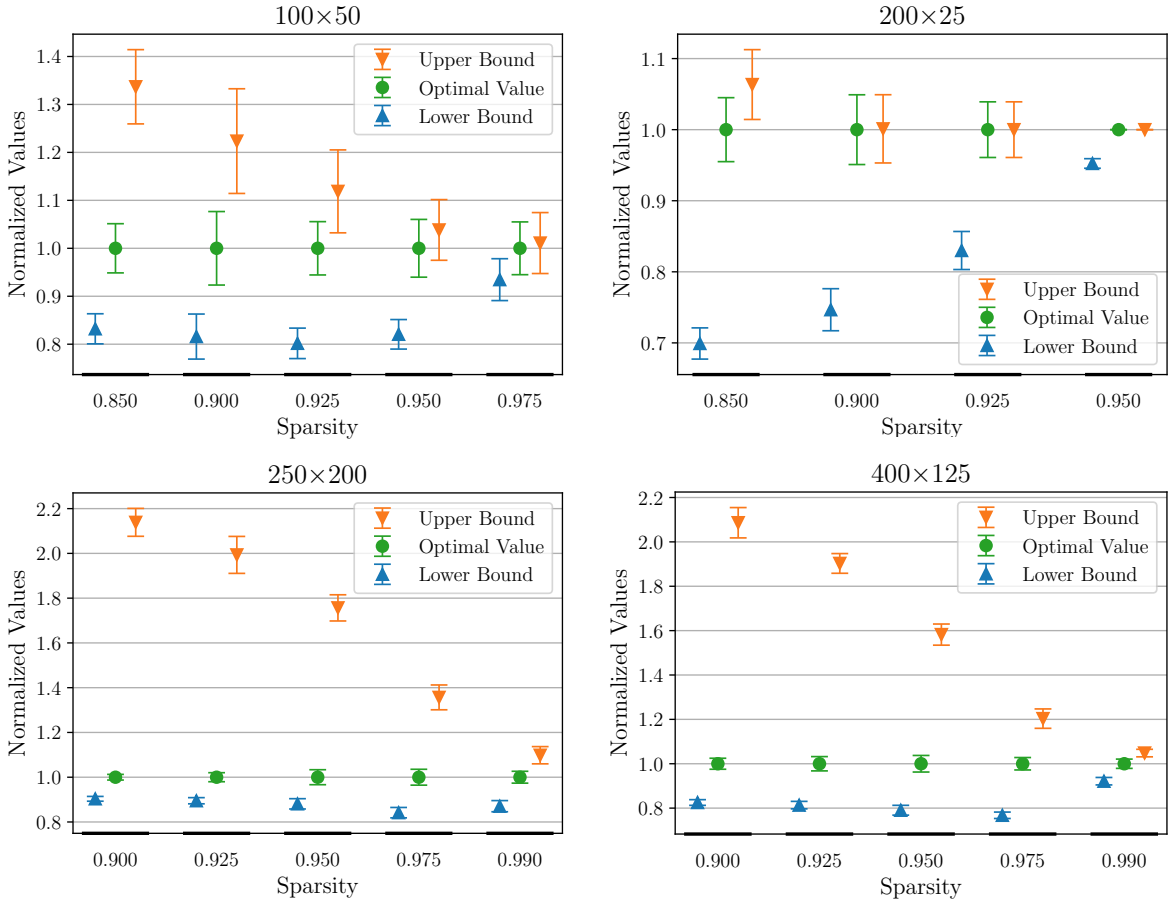


Figura 4.2: Qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze piccole e medie.

Se consideriamo le singole istanze, notiamo ancora una volta che l'aumento di sparsità è indicatore di un miglioramento della limitazione superiore ma non della limitazione inferiore, che ha un comportamento più irregolare.

Matrice	Sparsità	Limitazione Inferiore	Limitazione Superiore
100×50	0.850	0.832	1.337
	0.900	0.816	1.224
	0.925	0.802	1.119
	0.950	0.821	1.038
	0.975	0.935	1.011
200×25	0.850	0.699	1.063
	0.900	0.747	1.001
	0.925	0.830	1.000
	0.950	0.952	1.000
250×200	0.900	0.904	2.139
	0.925	0.895	1.993
	0.950	0.881	1.757
	0.975	0.842	1.357
	0.990	0.871	1.098
400×125	0.900	0.825	2.086
	0.925	0.814	1.903
	0.950	0.790	1.582
	0.975	0.768	1.203
	0.990	0.921	1.048

Tabella 4.2: Valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della forma della matrice di riferimento di istanze piccole e medie.

Osserviamo che, a parità di sparsità, la forma della matrice di riferimento influenza la qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe, anche se il numero degli elementi rimane lo stesso. In particolare, le limitazioni superiori migliori si ottengono per le istanze caratterizzate da un numero di variabili inferiori. Ad esempio, le matrici 200×25 producono limitazioni superiori migliori rispetto alle matrici 100×50, a prescindere dalla sparsità della matrice di riferimento. Lo stesso vale per le matrici 400×125, che producono limitazioni superiori migliori rispetto alle matrici 250×200.

Le limitazioni inferiori migliori si ottengono con le istanze caratterizzate da un numero di variabili maggiore, quando i valori di sparsità sono abbastanza bassi, e con le istanze caratterizzate da un numero di variabili minore, quando i valori di sparsità sono sufficientemente elevati.

Considerando istanze di taglia grande, otteniamo i risultati riassunti nella Figura 4.3 e riportati nella Tabella 4.3.

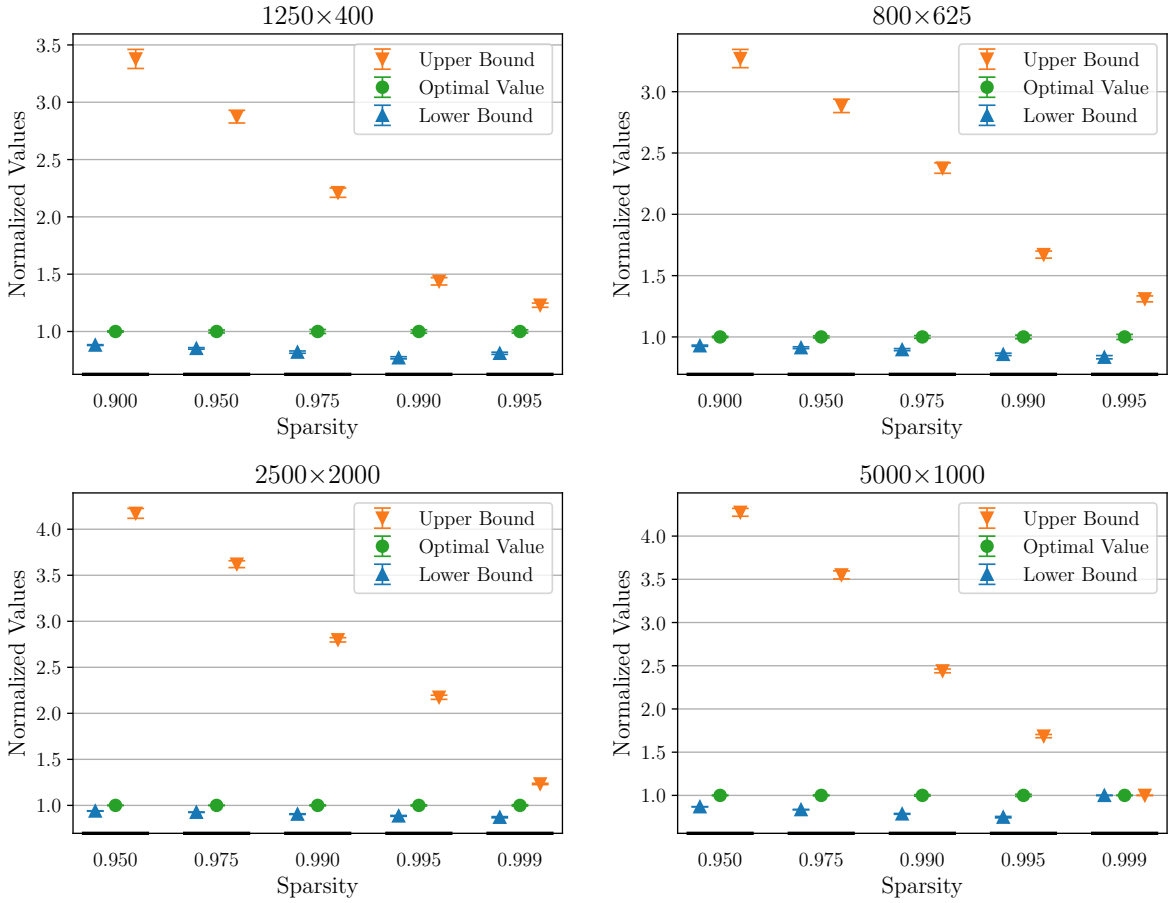


Figura 4.3: Qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento di istanze grandi.

Notiamo che il comportamento delle limitazioni superiori è lo stesso che abbiamo ottenuto per matrici più piccole, ma è meno evidente e si verifica solamente per valori di sparsità più elevati. Ad esempio, le matrici 1250×400 producono limitazioni superiori migliori rispetto alle matrici 800×625 quando la sparsità è maggiore o uguale a 0.950. Similmente, le matrici 5000×1000 producono limitazioni superiori migliori rispetto alle matrici 2500×2000, ma solo per valori di sparsità maggiori o uguali a 0.975.

Le limitazioni inferiori hanno un comportamento differente rispetto a prima. Ad esempio, le matrici 800×625 producono limitazioni inferiori migliori rispetto alle matrici 1250×400, a prescindere dal valore di sparsità. Similmente, le matrici 2500×2000 producono limitazioni inferiori migliori rispetto alle matrici 5000×1000, ma solo per valori di sparsità minori o uguali a 0.995.

La diversità nel comportamento delle limitazioni inferiori per matrici grandi rispetto a matrici più piccole suggerisce che la dimensione e la sparsità non costituiscono fattori così determinanti in questo caso.

Matrice	Sparsità	Limitazione Inferiore	Limitazione Superiore
800×625	0.900	0.928	3.270
	0.950	0.912	2.884
	0.975	0.897	2.377
	0.990	0.858	1.671
	0.995	0.835	1.310
1250×400	0.900	0.882	3.378
	0.950	0.853	2.874
	0.975	0.820	2.211
	0.990	0.771	1.438
	0.995	0.810	1.230
2500×2000	0.950	0.939	4.173
	0.975	0.926	3.620
	0.990	0.905	2.799
	0.995	0.886	2.175
	0.999	0.871	1.233
5000×1000	0.950	0.869	4.275
	0.975	0.836	3.551
	0.990	0.786	2.440
	0.995	0.748	1.686
	0.999	1.000	1.000

Tabella 4.3: Valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare della forma della matrice di riferimento di istanze grandi.

Il comportamento delle limitazioni inferiori non sembra dipendere tanto dal numero degli elementi della matrice di riferimento, ma quanto più dal bilanciamento di variabili e 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. Per confermare questa ipotesi, abbiamo utilizzato delle ulteriori istanze. I risultati sono riassunti nei grafici della Figura 4.4 e riportati nella Tabella 4.4.

In effetti, a parità di sparsità, la qualità delle limitazioni inferiori è maggiore per le matrici 1000×500 piuttosto che per le matrici 1000×100, nonostante la differenza nel numero totale di elementi. In aggiunta, se confrontiamo questi risultati con quelli ottenuti in precedenza osserviamo che le matrici 1000×1000 forniscono limitazioni inferiori ancora migliori, in accordo con quanto ipotizzato.

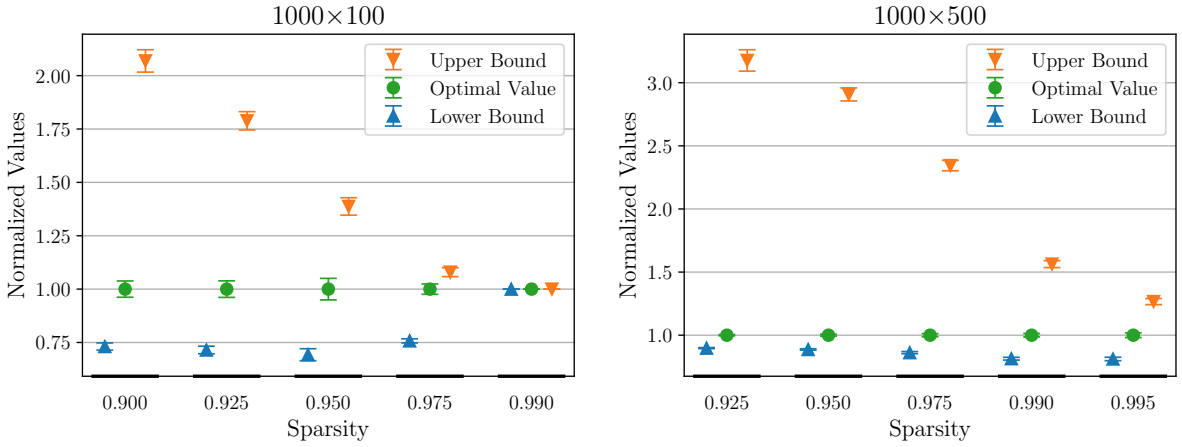


Figura 4.4: Qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento.

Matrice	Sparsità	Limitazione Inferiore	Limitazione Superiore
1000 × 100	0.900	0.731	2.069
	0.925	0.715	1.788
	0.950	0.693	1.387
	0.975	0.758	1.079
	0.990	1.000	1.000
1000 × 500	0.925	0.898	3.177
	0.950	0.887	2.907
	0.975	0.862	2.344
	0.990	0.814	1.563
	0.995	0.812	1.266

Tabella 4.4: Valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, al variare del bilanciamento di vincoli e variabili delle istanze.

Fino a questo momento abbiamo considerato matrici di forme differenti ma in tutti i casi il numero delle colonne era maggiore o 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 riassunti nei grafici della Figura 4.5 e riportati nella Tabella 4.5.

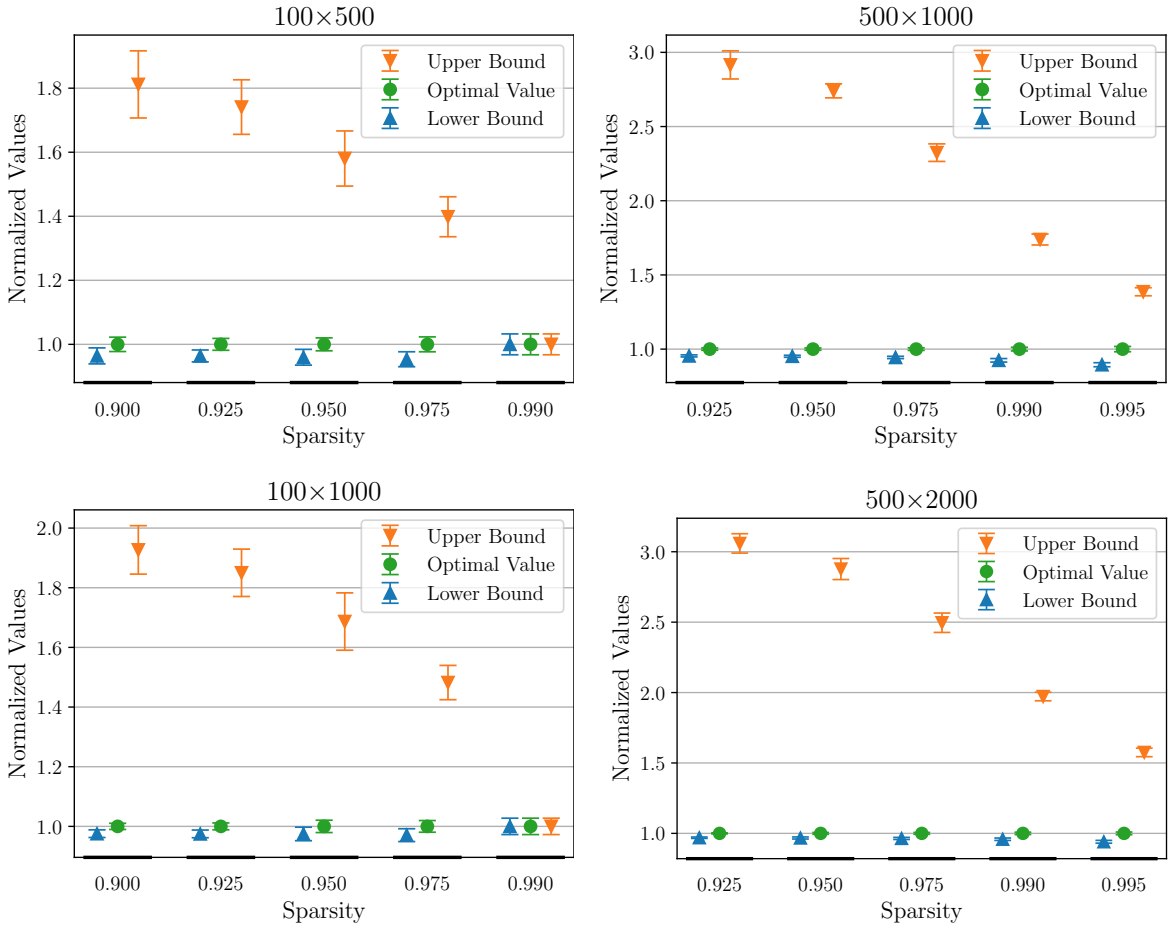


Figura 4.5: Qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe al variare della sparsità e della forma della matrice di riferimento per istanze caratterizzate da un numero di variabili dominante rispetto al numero dei vincoli.

Osserviamo che, a parità di sparsità della matrice di riferimento e numero di vincoli dell'istanza ad essa associata, l'aumentare del numero delle variabili è indicatore di un peggioramento nella qualità della limitazione superiore e di un miglioramento nella qualità della limitazione inferiore. Ad esempio, le matrici 100×1000 producono limitazioni inferiori migliori rispetto alle matrici 100×500 ma limitazioni superiori peggiori. Lo stesso vale per le matrici 500×2000 e 500×1000.

Se consideriamo istanze caratterizzate dallo stesso numero di variabili notiamo che, a parità di sparsità, le limitazioni migliori si ottengono per le istanze che hanno un numero minore di vincoli. Ad esempio, le matrici 100×1000 producono limitazioni migliori rispetto alle matrici 500×1000. Inoltre, quando il numero delle variabili è dominante rispetto al numero dei vincoli, il bilanciamento della matrice non ha influenza sulla qualità delle limitazioni inferiori. In realtà, le limitazioni ottenute con matrici 100×1000 sono anche migliori di quelle che si ottengono con la matrice bilanciata 1000×1000.

Matrice	Sparsità	Limitazione Inferiore	Limitazione Superiore
100×500	0.900	0.964	1.812
	0.925	0.964	1.741
	0.950	0.960	1.580
	0.975	0.953	1.398
	0.990	1.000	1.000
100×1000	0.900	0.976	1.927
	0.925	0.975	1.850
	0.950	0.975	1.687
	0.975	0.971	1.482
	0.990	1.000	1.000
500×1000	0.925	0.954	2.915
	0.950	0.952	2.741
	0.975	0.944	2.324
	0.990	0.925	1.739
	0.995	0.895	1.387
500×2000	0.925	0.969	3.059
	0.950	0.967	2.877
	0.975	0.964	2.496
	0.990	0.958	1.972
	0.995	0.940	1.575

Tabella 4.5: Valori delle limitazioni prodotte da Frank-Wolfe, normalizzati rispetto al valore ottimo fornito dal simplesso, per istanze caraterizzate da un numero di variabili dominante rispetto al numero dei vincoli.

Infine, se confrontiamo i risultati ottenuti con matrici caratterizzate dallo stesso numero di elementi ma con forme simmetriche, osserviamo che le limitazioni inferiori migliori si ottengono per le matrici in cui il numero delle variabili è dominante rispetto al numero dei vincoli, come accade ad esempio per le matrici 100×1000 e 1000×100 oppure 500×1000 e 1000×500.

Per le limitazioni superiori la situazione è diversa e l'andamento sembra dipendere molto dalla sparsità della matrice di riferimento. Per valori di sparsità abbastanza contenuti, le matrici 500×1000 producono limitazioni superiori migliori, mentre quando i valori di sparsità sono più elevati le limitazioni superiori migliori sono prodotte dalle matrici 1000×500.

Di conseguenza, l'influenza della forma della matrice di riferimento delle istanze dipende dalla struttura di quest'ultime, in quanto a numero di vincoli e di variabili.

4.3 Tempi di esecuzione

Gli esperimenti che abbiamo condotto fino a questo punto ci hanno permesso di capire come si comporta l'algoritmo di Frank-Wolfe al variare dei parametri che identificano le istanze che abbiamo considerato, come ad esempio la sparsità, il numero di elementi e la forma della matrice di riferimento. In particolare, abbiamo cercato di stabilire la qualità delle limitazioni prodotte da Frank-Wolfe confrontandole con il valore ottimo ottenuto attraverso l'algoritmo del simplesso. Analizzando i risultati che abbiamo ottenuto, siamo riusciti a concludere che la limitazione inferiore e quella superiore si comportano in modo diverso. In particolare, la prima è molto più stabile della seconda. In generale, abbiamo ottenuto delle limitazioni ragionevoli che richiedono ora di confrontare il tempo impiegato per ottenerle con il tempo necessario a trovare il valore ottimo con l'algoritmo del simplesso. L'obiettivo è quello di capire se l'algoritmo di Frank-Wolfe è in grado di produrre limitazioni di buona qualità in un tempo molto inferiore a quello impiegato dall'algoritmo del simplesso per determinare il valore ottimo.

In questa sezione analizzeremo i tempi di esecuzione dei due algoritmi, al variare dei parametri che abbiamo citato in precedenza. Per l'algoritmo del simplesso considereremo il tempo impiegato a trovare il valore ottimo, mentre per l'algoritmo di Frank-Wolfe considereremo il tempo impiegato a compiere un numero di iterazioni fissato. L'idea è quella di eseguire l'algoritmo di Frank-Wolfe molteplici volte per capire come il numero di iterazioni influenza le limitazioni prodotte. L'obiettivo è quello di trovare un compromesso tra la qualità delle limitazioni prodotte e il tempo impiegato per ottenerle.

4.3.1 Dimensione della matrice di riferimento

Iniziamo analizzando i risultati ottenuti dagli esperimenti svolti su istanze di dimensioni differenti, in quanto a numero di elementi della matrice di riferimento, che sono riassunti nei grafici della Figura 4.6 e riportati nella Tabella 4.6.

In primo luogo notiamo che la sparsità della matrice ha un'influenza significativa sul tempo di esecuzione dell'algoritmo del simplesso, soprattutto per le matrici più grandi. In particolare, l'aumento della sparsità è indicatore di una diminuzione nel tempo di esecuzione. Per l'algoritmo di Frank-Wolfe l'effetto è il medesimo, ma è meno marcato. Questi risultati non sono una sorpresa, poichè una matrice più sparsa richiede un numero inferiore di operazioni da eseguire rispetto ad una matrice che ha un valore di sparsità minore.

Per istanze caratterizzate da matrici di riferimento abbastanza piccole, i tempi di esecuzione di entrambi gli algoritmi sono molto bassi e quindi c'è poco spazio per le variazioni. Tuttavia, quando la matrice di riferimento è sufficientemente grande, osserviamo

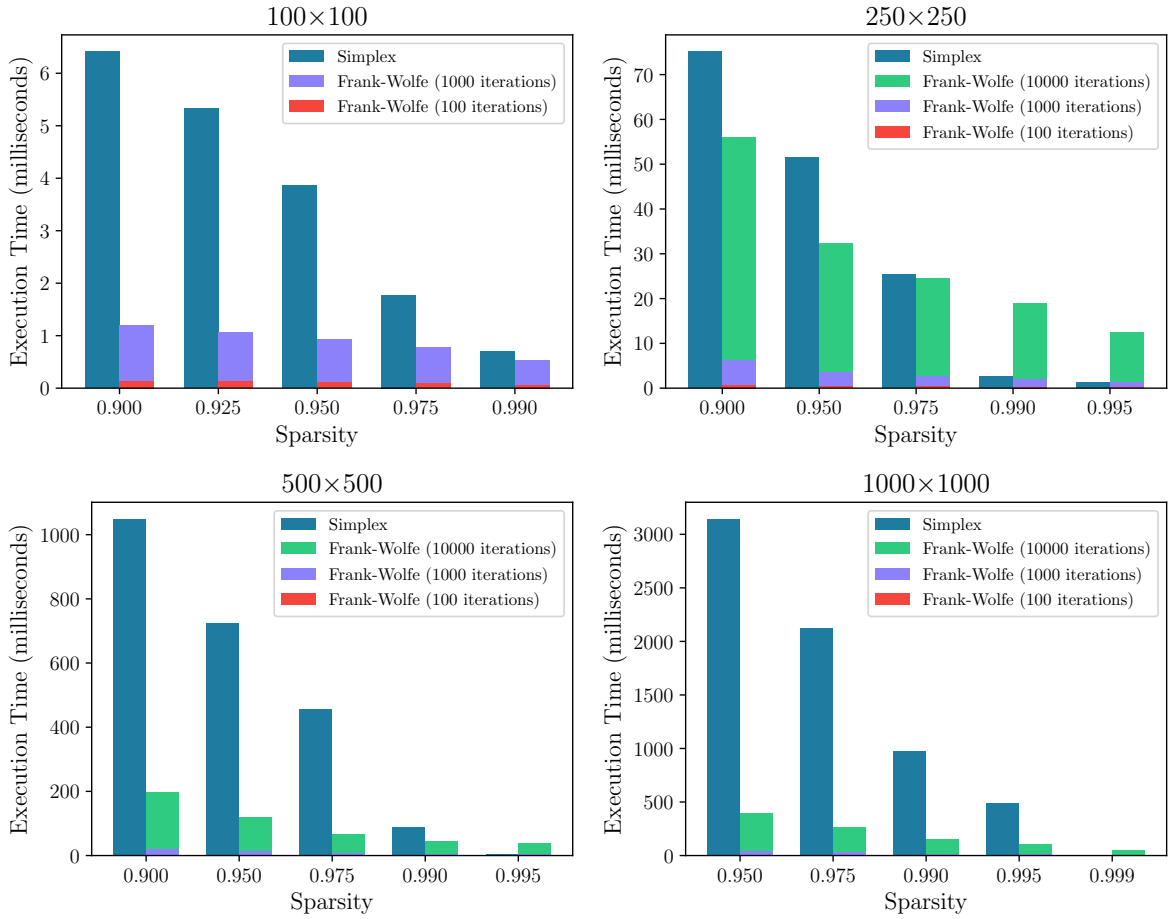


Figura 4.6: Tempi di esecuzione dell'algoritmo del simplesso e dell'algoritmo di Frank-Wolfe, al variare della sparsità e della dimensione della matrice di riferimento.

proprio una soglia di sparsità oltre la quale l'algoritmo del simplesso è notevolmente più veloce. L'andamento dei tempi di esecuzione dell'algoritmo di Frank-Wolfe è invece molto più stabile.

Il numero di iterazioni effettuate dall'algoritmo di Frank-Wolfe ha un'influenza significativa sul suo tempo di esecuzione. Per questo motivo, in diverse situazioni la competitività dell'algoritmo è vincolata alla scelta di un numero differente di iterazioni. Per le matrici più piccole, l'algoritmo del simplesso è molto veloce a prescindere dal valore di sparsità e quindi l'algoritmo di Frank-Wolfe è competitivo solo quando il numero delle iterazioni è contenuto. Quando le matrici sono più grandi, la sparsità gioca un ruolo fondamentale. Per valori di sparsità contenuti, l'algoritmo del simplesso fa molta fatica e Frank-Wolfe rimane competitivo anche quando il numero delle iterazioni è elevato. Per valori di sparsità sufficientemente elevati, l'algoritmo del simplesso si velocizza notevolmente e Frank-Wolfe rimane competitivo solo quando il numero delle iterazioni è abbastanza ridotto.

Nelle situazioni in cui Frank-Wolfe riesce a chiudere sul valore ottimo, tipicamente per valori di sparsità elevati, il numero delle iterazioni necessarie è spesso molto basso e questo permette di ottenere tempi inferiori rispetto all'algoritmo del simplesso.

In aggiunta, notiamo che per le matrici di dimensione fissata quando la sparsità aumenta il miglioramento nella qualità delle limitazioni prodotte da Frank-Wolfe diminuisce all'aumentare del numero di iterazioni effettuate. In altre parole, il miglioramento della qualità delle limitazioni ottenuto nel passaggio da 100 a 1000 iterazioni o da 1000 a 10000 è maggiore quando la sparsità della matrice di riferimento è più bassa e minore quando la sparsità è più alta.

In tutti i casi, l'aumento del numero di iterazioni è indicatore di un miglioramento più marcato della limitazione superiore piuttosto che della limitazione inferiore, che è invece molto più stabile. Inoltre, per tutte le istanze che abbiamo considerato, il passaggio da 1000 a 10000 iterazioni determina un miglioramento trascurabile rispetto al passaggio da 100 a 1000, soprattutto per le limitazioni inferiori delle matrici più piccole o delle matrici che hanno valori di sparsità abbastanza elevati. Di conseguenza, molto spesso l'esecuzione di Frank-Wolfe può essere limitata a 1000 iterazioni o poco più, per ottenere limitazioni ragionevoli in tempi molto ridotti. In alcuni casi, si può andare anche oltre e limitare il numero di iterazioni ancora di più. Le limitazioni ottenute sono comunque buone ma i tempi di esecuzione sono di molto inferiori, soprattutto per matrici più grandi.

Negli esperimenti non abbiamo considerato esecuzioni con un numero di iterazioni più elevato di 10000 semplicemente perché i tempi di esecuzione aumentavano troppo e il miglioramento della qualità delle limitazioni era trascurabile, soprattutto quello della limitazione inferiore. Utilizzare un numero di iterazioni maggiore di 10000 significa perdere la competitività dell'algoritmo di Frank-Wolfe praticamente per tutte le istanze considerate, senza però guadagnare nulla in termini di qualità delle limitazioni.

In conclusione, l'algoritmo di Frank-Wolfe può essere competitivo indipendentemente dal numero di elementi della matrice di riferimento, a patto di scegliere il giusto numero di iterazioni e di accontentarsi di una limitazione inferiore buona e di una limitazione superiore discreta. Nelle situazioni in cui Frank-Wolfe è molto più veloce del simplesso, si può considerare di aumentare il numero delle iterazioni effettuate per ottenere un miglioramento delle limitazioni, soprattutto quella superiore, a patto di mantenere un tempo di esecuzione ridotto che garantisca la competitività con l'algoritmo del simplesso.

		Limitazione Inferiore			Limitazione Superiore			Tempo Frank-Wolfe (ms)			
Matrice	Sparsità	100	1000	10000	100	1000	10000	100	1000	10000	Tempo Simplesso (ms)
100×100	0.900	0.8576	0.8865	0.8879	1.9539	1.6905	1.5373	0.1361	1.2024	11.6732	6.4107
	0.925	0.8576	0.8770	0.8791	1.7043	1.5328	1.4383	0.1225	1.0606	10.2862	5.3249
	0.950	0.8409	0.8542	0.8554	1.4336	1.3292	1.2457	0.1098	0.9214	8.9731	3.8551
	0.975	0.8598	0.8662	0.8666	1.1801	1.1163	1.1004	0.0880	0.7726	7.5559	1.7747
	0.990	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.0574	0.5323	5.0913	0.6971
250×250	0.900	0.8581	0.9096	0.9158	2.9567	2.4843	2.3332	0.7376	6.2793	55.9291	75.1152
	0.950	0.8618	0.8948	0.8985	2.2658	2.0222	1.9267	0.3853	3.4301	32.2335	51.4708
	0.975	0.8483	0.8657	0.8673	1.7121	1.5571	1.5112	0.2968	2.5420	24.4083	25.4179
	0.990	0.8541	0.8601	0.8606	1.2323	1.1907	1.1579	0.2333	1.9958	18.9113	2.5394
	0.995	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.1462	1.3031	12.4242	1.2011
500×500	0.900	0.8400	0.9204	0.9299	3.7099	3.2344	3.0214	2.2067	20.5444	198.6950	1048.5477
	0.950	0.8538	0.9127	0.9191	3.1156	2.7492	2.6017	1.4827	13.4749	121.3727	724.4617
	0.975	0.8648	0.8989	0.9027	2.3895	2.1962	2.1019	0.8559	7.5704	67.7478	456.2471
	0.990	0.8434	0.8579	0.8594	1.6196	1.5337	1.4915	0.6744	4.0800	46.5179	90.1103
	0.995	0.8531	0.8595	0.8602	1.2834	1.2430	1.2142	0.4950	4.2763	38.8078	3.5712
1000×1000	0.950	0.8346	0.9215	0.9312	3.8963	3.5183	3.3231	4.4801	41.5427	396.8238	3142.2952
	0.975	0.8553	0.9131	0.9195	3.2182	2.9260	2.8220	3.0410	28.8782	264.5643	2125.8350
	0.990	0.8621	0.8925	0.8957	2.2202	2.0965	2.0417	1.8980	17.1971	158.7026	975.0234
	0.995	0.8429	0.8571	0.8585	1.6542	1.5738	1.5479	1.3779	12.2042	111.4531	489.7301
	0.999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.6364	5.7191	51.2325	3.6620

Tabella 4.6: Qualità delle limitazioni (valori normalizzati rispetto all'ottimo ottenuto con l'algoritmo del simplesso) di Frank-Wolfe e tempo di esecuzione impiegato per ottenerle, al variare del numero di iterazioni.

4.3.2 Forma della matrice di riferimento

Procediamo ora con l'analisi dei risultati ottenuti considerando istanze caratterizzate da matrici di riferimento di forme differenti. Iniziamo considerando matrici di riferimento caratterizzate dallo stesso numero di elementi. I risultati relativi a istanze di taglia piccola e media sono riassunti nei grafici della Figura 4.7 e riportati nella Tabella 4.7, mentre quelli che riguardano le istanze di taglia grande sono riassunti nella Figura 4.8 e riportati nella Tabella 4.8.

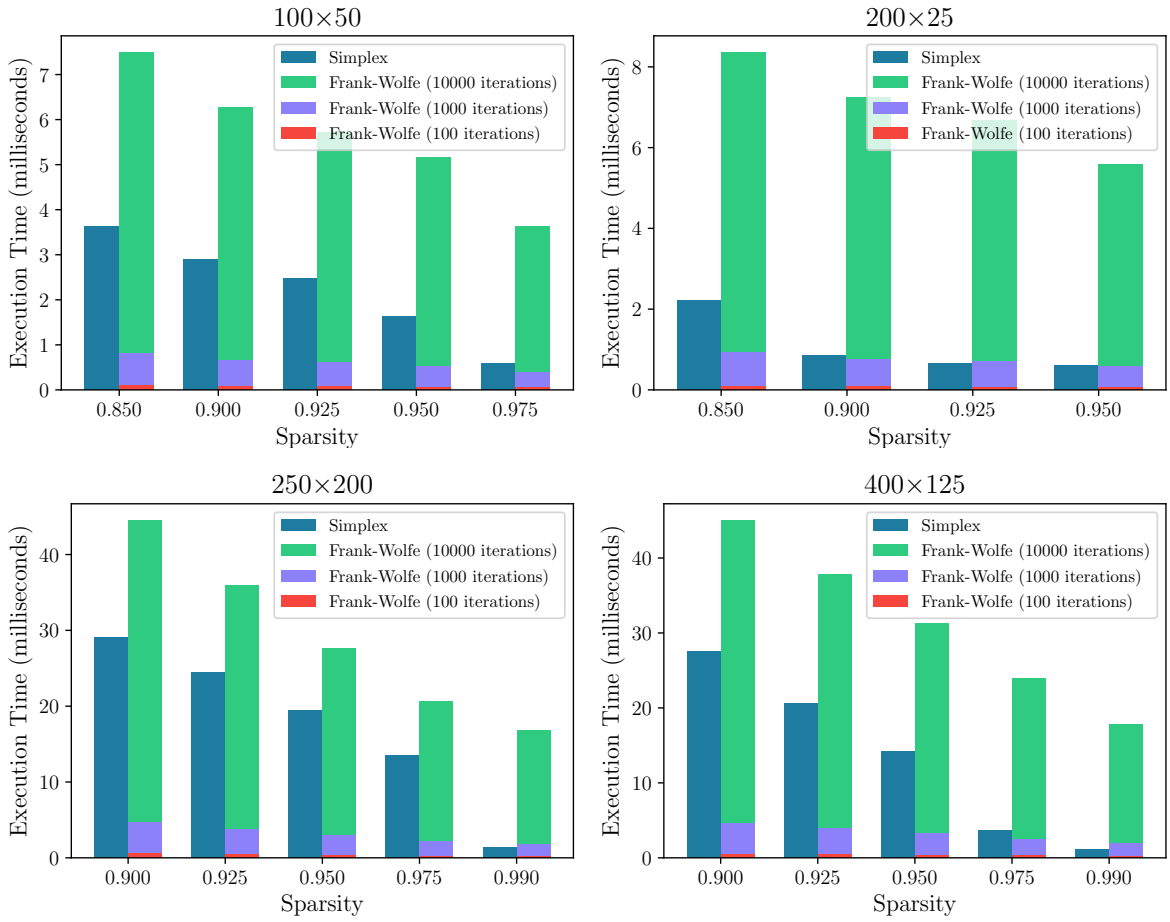


Figura 4.7: Tempo di esecuzione dell'algoritmo del simplesso e dell'algoritmo di Frank-Wolfe per istanze di taglia piccola e media, al variare della sparsità e della forma della matrice di riferimento.

Per le istanze di taglia piccola, a parità di sparsità e di numero di elementi della matrice di riferimento, variare la forma influenza in modo significativo i tempi di esecuzione dell'algoritmo del simplesso ma non quelli relativi all'algoritmo di Frank-Wolfe, che rimangono invece molto più stabili. La variazione dei tempi di esecuzione dell'algoritmo del simplesso è particolarmente evidente quando i valori di sparsità sono elevati. In par-

ticolare, l'algoritmo del simplesso impiega meno tempo a risolvere le istanze che hanno un numero minore di variabili. Ad esempio, il tempo impiegato per risolvere le istanze 200×25 è mediamente inferiore a quello necessario per risolvere le istanze 100×50, nonostante il numero degli elementi sia lo stesso. Lo stesso vale per le matrici 250×200 e 400×125, anche se l'effetto è meno marcato, soprattuto per i valori di sparsità più bassi.

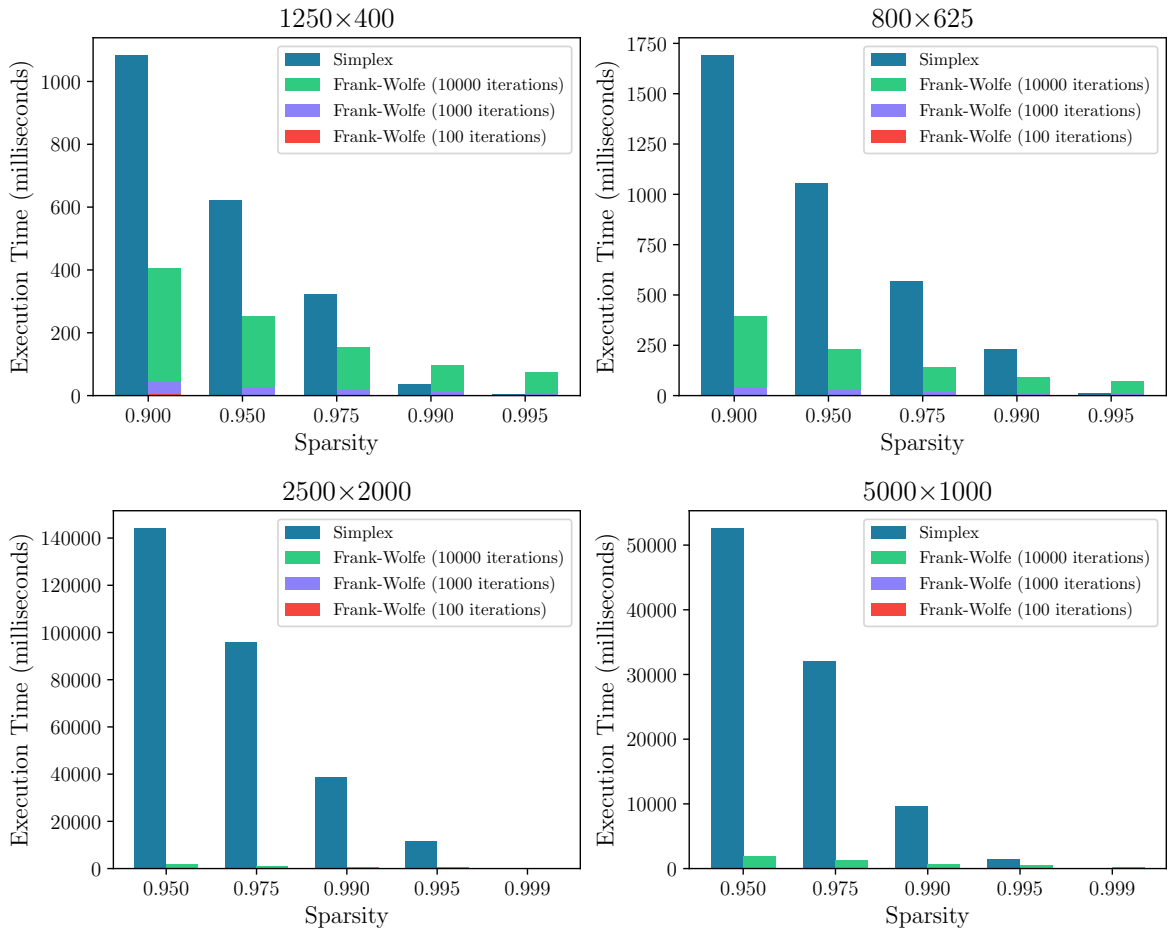


Figura 4.8: Tempo di esecuzione dell'algoritmo del simplesso e dell'algoritmo di Frank-Wolfe per istanze di taglia grande, al variare della sparsità e della forma della matrice di riferimento.

Per le matrici più grandi, il comportamento è lo stesso ma è più marcato. In particolare, i tempi dell'algoritmo del simplesso variano notevolmente a seconda della forma della matrice di riferimento. Ad esempio, per le matrici 2500×2000 e 5000×1000, i tempi di esecuzione sono molto differenti. L'effetto è uguale ma meno marcato per le matrici 800×625 e 1250×400. Di conseguenza, più è grande la matrice, e maggiore sarà l'influenza della sua forma sui tempi di esecuzione dell'algoritmo del simplesso. Anche in questo caso, l'algoritmo di Frank-Wolfe ha tempi molto più stabili. Inoltre, i tempi di Frank-Wolfe sono così ridotti rispetto a quelli del simplesso che per matrici sufficientemen-

te grandi non sono neanche visibili nei grafici. In questi casi l'algoritmo di Frank-Wolfe riesce comunque a trovare delle buone limitazioni, con il vantaggio di impiegare molto meno tempo rispetto all'algoritmo del simplesso.

Infine, consideriamo le istanze caratterizzate da matrici di riferimento simmetriche, i cui risultati sono riassunti nei grafici della Figura 4.9 e riportati nella Tabella 4.9.

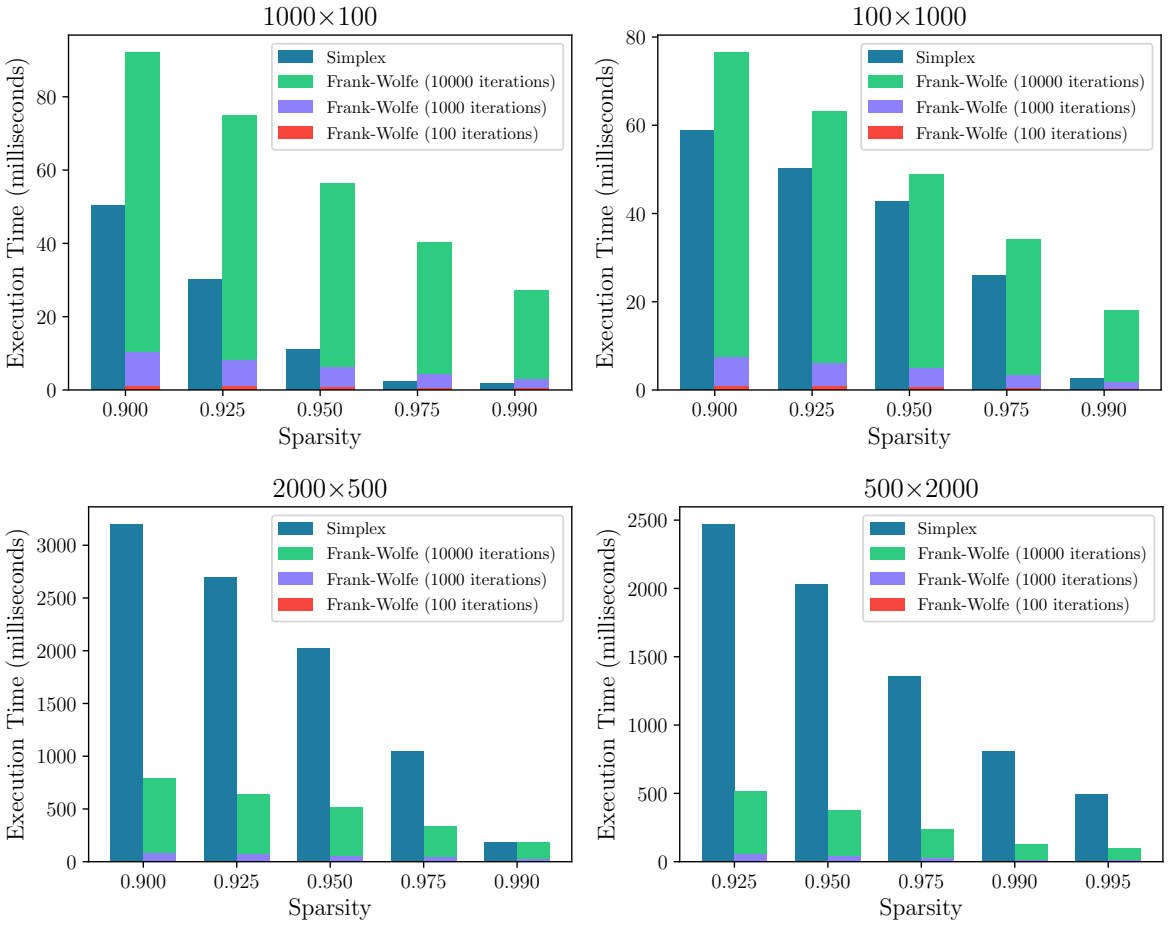


Figura 4.9: Tempo di esecuzione dell'algoritmo del simplesso e dell'algoritmo di Frank-Wolfe per istanze di taglia grande, al variare della sparsità e della forma della matrice di riferimento.

Osserviamo che i tempi di esecuzione dell'algoritmo di Frank-Wolfe sono migliori per le matrici che hanno un numero di variabili dominante rispetto al numero dei vincoli, mentre l'algoritmo del simplesso ha invece un comportamento più irregolare.

		Limitazione Inferiore			Limitazione Superiore			Tempo Frank-Wolfe (ms)			
Matrice	Sparsità	100	1000	10000	100	1000	10000	100	1000	10000	Tempo Simplesso (ms)
100×50	0.850	0.8075	0.8300	0.8321	1.7883	1.5568	1.4179	0.0893	0.8044	7.4851	3.6191
	0.900	0.8004	0.8148	0.8159	1.4754	1.3387	1.2667	0.0811	0.6531	6.2725	2.8984
	0.925	0.7909	0.8008	0.8018	1.2999	1.1949	1.1581	0.0682	0.5996	5.7267	2.4767
	0.950	0.8122	0.8201	0.8206	1.1685	1.0866	1.0625	0.0607	0.5266	5.1577	1.6346
	0.975	0.9309	0.9341	0.9345	1.0207	1.0109	1.0109	0.0455	0.3835	3.6199	0.5816
200×25	0.850	0.6809	0.6981	0.6991	1.2374	1.1422	1.1061	0.0923	0.9169	8.3504	2.2107
	0.900	0.7347	0.7461	0.7467	1.0633	1.0173	1.0035	0.0786	0.7532	7.2365	0.8551
	0.925	0.8204	0.8297	0.8300	1.0228	1.0021	1.0000	0.0732	0.6935	6.6782	0.6583
	0.950	0.9466	0.9522	0.9524	1.0000	1.0000	1.0000	0.0687	0.5865	5.5813	0.6166
250×200	0.900	0.8502	0.8974	0.9030	2.8143	2.3623	2.2692	0.5373	4.7315	44.4727	29.1073
	0.925	0.8477	0.8900	0.8946	2.5181	2.2540	2.0996	0.4176	3.7669	35.9471	24.5079
	0.950	0.8478	0.8777	0.8807	2.1247	1.9606	1.8297	0.3296	2.9376	27.6306	19.3998
	0.975	0.8262	0.8401	0.8416	1.5418	1.4524	1.3977	0.2566	2.2062	20.6300	13.4564
	0.990	0.8658	0.8699	0.8706	1.1950	1.1459	1.1137	0.2027	1.7696	16.8272	1.4226
400×125	0.900	0.7740	0.8207	0.8251	2.5543	2.3076	2.2024	0.5037	4.6682	44.9879	27.5289
	0.925	0.7712	0.8106	0.8134	2.2316	2.0595	1.9619	0.4126	3.9644	37.8337	20.6659
	0.950	0.7666	0.7891	0.7904	1.8273	1.6992	1.6352	0.3409	3.2158	31.3210	14.1838
	0.975	0.7541	0.7667	0.7677	1.3465	1.2681	1.2408	0.2768	2.4524	23.9097	3.6768
	0.990	0.9168	0.9208	0.9213	1.0490	1.0490	1.0490	0.2201	1.8997	17.8531	1.1396

Tabella 4.7: Qualità delle limitazioni (valori normalizzati rispetto all'ottimo ottenuto con l'algoritmo del simplesso) di Frank-Wolfe e tempo di esecuzione impiegato per ottenerle, al variare del numero di iterazioni e della forma della matrice di riferimento.

Matrice	Sparsità	Limitazione Inferiore			Limitazione Superiore			Tempo Frank-Wolfe (ms)			Tempo Simplesso (ms)
		100	1000	10000	100	1000	10000	100	1000	10000	
800 × 625	0.900	0.8184	0.9150	0.9270	4.1312	3.6840	3.4339	4.5415	44.5577	395.634	1693.215
	0.950	0.8327	0.9037	0.9115	3.4963	3.1305	2.9875	2.5951	26.1331	230.076	1057.753
	0.975	0.8444	0.8912	0.8961	2.7755	2.5332	2.4472	1.6854	15.1159	140.394	566.465
	0.990	0.8361	0.8559	0.8576	1.8471	1.7506	1.7032	1.1039	9.7862	91.465	228.820
	0.995	0.8247	0.8335	0.8345	1.4170	1.3630	1.3357	0.8481	7.4766	69.661	12.075
1250 × 400	0.900	0.7559	0.8695	0.8806	4.1034	3.7554	3.5263	4.4671	43.4733	406.711	1084.810
	0.950	0.7756	0.8451	0.8519	3.3665	3.0883	2.9520	2.7899	27.3705	252.102	621.661
	0.975	0.7768	0.8156	0.8198	2.4684	2.3355	2.2635	1.7517	16.4673	155.664	321.928
	0.990	0.7536	0.7690	0.7704	1.5636	1.5005	1.4694	1.1264	10.4208	97.371	37.720
	0.995	0.8002	0.8088	0.8097	1.2343	1.2343	1.2328	0.8521	7.8722	74.626	3.426
2500 × 2000	0.950	0.7961	0.9216	0.9372	5.1203	4.5322	4.2721	18.7118	178.5077	1770.390	144365.114
	0.975	0.8211	0.9139	0.9245	4.2592	3.8775	3.6957	11.3902	103.9968	1022.290	95927.598
	0.990	0.8420	0.8985	0.9044	3.1200	2.9310	2.8412	6.8700	62.3454	608.851	38566.928
	0.995	0.8478	0.8821	0.8855	2.3680	2.2480	2.2067	5.3173	48.9663	471.568	11814.630
	0.999	0.8654	0.8707	0.8713	1.2712	1.2583	1.2427	2.8082	25.5904	237.038	9.312
5000 × 1000	0.950	0.7182	0.8530	0.8674	4.9974	4.5726	4.4043	20.5998	197.0393	1911.559	52680.412
	0.975	0.7409	0.8262	0.8347	3.9821	3.7235	3.6208	13.0027	121.8646	1239.010	32048.006
	0.990	0.7378	0.7818	0.7858	2.5930	2.5263	2.4800	8.6056	80.4950	761.756	9718.694
	0.995	0.7253	0.7464	0.7482	1.7859	1.7366	1.7041	6.6523	61.8239	586.782	1440.216
	0.999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	2.0833	19.2386	168.014	7.510

Tabella 4.8: Qualità delle limitazioni (valori normalizzati rispetto all'ottimo ottenuto con l'algoritmo del simplesso) di Frank-Wolfe e tempo di esecuzione impiegato per ottenerle, al variare del numero di iterazioni e della forma della matrice di riferimento.

		Limitazione Inferiore			Limitazione Superiore			Tempo Frank-Wolfe (ms)			
Matrice	Sparsità	100	1000	10000	100	1000	10000	100	1000	10000	Tempo Simplesso (ms)
1000×100	0.900	0.6838	0.7278	0.7309	2.4425	2.2732	2.1281	1.0560	10.1338	92.2318	50.4139
	0.925	0.6804	0.7122	0.7146	2.0360	1.9230	1.8369	0.8351	8.1567	74.9928	30.0877
	0.950	0.6692	0.6912	0.6927	1.5582	1.4778	1.4341	0.6365	6.1303	56.3676	11.0046
	0.975	0.7410	0.7568	0.7575	1.1044	1.1038	1.0966	0.4589	4.3189	40.2606	2.2546
	0.990	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.3146	2.9617	27.2564	1.8022
100×1000	0.900	0.9270	0.9712	0.9752	2.7050	2.2530	2.0232	0.8954	7.4488	76.6151	58.8196
	0.925	0.9325	0.9708	0.9745	2.5234	2.1253	1.9730	0.7292	6.0458	63.0342	50.1840
	0.950	0.9411	0.9715	0.9743	2.2981	1.9397	1.7879	0.5587	4.7717	48.8442	42.7688
	0.975	0.9531	0.9690	0.9703	1.8491	1.6200	1.5382	0.3999	3.3857	34.2001	25.9526
	0.990	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.1775	1.7352	18.0140	2.6631
2000×500	0.900	0.7385	0.8657	0.8787	4.6149	4.1923	3.9683	8.1339	79.8394	788.0149	3203.9266
	0.925	0.7417	0.8545	0.8658	4.2667	3.9206	3.7215	6.8496	69.1491	639.2950	2693.2720
	0.950	0.7568	0.8387	0.8465	3.7923	3.4758	3.3291	5.2936	50.9565	519.3931	2027.2962
	0.975	0.7568	0.8098	0.8145	2.8693	2.6947	2.6113	3.6144	38.0744	336.7734	1045.2121
	0.990	0.7432	0.7635	0.7652	1.7597	1.6977	1.6525	2.1082	20.1108	188.2672	187.5982
500×2000	0.925	0.8667	0.9579	0.9679	4.1039	3.4251	3.1738	5.3659	51.5759	512.6423	2473.6085
	0.950	0.8836	0.9583	0.9662	3.7045	3.1601	3.0080	3.9270	38.4495	379.7790	2030.5913
	0.975	0.9061	0.9578	0.9635	3.0555	2.7320	2.5654	2.5957	23.9551	239.8961	1358.5758
	0.990	0.9272	0.9552	0.9581	2.2804	2.0995	2.0162	1.5630	12.9429	129.9060	807.6202
	0.995	0.9256	0.9388	0.9402	1.7528	1.6508	1.6046	1.2493	9.2556	95.7591	490.6100

Tabella 4.9: Qualità delle limitazioni (valori normalizzati rispetto all'ottimo ottenuto con l'algoritmo del simplesso) di Frank-Wolfe e tempo di esecuzione impiegato per ottenerle, al variare del numero di iterazioni e della forma della matrice di riferimento.

4.4 Convergenza dell'algoritmo di Frank-Wolfe

Concludiamo l'analisi dell'algoritmo di Frank-Wolfe studiando l'esecuzione più in dettaglio. In particolare, vogliamo capire come cambia la qualità delle limitazioni prodotte nelle varie iterazioni quando facciamo variare i parametri che identificano le istanze.

4.4.1 Dimensione della matrice di riferimento

I grafici della Figura 4.10 mostrano l'andamento delle limitazioni prodotte da Frank-Wolfe in funzione del numero delle iterazioni effettuate, al variare del numero di elementi della matrice di riferimento delle istanze.

Osserviamo che l'andamento delle limitazioni prodotte è lo stesso per tutte le istanze che abbiamo considerato, a prescindere dalla dimensione della matrice di riferimento. In particolare, in tutti i casi l'algoritmo ha una fase iniziale in cui le limitazioni prodotte sono pessime, seguita da un rapido miglioramento della qualità di quest'ultime. Infine, le limitazioni si stabilizzano e l'aumento del numero delle iterazioni non è più un fattore così determinante per la qualità, che migliora molto lentamente.

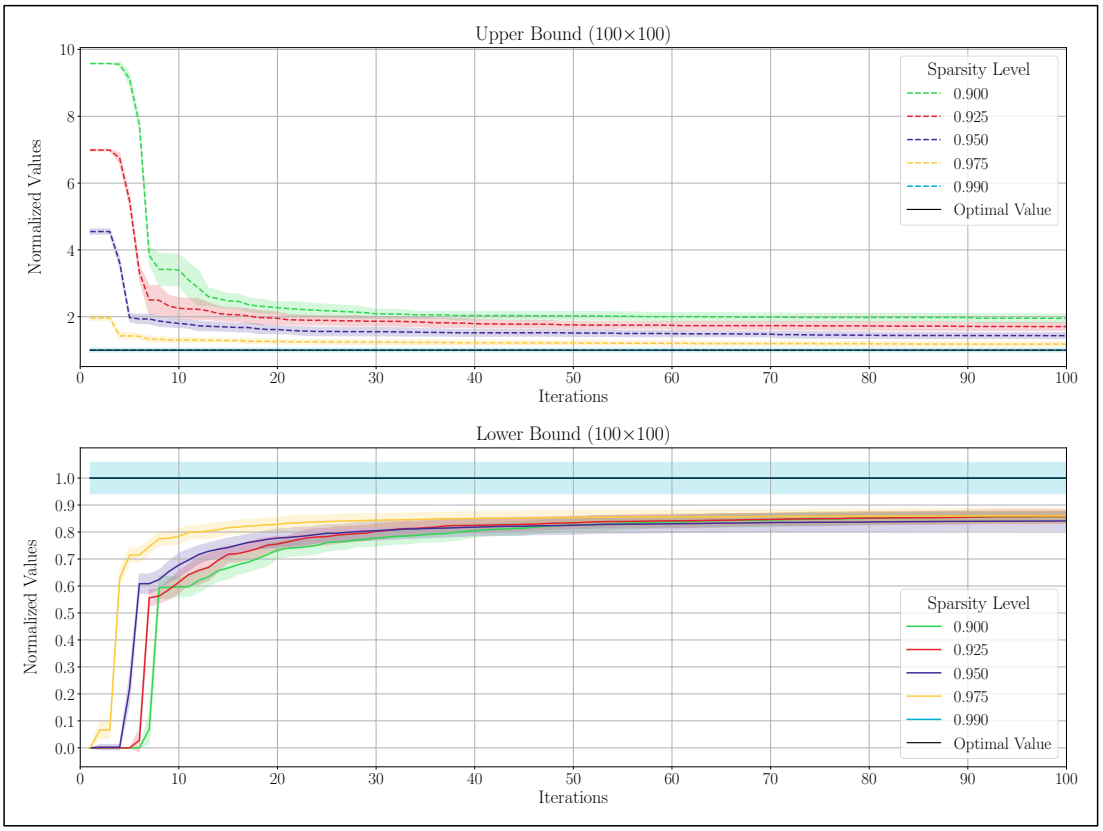
Per matrici di dimensione fissata, l'aumentare del valore di sparsità è indicatore di un minor numero di iterazioni richiesto per raggiungere la fase di rapido miglioramento della qualità delle limitazioni prodotte.

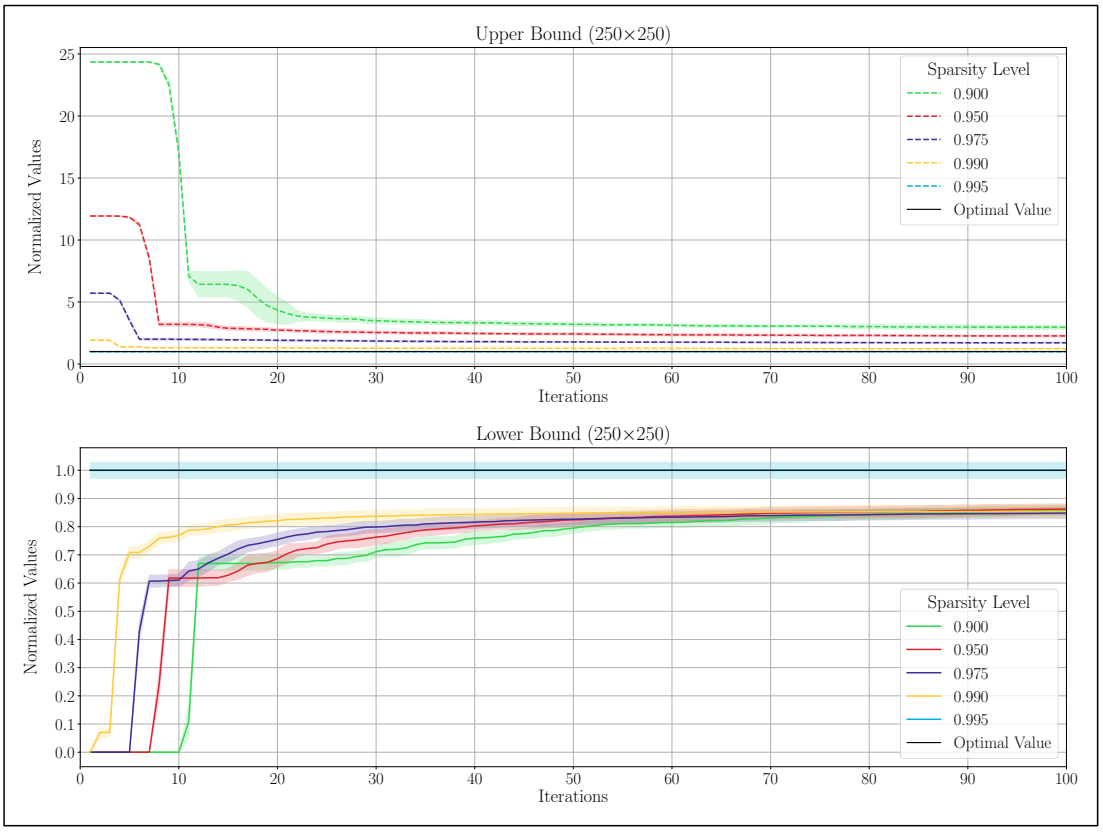
L'aumentare della dimensione della matrice di riferimento, a parità di numero delle iterazioni, è indicatore di un peggioramento nella qualità delle limitazioni prodotte. In altre parole, matrici più grandi richiedono un numero di iterazioni maggiore per produrre limitazioni di buona qualità rispetto alle matrici più piccole.

4.4.2 Forma della matrice di riferimento

I grafici della Figura 4.11 mostrano l'andamento delle limitazioni prodotte da Frank-Wolfe in funzione del numero delle iterazioni effettuate, al variare della forma della matrice di riferimento delle istanze.

Notiamo ancora una volta che la forma influenza notevolmente l'esecuzione dell'algoritmo. Ad esempio, le matrici 1000×100 producono limitazioni inferiori peggiori rispetto alle matrici 100×1000 e richiedono anche un numero di iterazioni maggiore. Lo stesso vale per le matrici 2000×500 e 500×2000. Per le limitazioni superiori la situazione è differente. Le istanze caratterizzate da un numero di variabili dominante rispetto al numero dei vincoli producono, nelle prime iterazioni, limitazioni superiori molto più alte rispetto alle istanze in cui è il numero dei vincoli a dominare. Tuttavia, la fase di miglioramento è più marcata e questo permette di compensare la differenza di qualità in poche iterazioni.


Upper Bound (100×100)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value
Lower Bound (100×100)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value


Upper Bound (250×250)
Normalized Values
Iterations
Sparsity Level
0.900
0.950
0.975
0.990
0.995
Optimal Value
Lower Bound (250×250)
Normalized Values
Iterations
Sparsity Level
0.900
0.950
0.975
0.990
0.995
Optimal Value

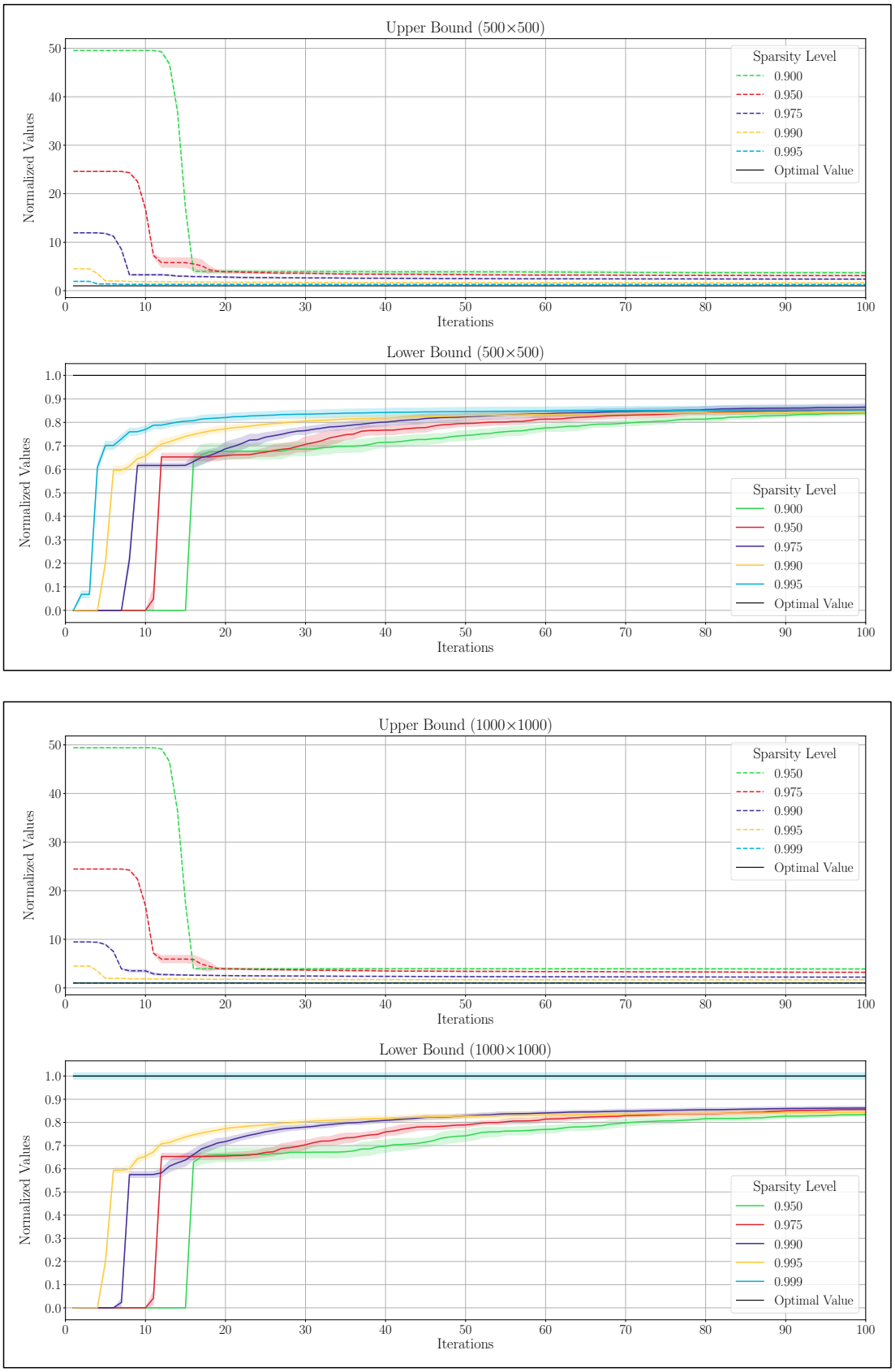
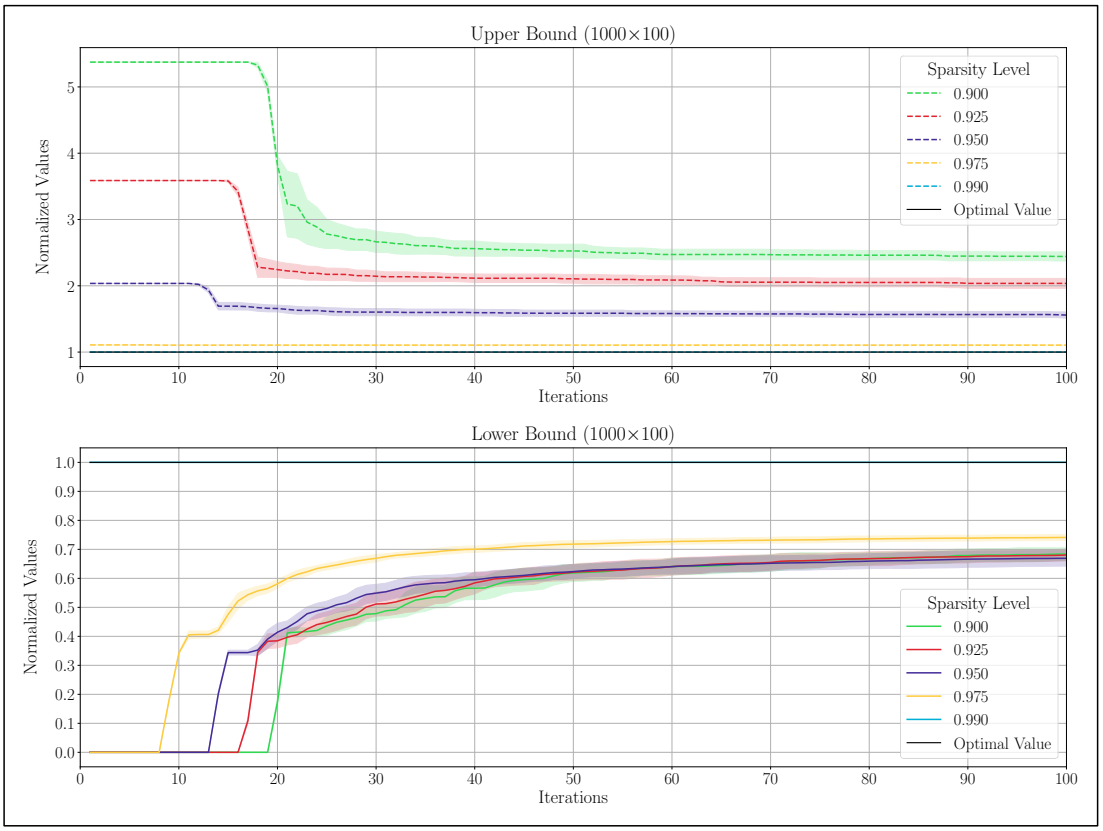
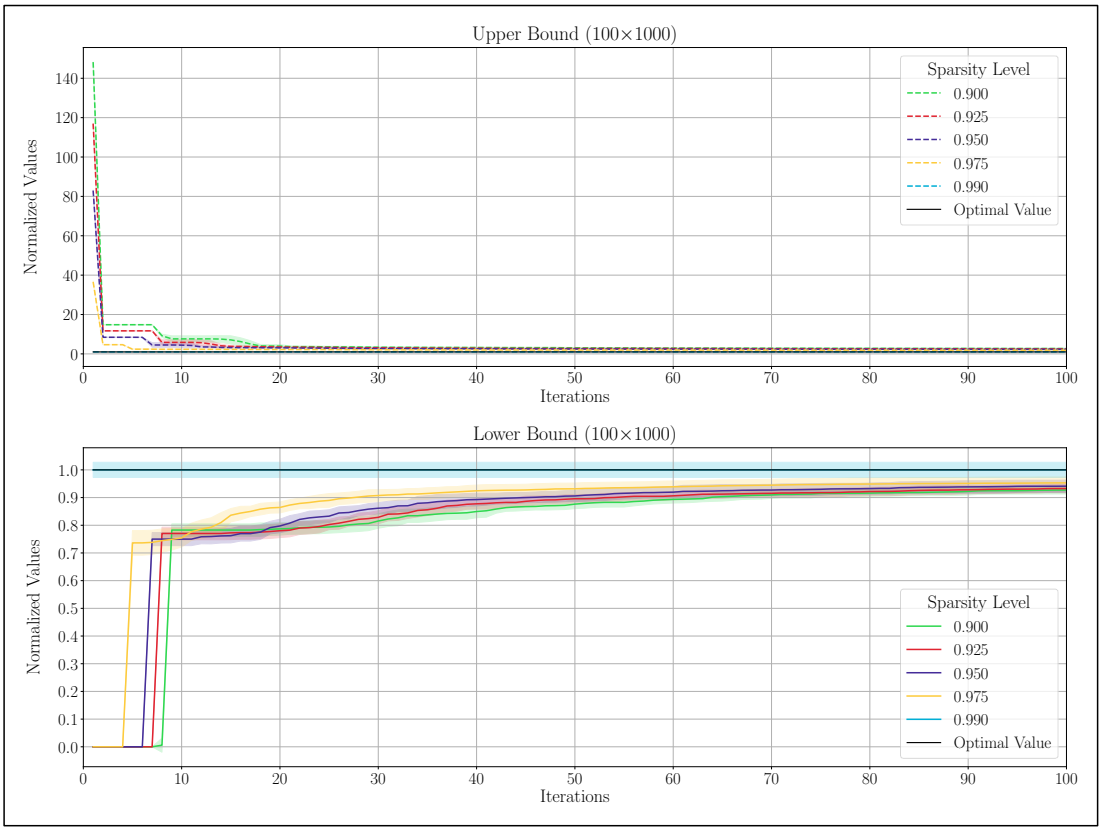


Figura 4.10: Limitazioni prodotte da Frank-Wolfe al variare del numero di elementi della matrice di riferimento delle istanze.


Upper Bound (1000×100)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value
Lower Bound (1000×100)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value


Upper Bound (100×1000)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value
Lower Bound (100×1000)
Normalized Values
Iterations
Sparsity Level
0.900
0.925
0.950
0.975
0.990
Optimal Value

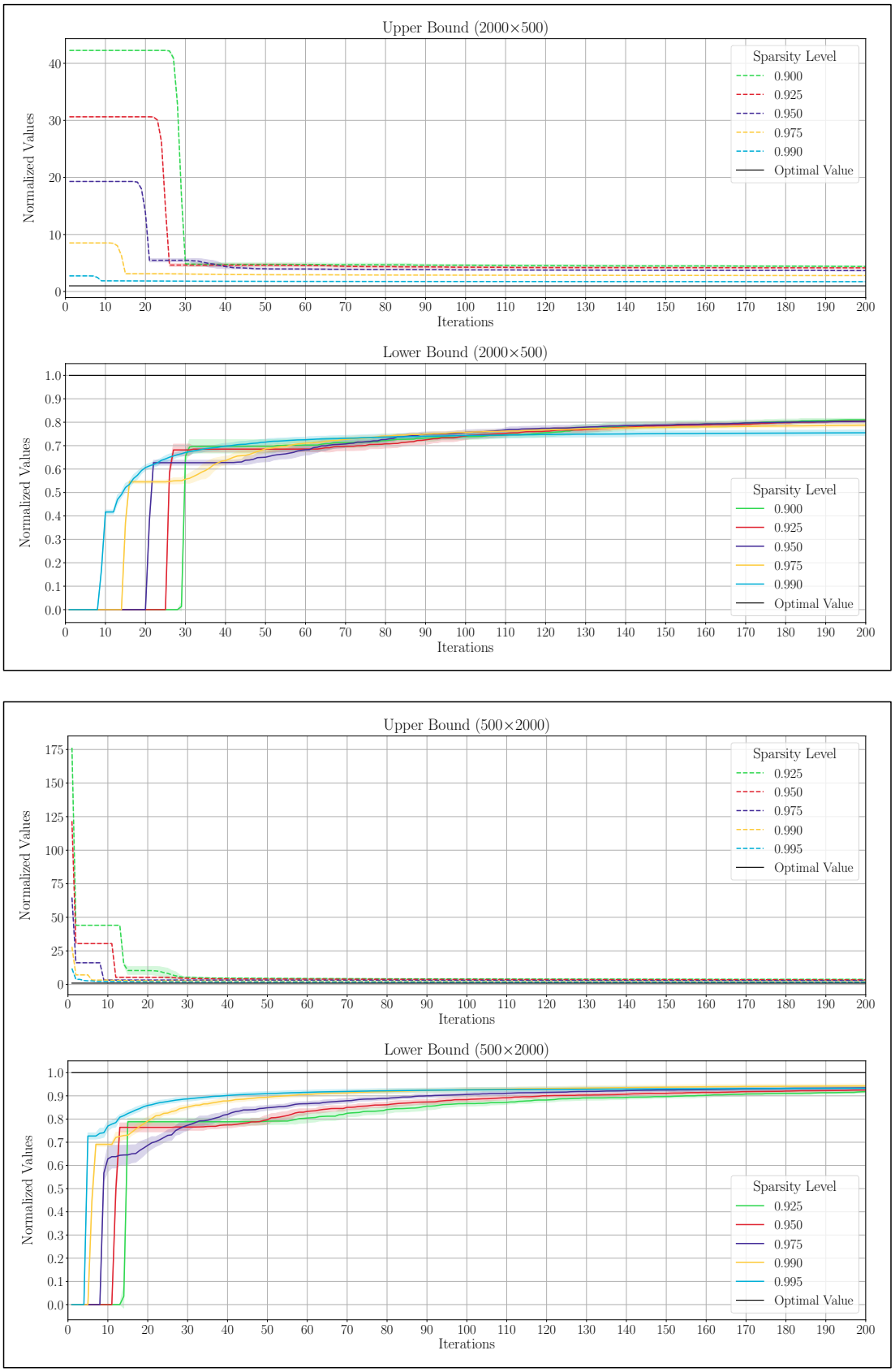


Figura 4.11: Limitazioni prodotte da Frank-Wolfe al variare della forma della matrice di riferimento delle istanze.

4.5 Conclusioni

In questo lavoro siamo partiti dalla formulazione del problema del set-covering e abbiamo utilizzato l'idea alla base del metodo di Frank-Wolfe per sviluppare l'implementazione di un algoritmo risolutivo per il rilassamento lineare. L'obiettivo era quello di ottenere delle limitazioni ragionevoli al valore ottimo in un tempo molto ridotto. Abbiamo utilizzato l'algoritmo del simplesso per ottenere le soluzioni ottime da utilizzare come riferimento per misurare la qualità delle limitazioni prodotte da Frank-Wolfe e abbiamo operato un confronto sui tempi di esecuzione per valutare la competitività dell'algoritmo che abbiamo implementato.

L'implementazione dell'algoritmo risolutivo è realizzata in linguaggio C e sfrutta la rappresentazione CSR per identificare la matrice binaria di riferimento di un'istanza senza la necessità di memorizzarla per intero.

Le numerose prove sperimentali che abbiamo svolto ci hanno permesso di ottenere risultati da utilizzare per analizzare il comportamento dell'algoritmo di Frank-Wolfe e confrontarlo con l'algoritmo del simplesso. Per ottenere risultati completi abbiamo ripetuto gli esperimenti variando tutti i parametri che caratterizzano le istanze del problema, come numero di elementi, sparsità e forma della matrice di riferimento. Per ottenere risultati accurati, abbiamo ripetuto gli esperimenti su molteplici istanze dello stesso tipo.

Analizzando i risultati sperimentali relativi alla qualità delle limitazioni prodotte dall'algoritmo di Frank-Wolfe abbiamo capito che parametri come sparsità e forma della matrice di riferimento hanno un'influenza significativa. Le limitazioni prodotte da Frank-Wolfe sono abbastanza buone in generale, con qualche eccezione per matrici caratterizzate da valori particolari per i parametri che abbiamo considerato. Inoltre, le limitazioni inferiori sono generalmente più stabili di quelle superiori quando variano i parametri.

I risultati relativamente ai tempi di esecuzione dell'algoritmo di Frank-Wolfe, che abbiamo confrontato con i tempi di esecuzione dell'algoritmo del simplesso, hanno evidenziato la competitività di Frank-Wolfe nella maggior parte delle situazioni, soprattutto per matrici meno sparse, con cui l'algoritmo del simplesso fa più fatica. Per le istanze caratterizzate da matrici di riferimento molto sparse, l'algoritmo di Frank-Wolfe è competitivo solo quando il numero delle iterazioni è molto ridotto. Tuttavia, questo non sembra avere effetti troppo negativi sulla qualità delle limitazioni prodotte. In effetti, l'analisi dell'esecuzione dell'algoritmo di Frank-Wolfe ha evidenziato che le prime 100 iterazioni sono molto più importanti di quelle successive, nelle quali il miglioramento delle limitazioni è trascurabile, soprattutto per le limitazioni inferiori delle matrici più piccole. Utilizzare un numero elevato di iterazioni ha senso solo quando le matrici sono particolarmente grandi oppure quando sono caratterizzate da un valore di sparsità abbastanza basso.

Bibliografia

[1] Domenico Salvagnin. Dispense dal corso “Modelli & software per l’ottimizzazione discreta”. Università degli Studi di Padova. Anno Accademico 2023-2024.

[2] Domenico Salvagnin. Frank-Wolfe Method for Set-Covering LPs. Università degli Studi di Padova. 2023.

[3] Python Programming Language. Versione 3.11.7. `https://www.python.org`.

[4] NumPy Python Package. Versione 1.26.4. `https://numpy.org`.

[5] SciPy Python Package. Versione 1.11.4. `https://scipy.org`.

[6] Matplotlib Python Package. Versione 3.9.1. `https://matplotlib.org`.

[7] PySCIPOpt Interface. Versione 4.4.0. `https://github.com/scipopt/PySCIPOpt`.

[8] GCC GNU Compiler. Versione 13.2.0. `https://gcc.gnu.org`.

[9] Leonardo Callegari. Frank-Wolfe Method For Set-Covering LPs. `https://github.com/lcalllegari1/Frank-Wolfe_Method_For_Set_Covering_LPs`