UNIVERSITÀ DEGLI STUDI DI TORINO

COMBINATORIAL OPTIMIZATION PROJECT

Modelli di programmazione lineare intera di dimensioni non polinomiali

COLUMN GENERATION PER IL 1D CUTTING STOCK PROBLEM

Frigato Luca

2023

Contents

Chapter 1 Introduction

La presente tesina esplorerà due metodologie avanzate utilizzate per affrontare problemi di dimensioni non polinomiali all'interno del campo della programmazione lineare intera. In particolare, si approfondiranno le strategie di column generation e l'approccio ad euristiche.

La programmazione lineare intera (Mixed-Integer Linear Programming, MILP) è un campo della ricerca operativa che si occupa di risolvere problemi di ottimizzazione in cui alcune o tutte le variabili decisionali devono assumere valori interi. Questo tipo di problemi presenta sfide aggiuntive rispetto alla programmazione lineare continua, poiché la natura discreta delle variabili comporta una complessità computazionale significativamente maggiore.

Due approcci ampiamente utilizzati per affrontare problemi di MILP di dimensioni non polinomiali sono la column generation e i modelli con piani di taglio. La column generation è una strategia iterativa che si concentra sulla generazione progressiva di soluzioni parziali chiamate colonne, che vengono successivamente integrate nel problema principale. Questo approccio è particolarmente efficace quando il numero di colonne possibili è molto grande, riducendo così la dimensione del problema e accelerando il processo di risoluzione.

D'altra parte, i modelli con piani di taglio si basano sulla generazione di vincoli addizionali, noti come piani di taglio, che limitano la regione ammissibile delle soluzioni. Questi piani di taglio vengono aggiunti al rilassamento lineare del problema, migliorandone la formulazione o eliminando soluzioni non ammissibili. L'obiettivo principale è ottenere un rilassamento lineare più stretto, che si avvicini di più alla soluzione intera ottima.

Nel contesto di questa tesina, verrà preso in considerazione il problema del 1D-CUT-AXES, un problema di taglio di materiali su assi unidimensionali. Sarà presentata un'implementazione pratica della column generation utilizzando il linguaggio di programmazione Python e la libreria PuLP, che semplifica la modellazione e la risoluzione dei problemi di programmazione lineare intera. Inoltre, si esplorerà l'applicazione del Genetic Algorithm come problema Slave generatore di colonne (pattern) per il 1D-CUT-AXES.

Chapter 2 MILP

La programmazione lineare intera (MILP) è una tecnica di ottimizzazione che si occupa di problemi in cui alcune o tutte le variabili decisionali devono assumere valori interi. A differenza della programmazione lineare continua, in cui le variabili possono assumere qualsiasi valore all'interno di un intervallo continuo, nella MILP le variabili devono essere vincolate ad assumere valori interi, come 0, 1, 2, ecc.

Nella formulazione di un problema di MILP, l'obiettivo è quello di minimizzare o massimizzare una funzione lineare soggetta a una serie di vincoli lineari, in cui alcune variabili possono essere intere.

Obiettivo:

$$\min c^T x$$

dove:

- c è il vettore dei coefficienti della funzione obiettivo
- x è il vettore delle variabili decisionali

Vincoli:

$$\begin{aligned} Ax &\leq b \\ A_{eq}x &= b_{eq} \\ x &\in Z^n \end{aligned}$$

dove:

- A è la matrice dei coefficienti dei vincoli di disuguaglianza
- b è il vettore dei termini noti dei vincoli di disuguaglianza
- A_{eq} è la matrice dei coefficienti dei vincoli di uguaglianza

- b_{eq} è il vettore dei termini noti dei vincoli di uguaglianza
- Z^n rappresenta l'insieme delle soluzioni intere per le variabili x

Questa formulazione rappresenta un problema di minimizzazione, ma può essere adattata per un problema di massimizzazione semplicemente invertendo il segno dei coefficienti della funzione obiettivo.

È importante notare che la presenza della condizione $x \in Z^n$ indica che le variabili decisionali devono assumere valori interi anziché continui.

Utilizzando questa rappresentazione matematica, è possibile definire un'ampia gamma di problemi di programmazione lineare intera, adattandoli alle specifiche esigenze e ai vincoli di un determinato dominio di applicazione.

2.1 Sfide

I problemi MILP presentano diverse sfide e problematiche nella loro risoluzione. Alcune delle caratteristiche che rendono i problemi MILP complessi includono la presenza di variabili intere, la grande dimensione del problema e la complessità dei vincoli.

Per risolvere i problemi MILP, sono disponibili diversi metodi di risoluzione. Tuttavia, i metodi tradizionali come l'enumerazione esaustiva e il branch and bound possono presentare limiti nella gestione di problemi di grandi dimensioni e con complessità elevata. L'enumerazione esaustiva, ad esempio, prova tutte le possibili combinazioni di valori delle variabili, il che può richiedere un tempo eccessivo per problemi di grandi dimensioni. Il branch and bound, invece, divide il problema in sotto-problemi più piccoli, ma anche questo metodo può essere inefficiente per problemi molto grandi.

I metodi tradizionali di risoluzione, come l'enumerazione esaustiva e il branch and bound, possono diventare computazionalmente costosi e inefficaci per problemi MILP di grandi dimensioni e complessi.

Inoltre, i problemi MILP possono essere NP-difficili, il che significa che non esiste un algoritmo efficiente in grado di risolvere il problema in tempi polinomiali. Ciò rende ancora più difficile trovare la soluzione ottimale per questi problemi.

2.2 NP-Hard

La complessità computazionale è un concetto fondamentale nella risoluzione dei problemi di programmazione lineare intera mista (MILP). La classe di complessità NP-hard rappresenta un insieme di problemi di ottimizzazione con complessità intrinsecamente alta che non possono essere risolti in tempo polinomiale.

I problemi MILP possono essere classificati come problemi NP-hard, il che significa che la loro soluzione ottimale richiede un tempo di calcolo esponenziale rispetto alla dimensione del problema. Questo rende difficile trovare la soluzione ottimale per problemi di grandi dimensioni in tempi ragionevoli. Tuttavia, ci sono anche problemi MILP che possono essere risolti in tempo polinomiale utilizzando metodi di risoluzione esatti.

2.3 Metodi di risoluzione

I problemi MILP possono essere risolti utilizzando due principali categorie di metodi di risoluzione: esatti e approssimati.

Metodi di risoluzione esatti: I metodi di risoluzione esatti cercano di trovare la soluzione ottimale del problema MILP. Alcuni esempi di questi metodi sono il simplesso intero e la programmazione a branch and cut. Tuttavia, questi metodi possono essere lenti e inefficienti per problemi di grandi dimensioni e complessi.

Il simplesso intero estende l'algoritmo del simplesso, utilizzato per la programmazione lineare continua, per gestire anche variabili intere. La programmazione a branch and cut sfrutta la programmazione lineare a numeri interi e tecniche di taglio per ridurre lo spazio di ricerca del problema.

Metodi di risoluzione approssimati I metodi di risoluzione approssimati cercano di trovare una soluzione vicina all'ottimo, ma non garantiscono la soluzione esatta. Alcuni esempi di questi metodi sono l'algoritmo genetico, la ricerca locale, l'ottimizzazione ad algoritmi paralleli e la generazione di colonne.

L'algoritmo genetico è un esempio di un metodo di risoluzione approssimato che utilizza tecniche di evoluzione biologica per cercare una soluzione vicina all'ottimo. Questo metodo funziona bene per problemi di grandi dimensioni e complessi, ma non garantisce la soluzione ottimale.

La ricerca locale è un altro metodo approssimato che cerca di migliorare la soluzione in modo iterativo. Questo metodo funziona bene per problemi con molti minimi locali, ma può essere inefficace per problemi di grandi dimensioni.

L'ottimizzazione ad algoritmi paralleli (PAO) è un metodo approssimato che utilizza tecniche di parallelizzazione per migliorare la velocità di calcolo. Questo metodo funziona bene per problemi di grandi dimensioni che richiedono molta potenza di calcolo.

Inoltre, la generazione di colonne è un altro metodo di risoluzione utilizzato per i problemi MILP NP-hard. Questo metodo utilizza la programmazione lineare continua per generare nuove variabili che possono essere aggiunte al problema originale, fino a quando la soluzione ottimale non è trovata. Anche se la generazione di colonne può essere considerata un metodo di risoluzione approssimato, può essere utilizzata in combinazione con la programmazione a branch and cut per risolvere i problemi MILP NP-hard in modo esatto.

Esempi di applicazione di queste tecniche sono in diversi contesti e settori industriali. Ad esempio, la programmazione a branch and cut viene spesso utilizzata per la pianificazione della produzione nel settore manifatturiero, mentre l'algoritmo genetico viene utilizzato per la pianificazione dei percorsi di trasporto nel settore della logistica. La ricerca locale è spesso utilizzata per la progettazione del layout delle fabbriche e l'ottimizzazione ad algoritmi paralleli viene utilizzata per la pianificazione delle risorse nell'industria degli impianti di produzione.

2.4 Metodi dei piani di taglio

L'approccio ai metodi dei piani di taglio è una tecnica utilizzata per risolvere problemi di programmazione lineare intera. Questi problemi coinvolgono variabili decisionali che devono assumere valori interi anziché continui.

Uno dei metodi più noti nel campo dei piani di taglio è il metodo di **Gomory**, sviluppato da Ralph E. Gomory negli anni '60. Questo metodo si basa sull'idea di generare tagli aggiuntivi per eliminare le soluzioni non intere nel poliedro delle soluzioni ammissibili del problema di programmazione lineare intera.

Il primo passo del metodo di Gomory consiste nell'eseguire una soluzione del rilassamento lineare del problema, che permette alle variabili di assumere valori continui invece di essere vincolate ad essere intere. Questo fornisce una soluzione di base per il problema.

Successivamente, viene verificato se la soluzione ottenuta è intera o meno. Se la soluzione è intera, allora è anche la soluzione ottima del problema intero e il processo termina. Altrimenti, viene identificata una variabile non intera nella soluzione e viene creato un taglio per eliminare questa soluzione non ammissibile.

Il taglio di Gomory è un vincolo lineare aggiuntivo che è valido per il poliedro delle soluzioni ammissibili del problema di programmazione lineare intera. Questo taglio viene creato utilizzando una combinazione lineare dei coefficienti non interi nella soluzione corrente. Questo taglio viene quindi aggiunto come restrizione al problema e il processo viene ripetuto con il nuovo problema ottenuto.

l taglio di Gomory può essere espresso come:
$\sum_{j \in J} a_j x_j \leq b$

dove J è l'insieme degli indici delle variabili non intere nella soluzione corrente, aj è il coefficiente associato alla variabile xj e b è un termine noto determinato in base alla soluzione corrente.

Il criterio di selezione per determinare quale variabile non intera deve essere utilizzata per generare il taglio di Gomory dipende dal metodo specifico di implementazione.
In generale, è comune selezionare *la variabile con il coefficiente frazionario più grande.* Questo perché una variabile con **un coefficiente frazionario significativo nella soluzione indica una forte violazione della condizione di interezza** e quindi è candidata ad essere soggetta a un taglio.

Se più variabili hanno coefficienti frazionari significativi, è possibile selezionarne una in base a un criterio specifico o generare più tagli corrispondenti alle diverse variabili. La scelta del criterio dipende dal problema specifico e dalle considerazioni di implementazione.

Il processo di generazione di tagli e aggiunta di restrizioni continua fino a quando una soluzione intera ottima è trovata o il problema è dimostrato essere non ammissibile. Questo metodo può richiedere un numero arbitrario di iterazioni, ma *garantisce alla fine di trovare una soluzione intera ottima* se esiste.

Un altro approccio comune ai metodi dei piani di taglio è il metodo di **branch and bound**. Questo metodo sfrutta la proprietà della decomposizione del problema in sottoinsiemi più piccoli per ridurre la dimensione del problema e ottenere una soluzione ottima.

Il metodo di branch and bound funziona creando una struttura ad albero in cui ogni nodo rappresenta una soluzione parziale del problema. Il problema viene suddiviso in sotto-problemi più piccoli, chiamati nodi figli, attraverso la creazione di vincoli che limitano il dominio delle variabili decisionali. I nodi figli vengono poi esaminati in modo ricorsivo fino a quando una soluzione ottima intera non viene trovata o il problema è dimostrato essere non ammissibile o non migliorabile.

Il **branching** si verifica quando un nodo figlio viene generato dividendo il problema originale in sotto-problemi più piccoli. Questo viene fatto introducendo un vincolo aggiuntivo che limita il valore di una variabile decisionale a un valore specifico. Ad esempio, se una variabile x può assumere solo valori interi, il branching può essere eseguito introducendo un vincolo x k o x k+1, dove k è un valore intero.

Il **bound** viene calcolato per ogni nodo figlio per ottenere un limite superiore (upper bound) e inferiore (lower bound) sul valore della funzione obiettivo. Il bound superiore viene calcolato utilizzando una soluzione ammissibile del problema rilassato, in cui le variabili possono assumere valori continui. Il bound inferiore viene calcolato utilizzando una soluzione ammissibile del problema intero, in cui le variabili assumono solo valori interi.

Il **pruning** viene eseguito per eliminare i nodi figli che non possono generare soluzioni migliori rispetto a quelle già trovate. Questo viene fatto confrontando il bound superiore e inferiore del nodo figlio con il miglior bound superiore trovato finora. Se il bound superiore è maggiore o uguale al miglior bound superiore, allora il nodo figlio viene potato (pruned), poiché non può generare una soluzione migliore.

Chapter 3 Column Generation

3.1 Introduzione

La Column Generation è una tecnica che è stata sviluppata nel campo della programmazione lineare intera mista (MILP) per affrontare problemi complessi e di grandi dimensioni. È stata introdotta per la prima volta negli anni '60 da Richard Bellman e Robert E. Dreyfus per la risoluzione di problemi di pianificazione della produzione.

Il concetto di Column Generation si basa sulla decomposizione del problema in sottoproblemi più piccoli e gestibili. Invece di risolvere direttamente l'intero problema MILP, si concentra sulla generazione di una "colonna" (ovvero una nuova variabile decisionale) che può migliorare la soluzione corrente. Questa colonna viene quindi aggiunta al problema principale, che viene risolto nuovamente per trovare una soluzione migliorata.

L'idea alla base della Column Generation è che alcune variabili possono essere generate in modo efficiente e indipendente dalle altre. Quindi, invece di considerare tutte le possibili variabili del problema iniziale, si generano solo quelle che sono rilevanti per la soluzione ottimale. Questo approccio riduce significativamente la dimensione del problema e permette di concentrare l'attenzione sulle variabili più importanti.

Più formalmente, la Column Generation è una tecnica utilizzata per risolvere problemi NP-completi, che prevede la risoluzione di un rilassamento di programmazione lineare (LP) ristretto del problema originale e quindi l'aggiunta iterativa di nuove variabili alla formulazione LP fino a ottenere una soluzione ottimale.

Supponiamo di avere un problema NP-completo che può essere formulato come un programma lineare misto-integrale (MILP) nella seguente forma:

$$\begin{gathered} minimizza\ c^T x \\ soggetto\ a\ Ax = b \\ x \in Z^m \times R^n \end{gathered}$$

dove c è un vettore di costi, A è una matrice di vincoli, b è un vettore del lato destro e x è un vettore di variabili decisionali che assumono valori interi in qualche sottoinsieme di Z^m o valori continui in R^n.

L'approccio di column generation inizia risolvendo un rilassamento di programmazione lineare (LP) del problema MILP:

$$\begin{gathered} \textit{minimizza } c^T x \\ \textit{soggetto a } Ax = b \\ x \in R^m \times R^n \end{gathered}$$

dove x assume ora valori continui.
La soluzione di questo rilassamento LP fornisce un limite inferiore per la soluzione ottimale del problema MILP.

Successivamente, generiamo una nuova variabile (colonna) y utilizzando un sotto-problema specifico. Questa nuova variabile rappresenta un possibile modo per migliorare il valore della funzione obiettivo. La formula per la nuova variabile può essere rappresentata come segue:

$$y = \arg\max\{c_y - a_y^T \lambda\}$$

dove c_y è il coefficiente corrispondente alla nuova variabile y, a_y è il vettore delle colonne di A associate alla nuova variabile y, e λ è il vettore dei moltiplicatori lagrangiani associati alle restrizioni di equilibrio $Ax = b$.

Una volta generata la nuova variabile y, la si aggiunge al rilassamento LP come una colonna aggiuntiva, ottenendo il seguente rilassamento LP esteso:

$$\begin{gathered} \text{minimizza } c^T x + c_y y \\ \text{soggetto a } Ax + Ay = b \\ x, y \in R^m \times R^n \end{gathered}$$

Quindi, il rilassamento LP esteso viene risolto nuovamente per ottenere una nuova soluzione continua x^* e un valore obiettivo migliorato.

Il processo di generazione di nuove variabili e aggiunta al rilassamento LP viene ripetuto finché nuove variabili che migliorano il valore obiettivo possono essere generate. L'approccio di column generation termina quando il rilassamento LP diventa intero-feasible, ovvero quando tutte le variabili assumono valori interi. La soluzione ottenuta in questo punto fornisce un limite inferiore per la soluzione ottimale del problema MILP e può essere arrotondata per ottenere una soluzione intera ammissibile.

In sintesi, l'approccio di column generation prevede la risoluzione iterativa di rilassamenti LP del problema MILP e la generazione di nuove variabili che migliorano il valore della funzione obiettivo fino a ottenere una soluzione intera ottimale. Questa tecnica può essere molto efficace per risolvere problemi NP-completi su larga scala, soprattutto quando il problema ha un elevato numero di variabili o vincoli che rendono difficile la risoluzione utilizzando i solutori standard per MILP.

3.2 Metriche di selezione

Le metriche di selezione delle colonne e le strategie di aggiornamento del problema principale sono componenti chiave nel framework di Column Generation per risolvere problemi di ottimizzazione. Vediamole in dettaglio:

Metriche di selezione delle colonne:

Le metriche di selezione delle colonne vengono utilizzate per determinare quali nuove variabili (colonne) generare e aggiungere al problema principale. L'obiettivo è selezionare le colonne che contribuiscono maggiormente al miglioramento della funzione obiettivo. Alcune metriche comuni includono:

- Costo ridotto: Questa metrica assegna un valore ai coefficienti delle variabili non presenti nel problema corrente e seleziona quelle con il costo ridotto più basso, ovvero quelle che possono apportare il maggior miglioramento alla soluzione corrente.
- Differenza duale: Questa metrica valuta la differenza duale associata alle variabili non presenti nel problema corrente. Le variabili che producono una differenza duale significativa vengono selezionate poiché possono influenzare in modo significativo la soluzione finale.
- Sensibilità: Questa metrica valuta l'effetto di una variazione nei coefficienti delle variabili non presenti nel problema corrente sulla soluzione ottimale. Le variabili che hanno un maggiore impatto sulla soluzione vengono selezionate.
- La scelta della metrica dipende dal tipo di problema e dalle sue caratteristiche specifiche. È possibile utilizzare una combinazione di metriche o adattarle in base alle esigenze del problema.

Strategie di aggiornamento del problema principale:

Le strategie di aggiornamento del problema principale riguardano il modo in cui vengono gestite le nuove colonne generate durante il processo di Column Generation. Alcune strategie comuni includono:

- Aggiunta immediata: Ogni volta che viene generata una nuova colonna, viene immediatamente aggiunta al problema principale. Questa strategia è semplice ma potrebbe comportare un numero elevato di iterazioni e rendere il processo più lento.
- Aggiunta in batch: Le nuove colonne vengono raccolte in un insieme (batch) e aggiunte al problema principale in un'unica operazione. Questa strategia riduce il numero di iterazioni e può migliorare l'efficienza computazionale.

- Aggiunta basata su una soglia: Le nuove colonne vengono aggiunte al problema principale solo se soddisfano determinati criteri di miglioramento. Ad esempio, potrebbe essere richiesto che una colonna produca una riduzione di costo significativa rispetto alla soluzione corrente prima di essere aggiunta.

- Rimozione di colonne non promettenti: Periodicamente, vengono esaminate le colonne attualmente presenti nel problema principale e quelle che non contribuiscono significativamente al miglioramento della soluzione vengono rimosse. Questo aiuta a ridurre la dimensione del problema e migliorare l'efficienza.

La scelta della strategia dipende dalla natura del problema, dalla complessità computazionale e dalla disponibilità di risorse.

È importante sottolineare che le metriche di selezione delle colonne e le strategie di aggiornamento del problema principale sono strettamente interconnesse e influenzano reciprocamente le prestazioni del framework di Column Generation. La scelta di metriche e strategie appropriate può migliorare l'efficienza del processo e garantire una soluzione di alta qualità.

3.3 Algoritmo di Dantzig-Wolfe nell'ambito della Column Generation

L'algoritmo di Dantzig-Wolfe è una tecnica specifica utilizzata all'interno del framework di Column Generation per generare e aggiungere nuove colonne al problema principale. L'implementazione dell'algoritmo di Dantzig-Wolfe all'interno del framework di Column Generation richiede la definizione di un sotto-problema noto come "problema master" e di un sotto-problema chiamato "sotto-problema duale". L'obiettivo dell'algoritmo è migliorare progressivamente la soluzione trovata risolvendo iterativamente il problema master e il sotto-problema duale.

Il problema master è un rilassamento LP del problema originale, ma con le variabili suddivise in gruppi corrispondenti alle colonne. La formulazione del problema master è la seguente:

$$\text{minimizza } c^T x \text{ soggetto a} \qquad \sum_j a_{ij} x_j = b_i, \quad \forall i \; x_j \geq 0, \quad \forall j$$

- dove x_j rappresenta la colonna j e a_{ij} è l'elemento corrispondente alla riga i e colonna j della matrice dei coefficienti A.

Il sotto-problema duale è un problema di programmazione lineare che dipende dal problema master. La sua formulazione è la seguente:

$$\text{massimizza } \sum_i b_i y_i \text{ soggetto a } \qquad \sum_i a_{ij} y_i \leq c_j, \quad \forall j \; y_i \geq 0, \quad \forall i$$

- dove y_i rappresenta il moltiplicatore lagrangiano associato alla riga i del problema master e c_j è il coefficiente corrispondente alla colonna j.

L'algoritmo di Dantzig-Wolfe si sviluppa iterativamente, alternando la risoluzione del problema master e del sotto-problema duale. Nel seguente ciclo, viene descritto il processo di aggiornamento delle colonne del problema master utilizzando il sotto-problema duale:

1. Risolvi il problema master utilizzando l'ultima soluzione duale ottimale trovata. Questo fornisce una soluzione primale ammissibile e un valore obiettivo.

2. Risolvi il sotto-problema duale utilizzando la soluzione primale ottimale del problema master. Questo fornisce una nuova soluzione duale ottimale.

3. Se la soluzione duale ottimale soddisfa determinati criteri di terminazione, termina l'algoritmo. Altrimenti, procedi al passo successivo.

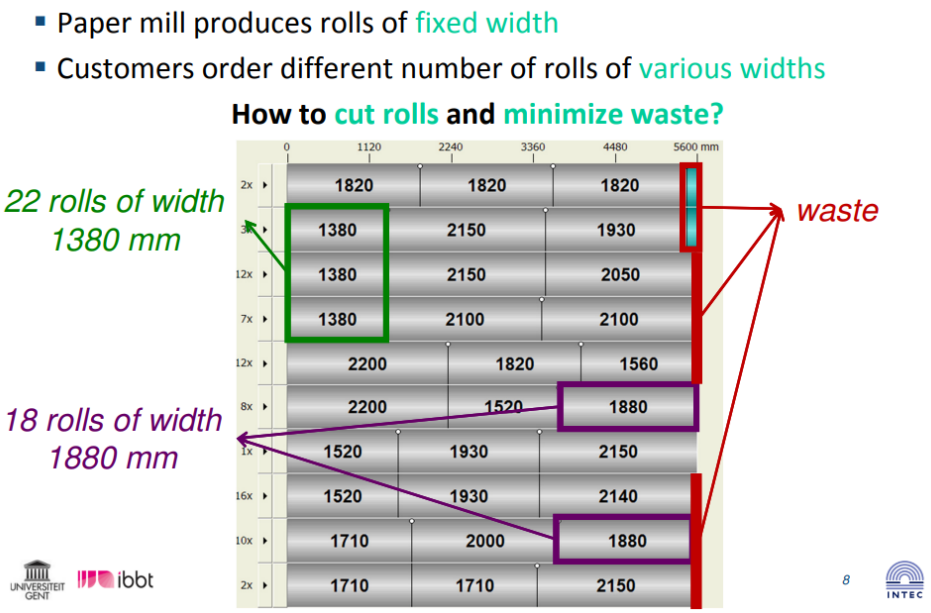
4. Genera una nuova colonna utilizzando la soluzione duale ottimale del sotto-problema duale. Aggiungi questa colonna al problema master.

5. Torna al passo 1.

L'algoritmo continua fino a quando la soluzione duale ottimale soddisfa i criteri di terminazione, che possono essere definiti in base alla differenza tra il valore obiet

Chapter 4 1-D Cutting Stock

Il problema del 1-D Cutting Stock è un problema di ottimizzazione combinatoria che riguarda il taglio efficiente di materiali (solitamente chiamati "rotoli" o "pezzi") di lunghezza fissa per soddisfare una serie di richieste di taglio di lunghezze specifiche, minimizzando lo spreco di materiale.



Questo problema è rilevante in diversi settori produttivi, come l'industria del legno, la lavorazione della lamiera, la produzione tessile e molti altri settori che coinvolgono il taglio di materiali in lunghezze specifiche per soddisfare le richieste dei clienti.

La complessità del problema del 1-D Cutting Stock risiede nella grande quantità di possibili combinazioni di taglio che devono essere esplorate per trovare la soluzione ottima. La sua complessità computazionale è classificata come NP-hard, il che significa che non esiste un algoritmo efficiente in grado di risolvere tutti i casi del problema in tempi ragionevoli.

La formula matematica del modello del problema originale può essere rappresentata come un problema di programmazione lineare intera (ILP) nel seguente modo:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} y_i \\ \text{soggetto a } & \sum_{i=1}^{n} x_{ij} \cdot l_i \leq L \quad \forall j \\ & \sum_{j=1}^{m} x_{ij} \geq d_i \quad \forall i \\ & x_{ij} \geq 0, \quad x_{ij} \in Z \quad \forall i, j \\ & y_i \geq 0 \quad \forall i \quad y_i \in Z \end{aligned}$$

dove:

- n è il numero di richieste di taglio
- m è il numero di combinazioni di taglio possibili
- l_i è la lunghezza richiesta per la richiesta di taglio i
- L è la lunghezza del rotolo da tagliare
- x_{ij} è la variabile binaria che indica se la combinazione di taglio j viene utilizzata per soddisfare la richiesta di taglio i
- d_i è la domanda, cioè il numero di pezzi necessari per la richiesta di taglio i
- y_i è una variabile che rappresenta il numero di boards utilizzate per la combinazione di taglio i

Il problema diventa NP-hard a causa della grande quantità di combinazioni di taglio possibili e della necessità di esplorare tutte le possibili combinazioni per trovare la soluzione ottima. Per comprendere meglio questo concetto, consideriamo un esempio semplificato.

Supponiamo di avere una lunghezza di tavola massima di 10 unità, le richieste di tagliocon le seguenti lunghezze: 2, 4 e 5 unità, in queste quantità 2 tavole di lunghezza 2, 1 di lunghezza 4 unità e 1 di lunghezza 5 unità. L'obiettivo è soddisfare tutte le richieste utilizzando la quantità minima di tavole.

Possiamo generare diverse combinazioni di taglio per soddisfare le richieste. Ad esempio:

Utilizzando una tavola di lunghezza 10, possiamo soddisfare le richieste 2+4+3. Utilizzando due tavole di lunghezza 5, possiamo soddisfare le richieste 5+3+2. Tuttavia, questa è solo una piccola istanza del problema. Quando il numero di richieste di taglio aumenta e le lunghezze

delle richieste diventano più variegate, il numero di possibili combinazioni di taglio cresce in modo esponenziale.

Ad esempio, se abbiamo 5 richieste di taglio con lunghezze 2, 3, 4, 5 e 6 unità, ci sono $2^5 = 32$ possibili combinazioni di taglio da esplorare. Se aumentiamo ulteriormente il numero di richieste di taglio, l'esplosione combinatoria diventa ancora più significativa. L'uso di tecniche di ottimizzazione avanzate, come la column generation o l'euristica di ricerca locale, può aiutare ad affrontare la complessità del problema e trovare soluzioni di buona qualità in tempi ragionevoli.

4.1 Applicazione della Column Generation

L'algoritmo di Dantzig-Wolfe, implementato all'interno del framework di Column Generation, può essere utilizzato per risolvere il problema del 1D Cut Stock. Per applicare l'algoritmo di Dantzig-Wolfe al problema del 1D Cut Stock, è necessario adattare il problema master e il sotto-problema duale alla specifica formulazione del problema. **Restricted Master Problem:** Il Restricted Master Problem (RMP) è un problema di programmazione lineare che rappresenta una formulazione rilassata del problema del 1D-CUT-AXES. L'obiettivo del RMP è minimizzare il numero totale di pattern di taglio utilizzati per soddisfare la domanda di taglio per ciascun elemento, soggetto ai vincoli di soddisfacimento della domanda e ai vincoli di quantità massima di taglio per ogni pattern. . Le colonne del problema master corrispondono ai possibili tagli che possono essere effettuati, e i coefficienti delle colonne rappresentano la quantità di materiale utilizzato per ciascun taglio. Le righe del problema master corrispondono agli ordini di pezzi da soddisfare, e i coefficienti delle righe rappresentano la quantità di pezzi di una determinata lunghezza prodotti da ciascun taglio.

Variabili:

x_j: Quantità di taglio utilizzata per il pattern di taglio j.

Funzione obiettivo:

$$\text{Minimize} \quad \sum_j x_j$$

L'obiettivo è minimizzare il numero totale di pattern di taglio utilizzati.

Vincoli:

$$x_j \geq \text{domanda di taglio per il pattern di taglio } j, \quad \forall j$$

I vincoli di soddisfacimento della domanda assicurano che la quantità di taglio utilizzata per ogni pattern sia sufficiente a soddisfare la domanda di taglio corrispondente.

Ogni variabile di pattern rappresenta la scelta di utilizzare quel particolare pattern di taglio. Quando viene aggiunta una variabile di pattern, i vincoli esistenti nel RMP devono essere adattati per includere l'utilizzo di questa variabile nelle equazioni.

Supponiamo di avere una nuova variabile di pattern x_{new}. Per aggiornare i vincoli esistenti, è necessario considerare come l'utilizzo di questa variabile influenzi il soddisfacimento della domanda e l'utilizzo complessivo dei materiali.

Ad esempio, se un vincolo precedentemente definito era espresso come una somma di variabili di pattern, sarà necessario includere anche la nuova variabile di pattern in questa somma.

L'aggiornamento dei vincoli esistenti consente al RMP di tenere traccia accuratamente dell'utilizzo dei pattern di taglio e di garantire che la soluzione ottimale rifletta correttamente la

combinazione di pattern più efficiente per soddisfare la domanda.

Il RMP viene risolto iterativamente insieme al Pricing Problem per trovare la soluzione ottima. Nella fase iniziale, vengono generati alcuni pattern di taglio iniziali. Successivamente, il Pricing Problem viene risolto per generare nuovi pattern di taglio che possono migliorare la soluzione corrente. I nuovi pattern di taglio vengono aggiunti al RMP come variabili decisionali, e il processo viene ripetuto fino a raggiungere una soluzione ottimale o soddisfare un criterio di terminazione.

Slave Problem:

Lo slave problem è responsabile della generazione di un pattern di taglio ottimale che soddisfi i vincoli di lunghezza massima degli assi consentiti. Il suo obiettivo è determinare la migliore combinazione di tagli da applicare per massimizzare l'utilizzo dei materiali disponibili.
Questa colonna sarà in grado di massimizzare i "costi ridotti", di conseguenza di minimizzare Z.

L'obiettivo dello slave problem è massimizzare i costi ridotti ovvero la somma dei prodotti tra i valori duali (duals$[i]$) e le variabili duali (S_{00i}):

$$\text{Maximise} \quad 1 - \sum_i \text{duals}[i] \cdot S_{00i} x_j \geq 0 \quad \forall j \quad x_j \in Z$$

I vincoli nello slave problem sono definiti come segue:

$$\text{Constraints} \quad \sum_i l_i \cdot S_{00i} \leq L$$

dove:

- S_{00i}: Rappresenta la variabile duale associata al tipo di taglio i. Indica la quantità di taglio del tipo i da applicare.
- duals$[i]$: Rappresenta il valore duale associato al tipo di taglio i. Questo valore indica l'importanza o il costo relativo di applicare il taglio del tipo i nell'ottimizzazione complessiva.
- l_i: Rappresenta la lunghezza dell'asse per il tipo di taglio i. Indica la lunghezza massima consentita per ciascun tipo di taglio.
- L: Rappresenta la lunghezza massima consentita per gli assi. È un parametro fisso che specifica la lunghezza massima dell'asse disponibile.

I passi necessari sono stati:

1. Definizione delle variabili: Vengono definite le variabili decisionali dello slave problem, che corrispondono alla quantità di ciascun tipo di taglio da applicare. Ad esempio, se ci sono 3 tipi di taglio (A, B, C), le variabili potrebbero essere indicate come S_{00A}, S_{00B} e S_{00C}.

2. Definizione della funzione obiettivo: L'obiettivo dello slave problem è minimizzare la quantità totale di materiale sprecato o non utilizzato. Questo viene rappresentato come la somma dei prodotti tra i valori duali (che rappresentano l'importanza di ciascun tipo di taglio) e le rispettive variabili duali. L'obiettivo è quindi quello di trovare la combinazione ottimale di tagli che riduca al minimo lo spreco di materiale.

3. Definizione dei vincoli: Uno dei vincoli principali nello slave problem riguarda la lunghezza massima consentita degli assi. Viene stabilito che la somma dei prodotti tra le lunghezze degli assi e le rispettive variabili duali deve essere inferiore o uguale alla lunghezza massima consentita. Questo vincolo garantisce che i tagli generati rispettino i limiti di lunghezza degli assi.

4. Risoluzione del problema: Una volta definiti gli obiettivi e i vincoli, lo slave problem viene risolto per determinare la combinazione ottimale di tagli che minimizzi lo spreco di materiale. La soluzione ottenuta fornisce i valori ottimali per le variabili duali, che indicano quanti tagli di ciascun tipo devono essere applicati per ottenere la migliore soluzione possibile.

5. In sintesi, lo slave problem genera un pattern di taglio ottimale considerando la lunghezza massima degli assi consentita e minimizzando lo spreco di materiale. Esamina le combinazioni di tagli possibili e assegna i valori ottimali alle variabili duali per determinare la soluzione migliore.

Chapter 5 Metauristiche e GeneticAlgorithm

Le metauristiche sono tecniche di ottimizzazione che possono essere utilizzate per risolvere problemi di ottimizzazione complessi, in cui la ricerca di una soluzione ottima richiede tempi di calcolo eccessivamente lunghi o in cui la soluzione ottima è difficile da raggiungere a causa della grande dimensione dello spazio di ricerca.

L'uso di metauristiche come approccio per risolvere lo slave problem del column generation è molto comune perché consente di affrontare problemi di ottimizzazione complessi in modo efficiente e veloce. In particolare, le metauristiche come il Genetic Algorithm sono efficaci perché offrono una soluzione flessibile e adattabile che si adatta alle diverse esigenze del problema.

Il Genetic Algorithm è una metaeuristica che emula il processo di selezione naturale per risolvere problemi di ottimizzazione. L'algoritmo utilizza una popolazione di soluzioni candidate, chiamati cromosomi, che vengono manipolati attraverso il crossover e la mutazione per generare nuove soluzioni candidate. Queste nuove soluzioni candidate sono valutate utilizzando una funzione di fitness, che valuta la qualità di una soluzione.

Il Genetic Algorithm è particolarmente adatto per la risoluzione di problemi di ottimizzazione combinatoria, in cui la soluzione ottima è difficile da trovare a causa della grande dimensione dello spazio di ricerca. Inoltre, l'algoritmo è in grado di gestire vincoli non lineari e funzioni di costo non lineari.

Nel contesto del column generation, il Genetic Algorithm può essere utilizzato per generare colonne ottime in modo efficiente e veloce. In particolare, il Genetic Algorithm può essere utilizzato per generare nuove colonne candidate attraverso la manipolazione dei cromosomi, che rappresentano le caratteristiche della colonna, come i coefficienti delle variabili e i vincoli associati. La funzione di fitness può essere utilizzata per valutare la qualità delle colonne candidate in modo da selezionare quelle migliori per l'inserimento nel Restricted Master Problem.

Tuttavia, come ogni algoritmo di ottimizzazione, l'efficacia del Genetic Algorithm dipende dalla correttaconfigurazione dei parametri e dalla scelta della funzione di fitness appropriata. È importante anche considerare la possibile presenza di trappole di ottimizzazione, ovvero soluzioni localmente ottimali che non corrispondono alla soluzione globale ottima del problema.

Chapter 6 Implementazione e risultati

6.1 Struttura progetto

La struttura del progetto è organizzata nel seguente modo:

1. Jupyter Notebook: Il progetto comprende un Jupyter Notebook principale che contiene gli script di esecuzione. Questo notebook può essere utilizzato per eseguire l'algoritmo di column generation e analizzare i risultati.

2. src: Questa directory contiene i modelli del Restricted Master Problem (MasterProblem) e dello Slave Problem (SlaveProblem). Questi modelli sono implementati come classi Python e contengono la logica per risolvere i rispettivi problemi.

3. models: Questa directory contiene tutti i modelli LP generati durante l'esecuzione dell'algoritmo di column generation. Dopo ogni iterazione, un modello LP viene generato e salvato in questa directory per analisi o debugging.

4. data: Questa directory contiene i dati di mock generati utilizzando lo script *datagen.py* e vengono utilizzati come input per l'algoritmo di column generation.

6.2 Restricted Master Problem

La classe *MasterProblem.py* definisce il problema principale e implementa le funzionalità per la modellazione, la risoluzione e l'aggiunta di nuovi pattern. Utilizza le funzioni e le classi fornite dalla libreria pulp per creare variabili di decisione, definire vincoli e obiettivi, risolvere il problema e ottenere il valore ottimo.

Nel costruttore della classe MasterProblem, vengono inizializzati i parametri del problema principale, come la massima dimensione della board, i tipi di taglio da effettuare e il numero minimo di tagli richiesti per ogni tipo. Vengono anche definite le variabili decisionali, i vincoli e l'obiettivo del problema utilizzando la libreria pulp. Inoltre, viene specificato il tipo di Solver da utilizzare per il così detto Pricing Problem.

```
1
2       self.prob = LpProblem('RMP',LpMinimize)
3       self.obj = LpConstraintVar("obj")
4       self.prob.setObjective(self.obj)
5
6       # imposta la quantit  minima di elementi da produrre Ax >= b (
            ↪ Greater/Equal)
7       for i,x in enumerate(itemDemands):
8         var=LpConstraintVar(
9           name=f'C{i}',
10          sense=LpConstraintGE,
11          rhs=x
12          )
13        self.constraintList.append(var)
14        self.prob+=var
15
16    # Salva i pattern iniziali, applicandoli al problema e alle sue
        ↪ constraints
17      for i,x in enumerate(self.initialPatterns):
18        affected_constraints = [j for j, y in enumerate(x) if y > 0]
19        LpVariable(
20          name=f'Init_Pat{i}',
21          lowBound= 0,
22          upBound= None,
23          cat= LpContinuous,
24          e= lpSum(self.obj+[self.constraintList[v] for v in
                ↪ affected_constraints])
25        )
26        self.PatternVars.append(var)
27        self.generatedPatterns.append(x)
```

Il metodo *solve*() risolve il problema principale chiamando il metodo solve() della classe LpProblem fornita da pulp.

Il metodo *addPattern*() viene utilizzato per aggiungere un nuovo pattern al modello esistente. Viene creato un oggetto LpVariable che rappresenta il nuovo pattern e viene aggiunto alla lista dei pattern generati e alla lista delle variabili decisionali.

Il metodo *startSlave*() crea o avvia un nuovo problema slave utilizzando il tipo di slave solver specificato e imposta i dati iniziali, tra cui i valori delle soluzioni duali. Cioè i costi marginali associati ai vincoli del problema primale, che indicano quanto aumenterebbe o dominuirebbe il valore della funzione obiettivo per unità aggiuntiva o in meno di una determinata risorsa vin-

colante . Successivamente, viene generato un nuovo pattern utilizzando il metodo generatePattern() del problema slave e viene restituito il pattern ottenuto.

Il metodo *setRelaxed*() viene utilizzato per impostare le variabili intere a False quando il parametro relaxed è False. In altre parole, se relaxed è False, le variabili vengono trattate come continue invece che come intere.

Altri metodi come *getObjective*(), *getUsedPatterns*() e *getComputedOptimal*() vengono utilizzati per ottenere informazioni sul problema principale, come il valore dell'obiettivo, la lista dei pattern utilizzati e il valore ottimo calcolato.

6.3 Slave Problem

La classe *SlaveProblem.py*, implementa un problema di ottimizzazione per trovare il miglior pattern da utilizzare per partizionare un asse.

```
1    '''
2            Definizione delle variabili decisionali
3                - S_00i : numero di unit  della risorsa i
4        '''
5        self.varList=[
6                LpVariable(
7                    name= f'S_{i:02d}',
8                    lowBound = 0,
9                    upBound=None,
10                   cat=LpInteger)
11           for i,x in enumerate(self.duals)]
12
13       '''
14           Utilizzo i valori duali nel calcolo dei coefficienti
               ↪ obiettivo dello Slave Problem
15       '''
16       self.slaveprob.setObjective(
17           -1 * lpSum([self.duals[i]*x for i, x in enumerate(self.
               ↪ varList)])
18       )
```

```
        '''
            Definizione del vincolo:
                -> l_i * S_00i  + .... + l_n * S_00n <= L
                - Vincola lo slave a creare partizionamenti all'
                    ↪ interno di un asse lungo  massimo L
        '''
        self.slaveprob += LpConstraint(
            e=lpSum([self.itemLengths[i]*x for i, x in enumerate(self
                ↪ .varList)]),
            sense=LpConstraintLE,
            rhs=self.maxValue
        )

        self.slaveprob.writeLP(f'models/slave/slaveprob{SlaveProblem.
            ↪ it}.lp')
        self.slaveprob.solve()
        self.slaveprob.roundSolution() #to avoid rounding problems
        SlaveProblem.it: int=SlaveProblem.it+1
```

In input riceve i valori duali computati dalla risoluzione del RMP, la dimensione di ciascun tipo di unità da produrre e la dimensione massima di un asse. Viene quindi definito il problema duale, ovvero minimizzare il costo di produzione utilizzando i valori duali nel calcolo dei coefficienti obiettivo, e viene impostato il vincolo chela somma dei prodotti tra le lunghezze dei tagli e il numero di unità di ciascun tipo non superi la dimensione massima dell'asse.

6.4 Column Generation

```
    # Initial solution (greedy):
    # - Use one axes for each demand of that axes length
    # - Start with a poor solution and expect the total required axes
      ↪ to decrease
    patterns = []
    for i in range(num_axes):
        pattern = np.zeros(num_axes)
        pattern[i] = axis_demands[i]
        patterns.append(pattern)
```

La funzione runColumnGeneration() implementa l'algoritmo di column generation completo. Inizializza la soluzione iniziale con un approccio greedy, dove viene assegnato un asse per ogni domanda di quella lunghezza dell'asse (tanti assi quanta è la domanda per quel taglio).

```

  while relaxed and iteration < max_iterations:
        duals = masterProblem.solve()
        if old_duals == duals:
            consecutive_unchanged_duals -= 1
        else:
            consecutive_unchanged_duals = 15
        if consecutive_unchanged_duals > 0:
            newPattern = masterProblem.startSlave(duals)
            if newPattern:
                masterProblem.addPattern(newPattern)
                if(debugMode):
                    print("New duals", duals)
                    print("New pattern added", newPattern)
        else:
            masterProblem.setRelaxed(False)
            masterProblem.solve()
            relaxed = False

        old_duals = duals
        iteration += 1
```

Successivamente, utilizza un'istanza del problema master per risolvere il problema e ottenere i duali fornendo in input il *tipo di SlaveSolver*. Se i duali non cambiano per un certo numero di iterazioni, il codice passa alla soluzione finale. In caso contrario, il codice utilizza il problema schiavo per generare una nuova soluzione e aggiungerla alla soluzione in corso. Il processo

continua fino a quando non viene trovata una soluzione finale o viene raggiunto il numero massimo di iterazioni consentite. Viene quindi avviato il solver per risolvere il problema e viene restituito il pattern ottenuto Infine, la funzione restituisce le colonne utilizzate nella soluzione ottimale, che sono rappresentate come pattern di taglio.

```
1
2       optimal = masterProblem.getComputedOptimal()
3       print("Optimal: %s axes to cut" % optimal)
4       print("Found in %s iterations" % iteration)
5       print("Computational time: %s seconds" % seconds)
```

6.5 Genetic Algorithm

Slavehe.py implementa l' algoritmo genetico per la generazione di un modello di taglio ottimo utilizzando i valori duali generati dalla risoluzione del Restricted Master Problem per guidare la generazione del pattern. La classe "GeneticAlgorithm" è una sottoclasse della classe "ISlave" e implementa i metodi necessari per risolvere il problema di taglio utilizzando un algoritmo genetico.

Non è stato creata una entità cromosoma ad hoc ma si è utilizzata la stessa struttura dati del pattern, quindi una lista con dimensione ' tipi di taglio'.

Il metodo "evaluate$_fitness$"$vieneutilizzatopervalutarelafitnessdiuncromosoma, calcolandolalunghezzato$

```
1       def evaluate_fitness(self, chromosome):
2
3           total_length = sum(chromosome[k] * self.itemLengths[k] for k
              ↪ in range(len(chromosome)))
4
5           # Check if the pattern exceeds the maximum resource usage
6           if total_length > self.maxValue:
7               return  0.0001
8
9           # Calculate the fitness score using the dual values
10          fitness =  (-sum( -self.duals[i] * chromosome[i]  for i in
              ↪ range(len(self.duals))))
11
12          return fitness if fitness>0 else 0.0001
```

Siccome il valore di fittness viene utilizzato per pesare la selezione di parent di una nuova offspring, il valore di fittness deve essere positivo e perciò è stato bounded a [0.0001.., Inf]

Il metodo "*generatePattern*" esegue l'algoritmo genetico per un numero specificato di generazioni. Viene valutata la fitness della popolazione corrente, creata una nuova popolazione attraverso crossover ed elitismo, generati nuovi individui offspring e sostituita la popolazione corrente con la nuova popolazione.

```
def generatePattern(self):
        self.initialize_population()
        for _ in range(self.generations):
            # Evaluate fitness for each individual in the population
            fitness_scores = [self.evaluate_fitness(chromosome) for
                ↪ chromosome in self.population]

            # Create a new population for the next generation
            new_population = []

            # Apply elitism - preserve the best individual in the
                ↪ current population
            best_index = fitness_scores.index(max(fitness_scores))
            best_chromosome = self.population[best_index]
            new_population.append(best_chromosome)

            # Generate offspring for the remaining population slots
            while len(new_population) < self.population_size:
                # Select parents for crossover
                # The selection is weighted by the fittness_scores
                parents = random.choices(self.population, weights=
                    ↪ fitness_scores, k=2)

                # Create offspring through crossover
                offspring = self.crossover(parents[0], parents[1])

                # Mutate the offspring
                self.mutate(offspring)

                # Add the offspring to the new population
                new_population.append(offspring)

            # Replace the current population with the new population
            self.population = new_population
```

Infine, il metodo "returnPattern" restituisce il miglior pattern di taglio ottenuto dall'ultima generazione della popolazione. Il miglior pattern viene selezionato sulla base della fitness più alta

6.6 Condizioni di stop

Durante l'implementazione degli algoritmi, sono state adottate specifiche condizioni di stop per determinare quando interrompere l'esecuzione e considerare la soluzione corrente come risultato finale. Queste condizioni di stop sono state impostate per garantire l'efficienza e prevenire l'esecuzione infinita dell'algoritmo.

Nel caso dell'algoritmo genetico, la condizione di stop è definita dalla fitness dell'attuale best solution generata. Se la fitness è inferiore a un determinato valore (ad esempio 1,0), l'algoritmo termina senza restituire un pattern valido. Inoltre, è stata riscontrata la possibilità di cadere in un loop quando tutte le soluzioni generate hanno un'affinità (fitness) non superiore a 1,0. Questa condizione impedisce che l'algoritmo continui a generare soluzioni non valide senza convergere verso una soluzione ottimale.

Per quanto riguarda il problema dello slave, la condizione di stop è legata alla funzione obiettivo. Se la funzione obiettivo non è strettamente negativa, non viene generato alcun pattern e l'algoritmo termina.

Per la column generation, sono state considerate diverse condizioni di stop. È possibile configurare le variabili "max iterations" e "consecutive unchanged duals" per definire il numero massimo di iterazioni consentite e il numero massimo di iterazioni consecutive senza cambiamenti nelle variabili duali. Queste condizioni consentono di interrompere l'esecuzione anticipatamente se l'algoritmo non mostra progressi significativi nel miglioramento della soluzione.

Chapter 7 Analisi dei risultati

7.1 Test e conclusioni

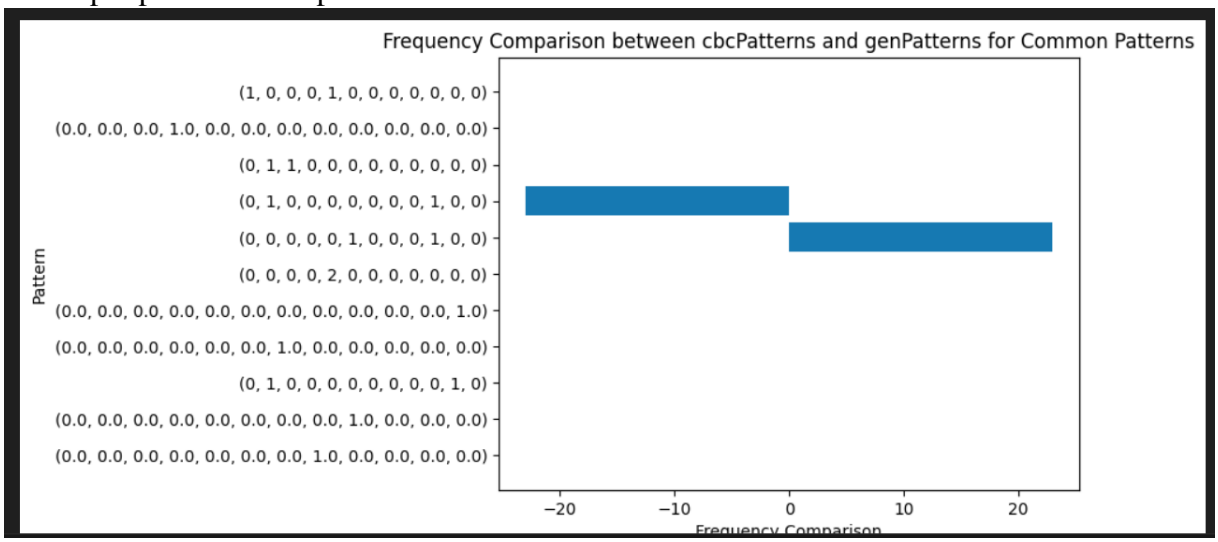
Algorithm	Max Axis Length	Axis Len	Axis Dem	Optimal	Iterati	Time
Slave Problem	25	12	[2, 3, 1]	5	17	00:00:01
Genetic Algorith	25	12	[2, 3, 1]	5	17	00:00:00
Slave Problem	25	45	[17, 89, 12, 4	456	27	00:00:01
Genetic Algorith	25	45	[17, 89, 12, 4	465	34	00:00:01
Slave Problem	50	20	[3, 1, 3, 1, 2]	5	21	00:00:01
Genetic Algorith	50	20	[3, 1, 3, 1, 2]	5	24	00:00:00
Slave Problem	50	172	[62, 60, 75, 8	1135	93	00:00:05
Genetic Algorith	50	172	[62, 60, 75, 8	1142	169	00:00:06
Slave Problem	100	181	[45, 78, 34, 9	1407	89	00:00:05
Genetic Algorith	100	181	[45, 78, 34, 9	1409	152	00:00:04
Slave Problem	250	692	[77, 6, 99, 5,	4424	253	00:00:19
Genetic Algorith	250	692	[77, 6, 99, 5,	4525	465	00:00:16
Slave Problem	500	1197	[75, 74, 83, 7	6678	590	00:01:06
Genetic Algorith	500	1197	[75, 74, 83, 7	6829	1000	00:00:47

In questo elaborato sono stati esaminati due approcci per risolvere il problema del 1D-cut-stock: l'algoritmo genetico e l'utilizzo del solver PULP. Sono state condotte analisi comparative dei risultati ottenuti da entrambi gli approcci e sono state discusse le loro caratteristiche e prestazioni. La tabella fornita mostra i risultati dell'esecuzione di due algoritmi, il Problema dello Slave e l'Algoritmo Genetico, per varie configurazioni di input. Ecco la ripartizione delle colonne:

- Algoritmo: Specifica l'algoritmo usato per il calcolo.
- Lunghezza massima dell'asse: Rappresenta la lunghezza massima dell'asse utilizzata nel calcolo.
- Lunghezze assi: indica le lunghezze degli assi utilizzate nel calcolo.
- Ottimale: Rappresenta la soluzione ottimale trovata dall'algoritmo.
- Iterazione: Specifica il numero di iterazione con cui è stata trovata la soluzione ottimale.
- Tempo: indica il tempo impiegato dall'algoritmo per trovare la soluzione ottimale.

Dai risultati ottenuti, è emerso che entrambi gli approcci hanno vantaggi e limitazioni. L'algoritmo genetico ha dimostrato di essere in grado di raggiungere soluzioni di buona qualità in tempi più brevi rispetto a PULP in alcuni casi. Tuttavia, l'algoritmo genetico rappresenta una soluzione approssimata e non garantisce la soluzione ottima. D'altra parte, PULP è un solver

esatto che garantisce soluzioni ottimali, ma può richiedere tempi di esecuzione più lunghi, specialmente per problemi complessi.



In tabella la "frequency comparison" tra i pattern "comuni" che sono generati sia dal Cbc-Solver (Pulp) e quelli generati tramite Genetic Algorithm. Si fa riferimento al problema con Lunghezza massima dell'asse 25 e Lunghezze assi 12. Solo 11 dei 12 pattern generati da entrambi i modelli, sono pattern comuni.

Un'analisi dettagliata delle soluzioni ottenute ha mostrato che entrambi gli approcci hanno generato pattern comuni, ma anche differenze nella frequenza di utilizzo di alcuni pattern. Questo può indicare che ci sono più modi per risolvere il problema e che gli algoritmi hanno preferenze diverse nella scelta dei pattern.

Sono inoltre stati fatti diversi test andando a modificare gli iperparametri dell'algoritmo genetico in quanto si è riscontrato che per i problemi affrontati, una popolazione troppo elevata ed un numero troppo grande di epoche, causavano enormi rallentamenti senza apportare a risultati troppo migliori. Vediamo in tabella:

Measure	50P_100G	%DIFF	5P_10G	MAX A	TYPES	%
TIME	0,738	28,46%	0,21			
OPTIMAL	5	100,00%	5	25	3	
ITERATIONS	17	100,00%	17			
TIME	1,726	35,92%	0,62			
OPTIMAL	456	100,00%	456	25	12	
ITERATIONS	26	100,00%	26			
TIME	1,111	28,80%	0,32			
OPTIMAL	5	100,00%	5	50	5	
ITERATIONS	21	100,00%	21			
TIME	13,8	40,58%	5,6			
OPTIMAL	1142	100,00%	1142	50	40	
ITERATIONS	97	208,25%	202			
TIME	11,5	29,04%	3,34			
OPTIMAL	1410	100,71%	1420	100	46	
ITERATIONS	84	144,05%	121			
TIME	90,75	15,23%	13,82			
OPTIMAL	4510	100,22%	4520	250	150	
ITERATIONS	264	154,17%	407			

In figura, sono confrontati i risultati ottenuti dall'algoritmo genetico allenato con 'popolazione=50' e 'epoche=100' rispetto all'algo allenato su 'popolazione=5' e 'epoche=10' Dai risultati ottenuti, si riscontra che la rimodulazione di popolazione/epoche ha permesso:

- un aumento di iterazioni dell'algoritmo genetico
- una riduzione considerevole del tempo di esecuzione dello stesso.
- soluzioni ottime molto simili rispetto a quelle prodotte dall'algoritmo genetico allenato su un campione più alto di popolazione.

L'uso dell'algoritmo genetico offre vantaggi come tempi di esecuzione più rapidi rispetto a PULP in alcuni casi, come evidenziato dai risultati riportati. Inoltre, l'algoritmo genetico può raggiungere soluzioni ottime con un numero inferiore di iterazioni rispetto a PULP. Questo potrebbe essere utile in scenari in cui è necessario ottenere rapidamente soluzioni di buona qualità senza richiedere una precisione assoluta.

Nel complesso, la scelta tra l'approccio approssimato dell'algoritmo genetico e l'approccio esatto di PULP dipende dalle specifiche del problema, come la complessità del problema, la necessità di soluzioni ottimali e la disponibilità di risorse computazionali. È importante valutare attentamente questi trade-off al fine di selezionare l'approccio più adatto alle esigenze del progetto.

7.2 Possibili ottimizzazioni

Per quanto riguarda le possibili ottimizzazioni, si potrebbe proporre l'introduzione dei piani di taglio nella column generation.

E' possibile utilizzare il metodo dei piani di taglio per generare piani di taglio che vengono successivamente incorporati nella procedura iterativa della column generation. In ogni iterazione, oltre a generare e aggiungere nuove colonne, si possono generare e aggiungere anche nuovi piani di taglio. Questi piani di taglio possono essere generati utilizzando informazioni sulle soluzioni parziali o su altre euristiche.

L'aggiunta dei piani di taglio nel processo di column generation può avere diversi vantaggi: Innanzitutto, i piani di taglio possono ridurre il numero di iterazioni necessarie per raggiungere la soluzione ottima, accelerando il processo di convergenza dell'algoritmo. Inoltre, i piani di taglio possono migliorare la formulazione del problema principale, rendendo il rilassamento lineare più stretto e fornendo soluzioni di migliore qualità siccome eliminano soluzioni non ammissibili .

Bibliography

[1] Lafifi, M. M. (2018). How to write ILP for two-stage column generation. *ResearchGate*. Retrieved from https://www.researchgate.net/profile/Mohamed-Mourad-Lafifi/post/How-to-write-ILP-for-two-stage-column-generation/attachment/5c2327193843b006754d3ec6/AS%3A707982715256832%401545807641551/download/Column+Generation+Tutorial.pdf

[2] l file Python su GitHub "cuttingstock.py": \bibitem{CGpython} Fontan, F. (2021). Cuttingstock.py [Python script]. GitHub. https://github.com/fontanf/columngenerationsolverpy/blob/main/examples/cuttingstock.py

[3] l video su YouTube "Introduction to Column Generation": \bibitem{CGvideo1} YouTube. (2017, January 11). Introduction to Column Generation [Video]. YouTube. https://www.youtube.com/watch?v=0SV99VJaGzw

[4] l file Python su GitHub "cuttingstock.py": \bibitem{CGpython2} Bart6114. (2013). cuttingstock.py [Python script]. GitHub Gist. https://gist.github.com/Bart6114/8414730

[5] l video su YouTube "Column Generation applied to the Cutting Stock Problem": \bibitem{CGvideo2} YouTube. (2017, January 11). Column Generation applied to the Cutting Stock Problem [Video]. YouTube. https://www.youtube.com/watch?v=FVZA28XZ7Mg

[6] l documento disponibile su ResearchGate "How to write ILP for two-stage column generation":
\bibitem{CGtutorial} Lafifi, M. M. (2018). How to write ILP for two-stage column generation. *ResearchGate*. Retrieved from https://www.researchgate.net/profile/Mohamed-Mourad-Lafifi/post/How-to-write-ILP-for-two-stage-column-generation/attachment/5c2327193843b006754d3ec6/AS%3A707982715256832%401545807641551/download/Column+Generation+Tutorial.pdf

[7] Taha, H. A. (2016). *Operations Research: An Introduction*. Pearson Education.

[8] Gilmore, P. C., Gomory, R. E. (1961). A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6), 849-859.

[9] Belov, G. V., Scheithauer, G., Werner, F. (2012). One-dimensional cutting stock problems: Survey and outlook. *European Journal of Operational Research*, 213(3), 361-374.

[10] Desaulniers, G., Desrosiers, J., Solomon, M. M. (2006). *Column Generation*. Springer Science Business Media.

[11] Barnhart, C., Hane, C. A. (1996). Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 44(6), 828-840.

[12] Rousseau, L. M., Desaulniers, G. (2007). A column generation algorithm for the one-dimensional cutting stock problem with usable leftovers. *INFORMS Journal on Computing*, 19(1), 57-67.

[13] Hanafi, S., Rousseau, L. M. (2018). Exact and heuristic approaches for the one-dimensional cutting stock problem with multiple stock lengths. *European Journal of Operational Research*, 271(3), 825-836.

[14] Beasley, J. E. (1985). An algorithm for solving large-scale 1-cutting-stock problems. *European Journal of Operational Research*, 19(3), 370-386.

[15] Martello, S., Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley Sons.

[16] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Professional.

[17] Parada, V., Lopes, M., Clautiaux, F. (2012). A genetic algorithm for the one-dimensional cutting stockproblem. *European Journal of Operational Research*, 223(3), 613-622.

[18] Martello, S., Pisinger, D. (1998). Algorithms for the one-dimensional cutting stock problem: A guided tour. *Operations Research*, 46(3), 346-357.

[19] Helber, S., Sahling, F., Haußner, H. (2008). Comparison of different solution approaches for the one-dimensional cutting stock problem. *European Journal of Operational Research*, 185(1), 282-292.

[20] Desaulniers, G., Desrosiers, J., Solomon, M. M. (2002). Column generation. *INFORMS Journal on Computing*, 14(4), 299-316.

[21] Belov, G., Rakhmanova, E. (2016). *Cutting Stock Problems: Theory and Solution Methods*. Springer International Publishing.