UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

MODELLI E ALGORITMI DI OTTIMIZZAZIONE PER PROBLEMI MULTI RISORSA CON PENALITÀ FAMILY-SPLIT

Relatore: Chiar.ma Prof. Renata Mansini

Laureando:
Fabio Perini
Matricola n. 715425

Anno Accademico 2023/2024

Indice

Capitolo 1: Introduzione

L'obiettivo di questa tesi è quello di introdurre una nuova variante di un problema già noto in letteratura: il *Multiple Multidimensional Knapsack Problem with Family-Split Penalties* (MMdKFSP). Si tratta di una generalizzazione del *Multiple Knapsack Problem* (MMKP) e del *Multidimensional Knapsack Problem* (MKP) definita come segue.

Si ha un insieme di oggetti, partizionati in famiglie, da inserire o meno in uno zaino. Gli zaini disponibili sono multipli e hanno delle capacità definite. Ogni oggetto occupa una serie di risorse, quindi una quantità per ogni risorsa, e, di conseguenza, anche gli zaini hanno una capacità per ognuna delle risorse definite. Ognuna delle famiglie ha un profitto e una penalità associati: il profitto si ottiene se la famiglia viene inserita negli zaini, la penalità viene sottratta al profitto una sola volta se la famiglia viene divisa tra più zaini (non viene sottratta se tutti gli oggetti della famiglia sono allocati nello stesso zaino). L'obiettivo del problema è di massimizzare la differenza tra la somma dei profitti e la somma delle penalità. Non è possibile né inserire parzialmente una famiglia, né superare le capacità degli zaini.

Quello che verrà osservato è che esistono delle situazioni in cui la formulazione precedente non cattura pienamente il problema reale. Per esser precisi, non sempre è realistico che la penalizzazione avvenga una sola volta anche se la famiglia viene divisa più volte, ma in certi casi è più appropriato fare in modo che venga sottratta una volta per ognuna delle divisioni (split). Per questo motivo si è deciso di introdurre una nuova variante del problema che tenga conto del numero di split di ogni famiglia.

Oltre all'introduzione della nuova variante, verranno proposti degli algoritmi euristici per la sua risoluzione. Prima verrà introdotto un metodo per risolvere la variante proposta da Mancini et. al (2021 *[1]*), poi saranno mostrati quattro algoritmi per risolvere quella nuova.

Un'ulteriore novità portata da questo lavoro sono le istanze utilizzate per valutare gli algoritmi nella risoluzione della seconda variante. Esse sono state create per essere sufficientemente impegnative da mettere in difficoltà un risolutore commerciale e quindi poter valutare l'efficacia dei metodi proposti. Invece, per la prima variante, sono state utilizzate le stesse istanze proposte in Mancini et al. (2021 *[1]*) e in Mancini et al. (2022 *[2]*)

Nel secondo capitolo si partirà da una descrizione dei problemi basilari da cui deriva il MMdKFSP, con riferimento ai corrispondenti articoli della letteratura. Verrà poi introdotta una descrizione più approfondita di entrambi i problemi trattati in questo lavoro, mostrando un esempio applicativo reale in ambito informatico e poi descrivendoli formalmente. A seguire, verrà fatta un'analisi dei problemi simili presenti in letteratura, evidenziando le differenze con quelli presentati.

Nel terzo capitolo saranno ripresi concetti e definizioni della teoria matematica della ricerca operativa, che rappresentano la base su cui si fondano gli algoritmi proposti. Poi verranno definiti rigorosamente i modelli matematici delle due varianti trattate.

Nel quarto capitolo verranno richiamate alcune definizioni relative alle categorie di algoritmi a cui appartengono quelli mostrati, per poi illustrare il funzionamento del metodo risolutivo implementato per la prima variante.

Nel quinto capitolo si passerà alla spiegazione degli algoritmi per la risoluzione della seconda variante. Essi saranno mostrati in ordine di implementazione, perché lo sviluppo di ognuno è motivato dai risultati ottenuti dai precedenti. In particolare, il primo sarà un adattamento dell'algoritmo sviluppato per la prima variante, il secondo riprenderà i concetti dell'algoritmo Kernel Search (Mansini et al. 2010 *[3]*) per migliorare i risultati ottenuti, il terzo sarà un'implementazione base dell'algoritmo Kernel Search e il quarto sarà una combinazione tra il secondo e il terzo.

Infine, nel sesto capitolo saranno descritte le istanze utilizzate per testare l'algoritmo implementato per la prima variante, per poi mostrare i risultati ottenuti, analizzarli e confrontarli con quelli di un risolutore commerciale (Gurobi). Dopodiché la stessa cosa verrà fatta per ognuno degli algoritmi implementati per la seconda variante, con l'aggiunta di confronti con i risultati degli algoritmi precedenti. Per ogni algoritmo verranno anche confrontate diverse configurazioni, per capire meglio come è stata raggiunta quella migliore.

Capitolo 2: Problemi knapsack con penalità

2.1 Introduzione

Il problema dello zaino è stato oggetto di studio nella letteratura scientifica già da molti anni. Il primo libro che va a indagare nel dettaglio questa classe di problemi è *Knapsack Problems: Algorithms and Computer Implementations*, a opera di Martello e Toth (1990 [4]), e viene seguito dal libro di Kellerer et al. (2004 [5]), *Knapsack Problems*. La sua formulazione base prevede che siano dati un insieme di n oggetti, ognuno provvisto di un profitto p_j e un peso w_j, con $j \in \{1, ..., n\}$, uno zaino di capacità c, e che l'obiettivo sia di trovare un sottoinsieme degli oggetti che abbia il massimo profitto totale possibile, senza eccedere la capacità dello zaino. Tra le varianti presenti, che aumentano la difficoltà del problema, ci sono: il problema dello zaino multiplo (MKP) e il problema dello zaino multidimensionale (MdKP) caratterizzato dall'avere più zaini (risorse) ciascuno con la propria capacità. Recentemente Cacchiani et al. hanno effettuato uno studio sugli avanzamenti recenti nell'affrontare i problemi a zaino singolo (2022a [6]) ma anche uno studio panoramico nell'ambito dei problemi con zaino multiplo, multidimensionale e quadratico (2022b [7]). In quest'ultimo articolo viene presentata una panoramica di MKP e MdKP, evidenziando che in letteratura è ben definito il problema dello zaino multiplo mentre esistono diverse interpretazioni del termine "multidimensionale".

Il problema dello zaino multiplo semplicemente prevede che siano presenti più zaini in cui inserire gli oggetti, quindi vanno determinati più sottoinsiemi di oggetti (uno per ogni zaino) che raggiungano il massimo profitto senza superare le capacità limite. Il problema dello zaino multidimensionale, invece, viene classificato in due tipologie: una in cui gli oggetti hanno più funzioni di peso (le risorse disponibili) e per ognuna viene definita una capacità limite, che in Cacchiani et al. (2022b [7]) viene chiamata *Multidimensional (vector) Knapsack Problems*; l'altra in cui oggetti e zaini sono delle scatole rettangolari e gli oggetti devono essere inseriti senza sovrapporsi, chiamata *Multidimensional Geometric Knapsack Problems*. Stando alla classificazione di Cacchiani et al. (2022b [7]), in questa tesi verranno affrontate due varianti che derivano dallo zaino multiplo e dal *Multidimensional (vector) Knapsack Problem*, dove si assume che tutti gli oggetti siano suddivisi in famiglie (gruppi) e sia previsto il pagamento di penalità qualora oggetti appartenenti alla stessa famiglia vengano allocati in zaini diversi.

La prima variante è stata introdotta da Mancini et al. (2021 [1]) e chiamata *Multiple Multidimensional Knapsack Problem with Family-Split Penalties* (MMdKFSP).

Essa prevede un insieme di oggetti, partizionati in famiglie, da allocare in un insieme di zaini. Ogni oggetto occupa una certa quantità di una risorsa e, di conseguenza, per ogni zaino è definita una capacità per ogni risorsa. Ogni famiglia ha due valori associati: un profitto e una penalità. Se tutti gli oggetti della famiglia vengono inseriti in uno o più zaini allora si ottiene il profitto associato. Se gli oggetti della famiglia vengono inseriti in zaini diversi, e non tutti nello stesso, allora si sottrae la penalità corrispondente dal profitto totale. Una famiglia non può essere inserita parzialmente, quindi: o tutti i suoi oggetti vengono associati a uno zaino, oppure nessuno viene inserito, senza eccedere le capacità degli zaini. L'obiettivo del problema è di massimizzare la differenza tra il totale dei profitti e il totale delle penalità.

Per comprendere meglio il problema, verrà presentato un esempio di applicazione reale in ambito informatico.

Esempio applicativo:
In un sistema di calcolo distribuito si hanno 3 processi da eseguire (famiglie), ognuno suddiviso in un certo numero di sottoprocessi (oggetti) distinti che richiedono CPU e memoria (risorse). Indicando con il primo numero la quantità di CPU richiesta e con il secondo numero la quantità di memoria richiesta, i processi hanno le seguenti caratteristiche:

Il processo A è diviso in 6 sottoprocessi:

$$A \leftarrow \begin{cases} \text{A1: 15-20} \\ \text{A2: 20-30} \\ \text{A3: 30-20} \\ \text{A4: 10-10} \\ \text{A5: 30-10} \\ \text{A6: 40-30} \end{cases}$$

Il processo B è diviso in 3 sottoprocessi:

$$B \leftarrow \begin{cases} \text{B1: 15-80} \\ \text{B2: 40-40} \\ \text{B3: 20-50} \end{cases}$$

Il processo C è diviso in 5 sottoprocessi:

$$C \leftarrow \begin{cases} \text{C1: 10-10} \\ \text{C2: 15-15} \\ \text{C3: 30-70} \\ \text{C4: 30-70} \\ \text{C5: 5-30} \end{cases}$$

Questi vanno assegnati a un nodo del sistema distribuito (zaino) per poter essere eseguiti e ogni nodo ha una quantità limitata di risorse a disposizione. Indicando con il primo numero la quantità di CPU massima e con il secondo numero la quantità di memoria massima, i nodi hanno le seguenti capacità:

$$\begin{aligned} &\text{N1: 100-150} \\ &\text{N2: 100-90} \\ &\text{N3: 100-120} \end{aligned}$$

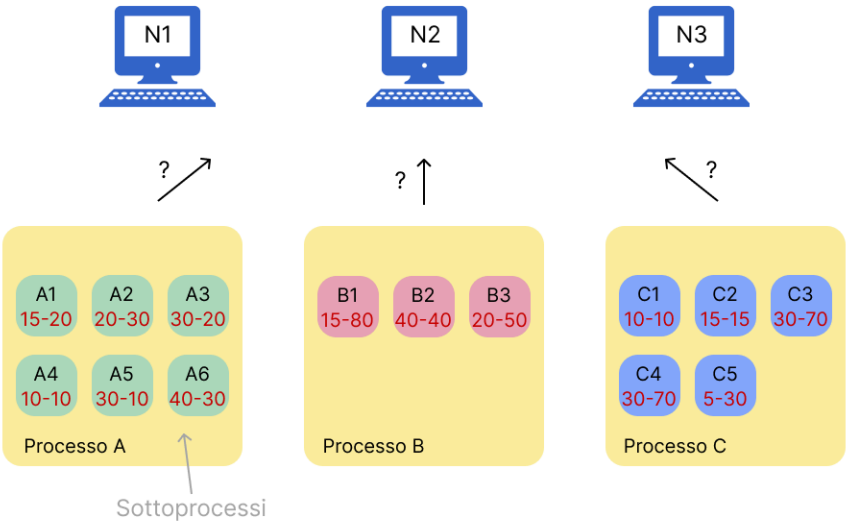


Figura 1: I sottoprocessi vanno assegnati a un nodo per essere eseguiti

Non è obbligatorio eseguire tutti i processi (condizione 3), ma se un processo viene eseguito allora tutti i suoi sottoprocessi devono essere eseguiti (condizione 1). Viceversa, se non viene eseguito allora nessuno dei suoi sottoprocessi può essere eseguito. I sottoprocessi possono essere eseguiti su nodi diversi (condizione 2) pagando una penalità.

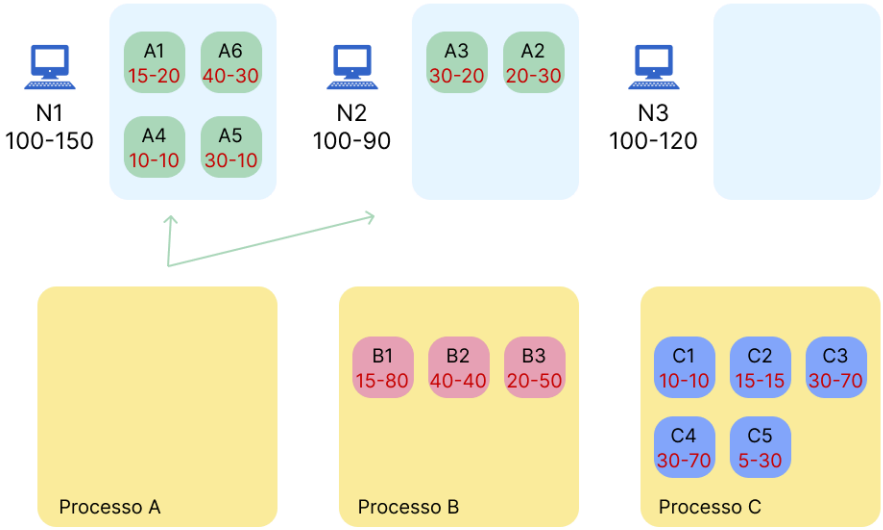


Figura 2: In figura il processo A viene diviso tra i nodi 1 e 2

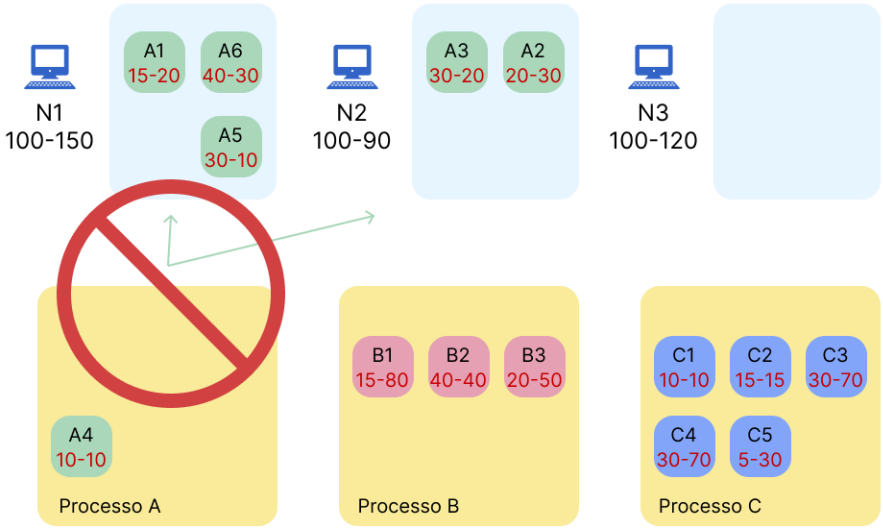


Figura 3: Non è possibile eseguire solo alcuni sottoprocessi di un processo

Ovviamente non è possibile assegnare troppi sottoprocessi a un nodo, altrimenti le sue risorse vengono esaurite (condizione 4).

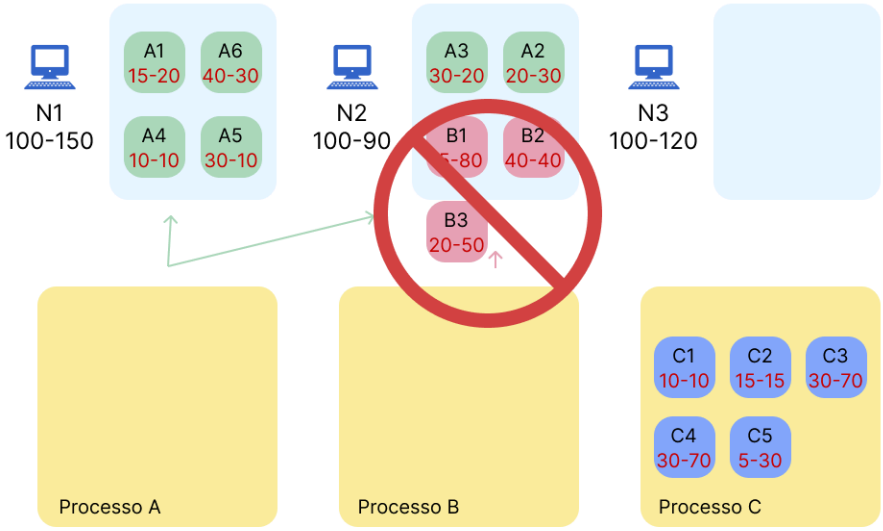


Figura 4: Non è possibile superare le capacità di un nodo

Riuscire ad eseguire più processi contemporaneamente è vantaggioso, perciò per ogni processo eseguito si ottiene un profitto (condizione 5). Esso viene associato al processo intero, piuttosto che ai singoli sottoprocessi, perché il vantaggio è legato al completamento del processo, che richiede l'esecuzione di tutti i sottoprocessi.

Normalmente si attribuirebbe un profitto a ogni oggetto, ma ci sono casi in cui il vantaggio si ottiene solo dall'insieme di oggetti, come viene spiegato da Mancini et al (2022 *[2]*) riferendosi a Chen e Zhang (2016 *[8]*): nel caso di una vacanza in montagna, il vantaggio che danno i singoli componenti della tenda, come picchetti, corde, bastoni e la tenda stessa, si ottiene solo se tutti gli oggetti vengono utilizzati; anche nel caso della consegna di un auto, il vantaggio si ottiene dividendo i pezzi in più trasporti, ma solo se tutte le parti vengono trasportate.

Inoltre, se un processo viene diviso tra più nodi (cioè se ha uno split), allora la sua esecuzione non sarà così efficiente come se fosse eseguito su un nodo singolo, perciò viene pagata una penalità (condizione 6). In particolare, la variante proposta da Mancini et al. (2021 *[1]*) prevede che la penalità associata alla famiglia venga applicata una volta sola, se la famiglia (il processo) non è contenuta in un singolo zaino (nodo). Questo significa che dividere una famiglia in due zaini comporta la stessa penalità che dividere la famiglia in tre o più zaini.

2.2 MMdKFSP seconda variante

Riprendendo il contesto dell'esempio precedente, siano dati 6 nodi e i due processi A e C. Indicando con il primo numero la quantità di CPU massima e con il secondo numero la quantità di memoria massima, i nodi hanno le seguenti capacità:

N1: 100-150
N2: 100-90
N3: 100-120
N4: 100-180
N5: 100-95
N6: 100-150

Assegnamo ad A un profitto pari a 50 e una penalità pari a 8, mentre a C un profitto di 35 e una penalità di 5.

Allocando i processi come in *Figura 5*, la somma dei due profitti, cioè 85, viene diminuita delle penalità delle due famiglie, cioè 8 e 5 (perché sono entrambe divise in due nodi), per ottenere un valore finale di 72.

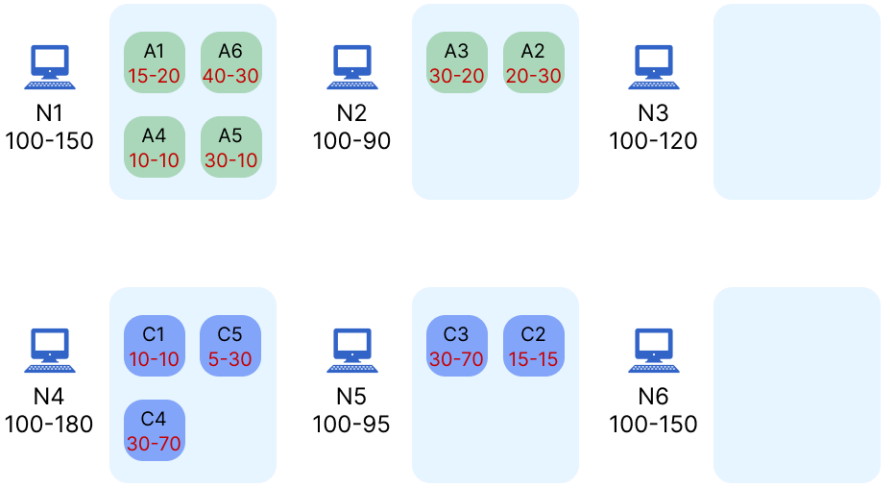


Figura 5: I processi A e C vengono divisi in due nodi ciascuno

Aumentando il numero di split dei processi (famiglie), come in figura *Figura 6*, la somma dei due profitti rimane la stessa, cioè 85, e viene diminuita anche in questo caso delle penalità dei due processi, cioè 8 e 5 (perché sono entrambi divisi in più nodi), per ottenere di nuovo un valore finale di 72.

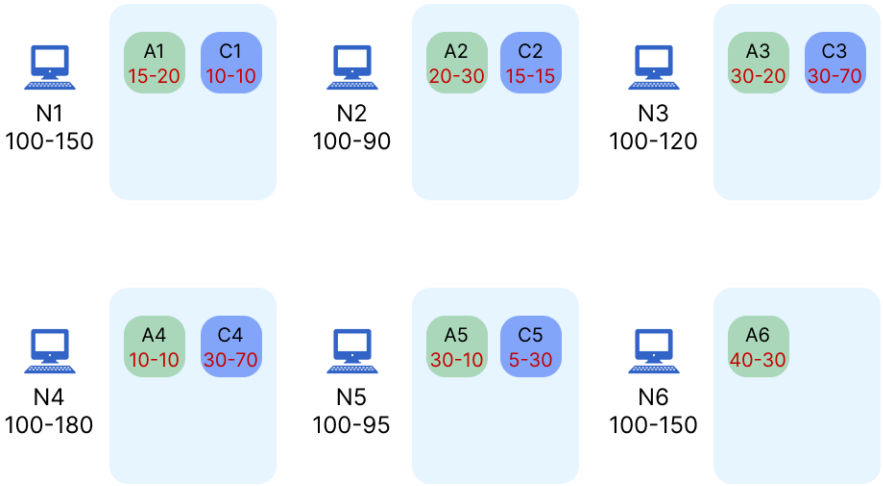


Figura 6: I processi A e C hanno tutti i sottoprocessi eseguiti su nodi diversi

In un caso reale i due esempi di assegnamento appena mostrati non possono avere la stessa penalità totale, perché il caso con tutti i sottoprocessi allocati su nodi diversi richiederebbe più comunicazione tra i nodi, risultando quindi più svantaggioso rispetto al caso in cui i sottoprocessi sono concentrati su pochi nodi (cioè con un numero ridotto di split).

Per questo motivo, in questa tesi viene proposta una seconda variante del problema, che deriva da una modifica della prima nel calcolo delle penalità: in questo caso, la penalità della

famiglia viene sottratta al profitto totale tante volte quante sono gli split della famiglia. Perciò, la penalità totale per ogni famiglia corrisponde al numero di zaini in cui gli oggetti della famiglia stessa sono inseriti, diminuito di uno (infatti, se la famiglia è tutta in uno zaino, non è divisa, e quindi non c'è penalità). Questo significa che, in questa formulazione, la penalità cresce linearmente al crescere del numero di zaini in cui la famiglia viene divisa.

Utilizzando questa definizione negli esempi in *Figura 5* e *Figura 6* si ha che:

- nel primo, i calcoli rimangono gli stessi del caso precedente, perché entrambi i processi hanno 1 split (sono divise una volta, in due nodi), ottenendo un valore finale di 72;
- nel secondo, il profitto totale dei due processi rimane 85, ma le penalità diventano $8 * 5 = 40$ (il processo A è diviso 5 volte in 6 nodi) e $5 * 4 = 20$ (il processo C è diviso 4 volte in 5 nodi), ottenendo un valore finale di 25.

In questo modo si riesce a catturare meglio il costo aggiuntivo che si ha eseguendo i sottoprocessi su più nodi, rispetto a quando sono concentrati su pochi nodi.

Lo stesso problema può essere applicato anche ad altre architetture informatiche, come quelle orientate ai servizi (SOA), ma anche in altri contesti, come la logistica o la produzione.

2.3 Descrizione formale prima variante

Dati $n, m \in N^+$, sia $I = \{1, ..., n\}$ un insieme di oggetti, sia $F = \{1, .., m\}$ l'insieme delle famiglie e sia $F_j \subseteq I$ l'insieme degli oggetti appartenenti alla famiglia $j \in F$. Si osservi che la collezione di insiemi $\{F_j\}_{j \in F}$ è una partizione dell'insieme degli oggetti I (ogni famiglia non può essere un insieme vuoto, ogni famiglia non può contenere oggetti che appartengano ad altre famiglie, l'unione di tutti i sottoinsiemi F_j deve corrispondere all'insieme degli oggetti I).

$$
\begin{aligned}
F_j \neq \emptyset, \qquad & \forall j \in F, \\
F_j \cap F_l = \emptyset, \qquad & \forall j, l \in F : j \neq l, \\
\bigcup_{j \in F} F_j = I. \qquad &
\end{aligned}
$$

Dati $k_{\max}, r_{\max} \in \mathbb{N}^+$, sia $K = \{1, ..., k_{\max}\}$ l'insieme degli zaini disponibili, ognuno con una capacità $C_{kr} \in \mathbb{N}$ per ogni tipo di risorsa $r \in R = \{1, ..., r_{\max}\}$.

Ogni oggetto $i \in I$ richiede una quantità $w_{ir} \in \mathbb{N}$ della risorsa $r \in R$ da allocare nello zaino $k \in K$.

La prima variante del problema dello zaino multidimensionale multiplo con penalità family-split (MMdKFSP) richiede di trovare la migliore assegnazione degli oggetti tra gli zaini disponibili al fine di massimizzare la differenza tra profitti e penalità di split, nelle seguenti condizioni:

1. Se un oggetto $i \in F_j (j \in F)$ è selezionato, allora tutti gli oggetti della famiglia F_j devono essere selezionati;

2. Gli oggetti di una qualsiasi famiglia possono essere assegnati a zaini diversi;
3. Ogni famiglia di oggetti può essere selezionata o meno;
4. Le capacità C_{kr} degli zaini $k \in K$ per ogni risorsa $r \in R$ non possono essere superate;
5. La selezione della famiglia $j \in F$ fornisce un profitto dato $p_j \in \mathbb{N}$;
6. La divisione (split) di una famiglia $j \in F$ tra più zaini comporta il pagamento di una penalità fissa che non dipende dal numero di split subito dalla famiglia ma dalla famiglia stessa, e il costo di penalità è indicato come δ_j;
7. Il numero massimo di split è pari al numero di zaini distinti in cui gli oggetti della famiglia sono stati inseriti, diminuito di uno.

2.4 Descrizione formale seconda variante

Dati $n, m \in N^+$, sia $I = \{1, ..., n\}$ un insieme di oggetti, sia $F = \{1, .., m\}$ l'insieme delle famiglie e sia $F_j \subseteq I$ l'insieme degli oggetti appartenenti alla famiglia $j \in F$. Si osservi che la collezione di insiemi $\{F_j\}_{j \in F}$ è una partizione dell'insieme degli oggetti I (ogni famiglia non può essere un insieme vuoto, ogni famiglia non può contenere oggetti che appartengano ad altre famiglie, l'unione di tutti i sottoinsiemi F_j deve corrispondere all'insieme degli oggetti I).

$$
\begin{aligned}
F_j \neq \emptyset, \qquad & \forall j \in F, \\
F_j \cap F_l = \emptyset, \qquad & \forall j, l \in F : j \neq l, \\
\bigcup_{j \in F} F_j = I. \qquad &
\end{aligned}
$$

Dati $k_{\max}, r_{\max} \in \mathbb{N}^+$, sia $K = \{1, ..., k_{\max}\}$ l'insieme degli zaini disponibili, ognuno con una capacità $C_{kr} \in \mathbb{N}$ per ogni tipo di risorsa $r \in R = \{1, ..., r_{\max}\}$.

Ogni oggetto $i \in I$ richiede una quantità $w_{ir} \in \mathbb{N}$ della risorsa $r \in R$ da allocare nello zaino $k \in K$.

La seconda variante del problema dello zaino multidimensionale multiplo con penalità family-split (MMdKFSP) richiede di trovare la migliore assegnazione degli oggetti tra gli zaini disponibili al fine di massimizzare la differenza tra profitti e penalità di split, nelle seguenti condizioni:

1. Se un oggetto $i \in F_j (j \in F)$ è selezionato, allora tutti gli oggetti della famiglia F_j devono essere selezionati;
2. Gli oggetti di una qualsiasi famiglia possono essere assegnati a zaini diversi;
3. Ogni famiglia di oggetti può essere selezionata o meno;
4. Le capacità C_{kr} degli zaini $k \in K$ per ogni risorsa $r \in R$ non possono essere superate;

5. La selezione della famiglia $j \in F$ fornisce un profitto dato $p_j \in \mathbb{N}$;
6. La divisione (split) di una famiglia $j \in F$ tra più zaini comporta il pagamento di una penalità fissa che non dipende dal numero di split subito dalla famiglia ma dalla famiglia stessa, e il costo di penalità è indicato come δ_j;
7. Il numero massimo di split è pari al numero di zaini distinti in cui gli oggetti della famiglia sono stati inseriti, diminuito di uno.

2.5 Letteratura

Per quanto riguarda il problema dello zaino, è già approfonditamente analizzato da Martello e Toth (1990 *[4]*) e Kellerer et al. (2004 *[5]*). L'articolo di Cacchiani et al. (2022a *[6]*) offre un'analisi dei problemi a zaino singolo, mentre una panoramica riguardo i problemi con zaini multipli e multidimensionali viene trattata da Cacchiani et al. (2022b *[7]*).

La prima comparsa del problema MMdKFSP avviene in Mancini et al. (2021 *[1]*), dove viene introdotto limitando il numero di risorse a due, viene proposto un algoritmo esatto per la sua risoluzione e viene fornito un primo gruppo di istanze benchmark. Successivamente Mancini et al. (2021 *[1]*) propongono nel 2022 un nuovo algoritmo esatto (in Mancini et al. 2022 *[2]*) che va a migliorare i risultati ottenuti nelle istanze di benchmark precedenti e, di conseguenza, a proporre nuove istanze. In questa tesi verranno usate istanze provenienti da entrambi i loro lavori per risolvere la prima variante.

Come indicato in Mancini et al. (2022 *[2]*), siccome MMdKFSP deriva direttamente dal problema dello zaino multiplo e dal problema dello zaino multidimensionale, è possibile trovare una trattazione di questi problemi in Chekuri and Khanna (2000 *[9]*), Dell'Amico et al. (2019 *[10]*), Ferreira et al. (1996 *[11]*), Fréville (2004 *[12]*), Pisinger (1999 *[13]*), e Martello e Toth (2003 *[14]*).

Alcuni altri problemi hanno delle somiglianze con questo, specialmente quelli in cui gli oggetti vengono raggruppati o partizionati, come hanno osservato anche Mancini et al. *[2]*. Infatti, affermano che, forse, quello più simile è il *packing Groups of Items into Multiple Knapsacks* (GMKP), affrontato da Chen e Zhang (2016 *[8]*), in cui devono essere selezionate partizioni di oggetti (come in questo caso) per massimizzare il profitto totale (assegnato al gruppo), tuttavia non viene considerata la penalità per la loro divisione tra gli zaini. Anche in Kataoka e Yamada (2014 *[15]*) gli oggetti vengono partizionati, nel problema *Multiple Knapsack Assignment Problem* (MKAP) che estende MKP, però il profitto è associato a ogni oggetto, a ogni zaino viene associata una partizione e possono essere inseriti oggetti appartenenti solo alla stessa partizione, e anche in questo caso non viene considerata la penalità per la divisione. Nel problema *Multiple Knapsack Problem with Setup* (MKPS), descritto da Cacchiani et al. (2022b *[7]*), gli oggetti appartengono a famiglie disgiunte e devono sostenere un costo (dipendente dalla coppia famiglia-zaino) di setup nel momento in cui vengono inserite in ogni zaino. Li et al. (2005 *[16]*) introducono il *Multidimensional Knapsack Problem with Generalized Upper Bound Constraints* (MdKP-GUB), in cui gli oggetti appartengono a categorie disgiunte, ma differisce dal problema corrente nel fatto che venga richiesto che al più un oggetto per sottoinsieme possa essere selezionato, quindi anche per il fatto che sia possibile non selezionare l'intera

famiglia, e, ancora una volta, perché non vengono considerate penalità per la divisione. In Yamada e Takeoka (2009 *[17]*) formulano il problema *Fixed-Charge Multiple Knapsack Problem* (FCMKP) come estensione di MKP (Kellerer et al. *[5]*, Martello e Toth *[4]*, Pisinger *[13]*), con l'aggiunta di un costo fisso associato a ogni zaino, quindi una sorta di penalità associata all'utilizzo dello zaino, che va perciò a vedere il problema da un altro lato, perché si deve decidere quali zaini utilizzare per minimizzare le penalità. Il *Class-Constrained Multiple Knapsack* definito in Shachnai e Tamir (2001 *[18]*) prevede nuovamente di massimizzare il profitto ottenuto dall'inserimento di oggetti in zaini, con l'aggiunta di un vincolo sul numero di classi di oggetti che uno zaino può contenere. Nel problema *Penalized Knapsack Problem* (Ceselli e Righini 2006 *[19]*), oltre a profitto e peso, a ogni oggetto viene associata anche una penalità, che viene poi usata nella funzione obiettivo per ridurre il profitto totale di una quantità pari alla penalità più alta tra gli oggetti selezionati.

Altre varianti associate a questo problema sono:

- il *Multi-Objective Multidimensional Knapsack Problem*, trattato da Lust e Teghem (2012 *[20]*), che deriva dal MdKP e definisce una funzione multi-obiettivo, in cui i profitti vengono massimizzati nel senso di Pareto, cioè l'obiettivo è allocare gli oggetti finché non si arriva a un punto in cui migliorare un qualunque obiettivo porterebbe al peggioramento di un altro;
- il *Multidimensional Multiple-Choice Knapsack Problem* (Ghasemi e Razzazi 2011 *[21]*), che generalizza il MdKP, in cui l'insieme degli oggetti è partizionato in classi e deve essere preso almeno un oggetto per ogni classe (trattato anche in Mansini e Zanotti 2020 *[22]*);
- il *All-Or-Nothing Generalized Assignment Problem* (Adany et al. 2016 *[23]*), in cui gli oggetti sono partizionati in gruppi e devono essere assegnati a un contenitore, ogni oggetto ha una dimensione e un'utilità associata al contenitore, ogni contenitore può contenere al massimo un oggetto per ogni gruppo, un gruppo risulta soddisfatto se tutti i suoi oggetti sono assegnati a un contenitore, lo spazio totale degli oggetti assegnati a un contenitore non può superare la sua capacità e l'obiettivo è trovare un assegnamento di un sottoinsieme degli oggetti ai contenitori in modo che l'utilità totale dei gruppi soddisfatti sia massima.

Capitolo 3: Modelli matematici

In questo capitolo saranno mostrate le formulazioni matematiche dei due problemi proposti nel capitolo precedente. A questo scopo, verranno introdotti alcuni concetti chiave di Programmazione Lineare e di Programmazione Lineare Intera.

3.1 Modelli di programmazione lineare

La programmazione lineare è una branca della ricerca operativa che si occupa di teoria e metodi per la ricerca di punti di massimo o di minimo di una funzione matematica lineare su un insieme definito. Il punto di partenza per la risoluzione di un problema di programmazione lineare (LP) è la definizione di un suo modello matematico. Un modello di programmazione lineare è costituito da:

- Variabili decisionali: variabili del modello di cui deve essere regolato il valore per ottenere il risultato ottimale.
- Funzione obiettivo: funzione matematica lineare, il cui valore è funzione delle variabili decisionali, che si vuole massimizzare o minimizzare.
- Vincoli: relazioni matematiche lineari che definiscono i limiti per i valori che le variabili decisionali possono assumere.

Se le variabili decisionali sono tutte continue, il problema è detto di programmazione lineare continua. Se le variabili decisionali sono tutte intere, si parla di programmazione lineare intera (PLI). Se, invece, alcune variabili decisionali sono continue e altre intere, si parla, più in generale, di un problema di programmazione lineare mista intera (PLMI).

Un modello di programmazione lineare continua espresso in forma canonica è rappresentato come segue:

$$\min \boldsymbol{c}^T \boldsymbol{x}$$

$$A\boldsymbol{x} \geq \boldsymbol{b}$$
$$\boldsymbol{x} \geq \boldsymbol{0}$$

$$\boldsymbol{c}, \boldsymbol{x} \in R^n, \boldsymbol{b} \in R^m, A \in R^{mxn}$$

Dove $\boldsymbol{x} \in R^n$ è il vettore delle variabili decisionali, $\boldsymbol{c}^T \in R^n$ è il vettore dei coefficienti della funzione obiettivo, $\boldsymbol{b} \in R^m$ è il vettore dei termini noti dei vincoli e $A \in R^{mxn}$ è la matrice dei coefficienti dei vincoli.

Un modello di programmazione lineare continua espresso in forma standard (con vincoli di uguaglianza), invece, è rappresentato come segue:

$$\min \boldsymbol{c}^T \boldsymbol{x}$$

$$A\boldsymbol{x} = \boldsymbol{b}$$
$$\boldsymbol{x} \geq \boldsymbol{0}$$

$$\boldsymbol{c}, \boldsymbol{x} \in R^n, \boldsymbol{b} \in R^m, A \in R^{mxn}$$

È possibile passare alla forma standard utilizzando delle variabili di slack, introducendo le seguenti modifiche.

Per i vincoli di $\leq$:
Il vincolo di $\leq$ diventa un vincolo di $=$, con l'aggiunta di una variabile di slack $s_1 \geq 0$.

$$3x_1 + 2x_2 \leq 18 \Longrightarrow 3x_1 + 2x_2 + s_1 = 18$$

Per i vincoli di $\geq$:
La stessa operazione si può fare per vincoli di $\geq$, cambiando il segno della disequazione e riconducendosi al caso precedente.

Per le variabili libere in segno:
Per le variabili che possono essere sia positive che negative, è sufficiente rappresentarle come differenza tra due variabili positive.

$$x_3 \in R \Longrightarrow x_3 = u_3 - v_3$$
$$u_3, v_3 \geq 0$$

Per i problemi di massimizzazione:
Un problema di massimizzazione può essere trasformato in un problema di minimizzazione moltiplicando la funzione obiettivo per -1, perché $\max(f(x)) = -\min(-f(x)))$

3.1.1 Variabili di base e non di base

Supponendo il problema in forma standard, con la matrice A di rango m:

- se $m = n \Longrightarrow$ esiste una sola soluzione del sistema di equazioni lineari $Ax = b \Longrightarrow x = A^{-1}b$.
- se $m < n \Longrightarrow$ esistono infinite soluzioni del sistema di equazioni lineari $Ax = b \Longrightarrow$ il valore di $n - m$ variabili può essere fissato arbitrariamente.

Se A è una matrice $m \times n$ con $m < n$, può essere suddivisa in due parti:

$$A = \left[\begin{array}{ccc|ccc} a_{1,1} & \dots & a_{1,m} & a_{1,m+1} & \dots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & \dots & a_{m,m} & a_{m,m+1} & \dots & a_{m,n} \end{array}\right] = (B|N)$$

$$B \in R^{mxm}, N \in R^{mx(m-n)}$$

Perciò anche il vettore delle variabili decisionali può essere suddiviso in due parti:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \\ \hline x_{m+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{pmatrix} \boldsymbol{x}_B \\ \boldsymbol{x}_N \end{pmatrix}$$

Quindi è possibile scomporre le variabili di una soluzione in due insiemi: le variabili di base sono quelle i cui indici sono presenti in B, mentre le variabili non di base sono quelle i cui indici sono presenti in N. Tramite delle permutazioni è possibile portare le variabili dentro o fuori dall'insieme di quelle di base.

Si definiscono, inoltre, *soluzioni di base* quelle per cui le variabili non di base sono nulle.

3.1.2 Metodo del simplesso

L'algoritmo più utilizzato per la risoluzione di problemi di programmazione lineare è il metodo del simplesso (Dantzig et al. 1955 *[24]*). È un algoritmo iterativo che parte da una soluzione di base iniziale, poi continua a spostarsi in una soluzione di base migliore della precedente finché non raggiunge l'ottimo globale.

Esso sfrutta il teorema fondamentale della programmazione lineare, il quale afferma che: se esiste una soluzione ammissibile, allora esiste una soluzione di base ammissibile; se esiste una soluzione ammissibile ottimale, allora esiste una soluzione di base ammissibile ottimale.

Perciò procede spostandosi da una soluzione di base all'altra, selezionando un insieme di variabili di base, e, iterativamente, sostituendo una di queste con una variabile non di base (tramite un'operazione chiamata *pivoting*). Per effettuare questa scelta il metodo si basa sul calcolo di alcuni coefficienti, detti *costi ridotti*, associati a ogni variabile, che indicano quanto varierebbe la funzione obiettivo se la variabile stessa venisse inserita in quelle di base (rimuovendone un'altra) in quell'iterazione. Le variabili che sono già tra quelle di base hanno costo ridotto nullo, mentre le altre possono avere un qualunque valore di costo ridotto: positivo, negativo o nullo. Per questo motivo, il metodo del simplesso termina quando tutti i costi ridotti delle variabili non di base indicano che la funzione obiettivo non può più essere migliorata ulteriormente.

Il vettore dei coefficienti di costo ridotto associato alla base B si definisce con la seguente formula:

$$\boldsymbol{c}^T - \boldsymbol{c}_B^T B^{-1} A = [\boldsymbol{c}_B^T - \boldsymbol{c}_B^T B^{-1} B, \boldsymbol{c}_N^T - \boldsymbol{c}_B^T B^{-1} N] = [\boldsymbol{0}^T, \boldsymbol{r}^T]$$

dove $\boldsymbol{c}_B^T$ e $\boldsymbol{c}_N^T$ derivano dalla divisione in due parti (di base e non di base), come visto in *Sezione 3.1.1*, dell'espressione della funzione obiettivo:

$$\boldsymbol{c}^T = \left[c_1 \ \cdots \ c_m \middle| c_{m+1} \ \cdots \ c_n\right] = (\boldsymbol{c}_B^T, \boldsymbol{c}_N^T)$$

Questi coefficienti sono un concetto chiave per il metodo della Kernel Search, che verrà illustrato in seguito. Infatti, per eseguire la fase di inizializzazione dell'algoritmo, sarà necessario ricavare i coefficienti di costo ridotto delle variabili fuori base con la formula:

$$\boldsymbol{r}^T = \boldsymbol{c}_N^T - \boldsymbol{c}_B^T B^{-1} N$$

L'informazione fornita dal costo ridotto riguardo ciascuna variabile sarà fondamentale per aiutare gli algoritmi proposti a trovare le variabili più importanti per la soluzione del problema.

3.2 Modelli di programmazione lineare intera

I problemi che verranno affrontati in questa tesi appartengono alla classe dei problemi di programmazione lineare intera, perché le variabili dei loro modelli matematici possono assumere soltanto valori interi.

Sia c una funzione di costo:

$$c : F \to N$$

Un problema di ottimizzazione discreta di massimo consiste nel trovare $x \in F$ con $F :=$ insieme discreto tale che:

$$c(x) \leq c(y), \forall y \in F$$

Un'istanza di un problema di ottimizzazione discreta consiste nella coppia (F, c) dove F è l'insieme delle soluzioni ammissibili e c è la funzione di costo.

Si definisce problema di programmazione lineare intera in forma canonica di minimo il seguente:

$$\min \boldsymbol{c}^T \boldsymbol{x}$$

$$A\boldsymbol{x} \geq \boldsymbol{b}$$
$$\boldsymbol{x} \geq \boldsymbol{0}$$

$$\boldsymbol{c}, \boldsymbol{x} \in Z^n, \boldsymbol{b} \in Z^m, A \in Z^{mxn}$$

Questa classe di problemi risulta essere più complessa da risolvere rispetto ai problemi di programmazione lineare continua. Per motivare questa affermazione, si consideri il seguente esempio in due dimensioni (due variabili decisionali). Nel caso della programmazione lineare continua, le soluzioni possibili sono infinite e si trovano nell'area colorata (regione ammissibile):

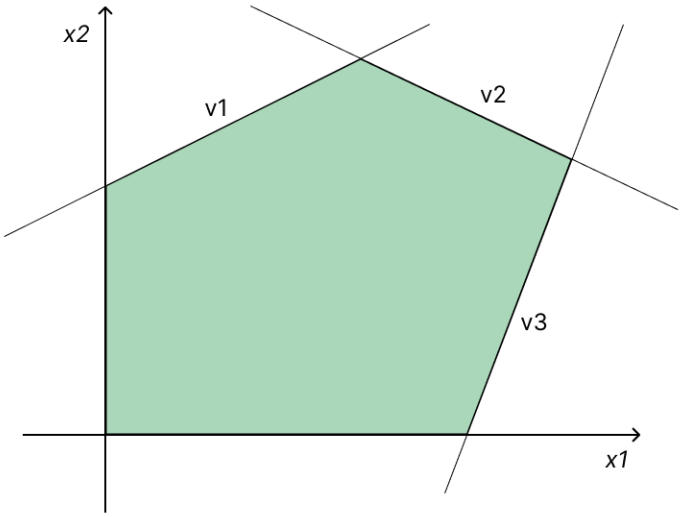


Figura 7: x1 e x2 sono le variabili decisionali, v1, v2 e v3 sono i vincoli del problema

La teoria della programmazione lineare continua garantisce che la soluzione ottima si trova in uno dei vertici della regione ammissibile, perciò è sufficiente valutare la funzione obiettivo in questi punti (che è la procedura attuata dal metodo del simplesso).

Nel caso della programmazione lineare intera, invece, le soluzioni ammissibili sono i punti con coordinate intere all'interno dell'area colorata, perciò non è detto che la soluzione ottima si trovi in uno dei vertici. Nella figura seguente è rappresentata la griglia delle soluzioni intere ammissibili per lo stesso esempio precedente:

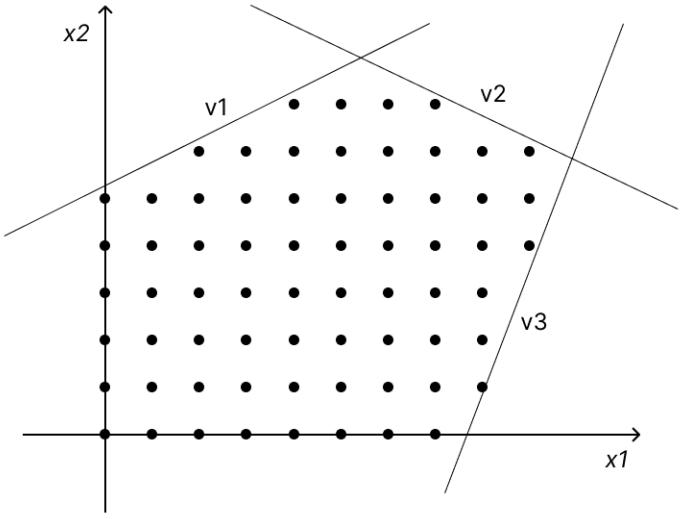


Figura 8: La regione ammissibile consiste esclusivamente nei punti a coordinate intere

Per trovare la soluzione ottima, in questo caso, si procede eliminando il vincolo di interezza delle variabili, ottenendo, quindi, un problema di programmazione lineare continua, che viene

anche chiamato rilassamento continuo del problema originale (oppure anche problema rilassato).

Il rilassamento continuo può essere risolto con il metodo del simplesso: se risulta impossibile allora anche il problema intero è impossibile; se risulta illimitato allora anche il problema intero è illimitato; se la soluzione ottima del rilassato ha tutte le variabili con valori interi allora è ottima anche per il problema intero. Il caso più probabile è che il rilassato abbia una soluzione con alcune o tutte le variabili che assumono un valore decimale, perciò è necessario procedere con ulteriori tecniche per capire come arrivare all'ottimo intero (per esempio, utilizzando un metodo di Branch and Bound).

Negli algoritmi che verranno illustrati, la soluzione del problema rilassato verrà utilizzata come punto di riferimento per indirizzare la ricerca della soluzione migliore.

Inoltre, i problemi di programmazione lineare intera, a differenza di quelli di programmazione lineare continua, hanno una complessità computazionale di classe NP-hard. Infatti, non esiste un algoritmo che possa risolvere in maniera esatta questi problemi in tempi polinomiali.

3.3 Modelli di programmazione lineare mista intera

Nel caso più generale, in cui il vincolo di interezza è relativo soltanto ad alcune delle variabili del modello, si parla di problemi di programmazione lineare mista intera.

Si definisce problema di programmazione lineare mista intera in forma canonica il seguente:

$$\min \boldsymbol{c}^T \boldsymbol{x} + \boldsymbol{d}^T \boldsymbol{y}$$

$$A\boldsymbol{x} + B\boldsymbol{y} \geq \boldsymbol{b}$$
$$\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}$$

$$\boldsymbol{x} \in Z^n \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{y} \in R^n \boldsymbol{b} \in R^m, A \in R^{mxn}, B \in R^{mxn}$$

Le metodologie risolutive per questi problemi sono le stesse dei problmi di PLI, trattandosi di un caso più generale. Tipicamente, per risolvere un problema PLMI, viene usato un algoritmo di Branch and Bound. Durante la sua esecuzione, esso va a generare una serie di sottoproblemi che, man mano, vengono risolti, per avvicinarsi sempre di più alla soluzione ottima.

3.3.1 Risoluzione di modelli di PL, PLI e PLMI mediante risolutori

Per risolvere le tipologie di problemi appena presentate (PL, PLI e PLMI) esistono dei programmi commerciali, che permettono di definire il modello matematico tramite un linguaggio di programmazione e implementano degli algoritmi esatti sofisticati per raggiungere la soluzione migliore il più velocemente possibile.

Il risolutore che verrà usato in questa tesi è Gurobi Optimizer. Questo risolutore viene fornito tramite un programma che mette a disposizione un'interfaccia (API) per diversi linguaggi di programmazione, tra cui Python. Tra le varie funzioni fornite da questa API, è possibile definire il modello matematico, impostare i parametri di esecuzione e ottenere la soluzione calcolata.

Per aumentare l'efficienza della risoluzione, Gurobi implementa un algoritmo esatto di Branch and Cut, che è una combinazione tra Branch and Bound e Cutting Planes, con ulteriori accorgimenti:

- Una fase di presolve, che permette di semplificare il modello matematico, per esempio eliminando variabili e vincoli ridondanti;
- Delle euristiche, che vanno a individuare ulteriori soluzioni durante l'esecuzione di Branch and Cut;
- Il parallelismo, infatti i sottoproblemi generati da Branch and Cut possono essere risolti in parallelo, sfruttando le potenzialità di macchine con più core;
- Altre tecniche, come selezione delle variabili o individuazione di simmetrie, che permettono di ridurre il numero di sottoproblemi che Branch and Cut deve esplorare.

3.4 Modello della prima variante di MMdKFSP

In questo paragrafo viene presentato il modello matematico della prima delle due varianti descritte nel capitolo precedente (descritta in *Sezione 2.3*).

Il modello fa uso delle seguenti variabili:

$$x_j = \begin{cases} 1 \text{ se la famiglia } j \in F \text{ è selezionata,} \\ 0 \text{ altrimenti.} \end{cases}$$

$$y_{ik} = \begin{cases} 1 \text{ se l'oggetto } i \in I \text{ è allocato nello zaino } k \in K, \\ 0 \text{ altrimenti.} \end{cases}$$

$$z_{jk} = \begin{cases} 1 \text{ se almeno un oggetto della famiglia } j \in F \text{ è allocato nello zaino } k \in K, \\ 0 \text{ altrimenti.} \end{cases}$$

$$u_j = \begin{cases} 1 \text{ se gli oggetti della famiglia } j \in F \text{ non sono allocati tutti nello stesso zaino,} \\ 0 \text{ se tutti gli oggetti della famiglia } j \in F \text{ sono allocati nello stesso zaino} \end{cases}$$

La formulazione proposta dal paper di Mancini et al. (2022 [2]), modificando il termine "dimensioni" con "risorse", è la seguente:

$$\max \sum_{j \in F} p_j x_j - \sum_{j \in F} \delta_j u_j \tag{1}$$

$$\sum_{k \in K} y_{ik} = x_j \qquad \forall i \in I, \forall j | i \in F_j, \tag{2}$$

$$\sum_{i \in I} c_i^r y_{ik} \leq C_{kr} \qquad \forall k \in K, \forall r \in \{1, ..., R\}, \tag{3}$$

$$z_{jk} \geq \frac{1}{|F_j|} \sum_{i \in F_j} y_{ik} \qquad \forall j | F_j \in F, \forall k \in K, \tag{4}$$

$$u_j \geq \frac{1}{|K|-1} \left(\sum_{k \in K} z_{jk} - 1 \right) \qquad \forall j | F_j \in F, \tag{5}$$

$$x_j, u_j \in \{0, 1\} \qquad \forall j | F_j \in F, \tag{6}$$

$$z_{jk} \in \{0, 1\} \qquad \forall j | F_j \in F, \forall k \in K, \tag{7}$$

$$y_{ik} \in \{0, 1\} \qquad \forall i \in I, \forall k \in K. \tag{8}$$

Il modello massimizza la somma dei profitti delle famiglie selezionate diminuito della somma delle penalità pagate per ogni famiglia che non è allocata interamente nello stesso zaino. I vincoli (2) assicurano che se una famiglia è selezionata, ogni oggetto che le appartiene sia allocato in uno e un solo zaino. I vincoli (3) costringono il rispetto dei limiti di capacità di tutti gli zaini, per tutte le risorse considerate. I vincoli (4) identificano se uno zaino contiene oggetti di una data famiglia e forzano la variabile z_{jk} ad assumere valore 1, se almeno un item della famiglia j è contenuto nello zaino k. I vincoli (5) permettono alla formulazione di rilevare se una famiglia è allocata in un solo zaino o divisa in due o più: se la sommatoria delle z_{jk} al secondo termine fosse pari a 1 (implicando che la famiglia j è contenuta in un solo zaino) il vincolo diventerebbe $u_j \geq 0$ e, grazie alla funzione obiettivo che penalizza il valore delle variabili u_j, quest'ultima verrebbe settata a zero. I vincoli dal (6) all'(8) definiscono i domini delle variabili.

3.5 Modello della seconda variante di MMdKFSP

In questo paragrafo viene presentato il modello matematico della seconda delle due varianti, quella che viene introdotta per la prima volta in questa tesi, descritte nel capitolo precedente (descritta in *Sezione 2.4*).

Il nuovo modello, proposto in questa tesi, fa uso delle seguenti variabili:

$$x_j = \begin{cases} 1 \text{ se la famiglia } j \in F \text{ è selezionata,} \\ 0 \text{ altrimenti.} \end{cases}$$

$$y_{ik} = \begin{cases} 1 \text{ se l'oggetto } i \in I \text{ è allocato nello zaino } k \in K \\ 0 \text{ altrimenti.} \end{cases}$$

$$z_{jk} = \begin{cases} 1 \text{ se almeno un oggetto della famiglia } j \in F \text{ è allocato nello zaino } k \in K \\ 0 \text{ altrimenti.} \end{cases}$$

$s_j \in \mathbb{N}$: quante volte la famiglia $j \in F$ è stata divisa tra più di uno zaino.

La formulazione è la seguente:

$$\max \sum_{j \in F} (p_j x_j - \delta_j s_j) \tag{1}$$

$$\sum_{k \in K} y_{ik} = x_j, \qquad \forall j \in F, \forall i \in F_j, \tag{2}$$

$$\sum_{i \in I} w_{ir} y_{ik} \leq C_{kr}, \qquad \forall k \in K, \forall r \in R, \tag{3}$$

$$\sum_{i \in F_j} y_{ik} \leq |F_j| z_{jk}, \qquad \forall j \in F, \forall k \in K, \tag{4}$$

$$\sum_{k \in K} z_{jk} - 1 \leq s_j, \qquad \forall j \in F \tag{5}$$

$$x_j, y_{ik}, z_{jk} \in \{0, 1\}, \qquad \forall i \in I, \forall j \in F, \forall k \in K, \tag{6}$$

$$s_j \in \mathbb{N} \qquad \forall j \in F. \tag{7}$$

Il modello massimizza la somma dei profitti delle famiglie selezionate diminuito della somma delle penalità pagate per ogni famiglia che non è allocata interamente nello stesso zaino. I vincoli (2) assicurano che ogni oggetto sia assegnato a esattamente uno zaino se e solo se la famiglia a cui appartiene è selezionata (tra uno o più zaini). I vincoli (3) assicurano che per ogni zaino non venga superata la capacità massima per ogni risorsa considerata. I vincoli (4) di tipo "BigM" impostano il valore delle variabili z_{jk} ($j \in F, k \in K$) a 1 se almeno un elemento $i \in F_j$ è assegnato allo zaino k. I vincoli (5) fissano il valore delle variabili s_j ($j \in F$) pari al numero di volte in cui ogni famiglia viene allocata in più di uno zaino. Entrambi i vincoli (4) e (5) si basano sulla massimizzazione della funzione obiettivo e sul fatto che δ_j sia positivo. Infine, (6) e (7) definiscono il dominio di ciascuna variabile. Va evidenziato che, rispetto al modello precedente, le variabili s_j relative agli split delle famiglie non sono più binarie ma intere non negative.

Capitolo 4: Algoritmo risolutivo per la variante con penalità fissa

In questo capitolo verrà mostrato l'algoritmo euristico sviluppato per risolvere la versione nota dei problemi descritti nel secondo capitolo, spiegandone il funzionamento e mostrando le parti di codice più importanti. Prima di fare ciò, verranno introdotte alcune nozioni teoriche sulle principali categorie di algoritmi esistenti, seguite dal richiamo alle idee di fondo dalla quale derivano gli algoritmi proposti anche nel prossimo capitolo.

Il codice sorgente degli algoritmi è scritto in Python ed è disponibile al seguente repository GitHub: https://github.com/perno97/mmdkfsp_unibs_master_degree

4.1 Problema di ottimizzazione combinatoria

Sia $N = \{1, ..., n\}$ un insieme finito e F una collezione di sottoinsiemi ammissibili di N. Un peso (costo) c_l è associato a ogni $j \in N$. Si definisce un problema di ottimizzazione combinatoria la ricerca di un sottoinsieme S con peso (costo) minimo:

$$\min_{S \subseteq N} \left\{ \sum_{j \in S} c_j : S \in F \right\}$$

Esso può essere formulato come un problema di programmazione lineare intera (oppure come problema di programmazione lineare mista intera).

4.2 Algoritmi euristici

Gli algoritmi esatti per la risoluzione di problemi di ottimizzazione combinatoria spesso richiedono tempi di calcolo esponenziali rispetto alla dimensione del problema, come, per esempio, gli algoritmi Branch and Bound e Branch and Cut. Per questo motivo, in molti casi, è preferibile utilizzare dei metodi che siano in grado di fornire soluzioni accettabili in tempi ragionevoli, pur non garantendo la soluzione ottima. Questi metodi sono chiamati algoritmi euristici.

Esistono due famiglie di algoritmi euristici:

- Algoritmi costruttivi: partono da una soluzione vuota e, iterativamente, aggiungono elementi fino a costruire una soluzione valida.
- Algoritmi di ricerca locale: partono da una soluzione valida e, iterativamente, cercano di migliorarla modificandola (muovendosi nel suo vicinato), finché non raggiungono una soluzione che non può più essere migliorata (ottimo locale).

Il vicinato si definisce come una funzione che associa a una soluzione un insieme di soluzioni ad essa vicine. Di fatto esso dipende da come viene definita una *mossa* per il problema affrontato. Nel problema che andremo a trattare saranno definiti due tipi di vicinato: nel primo, una mossa verrà definita come la rimozione di l famiglie dalla soluzione corrente, che verranno sostituite da altre casuali, quindi il vicinato di una soluzione sarà l'insieme di quelle che si possono ottenere rimuovendo l famiglie e aggiungendone altre; nel secondo, una mossa verrà definita come lo spostamento di l oggetti da uno zaino a un altro, quindi il vicinato di una soluzione sarà l'insieme di quelle che si possono ottenere spostando l oggetti qualunque;

Prima di mostrare un esempio numerico per entrambi i tipi di vicinato è necessario introdurre alcuni concetti: una soluzione viene definita come un vettore di n elementi, con n pari al numero di oggetti, in cui l'indice di ogni elemento rappresenta l'indice dell'oggetto e il valore dell'elemento rappresenta l'indice dello zaino in cui è stato inserito l'oggetto. Se un oggetto non è inserito in nessuno zaino, allora viene indicato il valore -1.

Negli esempi seguenti, i primi quattro oggetti appartengono alla prima famiglia, i successivi due alla seconda, il settimo alla terza, l'ottavo alla quarta e gli ultimi due alla quinta. Gli zaini disponibili sono numerati da 1 a 5.

Nel caso del primo vicinato, con $l = 2$ e soluzione corrente = `(1,2,2,1,-1,-1,4,-1,3,5)`, rimuovendo la prima e la terza famiglia e aggiungendo la seconda e la quarta, una possibile soluzione vicina è = `(-1,-1,-1,-1,3,1,-1,1,3,5)`. Rimuovendo, invece, la terza e la quinta famiglia e aggiungendo solo la seconda, una possibile soluzione vicina è = `(1,2,2,1,2,5,-1,-1,-1,-1)`. Si parla di "possibile" soluzione vicina perché gli zaini utilizzati dalle nuove famiglie potrebbero anche essere differenti da quelli indicati.

Nel caso del secondo vicinato, con $l = 2$ e soluzione corrente = `(1,2,2,1,-1,-1,4,-1,3,5)`, muovendo gli oggetti 3 e 7 negli zaini 4 e 1, rispettivamente, si ottiene la soluzione vicina = `(1,2,4,1,-1,-1,1,-1,3,5)`. Muovendo, invece gli oggetti 1 e 10 negli zaini 5 e 1, rispettivamente, si ottiene la soluzione vicina = `(5,2,2,1,-1,-1,4,-1,3,1)`.

4.3 Meta-euristiche

Il problema principale degli algoritmi euristici è che si bloccano facilmente in soluzioni non ottime che non riescono a migliorare (ottimi locali), per questo motivo sono state introdotte le tecniche meta-euristiche. Queste sono algoritmi che, come quelli euristici, cercano di trovare la soluzione migliore, implementando anche dei metodi per riuscire a evitare di rimanere bloccati in ottimi locali e dei metodi per evitare di esplorare delle soluzioni che sono già state visitate.

Alcuni degli algoritmi che verranno illustrati sono delle meta-euristiche. Essi si basano su Variable Neighborhood Decomposition Search (VNDS, Hansen et al. 2017 *[25]*), che è una variante di Variable Neighborhood Search (VNS), per cui verrà introdotto un breve approfondimento su entrambe le meta-euristiche e sulla loro differenza.

4.3.1 VNDS

La Variable Neighborhood Search (VNS), è un metodo meta-euristico per risolvere problemi di ottimizzazione combinatoria. Esso combina la ricerca locale con dei cambi sistematici del vicinato durante l'esecuzione, con fasi di allontanamento dagli ottimi locali.

Come descritto in Hansen et al. (2001 *[26]*) Il metodo riceve in ingresso una soluzione iniziale x ottenuta da un metodo costruttivo, richiede la definizione di $N_l, l = 1, ..., l_{\max}$ intorni diversi, caratterizzati da mosse diverse, e devono essere definiti i criteri di terminazione.

Verrà usata la notazione $N_l(x)$ per indicare l'insieme delle soluzioni vicine a x nel l-esimo vicinato.

L'algoritmo ripete le seguenti fasi finché non viene soddisfatto un criterio di terminazione:
1. Imposta $l = 1$
2. Finché $l \leq l_{\max}$:
   - Fase di **shaking**: genera casualmente una soluzione x' nel l-esimo vicinato della soluzione corrente ($x' \in N_l(x)$), in modo da non restare bloccati in un eventuale ottimo locale;
   - Fase di **improvement**: in cui utilizza un metodo di ricerca locale con x' come soluzione iniziale per trovare una soluzione migliore (ottimo locale), ottenendo x'';
   - Fase di **cambio del vicinato**: se la nuova soluzione è migliore di quella corrente, allora si sposta nella prima ($x \leftarrow x''$), cioè la soluzione corrente diventa la nuova soluzione, e continua la ricerca con N_1 ($l = 1$), altrimenti imposta $l = l + 1$

Una variante di questo metodo è la Variable Neighborhood Decomposition Search (VNDS) che va a introdurre un'ulteriore fase, chiamata di decomposizione, con lo scopo di ridurre la complessità del problema nella fase di improving.

Per decomposizione si intende che per una soluzione data x, durante la fase di improvement (ricerca locale) vengono fissati tutti i suoi attributi tranne l, scelti casualmente. Perciò, nel caso $l = 1$, l'algoritmo di ricerca locale dovrà risolvere un problema mono-dimensionale, variando soltanto l'attributo che è stato scelto casualmente. Quando termina la fase di improving, il nuovo valore dell'attributo viene restituito e inserito nella soluzione x, fissata prima della decomposizione.

Perciò, la fase di decomposizione riceve in ingresso la soluzione risultante dalla fase di shaking, la decompone, passa il risultato alla ricerca locale e, infine, ricompone la soluzione ottenuta dall'improvement con la parte fissata precedentemente.

VNDS è in grado di migliorare congiuntamente l'efficienza (tempi di calcolo) e l'efficacia (qualità delle soluzioni) dell'algoritmo VNS di base.

Come VNS, il metodo riceve in ingresso una soluzione iniziale x ottenuta da un metodo costruttivo, richiede la definizione di $N_l, l = 1, ..., l_{\max}$ intorni diversi, caratterizzati da mosse diverse, e devono essere definiti i criteri di terminazione.

L'algoritmo ripete le seguenti fasi finché non viene soddisfatto un criterio di terminazione:
1. Imposta $l = 1$
2. Finché $l \leq l_{\max}$:
   - Fase di **shaking**: genera casualmente una soluzione x' nel l-esimo vicinato della soluzione corrente ($x' \in N_l(x)$), in modo da non restare bloccati in un eventuale ottimo locale; in altre parole, sia y un insieme di l attributi presenti nella soluzione x' ma non in x ($y = x' \setminus x$);
   - Fase di **improvement**: in cui utilizza un metodo di ricerca locale nello spazio di y per trovare una soluzione migliore (ottimo locale); definiamo y' la soluzione trovata e x'' la corrispondente soluzione nello spazio complessivo delle soluzioni ($x'' = (x' \setminus y) \cup y'$)
   - Fase di **cambio del vicinato**: se la nuova soluzione è migliore di quella corrente, allora si sposta in quella ($x \leftarrow x''$), cioè la soluzione corrente diventa la nuova soluzione, e continua la ricerca con N_1 ($l = 1$), altrimenti imposta $l = l + 1$

In seguito viene mostrato lo pseudocodice dell'algoritmo VNDS:

```
def VNDS(current_solution, l_max)
  while not is_stopping_criterion_satisfied:
    l = 1
    while l <= l_max:
      # Shaking
      shaked_solution = genera casualmente una soluzione nel l_esimo
                        vicinato della soluzione corrente

      # Decomposizione
      decomposed_solution = decompone la shaked_solution fissando il
                            valore di n-l variabili

      # Improvement
      improvement_solution = esegue una ricerca locale sulla
                            decomposed_solution

      # Ricomposizione
      recomposed_solution = ricompone la improvement_solution con la
                            parte fissata della shaked_solution

      # Cambio del vicinato
      if new_solution > current_solution:
        current_solution = new_solution
        l = 1
      else
        l = l + 1
```

4.3.2 VND

Come algoritmo di ricerca locale per VNDS è stato scelto il Variable Neighborhood Descent (VND). Anche questo metodo è una variante di VNS, in cui, però, non viene implementata la fase di shaking.

Anche VND richiede una soluzione iniziale x (nel nostro caso verrà fornita dall'algoritmo VNS esterno) e la definizione di $N'_l, l = 1, ..., l'_{\max}$ intorni diversi, caratterizzati da mosse diverse. Siccome è stato scelto di usare una strategia di tipo *best improvement*, la terminazione di VND avviene quando il ciclo più interno non trova una soluzione migliore di quella corrente.

Quindi l'algoritmo ripete le seguenti fasi finché non avvengono più miglioramenti della soluzione.

1. Imposta $l = 1$
2. Finché $l \leq l'_{\max}$:
   - Fase di esplorazione del vicinato: in cui cerca il miglior vicino x' di x nell'intorno N'_l ($x' \in N'_l(x)$);
   - Fase di cambio del vicinato: se la nuova soluzione è migliore di quella corrente, allora si sposta in quella ($x \leftarrow x'$), cioè la soluzione corrente diventa la nuova soluzione, e continua la ricerca con N_1 (imposta $l = 1$), altrimenti imposta $l = l + 1$

```
def VND(current_solution, l_vnd_max)
  while not stop:
    l = 1
    while l <= l_vnd_max:
      # Improvement
      new_solution = cerca il miglior vicino della soluzione corrente
      # Cambio del vicinato
      if new_solution > current_solution:
        current_solution = new_solution
        l = 1
      else
        l = l + 1
    # Termina se non ci sono miglioramenti
    if new_solution <= current_solution:
      stop = True
```

4.4 Math-euristiche

Questa categoria di algoritmi nasce dopo i metodi esatti, le euristiche costruttive, gli algoritmi di ricerca locale e le meta-euristiche. Con questo gruppo si intendono quegli algoritmi che vanno a combinare metodi euristici con metodi esatti, per risolvere problemi di ottimizzazione combinatoria. L'obiettivo principale di questi algoritmi è quello di riuscire a risolvere problemi generali, basandosi solo sulla loro formulazione matematica e non essendo vincolati a particolari problemi (al più possono essere collegati a una classe di problemi). Appartengono a questa categoria metodi quali Local Branching (2003 [27]), RINS (2005 [28]), Kernel Search (2010 [3]) e Proximity Search (2014 [29]). Per la loro esecuzione sfruttano un risolutore di problemi di ottimizzazione lineare mista intera, quali CPLEX, Gurobi o altri.

4.4.1 Kernel Search

Il metodo Kernel Search (Mansini et al. [3]) è una matheuristic che verrà implementata e utilizzata in questa tesi. L'idea di base di questo metodo è che spesso ci sono poche variabili non nulle nella soluzione ottima di un problema, e la soluzione del rilassato è un buon indicatore di quali siano queste variabili. Pertanto il metodo ricerca un insieme di variabili promettenti (che

probabilmente non saranno nulle nella soluzione ottima) chiamato *kernel*, per poi costruire una serie di problemi di ottimizzazione lineare mista intera più piccoli, che includano le variabili del kernel e alcune altre, e risolverli con un programma di ottimizzazione.

Il metodo, quindi, è composto da due blocchi. A livello più alto c'è il framework euristico, che si occupa di preparare i sottoproblemi e di prendere le informazioni dalle loro soluzioni, mentre a livello più basso c'è il risolutore (Gurobi, CPLEX), che li dovrà risolvere.

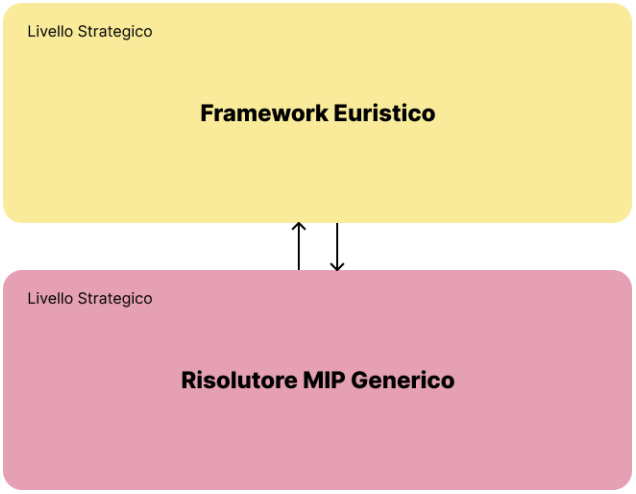


Figura 9: I due blocchi della Kernel Search

Il framework euristico svolge il suo compito in due fasi: la prima è detta di inizializzazione, la seconda è detta di improvement e cerca di migliorare quanto trovato (se trovato) dalla prima.

Fase 1: inizializzazione

Nella fase di inizializzazione deve riuscire a identificare le variabili che andranno inizialmente nel kernel e a ottenere, attraverso la risoluzione di un prima problema ristretto, una soluzione iniziale.

Per determinare l'ordinamento delle variabili, il metodo risolve il rilassamento continuo del problema iniziale (completo), ottenendo la sua soluzione ottima. Poi, ottiene il valore delle variabili di base e i valori dei costi ridotti di quelle non di base di questa soluzione. Infine, ordina le variabili, mettendo per prime quelle di base con valore non crescente, seguite da quelle non di base con modulo del costo ridotto non decrescente.

In *Figura 10* è possibile vedere un esempio di ordinamento delle variabili, nel caso esse siano nove: tramite i rettangoli con sfondo blu vengono rappresentati i valori assunti nella soluzione ottima del rilassato dalle variabili di base (ordinati in modo non crescente), mentre tramite i rettangoli con sfondo rosso vengono rappresentati i valori assoluti dei costi ridotti delle variabili non di base (ordinati in modo non decrescente).

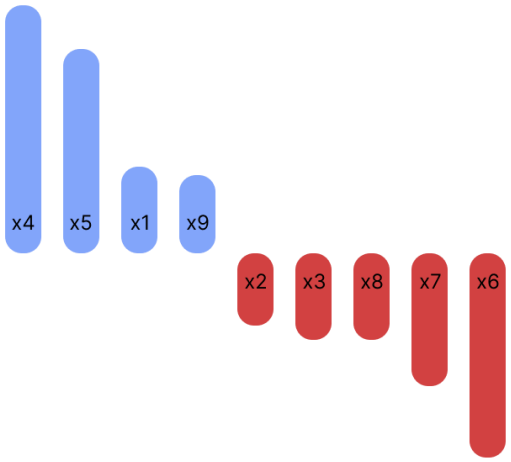


Figura 10: Esempio di ordinamento delle variabili

Per costruire il kernel, vengono prese dall'ordinamento le variabili che hanno valore non nullo nella soluzione ottima del rilassato.

Una volta ottenuto il kernel, il framework euristico costruisce il sottoproblema formato dalle sole variabili del kernel (tutte le rimanenti variabili presenti nel problema iniziale sono settate a zero) e utilizza il risolutore per ottenere la soluzione ottima iniziale e il suo valore.

L'ultimo passaggio della prima fase consiste nel partizionare le variabili non nel kernel in gruppi chiamati *bucket*, mantenendo l'ordine stabilito in precedenza. Come effettuare questo passaggio è un aspetto che va definito a seconda del problema affrontato. Infatti, è possibile decidere se: usare bucket grandi o piccoli; con dimensione fissa o variabile; avere tanti o pochi bucket; condividere le variabili tra i bucket o tenerli disgiunti; scorrere i bucket più di una volta; o altro.

Fase 2: miglioramento

Nella seconda fase, quella di miglioramento, il framework ripete i seguenti passaggi:

- Costruisce l'insieme di variabili formato dal kernel e dal bucket corrente (partendo dal primo bucket)
- Risolve il sottoproblema formato dalle variabili appena definite e da alcuni vincoli aggiuntivi
- Se il risolutore trova una soluzione valida per il sottoproblema:
  - Aggiorna la soluzione corrente
  - Aggiorna il kernel, inserendo le variabili del bucket corrente che hanno valore non nullo nella soluzione trovata
- Se ci sono ancora bucket da esaminare, ripete i passaggi con il bucket successivo, altrimenti termina.

I vincoli aggiuntivi che vengono passati al risolutore mirano a migliorarne l'efficienza. Essi sono due:

- il primo vincola il risolutore a ottenere una soluzione con valore della funzione obiettivo migliore o uguale a quello della soluzione corrente (quindi non peggiore);

- il secondo forza il risolutore a selezionare almeno una delle variabili presenti nel bucket corrente.

Perciò, la risoluzione di ogni sottoproblema può portare a trovare una soluzione migliore e/o trovare nuove variabili da inserire nel kernel.

4.5 Algoritmo risolutivo per il primo problema

Benché il problema fosse noto in letteratura, si è pensato di sviluppare un metodo euristico diverso da quelli esistenti. L'algoritmo proposto consiste direttamente in un'implementazione di VNDS, infatti nel codice è possibile trovare nomi di variabili e di funzioni che richiamano i concetti introdotti sopra.

4.5.1 Corpo dell'algoritmo

Le fasi principali di VNDS sono state implementate in funzioni separate che verranno descritte in maggior dettaglio nei capitoli successivi. Di seguito viene spiegato il ciclo principale dell'algoritmo, che va a richiamare le varie parti per analizzare il vicinato corrente.

L'algoritmo inizia costruendo la soluzione iniziale e memorizzando in quali zaini si trovano le famiglie della soluzione corrente. Questa informazione verrà utilizzata nelle fasi di shaking e di improvement per cercare di non aumentare il numero di zaini usati dalla famiglia, cercando di assegnare gli oggetti a degli zaini già in uso dalla famiglia stessa. Dopodiché inizia il ciclo principale dell'algoritmo, che si interromperà nel momento in cui verrà soddisfatto uno dei criteri di terminazione.

Seguendo la struttura di VNDS proposta in Hansen et al. (2017 *[25]*) e spiegata in *Sezione 4.3.1*, il ciclo interno va a scorrere le varie dimensioni del vicinato, cominciando da quella più piccola, finché non raggiunge la dimensione massima.

All'interno di questo ciclo, avvengono quattro fasi, descritte nei paragrafi successivi. La prima consiste nella fase di shaking (*Sezione 4.5.3*) oppure nel reset della soluzione (*Sezione 4.5.5*). La seconda è la fase di decomposizione, che dovrà ridurre la dimensione del problema per la fase di improvement, definendo quali oggetti potranno essere mossi e fissando i rimanenti. Perciò, la terza fase è quella di improvement (*Sezione 4.5.7*), che si occupa di muovere gli oggetti selezionati e di cercare la miglior soluzione possibile vicino a quella corrente. Infine, la quarta fase è il cambio del vicinato (*Sezione 4.5.9*), che dovrà decidere se muoversi nella nuova soluzione trovata o restare in quella corrente.

4.5.2 Costruzione della soluzione iniziale

La soluzione da cui parte l'algoritmo viene costruita selezionando le famiglie più promettenti, in base ai valori ottenuti nella soluzione del rilassamento continuo, e inserendole in ordine negli zaini, fino al loro riempimento. Questo approccio è stato scelto perché è stata notata una correlazione tra i valori delle variabili x ottenuti nel rilassamento continuo e la soluzione ottima del problema intero trovata da Gurobi.

L'algoritmo costruisce il modello del problema per Gurobi, lo trasforma nel corrispondente modello rilassato e lo risolve. Dalla soluzione ottenuta, vengono estrat-

te le variabili associate alla selezione delle famiglie (cioè le variabili x_j, $j \in F$ del modello matematico), poi, per ciascuna, viene memorizzata la coppia "indice della famiglia" – "valore della variabile" se è positiva, altrimenti viene memorizzata la coppia "indice della famiglia" - "valore assoluto del costo ridotto associato" se è nulla (non può essere negativa per definizione del dominio delle variabili x_j). Le due associazioni ottenute hanno la seguente struttura:

$$\text{xvars\_dict} = \{j : x_j, ...\}$$
$$\text{xvars\_rc\_module\_dict} = \{j : |x_j.\text{RC}|, ...\}$$

Con $j \in F$ e $x_j.\text{RC}$ il coefficiente di costo ridotto associato alla variabile nella soluzione ottima (Reduced Cost).

Queste due associazioni vengono poi ordinate e concatenate, la prima in ordine non crescente, la seconda in ordine non decrescente. Questo approccio prende spunto dall'algoritmo Kernel Search (*Sezione 4.4.1*), utilizzando, però, soltanto le variabili associate alle famiglie e non tutte le variabili del modello.

```
def build_initial_solution():
  # Risolve il rilassamento lineare del problema con Gurobi e ottiene i
  # valori delle variabili x non nulle e il modulo dei costi ridotti di
  # quelle nulle.
  gurobi_solver = GurobiSolver()
  relaxed_xvars, reduced_costs_module, relaxed_obj_value =
               gurobi_solver.solve_relaxed(instance, cpu_time_limit)

  # Ordina gli indici delle famiglie in base a:
  # - valori decrescenti delle variabili x non nulle
  # - valori crescenti del modulo dei costi ridotti delle variabili x
  # nulle
  sorted_families_indexes = sorted(relaxed_xvars) +
                            sorted(reduced_costs_module)

  # Costruisce la soluzione iniziale
  solution = build_solution(
    sorted_families_indexes, first_items, items
  )
  return solution
```

Una volta ottenuta la lista ordinata delle famiglie, l'algoritmo costruisce la soluzione iniziale (nel codice "build_solution") prendendone una per volta e cercando di assegnare ogni suo oggetto a uno zaino, nel seguente modo.

Gli zaini vengono ordinati per valore non crescente della somma delle loro capacità.

```
selected_families: list[int] = []
# Ordina gli zaini per capacità decrescente
temp_knapsacks1 = dict(
  sorted(
    enumerate(instance.knapsacks),
    key=lambda x: sum(x[1]), reverse=True
  )
)
```

Poi viene effettuato un tentativo di inserire ciascun oggetto in uno zaino, procedendo con lo zaino successivo in caso di fallimento. Per effettuare i tentativi di inserimento vengono definiti degli zaini temporanei. Viene controllato che il vincolo di capacità sia rispettato per ogni occupazione di risorsa dell'oggetto.

```
item_index = family_start_index
temp_knapsacks2 = temp_knapsacks1.copy()
family_can_be_selected = True
family_solution = []
while family_can_be_selected and item_index < family_end_index:
  item = items[item_index]
  knapsack_found = False
  for index, knapsack in temp_knapsacks2.items():
    constraint_violated = False
    i = 0
    while not constraint_violated and i < len(item):
      if knapsack[i] < item[i]:
        constraint_violated = True
      i += 1
    if not constraint_violated:
      knapsack_found = True
      family_solution.append(index)
      temp_knapsacks2[index] = [x - y for x, y in zip(knapsack, item)]
    if knapsack_found:
      break
  if not knapsack_found:
    family_can_be_selected = False
  item_index += 1
```

Se l'inserimento viene confermato allora viene aggiornata la capacità rimanente e la soluzione. Se un oggetto non può essere inserito in nessuno zaino, la famiglia viene rimossa dalla soluzione, inserendo tanti valori `-1` per quanti oggetti appartengono alla famiglia, e il processo continua con quella successiva.

```
if family_can_be_selected:
  selected_families.append(family_index)
  temp_knapsacks1 = temp_knapsacks2
  for i in range(family_start_index, family_end_index):
    solution[i] = family_solution[i - family_start_index]
else:
  for i in range(family_start_index, family_end_index):
    solution[i] = -1
family_index += 1
```

Per poter avere una soluzione coerente con il problema, essa deve avere lunghezza pari al numero di oggetti, perché deve indicare uno zaino per ogni oggetto. Per quelli che non sono stati inseriti, deve essere inserito il valore `-1`.

```
not_selected_families = []
for family in range(instance.n_families):
  if family not in families_indexes:
      not_selected_families.append(family)
for family in not_selected_families:
  family_start_index = first_items[family]
  family_end_index = first_items[family + 1] if family + 1 <
len(first_items) else len(items)
  for i in range(family_start_index, family_end_index):
    solution[i] = -1
```

La soluzione così costruita viene utilizzata come soluzione iniziale dell'algoritmo.

4.5.3 Fase di shaking

In questa fase l'obiettivo è perturbare la soluzione corrente in modo da generarne una casuale che si allontani da eventuali minimi locali. Questo viene fatto rimuovendo alcune famiglie dalla soluzione corrente per poi aggiungerne altre. Nel codice sorgente questa fase viene implementata mediante la classe *FamilySwitchNeighborhood*.

4.5.4 Vicinato di tipo FamilySwitchNeighborhood

L'algoritmo utilizza questa classe per ottenere un vicino casuale della soluzione corrente nella fase di shaking. Riceve come input:

- La soluzione corrente
- Il valore di l, che indica quante famiglie scambiare (più precisamente, quante famiglie rimuovere).

La procedura è divisa in due parti: prima rimuove l famiglie casuali dalla soluzione corrente, poi cerca di aggiungere più famiglie possibili tra quelle non selezionate. Il valore di l, perciò, non indica realmente quante famiglie scambiare, ma piuttosto quante famiglie rimuovere. Questo accade perché non è possibile sapere in anticipo quante famiglie potranno essere aggiunte: dipende dal numero di elementi appartenenti alle famiglie che saranno selezionate e dal loro peso.

```
def get_random_neighbor(input_solution, l):
  # Ottiene le famiglie selezionate e non selezionate nella soluzione
  # corrente
  selected_families, not_selected_families =
      get_selected_families(input_solution)

  # Rimuove j famiglie casuali dalla soluzione corrente
  solution_with_removed_families = remove_families(
      input_solution=input_solution,
      selected_families=selected_families,
      l=l
  )

  # Aggiunge famiglie casuali alla soluzione se ci sono famiglie non
  # selezionate da aggiungere
  if len(not_selected_families) != 0:
    solution_with_added_families = add_families(
        input_solution=solution_with_removed_families,
        not_selected_families=not_selected_families
    )
  else:
    # Se non ci sono famiglie da aggiungere
    if not solution_with_removed_families.empty():
      # Restituisce la soluzione ottenuta dalla rimozione delle
      # famiglie se non è vuota
      solution_to_return = solution_with_removed_families
    else:
      # Altrimenti restituisce la soluzione ricevuta in ingresso
      solution_to_return = input_solution
return solution_to_return
```

La rimozione delle famiglie avviene nel seguente modo. Innanzitutto viene ottenuta una lista delle famiglie selezionate nella soluzione corrente, tra cui verranno selezionate quelle da rimuovere. Poi viene preso come numero di famiglie da rimuovere il valore minimo tra l e il numero di famiglie selezionate. Dopodiché, per scegliere casualmente quali saranno rimosse, vengono calcolati i pesi da assegnare a ciascuna di quelle presenti, assegnando un peso maggiore alle famiglie con profitto minore.

È stato scelto di utilizzare il profitto come peso per la scelta delle famiglie da rimuovere e da aggiungere perché, dopo alcune osservazioni, è stata notata una correlazione tra il profitto delle famiglie e la loro presenza nella soluzione ottima trovata da Gurobi.

Infine, una volta selezionate le famiglie da rimuovere, vengono tolti gli oggetti corrispondenti dalla soluzione corrente. Va evidenziato che la parte di rimozione delle famiglie genera sempre una soluzione valida, anche se peggiore.

```
def remove_families(input_solution, selected_families, l):
  solution_with_removed_families = input_solution
  # Definisce il numero di famiglie da rimuovere
  removal_number = min(l, len(selected_families))
  max_profit = max(instance.profits)
  for family in selected_families:
    # Il peso è dato dalla differenza tra il profitto massimo e il
    # profitto della famiglia corrente, in questo modo cerca di
    # rimuovere famiglie con profitto minore
    weights.append(1 + max_profit - instance.profits[family])
  families_to_remove = random.choices(
      selected_families, weights, removal_number)

  # Rimuove gli oggetti delle famiglie selezionate
  for family_index in families_to_remove:
    for item_index in range(family_start_index, family_end_index):
      solution_with_removed_families[item_index] = -1
  return solution_with_removed_families
```

Se non ci sono famiglie da poter aggiungere (perché nella soluzione corrente erano selezionate tutte le famiglie possibili) allora questa fase termina restituendo la soluzione ottenuta dalla rimozione delle famiglie, se non è la soluzione vuota, altrimenti restituisce la soluzione ricevuta in ingresso.

Nella seconda parte, quella di aggiunta di nuove famiglie, l'algoritmo cerca di inserire più famiglie possibili tra quelle non selezionate, finché non trova una soluzione non valida, poi effettua ancora un certo numero di tentativi di aggiunta e infine restituisce la nuova soluzione generata. Il numero massimo di fallimenti per questa procedura corrisponde al valore del parametro `-fa`.

```
def add_families(input_solution, not_selected_families):
  solution_with_added_families = input_solution
  possible_families = not_selected_families

  is_feasible = True
  # Aggiunge famiglie finché non trova una soluzione non valida
  while is_feasible:
    # Sceglie una famiglia casuale
    family_index = select_random_family_for_addition(possible_families)
    # Prova ad aggiungerla alla soluzione
    solution_to_try, is_solution_to_try_feasible = try_to_add_family(
      solution_with_added_families,
      family_index
    )
    if is_solution_to_try_feasible:
      solution_with_added_families = solution_to_try
      possible_families.remove(family_index)
      is_feasible = is_solution_to_try_feasible

  # Prova ad aggiungere un'altra famiglia per un certo numero di volte
  while not is_feasible and
      not check_unfeasible_addition_counter_limit():
    family_index = select_random_family_for_addition(possible_families)
    solution_to_try, is_solution_to_try_feasible = try_to_add_family(
      solution_with_added_families,
      family_index
    )
    if is_solution_to_try_feasible:
      solution_with_added_families = solution_to_try
      is_feasible = is_solution_to_try_feasible
  return solution_with_added_families, is_feasible
```

Per effettuare un tentativo di aggiunta, l'algoritmo esegue i seguenti passaggi. Inizia scegliendo casualmente una famiglia tra quelle non selezionate nella soluzione corrente, dando a ciascuna un peso pari al proprio profitto.

```
for family in not_selected_families:
  weights.append(1 + instance.profits[family])
selected_family = random.choices(not_selected_families, weights)
```

Poi prova diverse selezioni casuali di zaini per la famiglia, finché non trova una soluzione valida, oppure finché non raggiunge il limite di tentativi. La scelta dello zaino è pesata dal numero di volte in cui è stato selezionato per contenere l'oggetto corrente, mentre il numero di tentativi di selezione di uno zaino durante l'aggiunta di una famiglia corrisponde al parametro `-ks`.

Siccome l'obiettivo è minimizzare il numero di split della famiglia, viene definito un insieme di zaini in cui la famiglia potrà essere inserita (family_knapsacks), che viene man mano ampliato se l'inserimento non riesce. Il seguente estratto di codice illustra in che modo vengono assegnati gli zaini alla famiglia da aggiungere.

```
# Lista di zaini che verrà ampliata man mano che se l'inserimento non
# riesce
family_knapsacks = []
while not is_solution_to_try_feasible and
    not check_unfeasible_addition_knapsack_counter_limit():
  add_new_knapsack = True
  for i in family_items:
    if add_new_knapsack:
      weights = []
      for kn in range(n_knapsacks):
        # Calcola il peso da assegnare allo zaino come la differenza
        # tra il numero totale di selezioni e il numero di volte in cui
        # è stato selezionato, in modo da cercare di assegnare lo zaino
        # meno selezionato
        w=select_knapsack_counter-knapsack_selection_counter[i][kn]
        weights.append(1 + w)
      selected_knapsack = random.choices(
          range(n_knapsacks),
          weights
      )
      family_knapsacks.append(selected_knapsack)
      add_new_knapsack = False
    else:
      # Uno zaino è già stato aggiunto, perché add_new_knapsack è
      # False, perciò ne sceglie uno tra quelli già usati dalla
      # famiglia
      selected_knapsack = random.choice(family_knapsacks)
    solution_to_try[i] = selected_knapsack
  is_solution_to_try_feasible = instance.is_feasible(solution_to_try)
```

La variabile add_new_knapsack serve per aggiungere uno zaino per tentativo, a quelli già usati dalla famiglia. Una volta aggiunto il primo oggetto a uno zaino casuale tra tutti quelli possibili, il tentativo prosegue inserendo gli altri oggetti in uno zaino casuale tra quelli della famiglia (compreso quello in cui è stato appena inserito il primo oggetto).

Un esempio di funzionamento di questa classe viene illustrato nella figura sottostante: in questo esempio viene mostrato uno scambio di due famiglie che genera una nuova soluzione dell'istanza `example.json`. Ogni oggetto è rappresentato da un quadrato con angoli arrotondati e due numeri all'interno. Ogni numero rappresenta la quantità di risorsa consumata dall'oggetto, ordinata in base all'indice della risorsa. Oggetti con lo stesso colore di sfondo appartengono alla stessa famiglia.

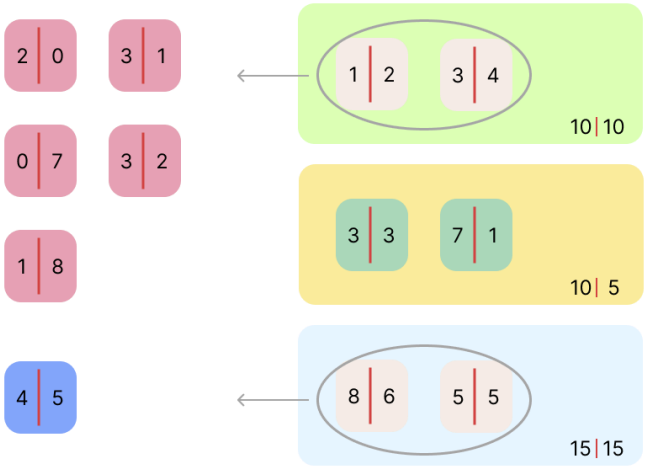


Figura 11: Una soluzione dell'istanza. Cerchiati gli oggetti della famiglia da rimuovere.

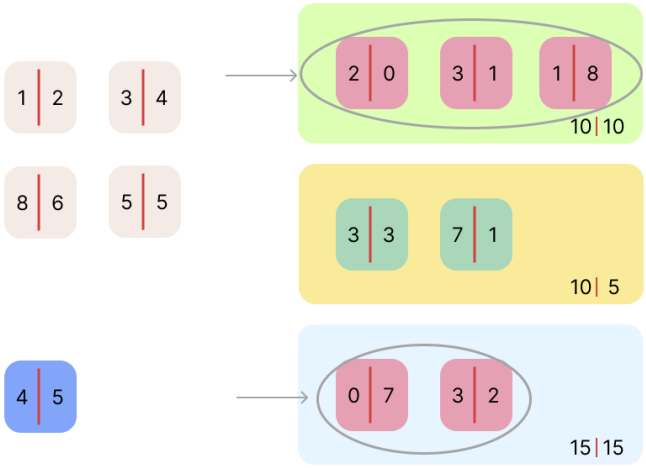


Figura 12: Una soluzione vicina a quella precedente. Cerchiati gli oggetti della famiglia aggiunta.

4.5.5 Reset della soluzione

Per inserire una strategia di intensificazione della ricerca, dopo un certo numero di iterazioni senza miglioramenti, l'algoritmo salta la fase di shaking e utilizza come soluzione da migliorare la miglior soluzione trovata fino a quel momento. Il parametro che definisce il numero di iterazioni senza miglioramenti prima di ripartire dalla soluzione migliore è `-r`.

```
if 0 < iterations_without_improvement_before_reset <=
          iterations_without_improvement_from_last_reset_counter:
        # Se sono passate "iterations_without_improvement_before
        # _reset" iterazioni senza miglioramenti allora resetta
        # il contatore e riparte dalla soluzione migliore trovata
        # finora
        iterations_without_improvement_from_last_reset_counter = 0
        x_shake = best_solution
      else:
        x_shake = shake(best_solution, l)
```

4.5.6 Fase di decomposizione

In questa fase l'obiettivo è selezionare quali oggetti muovere nella fase di improve. L'idea è di generare una soluzione non valida assegnando a una famiglia intera un solo zaino che la possa contenere, per poi passare alla fase di improvement i possibili oggetti da muovere per ottenere una soluzione valida. Il motivo di questa forzatura è dovuto al fatto che cercare semplicemente di muovere oggetti porterebbe difficilmente a un miglioramento del valore della funzione obiettivo, perché la penalità per gli split di una famiglia non cambia finché sono uno o più.

Innanzitutto vengono presi in considerazione soltanto gli oggetti che sono stati selezionati nella soluzione corrente, cioè quelli che hanno uno zaino assegnato.

```
valid_indexes = [i for i, x in enumerate(x_shake) if x != -1]
```

Poi cerca di fissare una famiglia che abbia almeno uno split in uno zaino che possa contenerla interamente (considerando lo spazio già occupato dalle famiglie che non hanno degli split). Prima ottiene la lista delle famiglie con almeno uno split e le capacità residue degli zaini.

```
split_families = {}
temp_knapsacks = instance.knapsacks.copy()
for family_index, first_item in enumerate(instance.first_items):
  if x_shake[first_item] != -1:
    if family_has_at_least_one_split:
      # Se la famiglia ha almeno uno split, allora la memorizzo
      split_families[family_index] = [first_item, last_item]
    else:
      # Se la famiglia non ha degli split, allora la considero
      # fissa nella soluzione, perciò decremento le capacità
      # disponibili degli zaini temporanei "temp_knapsacks"
      for index, values in items[first:last]:
        knapsack = temp_knapsacks[x_shake[first+index]]
        temp_knapsacks[x_shake[first+index]] =
            [x - y for x, y in zip(knapsack, item_values)]
```

In seguito, per ogni famiglia con almeno uno split, memorizza in quali zaini potrebbe essere contenuta:

```
possible_knapsacks_per_family = {}
for family_index in split_families:
  if x_shake[first_item] != -1:
    if family_has_at_least_one_split:
      family_capacity = [sum(valori) for valori in family_items]
      possible_knapsacks = {
        # Mapping di zaini con relative capacità disponibili che
        # possono contenere la famiglia corrente
        knapsack_index: knapsack for knapsack_index, knapsack in
            enumerate(temp_knapsacks)
            if all(knapsack[i] > family_capacity[i] for i in
                  range(len(family_capacity)))
      }
      if len(possible_knapsacks) > 0:
        # Se la famiglia ha almeno uno zaino in cui può essere
        # contenuta, allora memorizzo gli zaini possibili
        possible_knapsacks_per_family_dict[family_index] =
            possible_knapsacks
```

Infine seleziona l oggetti casuali, che non appartengano alle famiglie che non hanno split, per essere mossi nella fase di improvement. Questo perché muovere oggetti di famiglie che non hanno split potrebbe introdurne nuovi, che andrebbero sicuramente a peggiorare la soluzione.

Siccome la soluzione considerata in questo punto, molto probabilmente, risulta non valida, viene memorizzata quella ottenuta dalla fase di shaking, che verrà restituita come risultato di questa iterazione dell'algoritmo se la fase di improvement non dovesse riuscire a ottenere una soluzione valida.

```
x_improvement = x_shake
x_improvement_value = x_shake_value
# Se non ci sono zaini in cui è possibile inserire la famiglia, passa
al cambio del vicinato
if len(possible_knapsacks_per_family_dict) != 0:
  selected_family =
      random.choice(possible_knapsacks_per_family_dict.keys())
  knapsack_indexes =
      possible_knapsacks_per_family_dict[selected_family]
  selected_knapsack = random.choice(knapsack_indexes)

  # Fissa tutti gli oggetti della famiglia nello zaino selezionato
  for i in range(first_item, last_item):
```

```
    x_improvement[i] = selected_knapsack
    valid_indexes.remove(i)

  # Se non ci sono oggetti da poter muovere, passa al cambio del
  # vicinato
  if len(valid_indexes) > 0:
    variable_indexes =
      random.sample(valid_indexes, min(l), len(valid_indexes)))
    # Passa alla fase di improvement la soluzione e gli oggetti da
    # muovere
    x_improvement, x_improvement_value = improve(
        x_improvement, x_improvement_value, variable_indexes)
```

Perciò, tra gli oggetti che possono essere spostati, vengono selezionati l oggetti casuali che verranno utilizzati nella fase di improvement.

4.5.7 Fase di improvement

In questa fase l'obiettivo è cercare la soluzione migliore nel vicinato di quella ricevuta in ingresso (che, molto probabilmente, risulta essere non valida a causa della fase di decomposizione). Questo viene fatto cercando di spostare gli oggetti ricevuti in ingresso in zaini diversi.

L'algoritmo implementato è un Variable Neighborhood Descent (spiegato nell'introduzione del capitolo) il cui vicinato viene realizzato nel codice sorgente mediante la classe *KRandomRotateNeighborhood*.

Inizia fissando la soluzione corrente e il suo valore come i migliori trovati finora. Poi, per ogni dimensione del vicinato, partendo da quella più piccola, cerca il miglior vicino della soluzione corrente ed esegue la fase di cambio del vicinato (analoga a quella dell'algoritmo VNDS ma che va a modificare la dimensione del vicinato di VND e non quella dell'algoritmo principale).

```
def improve(solution_to_improve, variable_indexes):
  fixed_solution_before_improving = []
  current_solution = solution_to_improve
  stop = False
  while not stop:
    l_vnd = 1
    fixed_solution_before_improving = current_solution
    while l_vnd <= len(variable_indexes):
      best_neighbor, best_neighbor_value = get_best_neighbor(
        current_solution, current_solution_value, k, variable_indexes
      )
      current_solution, l_vnd = neighborhood_change_step(
        current_solution, best_neighbor, current_solution_value,
        best_neighbor_value, l_vnd
      )
```

```
    if fixed_solution_before_improving_value >= current_solution_value:
      stop = True
  return fixed_solution_before_improving
```

La ricerca locale termina quando non vengono trovate soluzioni migliori in nessuna delle dimensioni del vicinato e restituisce la soluzione migliore trovata (nel caso peggiore, quella ricevuta in ingresso).

4.5.8 Vicinato di tipo KRandomRotate

L'algoritmo utilizza questa classe per ottenere il miglior vicino della soluzione corrente nella fase di improvement. Riceve come input:

- La soluzione corrente
- Il valore di l, che indica il numero di elementi da spostare
- Gli indici degli elementi che possono essere spostati

Naturalmente, il numero di indici deve essere maggiore o uguale a l.

Sappiamo già che la soluzione corrente potrebbe non essere valida, per quanto già visto in *Sezione 4.5.6* sulla decomposizione, però questa classe si occupa soltanto di trovare il miglior vicino possibile, senza preoccuparsi della validità della soluzione iniziale. Infatti sarà il ciclo principale a occuparsi di gestire un'eventuale fallimento nel trovare un vicino valido.

Per cercare il miglior vicino, questo metodo seleziona l oggetti casuali tra quelli ricevuti in ingresso e prova ad assegnarli a l zaini casuali. Se trova una soluzione valida allora la restituisce, altrimenti ripete la scelta degli l zaini. Se dopo un certo numero di tentativi, che corrisponde a `-mi` non ha ancora trovato una combinazione di zaini valida, allora ripete anche la scelta degli l oggetti da muovere. Anche per questa scelta viene definito un limite massimo di tentativi, che corrisponde al parametro `-is`, perciò, complessivamente, verranno al massimo effettuati `-mi` $*$ `-is` tentativi.

```
def get_best_neighbor(input_solution, input_solution_value, l,
    variable_indexes):
  best_solution = input_solution
  best_value = input_solution_value
  stop = False
  # C'è un limite di tentativi per la scelta degli oggetti da muovere
  while not stop and not check_unfeasible_items_selection_limit():
    for item_index in variable_indexes:
      w = item_selection_counter - items_to_move_selected[item_index]
      weights.append(1 + w)
    indexes_to_change=random.choices(
      variable_indexes, weights, l
    )
    new_solution = input_solution
    is_feasible = False
```

```
    # C'è un limite di tentativi per la scelta degli zaini
    while not is_feasible and not check_unfeasible_move_items_limit():
      for i in indexes_to_change:
        for knapsack in range(n_knapsacks):
          w = select_knapsack_counter - knapsack_selection[i][knapsack]
          weights.append(1 + w)
        selected_knapsack = random.choices(
          range(n_knapsacks), weights
        )
        new_solution[i] = selected_knapsack
      if instance.is_feasible(new_solution):
        new_value = instance.evaluate_solution(new_solution)
        if new_value > best_value:
          best_value = new_value
          best_solution = new_solution
          stop = True
  return best_solution, best_value
```

Anche in questo caso le scelte degli oggetti e degli zaini sono pesate, in modo da dare una probabilità maggiore a quelli selezionati meno volte.

Un esempio di funzionamento di questa classe viene illustrato nelle figure sottostanti: in questo esempio viene mostrato uno spostamento di due elementi che genera una nuova soluzione dell'istanza `example.json`. Ogni oggetto è rappresentato da un quadrato con angoli arrotondati e due numeri all'interno. Ogni numero rappresenta la quantità di risorsa consumata dall'oggetto, ordinata in base all'indice della risorsa. Oggetti con lo stesso colore di sfondo appartengono alla stessa famiglia.

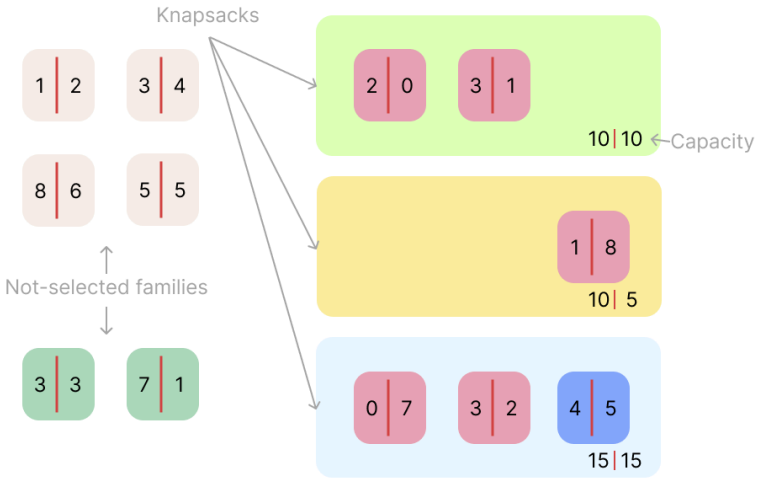


Figura 13: Una soluzione dell'istanza. In grigio le definizioni.

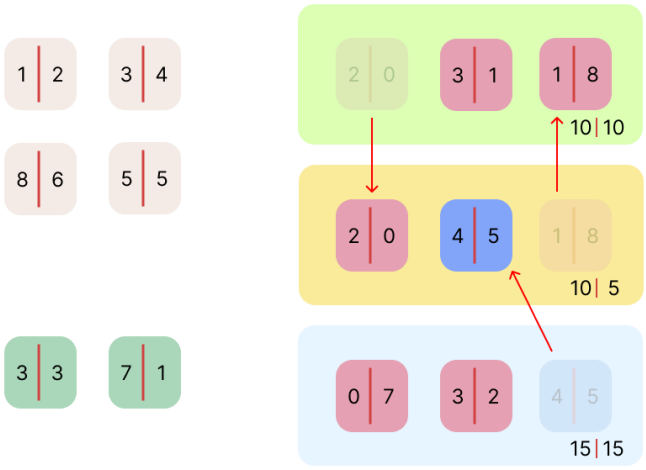


Figura 14: Una soluzione vicina a quella precedente. Le frecce rosse indicano gli spostamenti.

4.5.9 Cambio del vicinato

La scelta del vicinato avviene in base al valore della soluzione ottenuta nella fase di improvement. Se la soluzione ottenuta è migliore di quella corrente, allora l'algoritmo si sposta nella nuova soluzione e il vicinato viene resettato alla dimensione minima ($l = 1$). Altrimenti, la dimensione del vicinato viene incrementata di una unità e l'algoritmo resta nella soluzione corrente.

Sia il ciclo più esterno (VNS) che la fase di improvement (VND) utilizzano questo metodo per cambiare il vicinato.

```
def neighborhood_change_step(x_current, x_improvement, current_value,
  improvement_value, l):
  solution_to_return = x_current
  value_to_return = current_value
  has_improved = improvement_value > current_value

  if has_improved:
    # Aggiorna la soluzione corrente
    solution_to_return = x_improvement
    value_to_return = improvement_value
    # Resetta la dimensione del vicinato a quella minima
    l = 1
  else:
    # Altrimenti, incrementa la dimensione del vicinato
    l += 1
  return solution_to_return, value_to_return, l, has_improved
```

4.5.10 Criterio di terminazione

Ogni volta che l'algoritmo termina di esplorare tutte le dimensioni del vicinato utilizzato nella fase di shaking, esso raggiunge la verifica dei criteri di terminazione del ciclo principale. Se uno o più di questi criteri sono soddisfatti, l'algoritmo si ferma e restituisce la soluzione migliore trovata.

L'esecuzione si interrompe se:

- è stato raggiunto il valore ottimo indicato nell'istanza (solo se è stato fornito con l'istanza);
- è stato raggiunto il limite di tempo;
- è stato raggiunto il limite di iterazioni senza miglioramenti;
- è stato raggiunto il limite di iterazioni totali.

Il controllo del raggiungimento dell'ottimo è stato implementato per velocizzare la risoluzione delle istanze, perché, per la maggior parte di esse, è stato trovato da Gurobi.

4.5.11 Parametri dell'algoritmo

Parametri principali:

> **-t, –time [intero].** Limite di tempo di esecuzione in secondi, `0` per nessun limite. Viene verificato ad ogni controllo dei criteri di terminazione.

> **-i, –iterations, [intero].** Limite di iterazioni, inteso come numero di fasi di shaking effettuate. `0` per nessun limite.

> **-ni, –no-improvement, [intero].** Limite del numero di iterazioni senza miglioramento, `0` per nessun limite. Deve essere inferiore al limite di iterazioni.

> **-k, –k-max, [intero].** Valore massimo per il parametro l utilizzato nella ricerca di soluzioni vicine a quella corrente. Questo valore è proporzionale alla dimensione massima dei vicinati da esplorare.

> **-r, –restart, [intero].** Il numero di iterazioni senza miglioramento prima di ripartire dalla miglior soluzione trovata finora. Il valore `0` disattiva questa funzione.

> **-d, –debug.** Utilizzando questo parametro vengono mostrati ulteriori messaggi nei log, per dare più informazioni sull'esecuzione corrente.

Limiti per i tentativi che generano soluzioni non valide:

> **-is, –item-selection, [intero].** Il numero massimo di tentativi di selezionare gli oggetti da muovere nella fase di improvement. Il valore `0` significa nessun limite.

> **-mi, –move-item, [intero].** Il numero massimo di tentativi di muovere gli oggetti selezionati in uno zaino casuale, prima di provare un'altra selezione di oggetti (se il limite `-is` non è ancora stato raggiunto). Il valore `0` significa nessun limite.

-fa, –family-addition, [intero]. Il numero massimo di tentativi di aggiunta di una famiglia. Il valore `0` significa nessun limite.

-ks, –knapsack-selection, [intero]. Il numero massimo di tentativi di muovere gli oggetti della famiglia da aggiungere in uno zaino casuale. Il valore `0` significa nessun limite.

Se non vengono impostati limiti, l'algoritmo continuerà a effettuare tentativi finché non viene trovata una soluzione valida. Questo può richiedere molto tempo, perciò è meglio impostare un valore.

Altri parametri:
Alcuni parametri sono in realtà valori costanti all'interno del codice, perché non sono direttamente correlati alla risoluzione del problema:

INSTANCES_DIR. Percorso della cartella in cui cercare le istanze da risolvere (non ricorsivo).

OUTPUT_DIR. Percorso della cartella in cui salvare i risultati delle istanze risolte.

SKIP_GUROBI. Usato per scegliere se risolvere le istanze anche con Gurobi o meno.

GUROBI_TIME_LIMIT. Il tempo limite assegnato a Gurobi.

VNDS_MIP_GAP. Se impostato, l'algoritmo risolve il rilassamento continuo con Gurobi e, se viene trovata una soluzione, ottiene un limite superiore per calcolare il MIP gap della soluzione corrente. Grazie a questo valore esso può fermarsi quando viene raggiunta la tolleranza indicata.

4.5.12 Indicatori

Durante l'esecuzione dell'algoritmo vengono raccolte alcuni indicatori per valutarne le prestazioni e il comportamento.

Indicatori principali:

improve_top_solution_counter. Conta quante volte l'algoritmo riavvia la ricerca partendo dalla migliore soluzione trovata finora. Dipende dal parametro `-r`.

building_solution_time. Misura il tempo impiegato per costruire la soluzione iniziale, in secondi.

iterations_counter. Misura il numero di iterazioni dell'algoritmo. Si intende il numero di volte in cui vengono eseguite le tre fasi (shake, improve, cambio del vicinato) dell'algoritmo.

improvements_counter. Conta il numero di volte in cui l'algoritmo migliora la soluzione corrente.

not_improvements_counter. Conta il numero di volte in cui l'algoritmo non migliora la soluzione corrente.

no_families_to_remove_counter. Conta quante volte accade che non ci siano famiglie da rimuovere dalla soluzione, durante l'esecuzione di FamilySwitchNeighborhood.

unfeasible_add_random_families_counter. Conta il numero di soluzioni non valide incontrate quando si cerca di aggiungere famiglie casuali, eseguendo FamilySwitchNeighborhood.

removal_number_values. Conta quante volte ogni valore del numero di famiglie da rimuovere viene utilizzato durante l'esecuzione di FamilySwitchNeighborhood.

mean_shaking_time. Misura il tempo medio richiesto dalla fase di shaking, in secondi.

-mi limit reached. Nel codice denominato `unfeasible_move_items_limit_reached`. Conta quante volte l'algoritmo ha raggiunto il limite indicato dal parametro `-mi`.

-is limit reached. Nel codice denominato `unfeasible_items_selection_limit_reached`. Conta quante volte l'algoritmo ha raggiunto il limite indicato dal parametro `-is`.

-fa limit reached. Nel codice denominato `unfeasible_addition_counter_limit_reached`. Conta quante volte l'algoritmo ha raggiunto il limite indicato dal parametro `-fa`.

-ks limit reached. Nel codice denominato `unfeasible_addition_knapsack_counter_limit_reached`. Conta quante volte l'algoritmo ha raggiunto il limite indicato dal parametro `-ks`.

reached_k_max_counter. Conta quante volte l'algoritmo ha raggiunto il valore $l_{\max}$, indicato dal parametro `-k`.

discarded_get_best_neighbor. Conta il numero di soluzioni visitate durante la ricerca del miglior vicino, quindi durante la fase di improvement, che avevano un valore della funzione obiettivo peggiore della soluzione attuale.

improvement_failures_counter. Conta quante volte l'algoritmo non trova una soluzione migliore durante la fase di improvement, per ogni valore di l (cioè per ogni dimensione del vicinato esplorata).

improvement_successes_counter. Conta quante volte l'algoritmo VND della fase di improvement trova una soluzione migliore rispetto a quella ricevuta in ingresso (cioè la risultante dalla fase di shaking).

vnd_iterations_counter. Conta il numero di iterazioni eseguite durante la fase di improvement dall'algoritmo di ricerca locale (VND).

mean_get_best_neighbor_time. Misura il tempo medio impiegato per trovare il miglior vicino, in secondi.

final_iterations_without_improvement. Nel codice denominato `iterations_without_improvement_last`. Conta il numero di iterazioni senza miglioramenti dall'ultimo miglioramento fino al termine dell'algoritmo.

vnds_k_values_counter. Conta quante volte ogni valore di l viene utilizzato durante l'esecuzione dell'algoritmo VNDS.

top_solutions. Tiene traccia delle soluzioni migliori trovate durante l'esecuzione dell'algoritmo VNDS.

stopping_cause. Memorizza la causa di terminazione dell'algoritmo per VNDS e Gurobi. I valori possibili sono: stop_cpu_time_limit, stop_iterations_counter_limit, stop_iterations_without_improvement_limit, stop_mip_gap_limit, stop_instance_optimum, stop_gurobi_optimal, stop_gurobi_infeasible, stop_gurobi_node_limit e unknown (valore predefinito).

initial_solution_obj_value. Il valore obiettivo della soluzione iniziale.

execution_time. Il tempo di esecuzione totale, in secondi.

obj_value. Il valore obiettivo della migliore soluzione trovata.

best_solution. Memorizza la migliore soluzione trovata durante l'esecuzione dell'algoritmo.

Altri indicatori:

mip_gap. Memorizza il MIP gap calcolato, quando l'algoritmo calcola il rilassamento, quindi quando il parametro VNDS_MIP_GAP è impostato.

relaxed_compute_time. Tempo impiegato per calcolare il rilassamento dal solutore di Gurobi, se calcolato.

relaxed_optimum_value. Il valore ottimo del rilassamento, se calcolato.

start_time. Riferimento temporale all'inizio dell'algoritmo. Utilizzato per calcolare il tempo di esecuzione.

cumulative_shaking_time. La somma del tempo impiegato da tutte le esecuzioni della fase di shaking, durante questa esecuzione dell'algoritmo. Utilizzato con `shakes_counter` per calcolare il tempo medio impiegato dalla fase di shaking.

shakes_counter. Il numero di volte in cui la fase di shaking è stata eseguita durante l'esecuzione dell'algoritmo. Utilizzato con `cumulative_shaking_time` per calcolare il tempo medio impiegato dalla fase di shaking.

families_addition_counter. Conta quante volte ogni famiglia è stata aggiunta alla soluzione corrente. Utilizzato per calcolare i pesi delle famiglie.

removed_families_counter. Conta quante volte ogni famiglia è stata rimossa dalla soluzione corrente. Utilizzato per calcolare i pesi delle famiglie.

not_selected_families_counter. Conta quante volte ogni famiglia non è stata presente nella soluzione corrente. Utilizzato per calcolare i pesi delle famiglie.

selected_families_counter. Conta quante volte ogni famiglia è stata presente nella soluzione corrente. Utilizzato per calcolare i pesi delle famiglie.

add_family_counter. Conta quante volte l'algoritmo ha aggiunto una famiglia alla soluzione con successo. Utilizzato con `families_addition_counter` per calcolare quante volte una famiglia non è stata aggiunta alla soluzione quando si calcolano i pesi delle famiglie.

remove_counter. Conta quante volte l'algoritmo ha rimosso una famiglia dalla soluzione con successo. Utilizzato con `removed_families_counter` per calcolare quante volte una famiglia non è stata rimossa dalla soluzione quando si calcolano i pesi delle famiglie.

cumulative_improvement_found_time. La somma del tempo impiegato da tutte le esecuzioni della fase di improvement che ha portato a un miglioramento, durante questa esecuzione dell'algoritmo. Usato con `improvement_found_counter` per calcolare il tempo medio impiegato dalle fasi di improvement che hanno avuto successo.

cumulative_improvement_not_found_time. La somma del tempo impiegato da tutte le esecuzioni della fase di improvement che non hanno portato a un miglioramento, durante questa esecuzione dell'algoritmo. Usato con `improvement_not_found_counter` per calcolare il tempo medio impiegato dalle fasi di improvement non riuscite.

get_best_neighbor_counter. Conta quante volte l'algoritmo ha cercato il miglior vicino nel KRandomRotateNeighborhood durante questa esecuzione. Utilizzato per calcolare il tempo medio impiegato dalla fase di improvement.

cumulative_get_best_neighbor_time. La somma del tempo impiegato da tutte le esecuzioni della fase di improvement, durante questa esecuzione dell'algoritmo. Utilizzato con `get_best_neighbor_counter` per calcolare il tempo medio impiegato dalla fase di improvement.

items_to_move_selected. Conta quante volte ogni oggetto è stato selezionato per essere spostato. Utilizzato per le scelte pesate degli oggetti da selezionare.

knapsack_selection_counter. Conta quante volte ogni zaino è stato selezionato per spostare un oggetto. Utilizzato per le scelte pesate degli zaini in cui inserire un oggetto.

item_selection_counter. Conta quante volte l'algoritmo ha selezionato un oggetto da spostare. Utilizzato con `items_to_move_selected` per calcolare quante volte un oggetto non è stato selezionato per poter dare più peso a quelli meno selezionati.

select_knapsack_counter. Conta quante volte l'algoritmo ha selezionato uno zaino per spostare un oggetto. Usato con `knapsack_selection_counter` per calcolare quante volte uno zaino non è stato selezionato per poter dare più peso a quelli meno selezionati.

try_family_add_counter. Conta quante volte l'algoritmo ha tentato di aggiungere una famiglia alla soluzione.

4.5.13 Leggere i risultati

I risultati ottenuti si trovano all'interno della directory dell'esecuzione corrente (il nome della cartella corrisponde a data e ora di avvio dell'esecuzione nel formato `YYYY-MM-DDTHH_MM_SS.FFFFFF`), che viene creata dallo script `solve_instances.py` quando viene eseguito all'interno della cartella `results`.

All'interno di questa cartella si possono trovare delle sottocartelle: una chiamata `logs`, che contiene i file di logging generati dall'algoritmo VNDS, utili per capire come l'algoritmo ha risolto il problema e se ci sono stati problemi, l'altra chiamata `gurobi_json_files`, che viene creata solo se viene eseguito Gurobi e contiene i file di output dell'esecuzione di ogni istanza.

Nella stessa cartella, viene salvata una tabella nel formato ".csv". Essa contiene: nella prima riga i nomi dei valori mostrati per ogni colonna, nelle righe successive i valori misurati per ogni istanza (un'istanza per riga). Il nome dell'istanza viene mostrato nella prima colonna, mentre il valore obiettivo della miglior soluzione trovata viene mostrato nell'ultima.

In figura uno scorcio della tabella, visualizzata con un programma di visualizzazione di fogli elettronici.

	A	B	C	D	E	F	G	H	I	J	K
1	instance_id	solver_name	-t	-k	-i	-ni	-is	-mi	-fa	-ks	-r
2	G1_SC1_instance1-k3_f7_r2_i400	vnds	600	3	0	0	10	10	30	20	3
3	G1_SC1_instance2-k3_f8_r2_i400	vnds	600	3	0	0	10	10	30	20	3
4	G1_SC1_instance3-k3_f10_r2_i500	vnds	600	3	0	0	10	10	30	20	3
5	G1_SC1_instance4-k3_f8_r2_i500	vnds	600	3	0	0	10	10	30	20	3
6	G1_SC1_instance5-k3_f10_r2_i600	vnds	600	3	0	0	10	10	30	20	3
7	G1_SC1_instance6-k3_f13_r2_i600	vnds	600	3	0	0	10	10	30	20	3
8	G1_SC1_instance7-k5_f9_r2_i400	vnds	600	3	0	0	10	10	30	20	3
9	G1_SC1_instance8-k5_f8_r2_i400	vnds	600	3	0	0	10	10	30	20	3
10	G1_SC1_instance9-k5_f9_r2_i500	vnds	600	3	0	0	10	10	30	20	3
11	G1_SC1_instance10-k5_f9_r2_i500	vnds	600	3	0	0	10	10	30	20	3
12	G1_SC1_instance11-k5_f11_r2_i600	vnds	600	3	0	0	10	10	30	20	3
13	G1_SC1_instance12-k5_f10_r2_i600	vnds	600	3	0	0	10	10	30	20	3
14	G1_SC1_instance13-k10_f7_r2_i400	vnds	600	3	0	0	10	10	30	20	3
15	G1_SC1_instance14-k10_f7_r2_i400	vnds	600	3	0	0	10	10	30	20	3
16	G1_SC1_instance15-k10_f11_r2_i500	vnds	600	3	0	0	10	10	30	20	3
17	G1_SC1_instance16-k10_f9_r2_i500	vnds	600	3	0	0	10	10	30	20	3
18	G1_SC1_instance17-k10_f12_r2_i600	vnds	600	3	0	0	10	10	30	20	3
19	G1_SC1_instance18-k10_f12_r2_i600	vnds	600	3	0	0	10	10	30	20	3

Figura 15: Tabella dei risultati dell'esecuzione di VNDS sulle istanze della prima variante.

Capitolo 5: Algoritmi risolutivi per la variante con penalità per ogni split

Per risolvere il secondo problema sono stati implementati e testati diversi algoritmi. Il primo è una variante dell'algoritmo già presentato per risolvere il primo problema, quindi un'implementazione di VNDS. Il secondo è una versione ibrida dell'algoritmo Kernel Search che utilizza VNDS per la risoluzione dei problemi ristretti generati dal metodo. Come terzo algoritmo è stata implementata una versione base della Kernel Search e, infine, il quarto è una combinazione tra Kernel Search standard e la versione ibrida con VNDS.

5.1 Algoritmo 1: VNDS

Per quanto riguarda il primo metodo, buona parte delle componenti è la stessa:

- Corpo dell'algoritmo: lo stesso indicato in *Sezione 4.5.1*
- Reset della soluzione: lo stesso indicato in *Sezione 4.5.5*
- Cambio del vicinato: lo stesso indicato in *Sezione 4.5.9*

In questo capitolo verranno evidenziate le differenze rispetto a quello implementato per il primo problema.

5.1.1 Costruzione della soluzione iniziale

La costruzione della soluzione iniziale è la stessa descritta in *Sezione 4.5.2*. Quello che cambia è che stavolta vengono prese più informazioni dalla risoluzione del rilassamento continuo del problema, per poi utilizzarle durante l'esecuzione dell'algoritmo.

In questa variante del problema, utilizzare il profitto delle famiglie come indicazione per raggiungere soluzioni migliori non funziona. Analizzando le soluzioni dei rilassamenti continui e comparandole ai risultati ottenuti da Gurobi, è stato osservato che molte delle famiglie selezionate dalle variabili x_j venivano selezionate anche da Gurobi. Il problema di queste, però, è che la maggior parte di esse risulta avere valore nullo e anche costo ridotto associato nullo. Per questo motivo le variabili utilizzate per capire quali famiglie siano le più promettenti, e quindi per assegnare a ciascuna un peso, sono le y_{ik} con $i \in I$ e $k \in K$.

Per questo motivo, oltre alle variabili x_j associate alla selezione delle famiglie, vengono estratte anche le variabili che indicano a quali zaini sono stati assegnati gli oggetti (cioè le variabili y_{ik} del modello matematico), e, per ciascuna, viene incrementato o decrementato un punteggio associato alla famiglia dell'oggetto corrispondente: se la variabile ha valore positivo nella soluzione ottima del rilassato, viene incrementato il punteggio della famiglia di una quantità pari al valore della variabile moltiplicato per 10; se la variabile ha valore nullo nella soluzione ottima del rilassato, viene diminuito il punteggio della famiglia di una quantità pari al valore assoluto del costo ridotto associato alla variabile.

La moltiplicazione per 10 è motivata dal fatto che i valori delle variabili y_{ik} sono compresi tra 0 e 1, mentre quelli dei costi ridotti possono avere valore assoluto più elevato (anche qualche centinaio), quindi moltiplicandoli per 10 si ottengono valori compresi tra 0 e 10, che riescono a discriminare meglio le famiglie.

L'associazione $\text{famiglia} - \text{punteggio}$ ottenuta ha la seguente struttura:

$$\text{families\_selection\_score} = \begin{cases} 0 : \text{family\_score} \\ 1 : \text{family\_score} \\ \vdots \\ j : \text{family\_score} \end{cases}$$
$$j \in F$$

Questa verrà usata per assegnare un peso a ogni famiglia nelle scelte casuali di aggiunta e rimozione delle famiglie, nella fase di shaking.

Viene riportato il codice che si occupa di calcolare i punteggi delle famiglie.

```
family_index = -1
for i in range(instance.n_items):
  if i in instance.first_items:
    family_index += 1
    families_selection_score.append(0)
  for k in range(instance.n_knapsacks):
    # Ottiene la variabile y[i,k] del modello matematico
    y_ik = model.getVarByName(f'y[{i},{k}]')
    # Ne controlla il valore nella soluzione ottima del rilassato
    if y_ik.X > 0.0:
      families_selection_score[family_index] += y_ik.X * 10
    else:
      families_selection_score[family_index] -= abs(y_ik.RC)
```

5.1.2 Fase di shaking

L'unica differenza dalla prima versione risiede nel valore utilizzato come peso per la scelta delle famiglie da rimuovere e da aggiungere.

Stavolta corrisponde al peso calcolato durante la costruzione della soluzione iniziale, perciò risulta diversa l'implementazione della classe *FamilySwitchNeighborhood*.

Inoltre, in questo caso, è stato invertito il peso assegnato alle famiglie in fase di rimozione. Infatti, viene dato più peso alle famiglie con un punteggio più alto per rimuoverle più facilmente, perché poi verranno aggiunte nuovamente in un altro zaino. Questo approccio ha dato risultati migliori rispetto a dare più peso alle famiglie con un punteggio più basso.

```
weights: list[int] = []
for family_index in selected_families:
  weights.append(measures.families_selection_score[family_index]/2)
```

I pesi possono essere negativi, per cui vanno traslati.

```
offset = min(weights)
positive_weights = [w - offset + 1 for w in weights]
families_to_remove = random.choices(
  selected_families, positive_weights, removal_number
)
```

5.1.3 Fase di decomposizione

Anche in questo caso l'obiettivo è selezionare quali oggetti muovere nella fase di improve. Stavolta è stato deciso di non forzare una soluzione, che probabilmente risulta non valida (come nella versione precedente), perché in questa variante del problema il numero di split va a influenzare la funzione obiettivo, anche se è maggiore di 1, perciò è possibile ottenere più miglioramenti intermedi.

Come precedentemente, l'algoritmo considera tutti gli oggetti con uno zaino associato nella soluzione corrente, però cerca di spostarne un numero maggiore. Per questo viene introdotto il parametro `variable_indexes_number_multiplier`, che moltiplica il valore di l per ottenere la quantità di oggetti da muovere.

```
if len(valid_indexes) > 0:
  variable_indexes_number = l * variable_indexes_number_multiplier
  variable_indexes = random.choices(
    valid_indexes, min(variable_indexes_number, len(valid_indexes))
  )

  x_improvement, x_improvement_value = improve(
      x_improvement, x_improvement_value, variable_indexes
  )
```

5.1.4 Fase di improvement

Il vicinato di tipo *KRandomRotate* utilizzato nella fase di improvement è ancora realizzato muovendo gli oggetti selezionati in zaini casuali. Stavolta, però, vengono effettuati una serie di tentativi di spostamento di ogni oggetto in uno zaino che la sua famiglia sta già occupando, in modo da cercare di non aumentare il numero di zaini utilizzati dalla famiglia stessa.

Prima ottiene la famiglia a cui appartiene ogni oggetto da muovere e gli zaini occupati da ognuna.

```
for i in items_to_move:
  family_index = len(instance.first_items) - 1
  item_family_dict[i] = family_index
  for family, first_item in enumerate(instance.first_items):
    if first_item > i:
```

```
      family_index = family - 1
      item_family_dict[i] = family_index
      break
  current_families_knapsacks[family_index] =
      families_knapsacks[family_index]
```

Poi cerca di muovere gli oggetti in uno zaino che la famiglia sta già occupando, dando più peso agli zaini selezionati meno volte per quell'oggetto.

```
is_feasible = False
while not is_feasible and not check_unfeasible_move_items_limit():
  for i in indexes_to_change:
    for knapsack in current_families_knapsacks[family_index]:
      w = select_knapsack_counter - knapsack_selection[i][knapsack]
      weights.append(1 + w)
    selected_knapsack = random.choices(
      current_families_knapsacks[family_index], weights
    )
    new_solution[i] = selected_knapsack
```

Infine, siccome potrebbero essere stati cambiati gli zaini occupati dalla famiglia se viene trovato un miglioramento, è necessario aggiornare la lista corrispondente.

```
if new_value > best_value:
  best_value = new_value
  best_solution = new_solution
  stop = True
  update_families_knapsacks(new_solution)
```

5.1.5 Criterio di terminazione

I criteri di terminazione e il loro controllo sono gli stessi descritti in *Sezione 4.5.10*, ad eccezione della terminazione per raggiungimento del valore ottimo. Questa condizione non è stata implementata poiché per nessuna istanza è stato trovato o fornito il suddetto valore.

5.1.6 Parametri dell'algoritmo

La maggior parte dei parametri corrisponde a quelli descritti in *Sezione 4.5.11*. I seguenti sono stati aggiunti in questa nuova versione:

> `variable_indexes_number_multiplier`. Valore per cui moltiplicare l per ottenere il numero di oggetti da muovere nella fase di improvement, scelti dalla fase di decomposizione.

> `initial_solution_builder`. Le funzioni che si occupano della costruzione della soluzione iniziale sono state spostate in una classe a parte, perciò è necessario passare il riferimento a questa classe.

> `vnds_mip_gap`. Corrisponde a quella che era una costante interna nella versione precedente (vedere *Sezione 4.5.11*).

5.1.7 Esecuzione dell'algoritmo

L'algoritmo VNDS deve essere eseguito su tutte le istanze fornite nella cartella `instances`, quindi viene avviato un ciclo che, iterativamente, legge i dati contenuti nei file, costruisce gli oggetti necessari e risolve l'istanza. Questo ciclo è implementato nello script `solve_instances.py` e, a ogni avvio, crea una nuova cartella nella cartella `OUTPUT_DIR` per memorizzare i file di output dell'esecuzione corrente.

Durante l'esecuzione, l'algoritmo salva alcuni indicatori, spiegati in *Sezione 4.5.12*, e scrive alcune informazioni di log nello *standard output* e in un file con estensione ".log" all'interno della cartella `log`. Inoltre, se la costante `SKIP_GUROBI` non è impostata a `True`, il programma esegue anche Gurobi per risolvere le stesse istanze e confrontare i risultati.

Per poter eseguire intercambiabilmente gli algoritmi, sono stati introdotte delle costanti che permettono di scegliere quale utilizzare.

Per poter avviare quello appena descritto, è necessario impostare le costanti `SEQUENTIAL_VNDS_KERNEL`, `STANDARD_KERNEL_SEARCH` e `USE_VNDS` al valore `False`, mentre la costante `USE_KERNEL_SEARCH` al valore `True`.

L'algoritmo `MkfspVnds.py` restituisce la migliore soluzione trovata, il suo valore obiettivo e gli indicatori relativi all'esecuzione. Lo script `solve_instances.py`, invece, scrive i risultati della risoluzione di ogni istanza in un file con estensione ".csv".

Per eseguire l'algoritmo su tutte le istanze, si può avviare lo script main.py tramite il comando sottostante, seguito da eventuali parametri.

```
python -m source.main
```

Se per esempio si vuole definire un tempo limite di esecuzione diverso da quello di default, si può eseguire il comando seguente.

```
python -m source.main -t 300
```

I parametri dell'algoritmo VNDS possono essere passati come argomento via riga di comando, oppure modificando direttamente i loro valori di default all'interno della classe `ParametersParser`.

Lo script `main.py` va a eseguire lo script `solve_instances.py` con il logging abilitato.

5.1.8 Indicatori

Sono state introdotte alcune ulteriori indicatori rispetto a quelli descritte in *Sezione 4.5.12*. Questi sono:

> `improve_top_solution_success_counter`. Conta quante volte risulta efficace la fase di reset della soluzione, cioè quante volte ripartire dalla miglior soluzione trovata porta un miglioramento.

> `max_solutions_distance`. Misura quante iterazioni passano al massimo tra due soluzioni successive. Viene usato per il tuning del parametro `-ni`.

> `families_selection_score`. Punteggio assegnato a ciascuna famiglia durante la costruzione della soluzione iniziale, come spiegato in *Sezione 4.5.2*.

> TTB (Time To Best). Tempo impiegato dall'algoritmo per raggiungere la migliore soluzione trovata.

> ITB (Iterations To Best). , Iterazioni impiegate per raggiungere la miglior soluzione.

5.1.9 Leggere i risultati

Il metodo di scrittura dei risultati dell'esecuzione è lo stesso descritto in *Sezione 4.5.13*, con l'aggiunta dei nuovi indicatori raccolti.

5.2 Algoritmo 2: Kernel Search ibrida

Come secondo approccio per risolvere il secondo problema, è stato scelto di avvicinarsi alla tecnica utilizzata nella Kernel Search, spiegata in *Sezione 4.4.1*. In sostituzione al risolutore MIP che dovrebbe risolvere il sottoproblema creato a ogni iterazione, viene usato l'algoritmo euristico presentato nel capitolo precedente (*Sezione 5.1*).

5.2.1 Corpo dell'algoritmo

La struttura principale riprende i concetti introdotti riguardo la Kernel Search (*Sezione 4.4.1*): vengono costruiti il kernel e i bucket, poi viene definita una soluzione iniziale, infine vengono risolti iterativamente i sottoproblemi usando l'algoritmo euristico, aggiornando la soluzione corrente e il kernel se viene trovata una soluzione migliore.

```
def KernelVndsSearch(families_per_bucket, buckets_overlapping,
    cpu_time_limit):
  kernel, buckets, families_selection_score = initialization(
    families_per_bucket, buckets_overlapping
  )
  best_solution, best_solution_value = build_initial_solution(kernel)
```

```
# Il primo sottoproblema deve essere risolto usando solo il kernel
# iniziale
buckets.insert(0, [])
bucket_number=0
```

Si evidenzia che alla lista dei bucket viene aggiunto in testa un bucket vuoto. Questo viene fatto per cercare di migliorare la soluzione iniziale utilizzando solo le variabili del kernel alla prima iterazione dell'algoritmo.

```
for b in buckets:
  if current_time > cpu_time_limit:
    break
  bucket_number += 1
  current_solution, current_solution_value = solve_bucket(
    kernel,
    b,
    bucket_number,
    best_solution,
    best_solution_value,
    families_selection_score
  )
  if current_solution_value >= best_solution_value:
    # Aggiorna la soluzione e il kernel solo se non ha trovato una
    # soluzione peggiore
    best_solution = current_solution
    best_solution_value = current_solution_value
    # Aggiunge al kernel le famiglie che sono state aggiunte in
    # questo bucket, se ce ne sono
    added_bucket_variables = get_added_bucket_variables(
      current_solution, b
    )
    kernel.extend(added_bucket_variables)
return best_solution, best_solution_value
```

Ogni volta che la risoluzione di un sottoproblema termina (cioè quando termina la funzione `solve_bucket`), viene controllato se la soluzione trovata è migliore o uguale a quella precedente. Se così fosse, la soluzione corrente viene aggiornata, poi vengono analizzate le famiglie selezionate. Se tra queste ce ne sono alcune che erano nel bucket corrente, e che quindi sono state aggiunte in questa iterazione, allora esse vengono aggiunte al kernel.

L'esecuzione termina quando tutti i bucket sono stati utilizzati, oppure quando il tempo massimo di esecuzione è stato raggiunto.

5.2.2 Fase di inizializzazione della Kernel Search

Questa prima parte riceve come input la dimensione del kernel, il numero di famiglie da inserire in ogni bucket e il numero di famiglie che compaiono sia in un bucket che nel successivo (per catturare eventuali correlazioni tra le variabili).

Inizia risolvendo il rilassamento continuo del problema con Gurobi, per ottenere gli indici delle famiglie più promettenti secondo la soluzione trovata. Questo procedimento è il medesimo descritto in *Sezione 5.1.1*, infatti si ricavano le stesse informazioni dal rilassamento continuo: le associazioni "xvars_dict", "xvars_rc_module_dict" e i punteggi assegnati alle famiglie "families_selection_score".

Invece di inserire tutte le famiglie che hanno variabile x_j non nulla nel kernel, vengono selezionate soltanto quelle la cui variabile x_j assume valore 1 nella soluzione ottima del rilassato. Questo significa che le famiglie selezionate molto probabilmente sono in numero inferiore rispetto a tutte quelle con valore x_j non nullo, infatti l'unico caso in cui sarebbero in egual numero è quando tutte le variabili x_j valgono `0` oppure `1`. Questa scelta deriva dall'osservazione di risultati migliori ottenuti in questo modo.

Una volta definito il kernel, le famiglie che rimangono nella lista ordinata vengono divise nei bucket, mantenendone l'ordine. Viene considerato anche il parametro `buckets_overlapping`, che indica quante famiglie devono essere presenti in un bucket e nel successivo.

```
def split_into_buckets(vars, bucket_size, overlap):
  buckets = []
  bucket = []
  for i in range(0,len(vars),bucket_size - overlap):
    bucket = vars[i:i + bucket_size]
    if len(bucket) == bucket_size:
        buckets.append(bucket)
  return buckets
```

5.2.3 Costruzione della soluzione iniziale

Dopo la costruzione del kernel e dei bucket, è possibile costruire la soluzione iniziale. Vengono prese una per volta le famiglie indicate dal kernel per poi inserire i loro oggetti in uno o più zaini. Il metodo attuato è lo stesso descritto in *Sezione 4.5.2* ma usando solo le famiglie presenti nel kernel iniziale.

Da notare che la soluzione costruita in questo modo potrebbe non riempire tutti gli zaini, perché non è detto le famiglie indicate nel kernel abbiano una capienza sufficiente. In tal caso, gli zaini verranno comunque riempiti nelle iterazioni successive dell'algoritmo.

5.2.4 Risoluzione di un sottoproblema

Prima di poter dare in pasto il sottoproblema all'algoritmo euristico, è necessario riuscire a ridurlo senza perdere informazioni e mantenendo l'ammissibilità delle soluzioni.

Quello che è stato scelto di fare è di costruire una nuova istanza del problema ad hoc, che contenga soltanto le famiglie presenti nel kernel e nel bucket corrente.

Come prima cosa viene specificato nell'identificativo dell'istanza che questa corrisponde al bucket corrente (indicando il numero del bucket), poi vengono definite le parti invarianti dell'istanza, cioè il numero di zaini, il numero di risorse e gli zaini stessi.

```
id = instance.id + f"_bucket{bucket_number}"
n_knapsacks = instance.n_knapsacks
n_resources = instance.n_resources
knapsacks = instance.knapsacks
```

Successivamente, si aggiungono iterativamente all'istanza le famiglie presenti nel kernel e nel bucket corrente, prendendo le relative informazioni dall'istanza originale. Siccome la codifica dell'istanza è legata all'ordine delle famiglie, è necessario mantenerlo, per poter poi ricostruire la soluzione ottenuta.

```
for j in range(instance.n_families):
  if j in kernel or j in bucket:
    family_mapping[families_count] = j
    if j in bucket:
      bucket_kernel_search.append(families_count)
    families_count += 1
    # Inserisce le caratteristiche della famiglia
    profits.append(instance.profits[j])
    penalties.append(instance.penalties[j])
    first_item = instance.first_items[j]
    last_item = instance.first_items[j + 1] if j + 1 <
        instance.n_families else instance.n_items
    first_items.append(len(solution_to_improve))
    # Inserisce gli oggetti della famiglia
    items.extend(instance.items[first_item:last_item])
    solution_to_improve.extend(best_solution[first_item:last_item])
n_families = families_count
```

Si evidenzia che viene memorizzato anche il mapping tra le famiglie dell'istanza originale e quelle dell'istanza ridotta, per poter scrivere i messaggi di logging con il corretto indice delle famiglie (quello del problema complessivo).

Una volta creata l'istanza ridotta, viene avviata la sua risoluzione mediante l'algoritmo VNDS descritto in *Sezione 5.1*. Siccome questa volta la soluzione iniziale viene passata come input, l'algoritmo VNDS parte da essa e salta la fase di costruzione della soluzione iniziale. Vengono passati anche i punteggi delle famiglie, il loro mapping con quelle dell'istanza originale e il tempo per l'esecuzione. Quest'ultimo deve essere il minimo tra il tempo standard concesso a VNDS per ogni bucket e il tempo rimanente prima di raggiungere il tempo limite dell'intera esecuzione, altrimenti l'algoritmo potrebbe terminare oltre il tempo massimo.

```
cpu_time_limit = min(remaining_time, parameters.time)
reduced_solution = vnds_solver.solve(
  solution_to_improve,
  family_mapping,
  families_selection_score
)
```

Alla fine della risoluzione, il risultato ottenuto deve essere ricongiunto all'istanza originale, perché al suo interno mancano le famiglie che sono state escluse dal calcolo. Queste non devono essere assegnate a nessuno zaino, perciò i loro oggetti vengono inseriti nella soluzione con valore -1.

```
reduced_family_index = 0
solution = []
for j in range(instance.n_families):
  if j in kernel or j in bucket:
    # Questa famiglia è nell'istanza ridotta, quindi va aggiunta la sua
    # soluzione nel problema ridotto
    first_item = reduced_instance.first_items[reduced_family_index]
    last_item = reduced_instance.first_items[reduced_family_index + 1]
        if reduced_family_index+1<len(reduced_instance.first_items)
        else len(reduced_instance.items)
    for i in range(first_item, last_item):
        solution.append(reduced_solution[i])
    # Incrementa l'indice che scorre sulle famiglie della soluzione
    # ridotta
    reduced_family_index += 1
  else:
    # Questa famiglia non è nell'istanza ridotta, quindi va tenuta
    # fuori dagli zaini
    first_item = instance.first_items[j]
    last_item = instance.first_items[j + 1] if j + 1 <
        len(instance.first_items) else len(instance.items)
    solution.extend([-1] * (last_item - first_item))
```

Infine, la soluzione ottenuta viene valutata e restituita.

5.2.5 Parametri dell'algoritmo

Siccome in questo caso viene comunque inglobato l'algoritmo VNDS, l'esecuzione richiede che vengano impostati anche i suoi parametri (si veda *Sezione 5.1.6*).

> `cpu_time_limit`. Limite di tempo massimo per l'esecuzione dell'algoritmo complessivo. Viene controllato prima di passare al bucket successivo.

`families_per_bucket`. Numero di famiglie da inserire in ogni bucket.

`overlapping`. Numero di famiglie che compaiono sia in un bucket che nel successivo.

`parameters`. Parametri dell'algoritmo VNDS.

5.2.6 Esecuzione dell'algoritmo

Anche in questo caso l'algoritmo deve essere eseguito su tutte le istanze fornite nella cartella `instances`, quindi il funzionamento è analogo a quello descritto in *Sezione 5.1.7*.

Per poter avviare l'algoritmo appena descritto, è necessario impostare le costanti `SEQUENTIAL_VNDS_KERNEL`, `STANDARD_KERNEL_SEARCH` e `USE_VNDS` al valore `False`, mentre la costante `USE_KERNEL_SEARCH` al valore `True`.

Prima di eseguire il programma è necessario impostare i valori delle costanti presenti all'inizio dello script `solve_instances.py`, perché i loro valori verranno utilizzati come parametri dell'algoritmo.

È stata mantenuto il codice per cui se la costante `SKIP_GUROBI` non è impostata a `True`, il programma esegue anche Gurobi per risolvere le stesse istanze e confrontare i risultati.

Per eseguire l'algoritmo su tutte le istanze, si può avviare lo script main.py tramite il comando sottostante, seguito da eventuali parametri.

```
python -m source.main
```

I parametri dell'algoritmo VNDS possono essere passati come argomento via riga di comando, oppure modificando direttamente i loro valori di default all'interno della classe `ParametersParser`.

Lo script `main.py` va a eseguire lo script `solve_instances.py` con il logging abilitato.

5.2.7 Indicatori

Oltre agli indicatori raccolti da VNDS, vengono memorizzati anche i seguenti dati:

`families_per_bucket`. Numero di famiglie da inserire in ogni bucket.

`top_solutions`. Lista delle migliori soluzioni trovate dall'algoritmo per ogni istanza. Non corrisponde alla lista già memorizzata in VNDS ma viene costruita considerando VNDS come una blackbox, quindi aggiungendo soltanto le soluzioni risultanti dalla risoluzione dei bucket, che hanno portato un miglioramento.

`iterations_with_added_variable_counter`. Conta il numero di iterazioni in cui è stata aggiunta al kernel almeno una famiglia che si trovava nel bucket corrente.

`added_variables_counter`. Conta il numero totale di famiglie aggiunte al kernel durante l'esecuzione dell'algoritmo.

> **TTB (Time To Best).** Tempo impiegato dall'algoritmo per raggiungere la migliore soluzione trovata.

5.2.8 Leggere i risultati

Anche in questo caso, i risultati ottenuti si trovano nella directory dell'esecuzione corrente, come spiegato in *Sezione 4.5.13*. Siccome l'esecuzione di questo algoritmo comporta anche più esecuzioni annidate di VNDS, vengono generati più file per ogni istanza.

In particolare, per ogni istanza viene generato un file di logging, nella cartella `log`, analogo a quelli generati da VNDS, e un file ".csv" con indicatori e risultati delle risoluzioni dei bucket di quell'istanza, strutturato come spiegato in *Sezione 4.5.13*. Infine, al termine della risoluzione di tutte le istanze, viene memorizzato un file ".csv" con gli indicatori e i risultati dell'esecuzione complessiva dell'algoritmo spiegato in questo capitolo, per ogni istanza.

5.3 Algoritmo 3: Kernel Search

Dopo aver implementato la Kernel Search ibridizzata con VNDS, e per verificare ulteriormente le prestazioni dell'euristica introdotta, è stato scelto di osservare quali risultati avrebbe ottenuto l'algoritmo Kernel Search standard, come descritto in *Sezione 4.4.1*. Per la sua realizzazione sono state apportate alcune modifiche al precedente algoritmo presentato.

Quello che si intende fare in questo approccio è di sostituire l'algoritmo VNDS euristico, come risolutore del sottoproblema, con Gurobi.

L'inizializzazione, la costruzione della soluzione iniziale, la costruzione del sottoproblema e la ricostruzione della soluzione ottenuta sono rimaste invariate rispetto all'algoritmo precedente.

Quello che viene introdotto al posto di VNDS è la costruzione del modello del problema per Gurobi, a cui vengono aggiunti i due vincoli descritti in *Sezione 4.4.1*.

Il primo vincolo impone che la soluzione trovata non sia peggiore di quella corrente. Sia w^* il valore della soluzione corrente, allora il vincolo è il seguente:

$$\sum_{j \in F} (p_j x_j - \delta_j s_j) \geq w^*$$

Nel codice definito come segue:

```
lhs = quicksum(
  [xvars[j]*instance.profits[j]-svars[j]*instance.penalties[j]
  for j in range(instance.n_families)]
)
model.addConstr(lhs, GRB.GREATER_EQUAL, input_obj_value,
    '_kernel_search_obj_constraint')
```

Il secondo vincolo impone che almeno una famiglia del bucket corrente venga selezionata.

Sia B l'insieme delle famiglie appartenenti al bucket corrente, allora il vincolo è il seguente:

$$\sum_{j \in B} x_j \geq 1$$

Nel codice definito come segue:

```
buckets_xvars = [xvars[j] for j in bucket_indexes]
lhs = quicksum(buckets_xvars)
model.addConstr(lhs, GRB.GREATER_EQUAL, 1,
    '_kernel_search_select_constraint')
```

Il primo vincolo viene aggiunto in ogni caso, mentre il secondo solo se il bucket considerato non è vuoto. Infatti, alla prima iterazione della Kernel Search, il bucket risulta vuoto, per poter risolvere il sottoproblema considerando soltanto le variabili del kernel iniziale.

I parametri e gli indicatori raccolti sono gli stessi dell'algoritmo precedente (*Sezione 5.2*).

5.3.1 Parametri dell'algoritmo

I parametri dell'algoritmo sono i seguenti:

> `cpu_time_limit`. Limite di tempo massimo per l'esecuzione dell'algoritmo complessivo. Viene controllato prima di passare al bucket successivo.

> `families_per_bucket`. Numero di famiglie da inserire in ogni bucket.

> `overlapping`. Numero di famiglie che compaiono sia in un bucket che nel successivo.

> `bucket_time_limit`. Limite di tempo massimo per la risoluzione di un sottoproblema.

5.3.2 Esecuzione dell'algoritmo

Anche in questo caso l'algoritmo deve essere eseguito su tutte le istanze fornite nella cartella `instances`, quindi il funzionamento è analogo a quello descritto in *Sezione 5.1.7*.

Per poter avviare l'algoritmo appena descritto, è necessario impostare le costanti `SEQUENTIAL_VNDS_KERNEL` e `USE_VNDS` al valore `False`, mentre le costanti `USE_KERNEL_SEARCH` e `STANDARD_KERNEL_SEARCH` al valore `True`.

Prima di eseguire il programma è necessario impostare i valori delle costanti presenti all'inizio dello script `solve_instances.py`, perché i loro valori verranno utilizzati come parametri dell'algoritmo.

È stata mantenuto il codice per cui se la costante `SKIP_GUROBI` non è impostata a `True`, il programma esegue anche Gurobi per risolvere le stesse istanze e confrontare i risultati.

Per eseguire l'algoritmo su tutte le istanze, si può avviare lo script main.py tramite il comando sottostante, seguito da eventuali parametri.

```
python -m source.main
```

I parametri dell'algoritmo VNDS possono essere passati come argomento via riga di comando, oppure modificando direttamente i loro valori di default all'interno della classe `ParametersParser`.

Lo script `main.py` va a eseguire lo script `solve_instances.py` con il logging abilitato.

5.3.3 Leggere i risultati

Anche in questo caso, i risultati ottenuti si trovano nella directory dell'esecuzione corrente, come spiegato in *Sezione 4.5.13*. Siccome l'esecuzione di questo algoritmo comporta anche più esecuzioni annidate di Gurobi, vengono generati più file per ogni istanza.

In particolare, per ogni istanza: viene generato un file di logging, nella cartella `logs`, analogo a quelli generati da VNDS; un ulteriore file di log per ogni bucket analizzato da Gurobi, sempre nella cartella `logs`; un file ".csv" con indicatori e risultati delle risoluzioni dei bucket di quell'istanza, strutturato come spiegato in *Sezione 4.5.13*. Infine, al termine della risoluzione di tutte le istanze, viene memorizzato un file ".csv" con gli indicatori e i risultati dell'esecuzione complessiva dell'algoritmo spiegato in questo capitolo, per ogni istanza.

5.4 Algoritmo 4: Sequential VNDS Kernel Search

I risultati ottenuti testando le performance dei precedenti algoritmi hanno evidenziato che:

- l'algoritmo Kernel Search riesce a ottenere ottimi risultati nelle istanze piccole e medie;
- l'algoritmo Kernel Search ibrida riesce a ottenere ottimi risultati nelle istanze grandi;
- la soluzione iniziale costruita dall'implementazione di VNDS è molto buona e richiede poco tempo per essere ottenuta.

Quello a cui questo algoritmo punta è di sfruttare le potenzialità di entrambi gli algoritmi, cioè utilizzare la Kernel Search ibrida per trovare una buona soluzione iniziale e selezionare le famiglie migliori, per poi utilizzare la Kernel Search e raffinare la soluzione trovata.

Questo metodo semplicemente va ad eseguire prima l'algoritmo Kernel Search ibrida, per selezionare le famiglie più promettenti, poi esegue l'algoritmo Kernel Search per trovare la soluzione migliore possibile nel resto del tempo a disposizione.

Come prima cosa vengono ottenute le informazioni fornite dal rilassamento continuo, come in *Sezione 5.1.1*. Dopodiché viene avviato l'algoritmo euristico (Kernel Search ibrida *Sezione 5.2*) passando le informazioni appena calcolate e dando un tempo ridotto rispetto al limite complessivo. Sarà l'algoritmo euristico a occuparsi della costruzione della soluzione iniziale e del suo successivo miglioramento (si veda *Sezione 5.2.3*). Vanno considerati anche i parametri dell'algoritmo VNDS perché è da essi che dipende il tempo di esecuzione di ogni sottoproblema.

```
# Ottiene le informazioni dalla soluzione del rilassamento continuo
gurobi_solver = GurobiSolver()
relaxation_info, relaxation_measures =
gurobi_solver.solve_relaxed_and_get_info(instance,
relaxation_time_limit)
# Avvia l'algoritmo Kernel Search ibrido
best_solution, best_solution_value = kernel_search_vnds_solver.solve(
    kernel_size=kernel_size,
    families_per_bucket=families_per_bucket,
    overlapping=overlapping,
    standard_kernel_search=False,
    input_relaxation_info=relaxation_info
)
# Avvia l'algoritmo Kernel Search standard, con la soluzione trovata
# dall'algoritmo precedente
best_solution, best_solution_value =
kernel_search_standard_solver.solve(
    families_per_bucket=families_per_bucket,
    overlapping=overlapping,
    standard_kernel_search=True,
    input_solution=best_solution,
    input_relaxation_info=relaxation_info
)
```

Una volta che il primo algoritmo termina, il risultato viene passato all'algoritmo Kernel Search (*Sezione 5.3*), insieme alle informazioni sul rilassamento continuo. In questo caso, non è ne utile ne necessario costruire un nuovo kernel e una nuova soluzione iniziale, infatti, vengono utilizzate le famiglie già selezionate dalla Kernel Search ibrida per definire il kernel, e il valore della soluzione ottenuta come valore da migliorare. Quest'ultimo, perciò, viene utilizzato come valore per il vincolo aggiuntivo della Kernel Search (quello che mira ad aumentarne l'efficienza, come spiegato in *Sezione 4.4.1*).

```
# Aggiunge il vincolo per il valore della soluzione
lhs = quicksum(
  [xvars[j]*instance.profits[j]-svars[j]*instance.penalties[j]
  for j in range(instance.n_families)]
)
model.addConstr(lhs, GRB.GREATER_EQUAL, input_obj_value,
    '_kernel_search_obj_constraint')
# I vincoli di selezione vengono aggiunti solo se il bucket non è vuoto
if len(bucket_indexes) > 0:
    buckets_xvars = [xvars[j] for j in bucket_indexes]
    lhs = quicksum(buckets_xvars)
    model.addConstr(lhs, GRB.GREATER_EQUAL, 1,
```

```
        '_kernel_search_select_constraint')
    model.update()
```

5.4.1 Parametri dell'algoritmo

Dato che l'algoritmo internamente esegue VNDS, Kernel Search ibrida e Kernel Search standard, è necessario fornire i parametri per ciascuno di essi.

`parameters`. Parametri per l'algoritmo VNDS.

`bucket_time_limit`. Tempo massimo da dedicare alla risoluzione di un bucket per l'algoritmo Kernel Search standard.

`heuristic_time_limit`. Tempo massimo da dedicare all'intera esecuzione, quindi all'algoritmo Sequential VNDS Kernel.

`vnds_kernel_time_limit`. Tempo massimo da dedicare all'esecuzione del primo algoritmo, cioè della Kernel Search ibrida.

`vnds_variable_indexes_number_multiplier`. Parametro che viene passato a VNDS (vedere *Sezione 5.1.6*).

`relaxation_time_limit`. Tempo massimo per la risoluzione del rilassamento continuo per la costruzione della soluzione iniziale.

`kernel_size`. Dimensione del kernel da utilizzare per l'algoritmo Kernel Search ibrido. Non viene usato da Kernel Search standard perché la dimensione del suo kernel è definita dalle famiglie selezionate dal primo algoritmo.

`families_per_bucket`. Dimensione dei bucket. Viene usato per entrambi gli algoritmi, quindi i bucket avranno la stessa dimensione in entrambi.

`overlapping`. Numero di famiglie che compaiono sia in un bucket che nel successivo. Viene usato per entrambi gli algoritmi, quindi i bucket avranno lo stesso overlapping in entrambi.

Il limite totale di tempo per la Kernel Search standard viene calcolato come differenza tra `heuristic_time_limit` e `vnds_kernel_time_limit`.

5.4.2 Esecuzione dell'algoritmo

Anche in questo caso l'algoritmo deve essere eseguito su tutte le istanze fornite nella cartella `instances`, quindi il funzionamento è analogo a quello descritto in *Sezione 5.1.7*.

Per poter avviare l'algoritmo appena descritto, è necessario impostare le costanti `USE_KERNEL_SEARCH`, `STANDARD_KERNEL_SEARCH` e `USE_VNDS` al valore `False`, mentre la costante `SEQUENTIAL_VNDS_KERNEL` al valore `True`.

Prima di eseguire il programma è necessario impostare i valori delle costanti presenti all'inizio dello script `solve_instances.py`, perché i loro valori verranno utilizzati come parametri dell'algoritmo.

È stata mantenuto il codice per cui se la costante `SKIP_GUROBI` non è impostata a `True`, il programma esegue anche Gurobi per risolvere le stesse istanze e confrontare i risultati.

Per eseguire l'algoritmo su tutte le istanze, si può avviare lo script main.py tramite il comando sottostante, seguito da eventuali parametri.

```
python -m source.main
```

I parametri dell'algoritmo VNDS possono essere passati come argomento via riga di comando, oppure modificando direttamente i loro valori di default all'interno della classe `ParametersParser`.

Lo script `main.py` va a eseguire lo script `solve_instances.py` con il logging abilitato.

5.4.3 Leggere i risultati

Anche in questo caso, i risultati ottenuti si trovano nella directory dell'esecuzione corrente, come spiegato in *Sezione 4.5.13*. Siccome l'esecuzione di questo algoritmo non introduce nuovi indicatori ma va a eseguire due algoritmi, è stato scelto di fare in modo che i risultati di entrambi vengano salvati nella stessa cartella con nomi diversi.

Per ogni istanza: vengono generati due file di logging, uno per la Kernel Search ibrida e uno per quella standard, nella cartella `logs`, analoghi a quelli generati da VNDS; un ulteriore file di log per ogni bucket analizzato da Gurobi, sempre nella cartella `logs`; due file ".csv", uno per ognuna dei due algoritmi, con indicatori e risultati delle risoluzioni dei bucket di quell'istanza, strutturato come spiegato in *Sezione 4.5.13*. Infine, al termine della risoluzione di tutte le istanze, vengono memorizzati due file ".csv", uno per la Kernel Search ibrida e uno per quella standard, ognuno con le misure e i risultati ottenuti in ogni istanza dall'algoritmo corrispondente.

Per poter leggere il valore finale della funzione obiettivo bisogna guardare la tabella dei risultati della Kernel Search standard, perché è l'ultimo algoritmo eseguito.

Capitolo 6: Risultati computazionali

In questo capitolo vengono descritte le istanze utilizzate e presentati i risultati ottenuti risolvendo con metodi esatti ed euristici i due problemi analizzati.

I calcoli sono stati eseguiti su un sistema con processore Intel Core i5-4670 CPU 3.40GHz, con 16 GB di RAM e sistema operativo Windows 10. La versione di Gurobi Optimizer utilizzata è la 11.0.2, mentre la versione di Python è la 3.10.13 con il modulo gurobipy alla versione 11.0.2.

La risoluzione di ogni istanza tramite Gurobi è stata effettuata imponendo soltanto un tempo limite di un'ora, senza ulteriori vincoli, perciò ha utilizzato i quattro core a disposizione. Gli algoritmi implementati in Python, invece, sono stati programmati per essere eseguiti su un singolo core e gli è stato limitato il tempo di esecuzione a dieci minuti.

Questa differenza di tempo limite tra i due programmi serve per giustificare l'utilizzo di un algoritmo euristico piuttosto che di un risolutore esatto. Infatti esso potrebbe raggiungere una soluzione buona, o a volte anche migliore, in un tempo ridotto.

Riguardo alle istanze che verranno analizzate, esse sono diverse per le due varianti del problema. Per alcune delle istanze relative alla prima variante del problema, Gurobi ha determinato la soluzione ottima in un tempo inferiore al tempo limite. Nessuna istanza relativa alla seconda variante è stata, invece, risolta all'ottimo.

6.1 Descrizione delle istanze

Le istanze sono state memorizzate in formato `JSON` e seguono tutte la stessa struttura, poiché sono caratterizzate dagli stessi attributi, ma vengono poi usate in modo diverso per il calcolo della funzione obiettivo e per il controllo dei vincoli in ciascuna delle due varianti del problema.

6.1.1 Formato delle istanze

> **Importante.**
> Sebbene nella dichiarazione formale del problema gli indici partano da 1, tutte le istanze utilizzano indici basati su 0 (come tutte le strutture di dati di tipo array in Java e Python). Per esempio, data una lista di n elementi, il primo elemento ha indice 0, il secondo elemento ha indice 1 e così via… l'ultimo elemento ha indice `n-1`.

In ogni file di istanza gli elementi sono ordinati per famiglia: se la prima famiglia ha dimensione f_1, i suoi elementi sono indicizzati da 0 a $f_1 - 1$, f_1 è l'indice del primo elemento della seconda famiglia e così via. Le istanze di benchmark sono memorizzate in file di testo in formato `JSON`. Ogni file descrive un'istanza e contiene un oggetto `JSON` con le seguenti proprietà.

- `id`: una stringa, un nome che identifica in modo univoco l'istanza,
- `n_items`: un intero, il numero di oggetti disponibili, cioè n,
- `n_families`: un intero, il numero di famiglie disponibili, cioè m,
- `n_knapsacks`: un intero, il numero di zaini disponibili, cioè $k_{\max}$,

- `n_resources`: un intero, il numero di risorse disponibili, cioè $r_{\max}$,
- `profits`: una lista di interi di lunghezza m, il valore in posizione $0 \leq j \leq m-1$ è il profitto ottenuto quando viene selezionata la famiglia j,
- `penalties`: una lista di interi di lunghezza m, il valore in posizione $0 \leq j \leq m-1$ è la penalità associata alla famiglia j,
- `first_items`: una lista di interi di lunghezza m, il valore in posizione $0 \leq j \leq m-1$ è l'indice del primo elemento che appartiene alla famiglia j,
- `items`: una lista di lunghezza n di liste di lunghezza $r_{\max}$ di interi, per $0 \leq i \leq n-1$ e $0 \leq r \leq r_{\max}-1$ il valore di `items[i][r]` è uguale a w_{ir}, cioè la quantità di risorsa di tipo r necessaria per assegnare l'oggetto i in uno zaino,
- `knapsacks`: una lista di lunghezza $k_{\max}$ di liste di lunghezza $r_{\max}$ di interi, per $0 \leq k \leq k_{\max}-1$ e $0 \leq r \leq r_{\max}-1$ il valore di `knapsacks[i][r]` è uguale a C_{kr}, cioè la capacità massima dello zaino k rispetto al tipo di risorsa r.

6.2 Istanze per la variante del problema con penalità fissa

Per affrontare la prima variante sono state prese come riferimento le istanze già presenti in letteratura, indicate in Mancini et al. (2021 *[1]*) e Mancini et al. (2022 *[2]*). Nel primo articolo vengono descritte nel dettaglio mentre nel secondo articolo vengono solo indicate le loro caratteristiche principali. Le istanze fornite e il programma utilizzato per la loro generazione sono presenti nel repository GitHub di uno degli autori dei due articoli (https://github.com/miciav/instance-generator).

Ho richiesto direttamente agli autori i file delle istanze descritte nei due documenti e ho confrontato le caratteristiche indicate nei paper con quelle delle istanze fornite e:

- Riguardo le istanze indicate nel dettaglio in Mancini et al. (2021 *[1]*), sono riuscito a trovare tutte quelle dei gruppi SC-1, M1, M2, M3, M4, Y1, Y2, W1, W2. Ne ho trovate anche per SC-4 ma con il numero di famiglie leggermente diverso, perciò penso non siano le stesse usate per i risultati mostrati nel paper. Del gruppo SC-2 ho trovato solo 10 istanze, mentre secondo il paper dovrebbero essere 90, e anche in questo caso il numero di famiglie è leggermente diverso rispetto a quelle indicate nel paper. Non sono riuscito a trovare le istanze del gruppo SC-3.
- Riguardo le istanze riportate in Mancini et al. (2022 *[2]*), di cui non è stato fornito il dettaglio, sono riuscito a trovare tutte quelle dei gruppi G1, G2, G3 (riusando alcune istanze già usate in G2), G4 (riusando alcune istanze già usate in G2) e G5. Non sono riuscito a trovare le istanze del gruppo G6.

Per risolvere queste inconsistenze, le istanze mancanti sono state generate con lo stesso programma utilizzato dagli autori dei due paper.

Per l'analisi sperimentale, visto che il numero di istanze usate in Mancini et al. (2022 *[2]*) ammonta a più di cinquecento, ne ho selezionate casualmente due per ogni gruppo ottenenedo 104 istanze. Nella seguente tabella ne vengono riportati i dettagli.

Va evidenziato che, siccome per ogni tipologia di istanza sono state selezionate due istanze casuali, ogni riga della tabella rappresenta due istanze precise. Per questo motivo nella colonna "Famiglie" sono indicate due cifre separate da una virgola, che rappresentano il numero di famiglie delle due istanze selezionate.

Inoltre, per queste istanze, sono state memorizzate due ulteriori proprietà:

- `betas`: una lista di lunghezza $k_{\max}$ di liste di lunghezza $r_{\max}$ di valori reali, per $0 \leq k \leq k_{\max} - 1$ e $0 \leq r \leq r_{\max} - 1$ il valore di `betas[i][r]` è uguale a un parametro β_{kd} utilizzato nei paper Mancini et al. (2021 *[1]*) e Mancini et al. (2022 *[2]*) per definire le capacità degli zaini in fase di generazione delle istanze. Questi valori sono stati mantenuti durante la conversione delle istanze poiché erano memorizzati nei file di testo originali.
- `optimum`: un intero, il valore della funzione obiettivo della soluzione ottima per l'istanza (se trovata). Questo valore (se presente) è stato calcolato con Gurobi e memorizzato nei file di istanza per poter terminare anzitempo l'esecuzione dell'algoritmo presentato in questo lavoro se il valore della funzione obiettivo raggiunge il valore ottimo, per diminuire i tempi di calcolo a causa dell'elevato numero di istanze da risolvere.

Gruppo	Sottogruppo	Zaini	Oggetti	Famiglie	Risorse
G1	SC1	3	400	7,8	2
		3	500	8,10	2
		3	600	10,13	2
		5	400	8,9	2
		5	500	9,9	2
		5	600	10,11	2
		10	400	7,7	2
		10	500	9,11	2
		10	600	12,12	2
	SC2	3	400	8,7	2
		3	500	9,9	2
		3	600	9,13	2
		5	400	8,8	2
		5	500	9,10	2
		5	600	11,12	2
		10	400	8,9	2
		10	500	9,10	2
		10	600	11,12	2

	SC3	3	400	6,8	2
		3	500	8,9	2
		3	600	9,10	2
		5	400	6,8	2
		5	500	8,9	2
		5	600	9,13	2
		10	400	5,8	2
		10	500	8,9	2
		10	600	8,10	2
	SC4	3	400	8,10	2
		3	500	9,9	2
		3	600	11,12	2
		5	400	7,10	2
		5	500	9,10	2
		5	600	10,12	2
		10	400	7,7	2
		10	500	8,9	2
		10	600	9,12	2
G2		10	600	10,11	2
		10	600	12,12	4
		10	600	10,11	6
		10	600	9,10	8
G3		10	600	11,12	2
		15	600	10,12	2
		20	600	9,11	2
G4		10	600	12,13	2
		10	800	15,16	2
		10	100	18,19	2
G5	1	10	100	5,5	2
	2	10	100	10,10	2
	3	10	100	15,15	2
	4	10	100	20,20	2
G6	10	10	100	33,33	2
	20	20	100	33,33	2

6.3 Risultati relativi alla prima variante del problema con VNDS

Vengono riportati e commentati i risultati ottenuti dall'implementazione di VNDS descritta in *Sezione 4.5*.

I parametri che mediamente hanno ottenuto risultati migliori nella risoluzione delle istanze del primo problema sono i seguenti:

-t	-k	i	-ni	-is	-mi	-fa	-ks	-r
600	3	0	0	10	10	30	20	30

Per capire come sia stato effettuato il tuning dei parametri e siano stati impostati questi valori, va precisato che sono state eseguite molte prove, variandone i valori e osservando alcuni indicatori risultanti dall'esecuzione e i messaggi di logging. Prendendo come esempio il parametro `-is`, per decidere il valore da assegnare è stato analizzato il valore assunto dal contatore che misura quante volte viene raggiunto il limite definito dal parametro nelle istanze; è stato considerato il tempo impiegato per l'analisi del vicinato di tipo KRandomRotate (dove viene usato questo limite) e quanto venga influenzato dal valore del parametro; sono stati osservati i file di log per capire quanti tentativi vale la pena fare per ottenere un miglioramento. La stessa cosa è stata fatta anche per gli altri parametri che vanno a definire il numero di tentativi (come `-mi`, `-fa`, `-ks`). Ovviamente è stato valutato anche il valore raggiunto dalla soluzione ottima e per quante istanze l'ottimo veniva raggiunto.

I risultati ottenuti per ogni istanza sono riportati nella tabella seguente. La colonna *Istanza* contiene il nome di riferimento dell'istanza, le colonne *T(s)* indicano il tempo impiegato per risolvere l'istanza e le colonne *Obj* il valore della funzione obiettivo della miglior soluzione trovata dall'algoritmo. In verde sono evidenziate le istanze per le quali è stata trovata la soluzione ottima (nel caso di VNDS, 20 istanze). La colonna *%* indica il rapporto percentuale tra la soluzione trovata da VNDS e quella trovata da Gurobi.

Istanza	Gurobi		VNDS		%
	T(s)	Obj	T(s)	Obj	
G1_SC1_instance1	0,1855	216	378,5623	216	100%
G1_SC1_instance2	0,1787	281	600,0769	243	86%
G1_SC1_instance3	0,9626	309	600,2131	289	93%
G1_SC1_instance4	0,3975	235	2,0685	235	100%
G1_SC1_instance5	0,811	432	600,2569	417	96%
G1_SC1_instance6	1,071	413	600,0353	404	97%
G1_SC1_instance7	1,1524	223	169,6432	223	100%
G1_SC1_instance8	0,3573	302	6,093	302	100%
G1_SC1_instance9	0,8079	291	600,4645	259	89%
G1_SC1_instance10	0,9943	323	600,2926	304	94%
G1_SC1_instance11	1,3267	421	600,686	409	97%

G1_SC1_instance12	1,2189	326	600,6512	301	92%
G1_SC1_instance13	1,0261	131	600,3012	120	91%
G1_SC1_instance14	1,1778	112	66,2396	112	100%
G1_SC1_instance15	3,4204	270	0,1832	270	100%
G1_SC1_instance16	2,0307	231	600,0404	226	97%
G1_SC1_instance17	5,2606	256	600,2792	240	93%
G1_SC1_instance18	5,8614	258	600,1796	248	96%
G1_SC2_instance1	1,0497	192	600,3224	186	96%
G1_SC2_instance2	0,363	141	8,5707	141	100%
G1_SC2_instance3	0,1972	359	600,4554	311	86%
G1_SC2_instance4	0,3574	218	16,995	218	100%
G1_SC2_instance5	1,5526	196	0,0549	196	100%
G1_SC2_instance6	0,5268	345	600,4049	320	92%
G1_SC2_instance7	0,7528	191	600,4452	182	95%
G1_SC2_instance8	0,7074	298	600,0866	281	94%
G1_SC2_instance9	0,9037	177	600,0725	141	79%
G1_SC2_instance10	2,7341	331	600,4519	308	93%
G1_SC2_instance11	1,8513	409	600,5896	346	84%
G1_SC2_instance12	3,2873	308	600,6809	280	90%
G1_SC2_instance13	1,1708	227	600,1056	208	91%
G1_SC2_instance14	1,6883	271	600,0761	254	93%
G1_SC2_instance15	8,4934	261	600,3353	209	80%
G1_SC2_instance16	3,2197	239	600,1531	210	87%
G1_SC2_instance17	14,7438	254	7,3062	254	100%
G1_SC2_instance18	8,8238	289	600,2531	249	86%
G1_SC3_instance1	0,1429	96	0,6177	96	100%
G1_SC3_instance2	0,3286	180	600,2418	170	94%
G1_SC3_instance3	0,2954	239	0,0857	239	100%
G1_SC3_instance4	0,2555	227	600,3382	220	96%
G1_SC3_instance5	0,3791	143	600,0673	136	95%
G1_SC3_instance6	0,4088	124	0,0583	124	100%
G1_SC3_instance7	1,0171	217	535,15	217	100%
G1_SC3_instance8	0,2589	106	600,2059	94	88%
G1_SC3_instance9	0,5824	125	600,3448	115	92%
G1_SC3_instance10	0,6514	153	0,1196	153	100%

G1_SC3_instance11	1,327	332	36,3146	332	100%
G1_SC3_instance12	1,3584	114	600,3802	113	99%
G1_SC3_instance13	0,4995	48	0,1538	48	100%
G1_SC3_instance14	1,4581	141	600,2112	140	99%
G1_SC3_instance15	1,0059	151	600,2056	138	91%
G1_SC3_instance16	1,9866	223	0,1561	223	100%
G1_SC3_instance17	4,3218	125	600,1589	122	97%
G1_SC3_instance18	97,6178	225	600,1599	172	76%
G1_SC4_instance1	0,1493	251	600,0013	231	92%
G1_SC4_instance2	0,1265	277	600,1353	270	97%
G1_SC4_instance3	0,3262	291	1,6948	291	100%
G1_SC4_instance4	0,2417	227	600,0769	209	92%
G1_SC4_instance5	1,6266	375	600,4729	361	96%
G1_SC4_instance6	1,0763	393	600,2282	376	95%
G1_SC4_instance7	0,8069	270	600,1299	235	87%
G1_SC4_instance8	0,3178	283	600,1237	248	87%
G1_SC4_instance9	2,1586	455	600,0457	427	93%
G1_SC4_instance10	1,778	424	600,3321	394	92%
G1_SC4_instance11	50,6395	503	600,3658	492	97%
G1_SC4_instance12	2,5309	475	600,1454	398	83%
G1_SC4_instance13	2,5559	164	2,8701	164	100%
G1_SC4_instance14	2,6348	201	600,6163	174	86%
G1_SC4_instance15	8,942	332	600,259	282	84%
G1_SC4_instance16	11,9119	284	600,5176	279	98%
G1_SC4_instance17	4,9749	352	600,094	315	89%
G1_SC4_instance18	17,8507	530	600,1837	463	87%
G2_instance1	17,2489	309	600,5963	278	89%
G2_instance2	7,1678	278	600,0631	213	76%
G2_instance3	19,0656	407	600,5016	382	93%
G2_instance4	23,1354	349	600,0587	282	80%
G2_instance5	15,678	321	600,3319	264	82%
G2_instance6	38,2466	282	600,4045	225	79%
G2_instance7	24,222	280	600,069	218	77%
G2_instance8	17,4905	307	600,6225	274	89%
G3_instance1	19,1918	366	600,4928	337	92%

G3_instance2	8,3513	331	600,1793	308	93%
G3_instance3	31,8746	470	600,418	385	81%
G3_instance4	8,5603	329	600,2639	288	87%
G3_instance5	22,045	340	600,2454	290	85%
G3_instance6	3600,4681	251	600,1723	225	89%
G4_instance1	18,3954	384	600,3371	338	88%
G4_instance2	10,6172	440	600,1257	390	88%
G4_instance3	27,6514	382	600,2202	338	88%
G4_instance4	70,5881	421	600,1077	400	95%
G4_instance5	39,8635	474	600,6552	422	89%
G4_instance6	96,7533	514	600,6983	456	88%
G5_1_instance1	1,1481	145	600,0193	137	94%
G5_1_instance2	0,188	204	600,0694	202	99%
G5_2_instance1	28,1131	328	600,0352	301	91%
G5_2_instance2	17,9712	336	600,0459	306	91%
G5_3_instance1	112,3363	473	600,1535	435	91%
G5_3_instance2	2,709	427	600,0797	398	93%
G5_4_instance1	3600,0613	590	600,1179	532	90%
G5_4_instance2	1359,6313	764	600,0417	581	76%
G6_10_instance1	2,3529	393	600,1496	357	90%
G6_10_instance2	11,3533	295	600,0842	277	93%
G6_20_instance1	76,0126	630	600,1527	551	87%
G6_20_instance2	132,0964	639	600,0534	560	87%

I risultati ottenuti mostrano che VNDS è in grado di trovare in poco tempo dei risultati buoni rispetto a Gurobi (in media 91% rispetto al miglior risultato trovato da Gurobi), nonostante le istanze prese dalla letteratura siano di dimensioni ridotte e quindi estremamente semplici da risolvere per risolutori avanzati come Gurobi. Si può osservare che nei due casi in cui Gurobi non trova la soluzione ottima, VNDS arriva comunque al 90% della miglior soluzione di Gurobi.

Nella prossima tabella vengono riportati i risultati migliori ottenuti per tre configurazioni di VNDS, di cui l'ultima corrisponde a quella dei risultati migliori (visti sopra).

Istanza	Gurobi	VNDS (1)	VNDS (2)	VNDS (3)
	Obj	Obj	Obj	Obj
G1_SC1_instance1	216	183	216	216
G1_SC1_instance2	281	281	243	243
G1_SC1_instance3	309	305	289	289

G1_SC1_instance4	235	235	235	235
G1_SC1_instance5	432	380	385	417
G1_SC1_instance6	413	361	410	404
G1_SC1_instance7	223	176	201	223
G1_SC1_instance8	302	271	302	302
G1_SC1_instance9	291	269	259	259
G1_SC1_instance10	323	283	261	304
G1_SC1_instance11	421	381	359	409
G1_SC1_instance12	326	294	276	301
G1_SC1_instance13	131	107	120	120
G1_SC1_instance14	112	97	112	112
G1_SC1_instance15	270	245	270	270
G1_SC1_instance16	231	201	226	226
G1_SC1_instance17	256	225	240	240
G1_SC1_instance18	258	224	248	248
G1_SC2_instance1	192	186	168	186
G1_SC2_instance2	141	141	141	141
G1_SC2_instance3	359	311	311	311
G1_SC2_instance4	218	218	218	218
G1_SC2_instance5	196	158	196	196
G1_SC2_instance6	345	273	320	320
G1_SC2_instance7	191	182	182	182
G1_SC2_instance8	298	296	265	281
G1_SC2_instance9	177	176	141	141
G1_SC2_instance10	331	311	308	308
G1_SC2_instance11	409	326	346	346
G1_SC2_instance12	308	232	254	280
G1_SC2_instance13	227	168	208	208
G1_SC2_instance14	271	271	232	254
G1_SC2_instance15	261	244	209	209
G1_SC2_instance16	239	218	210	210
G1_SC2_instance17	254	220	254	254
G1_SC2_instance18	289	215	249	249
G1_SC3_instance1	96	91	91	96
G1_SC3_instance2	180	180	170	170

G1_SC3_instance3	239	232	239	239
G1_SC3_instance4	227	220	220	220
G1_SC3_instance5	143	136	136	136
G1_SC3_instance6	124	121	124	124
G1_SC3_instance7	217	185	217	217
G1_SC3_instance8	106	87	94	94
G1_SC3_instance9	125	125	115	115
G1_SC3_instance10	153	138	153	153
G1_SC3_instance11	332	310	309	332
G1_SC3_instance12	114	114	113	113
G1_SC3_instance13	48	46	48	48
G1_SC3_instance14	141	126	140	140
G1_SC3_instance15	151	131	138	138
G1_SC3_instance16	223	162	223	223
G1_SC3_instance17	125	76	122	122
G1_SC3_instance18	225	165	172	172
G1_SC4_instance1	251	156	231	231
G1_SC4_instance2	277	270	270	270
G1_SC4_instance3	291	232	273	291
G1_SC4_instance4	227	184	227	209
G1_SC4_instance5	375	333	359	361
G1_SC4_instance6	393	349	346	376
G1_SC4_instance7	270	220	235	235
G1_SC4_instance8	283	263	248	248
G1_SC4_instance9	455	448	435	427
G1_SC4_instance10	424	369	394	394
G1_SC4_instance11	503	337	443	492
G1_SC4_instance12	475	455	398	398
G1_SC4_instance13	164	133	164	164
G1_SC4_instance14	201	159	174	174
G1_SC4_instance15	332	260	286	282
G1_SC4_instance16	284	243	279	279
G1_SC4_instance17	352	251	315	315
G1_SC4_instance18	530	460	458	463
G2_instance1	309	245	278	278

G2_instance2	278	234	211	213
G2_instance3	407	348	364	382
G2_instance4	349	270	270	282
G2_instance5	321	264	252	264
G2_instance6	282	224	231	225
G2_instance7	280	223	218	218
G2_instance8	307	214	274	274
G3_instance1	366	315	319	337
G3_instance2	331	289	308	308
G3_instance3	470	342	400	385
G3_instance4	329	275	288	288
G3_instance5	340	264	290	290
G3_instance6	251	203	225	225
G4_instance1	384	316	338	338
G4_instance2	440	361	371	390
G4_instance3	382	338	345	338
G4_instance4	421	324	400	400
G4_instance5	474	389	400	422
G4_instance6	514	372	456	456
G5_1_instance1	145	83	137	137
G5_1_instance2	204	174	202	202
G5_2_instance1	328	272	312	301
G5_2_instance2	336	291	306	306
G5_3_instance1	473	366	428	435
G5_3_instance2	427	381	395	398
G5_4_instance1	590	442	520	532
G5_4_instance2	764	585	577	581
G6_10_instance1	393	312	323	357
G6_10_instance2	295	245	247	277
G6_20_instance1	630	490	523	551
G6_20_instance2	639	471	525	560
Numero Ottimi		8	17	20

Nelle prime due configurazioni, i parametri utilizzati sono:

-t	-k	i	-ni	-is	-mi	-fa	-ks	-r
600	7	0	0	70	15	5	50	15

La loro differenza sta nei pesi utilizzati per la selezione delle famiglie da rimuovere. Infatti, per VNDS (1) mostrata nella prima colonna veniva usato come peso il numero di volte in cui una famiglia era stata selezionata, quindi: durante l'aggiunta veniva dato più peso alle famiglie selezionate meno volte, mentre durante la rimozione veniva dato più peso alle famiglie selezionate più volte. Per VNDS (2) riportata nella seconda colonna, invece, è stato introdotto l'utilizzo del profitto come peso, per cui viene dato più peso alle famiglie con profitto maggiore.

Per VNDS (3) in terza colonna è stato mantenuto il profitto come peso, ma sono stati cambiati i valori dei parametri. Poiché questa variante ha ottenuto ottenere un leggero miglioramento, questi sono stati scelti come parametri migliori (visti all'inizio di *Sezione 6.3*).

Si può notare che la differenza nei valori tra VNDS (1) e VNDS (2) risulta maggiore rispetto a quella ottenuta tra la VNDS (2) e VNDS (3). Questo perché l'implementazione dell'utilizzo dei profitti come peso riesce a fornire una buona indicazione riguardo le famiglie da selezionare.

6.4 Istanze per la variante con penalità dipendente dal numero di split

Per analizzare la performance dei quattro algoritmi proposti per la variante con costi di penalità dipendenti dal numero di split per famiglia, essendo il problema non noto in letteratura, sono state utilizzate istanze generate nel Laboratorio di Ricerca Operativa coordinato dalla Prof.ssa Mansini. Si tratta di dodici istanze costruite facendo riferimento a quelle pubblicate in *[30]* da J E Beasley riguardo agli zaini multidimensionali, e quindi di elevata complessità computazionale.

Sarebbe stato possibile utilizzare anche le istanze della prima variante, perché gli attributi principali sono gli stessi, ma, dato il tempo molto ristretto impiegato da Gurobi per risolverle all'ottimo (a eccezione di due casi in cui non è stato raggiunto) già nella prima variante con costo fisso di split, è stato ritenuto poco significativo utilizzarle anche per la seconda.

Nella prima variante del problema sono state utilizzate perché già fornite dalla letteratura, mentre nella seconda variante sono state studiate istanze in grado di mettere in difficoltà risolutori commerciali estremamente potenti, come Gurobi.

In tabella sono riportati i dettagli delle istanze utilizzate.

Istanza	Zaini	Oggetti	Famiglie	Risorse
instance01	5	500	67	10
instance02	10	1000	131	10
instance03	15	1500	202	10
instance04	5	500	67	30
instance05	10	1000	132	30
instance06	15	1500	202	30
instance07	5	1250	167	30
instance08	10	2500	330	30
instance09	15	3750	502	30
instance10	5	2500	334	30
instance11	10	5000	670	30
instance12	15	7500	1000	30

6.5 Risultati per la variante con penalità dipendente dal numero di split

In questo paragrafo verranno analizzate le performance dei quattro algoritmi proposti per la risoluzione della variante del problema, non nota in letteratura, che introduce l'assegnazione di una penalità per ogni split subito dalle famiglie selezionate.

6.5.1 Analisi computazionale dell'algoritmo VNDS

I parametri che mediamente hanno ottenuto risultati migliori nella risoluzione delle istanze del secondo problema sono i seguenti:

-t	-k	i	-ni	-is	-mi	-fa	-ks	-r
600	2	0	0	10	100	30	20	80

Anche in questo caso sono stati eseguiti diversi test di tuning per ottenere i parametri migliori, osservando ogni volta i risultati ottenuti, gli indicatori raccolti e i messaggi di logging.

I migliori risultati ottenuti sono riportati nella tabella seguente. Vengono comparati i valori delle funzioni obiettivo per le migliori soluzioni trovate da Gurobi e dall'implementazione di VNDS. La colonna % indica il rapporto percentuale tra la soluzione trovata dal metodo VNDS e quella trovata da Gurobi. In verde sono evidenziate le istanze in cui l'algoritmo euristico è riuscito a ottenere un risultato migliore o uguale a Gurobi, cioè se il rapporto percentuale risulta maggiore o uguale a 100%. Il tempo di esecuzione è stato omesso perché è sempre stato raggiunto il limite massimo (3600s per Gurobi, 600s per VNDS), trattandosi, come già detto, di istanze molto complesse. È comunque importante sottolineare che il tempo a disposizione per il metodo euristico è un sesto rispetto a quello assegnato al risolutore MIP.

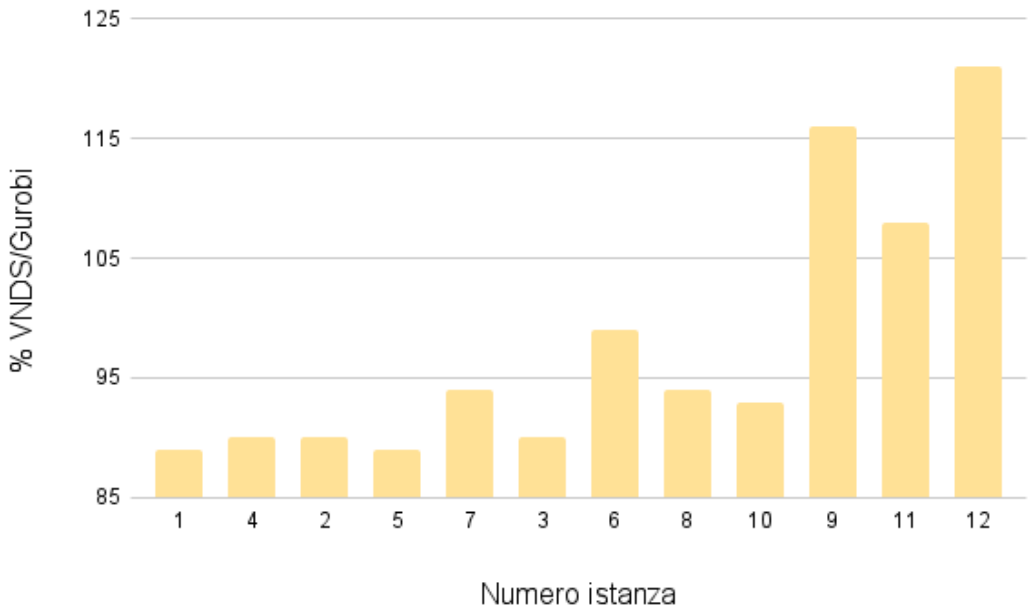
Istanza	Gurobi	VNDS	%
	Obj	Obj	
instance01	93103	82543	88%
instance02	272718	246412	90%
instance03	723224	653127	90%
instance04	85036	76695	90%
instance05	259140	229453	88%
instance06	645278	639268	99%
instance07	234578	221360	94%
instance08	696121	656055	94%
instance09	1493885	1739759	116%
instance10	496636	463693	93%
instance11	1256441	1359529	108%
instance12	2929729	3556505	121%

Dai valori percentuali si vede subito come l'algoritmo euristico faccia fatica a stare al passo con Gurobi nelle istanze iniziali, caratterizzate in media dall'avere meno oggetti, famiglie e risorse, mentre riesce a batterlo nelle istanze più grandi.

Nei grafici seguenti sono stati riportati i valori della colonna % mostrati nella tabella precedente, per ogni istanza. Sull'asse delle ordinate vengono indicati i valori percentuali del rapporto tra la soluzione trovata da VNDS e quella trovata da Gurobi per l'istanza corrispondente.

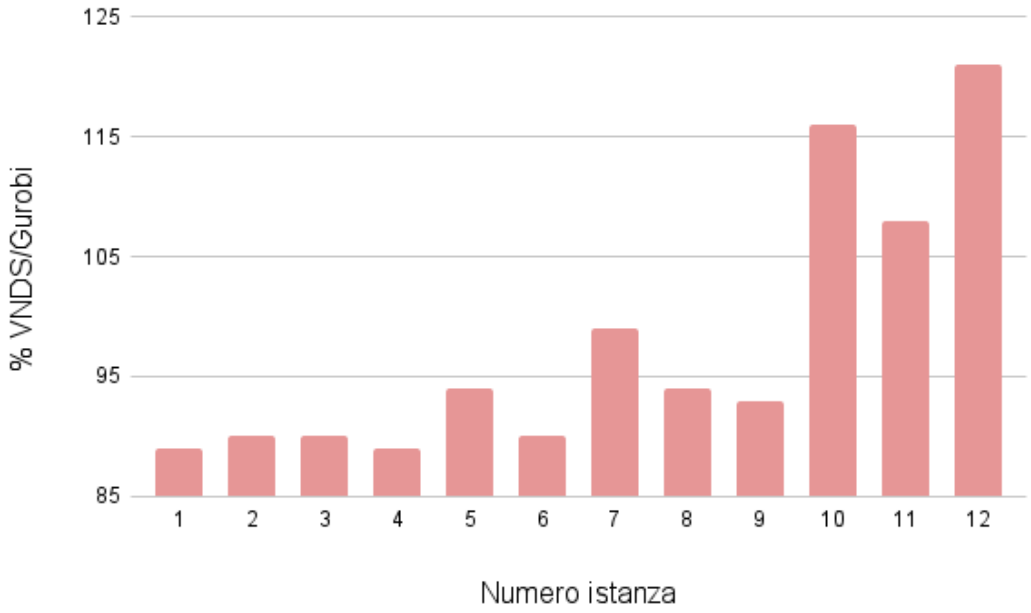
Sull'asse delle ascisse, invece: nel primo grafico vengono indicati i numeri di istanza ordinati per numero crescente di oggetti; nel secondo grafico sono ordinati per numero crescente di risorse.

Nel grafico sottostante si può notare l'aumento della percentuale all'aumentare del numero di oggetti dell'istanza, a supporto di quanto già detto sopra. Lo stesso grafico vale anche per il numero di famiglie, perché l'ordinamento delle istanze coincide.



Questo supporta ulteriormente l'idea che l'algoritmo VNDS sia più adatto a risolvere istanze più complesse, in cui il numero di oggetti e famiglie è maggiore.

Ordinando, invece, per numero di risorse crescente, risulta più difficile trovare una correlazione con la qualità della soluzione, perché non c'è differenza tra le prime tre istanze, che hanno 10 risorse, e tutte le altre, che ne hanno 30.



Di seguito viene riportato un confronto tra i risultati migliori ottenuti da due diverse configurazioni di VNDS, di cui la seconda corrisponde a quella dei risultati migliori (visti sopra).

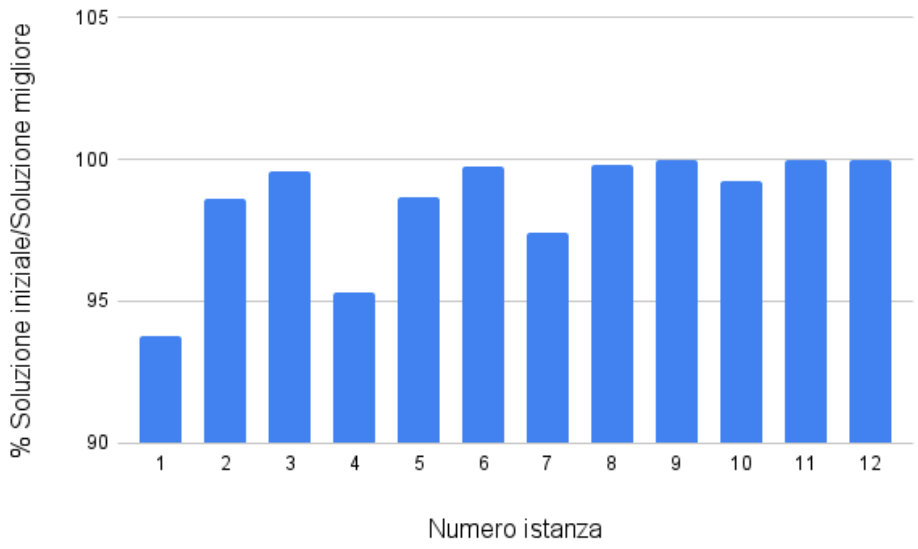
Istanza	Gurobi	VNDS (1)	VNDS (2)
	Obj	Obj	Obj
instance01	93103	80489	82543
instance02	272718	242924	246412
instance03	723224	652316	653127
instance04	85036	74194	76695
instance05	259140	228321	229453
instance06	645278	637803	639268
instance07	234578	216304	221360
instance08	696121	647438	656055
instance09	1493885	1739233	1739759
instance10	496636	460100	463693
instance11	1256441	1359529	1359529
instance12	2929729	3556505	3556505

Nella prima configurazione, cioè quella dei risultati nella colonna *VNDS (1)*, non sono stati usati i punteggi delle famiglie per la scelta pesata ma veniva data priorità alle famiglie in base al numero di volte in cui sono state aggiunte alla soluzione, cioè: nella fase di aggiunta di una famiglia il metodo preferiva quelle che erano state aggiunte meno volte, mentre nella fase di rimozione pesava di più quelle che erano state aggiunte più volte. Inoltre, in questa configurazione, sono stati mantenuti i parametri migliori ottenuti per il primo problema (*Sezione 6.3*), per vedere se fossero ancora validi.

Nella seconda configurazione, riportata in colonna VNDS (2), è stato introdotto il punteggio delle famiglie, calcolato in base ai valori della soluzione ottima del rilassamento continuo (*Sezione 5.1.1*), e sono stati modificati i parametri.

I risultati sono migliorati di poco tra le due. Soltanto nelle prime istanze (più piccole) si nota qualche differenza.

Nel grafico seguente viene mostrato, per ogni istanza, il rapporto tra il valore della funzione obiettivo nella soluzione migliore trovata e il valore della funzione obiettivo nella soluzione iniziale costruita dall'algoritmo. Sull'asse delle ascisse viene indicato il numero dell'istanza, mentre sull'asse delle ordinate viene indicato il valore percentuale del rapporto appena descritto.



I valori del grafico sono sempre superiori al 90%, il che significa che la soluzione costruita all'avvio del programma è già molto buona. Inoltre, essa viene trovata in poco tempo, perché la fase di costruzione della soluzione iniziale impiega, nelle istanze piccole, alcuni decimi di secondo; nella maggior parte delle istanze, alcuni secondi; nell'istanza più grande (la numero dodici), circa 80 secondi.

6.5.2 Analisi computazionale dell'algoritmo Kernel Search Ibrida

Per cercare di migliorare i risultati ottenuti da VNDS, è stata implementata questa variante della Kernel Search che non utilizza un risolutore esatto. Questo algoritmo richiede alcuni parametri che derivano dalla Kernel Search, ma vuole anche i parametri richiesti dal metodo VNDS, come già visto in *Sezione 5.2.5*.

I parametri che mediamente hanno ottenuto risultati migliori nella risoluzione delle istanze del secondo problema sono descritti in seguito. Per quanto riguarda quelli legati alla Kernel Search, invece di dare un valore specifico a `families_per_bucket`, è stata usata la quantità che corrisponde al 10% delle famiglie dell'istanza; il parametro `overlapping` non ha portato miglioramenti, per cui è stato impostato a `0`. Per quanto riguarda i parametri di VNDS, è stato ridotto il tempo di esecuzione limite, per poter analizzare più bucket, ed è stata impostata a un valore sufficientemente basso anche la terminazione per iterazioni senza miglioramento.

In tabella vengono riportati i parametri usati per VNDS:

-t	-k	i	-ni	-is	-mi	-fa	-ks	-r
100	2	0	200	10	100	30	20	80

I migliori risultati ottenuti sono riportati nella tabella seguente. Vengono comparati i valori delle funzioni obiettivo per le migliori soluzioni trovate da Gurobi e dalla Kernel Search ibrida. Nel caso della Kernel Search ibrida, viene riportato anche il tempo di esecuzione perché, a causa dei limiti di tempo e di iterazioni senza miglioramento per i sottoproblemi, l'algoritmo termina prima del limite globale di 600s. La colonna % indica il rapporto percentuale tra la soluzione trovata dalla Kernel Search e quella trovata da Gurobi. In verde sono evidenziate le istanze in cui l'algoritmo euristico è riuscito a ottenere un risultato migliore o uguale a Gurobi, cioè se il rapporto percentuale risulta maggiore o uguale a 100%.

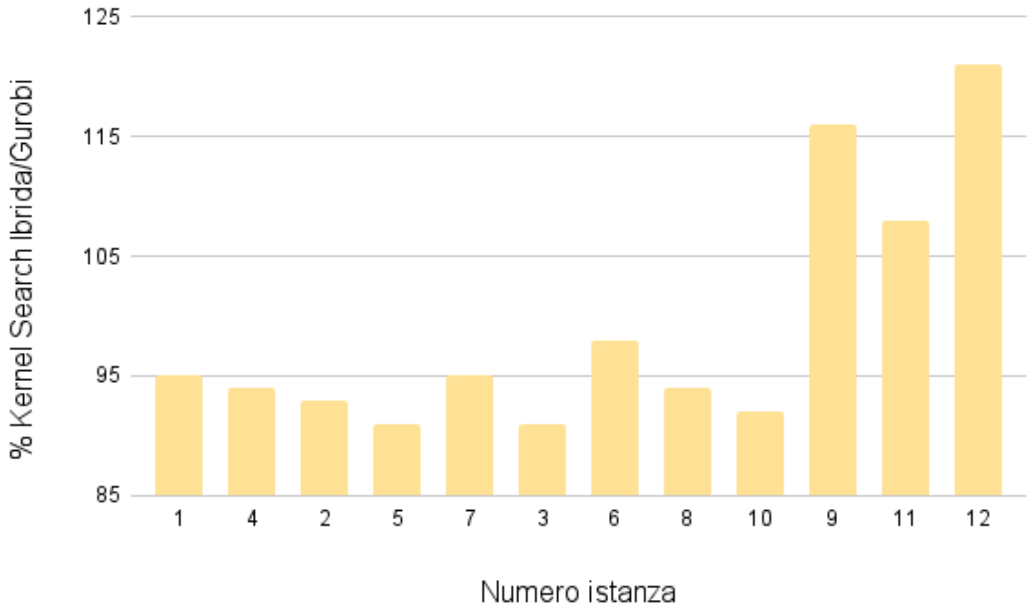
Istanza	Gurobi	Kernel Search ibrida		%
	Obj	Obj	T(s)	
instance01	93103	88898	365.2	95%
instance02	272718	255323	600.1	93%
instance03	723224	663260	405.7	91%
instance04	85036	80384	472.4	94%
instance05	259140	236138	601	91%
instance06	645278	636953	510.3	98%
instance07	234578	223145	600.6	95%
instance08	696121	655294	600.9	94%
instance09	1493885	1740078	424.6	116%
instance10	496636	460615	600.2	92%
instance11	1256441	1365507	602.3	108%
instance12	2929729	3552813	511.9	121%

Anche in questo caso l'algoritmo fatica nelle istanze più piccole, ma riesce comunque a ottenere risultati migliori di VNDS.

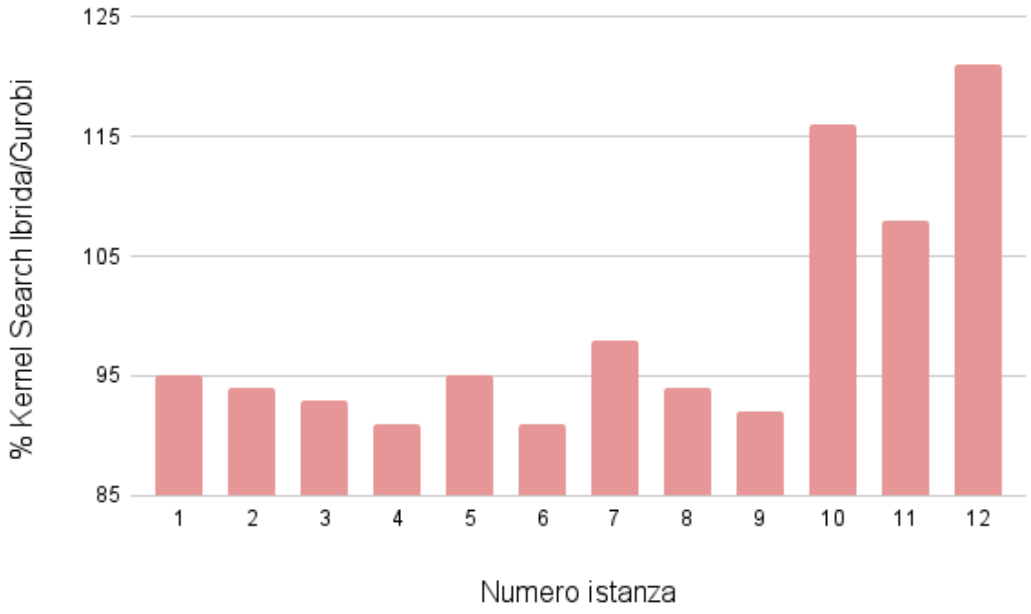
Nei grafici seguenti sono stati riportati i valori della colonna % mostrati nella tabella precedente, per ogni istanza. Sull'asse delle ordinate vengono indicati i valori percentuali del rapporto tra la soluzione trovata dalla Kernel Search ibrida e quella trovata da Gurobi per l'istanza corrispondente.

Sull'asse delle ascisse, invece: nel primo grafico vengono indicati i numeri di istanza ordinati per numero crescente di oggetti; nel secondo grafico sono ordinati per numero crescente di risorse.

Nel grafico seguente si vede che l'andamento è simile a quello di VNDS, con un miglioramento nelle istanze più piccole. Lo stesso grafico vale anche per il numero di famiglie, perché l'ordinamento delle istanze coincide.



Anche in questo caso, ordinando per numero di risorse crescente, non si vede un andamento particolare.



Nella seguente tabella viene mostrato un confronto con l'algoritmo precedente. Le colonne % indicano il rapporto tra il risultato di Gurobi e quello dell'algoritmo corrispondente alla colonna. In verde vengono evidenziate le percentuali che battono il risultato di Gurobi.

Istanza	Gurobi	VNDS		Kernel Search ibrida	
	Obj	Obj	%	Obj	%
instance01	93103	82543	88%	88898	95%
instance02	272718	246412	90%	255323	93%
instance03	723224	653127	90%	663260	91%
instance04	85036	76695	90%	80384	94%
instance05	259140	229453	88%	236138	91%
instance06	645278	639268	99%	636953	98%
instance07	234578	221360	94%	223145	95%
instance08	696121	656055	94%	655294	94%
instance09	1493885	1739759	116%	1740078	116%
instance10	496636	463693	93%	460615	92%
instance11	1256441	1359529	108%	1365507	108%
instance12	2929729	3556505	121%	3552813	121%

Si vede che la Kernel Search ibrida riesce a migliorare quasi tutte le soluzioni di VNDS, soprattutto nelle istanze più piccole. I casi in cui il valore della funzione obiettivo risulta leggermente più basso sono dovuti alla casualità delle selezioni all'interno dell'algoritmo, che possono portare a risultati leggermente diversi.

Per completezza è stato provato l'utilizzo dei parametri usati per la Kernel Search standard (spiegati nel capitolo successivo) anche in questo algoritmo.

Nel dettaglio, invece di usare le famiglie con variabile x_j associata che abbia valore 1 nell'ottimo del rilassamento continuo, è stato preso un numero di variabili pari al 15% della dimensione dell'istanza, seguendo comunque l'ordinamento spiegato in *Sezione 4.4.1*. Anche nel definire il numero di famiglie per bucket è stato usato il metodo della Kernel Search standard, con qualche famiglia in meno.

Vengono prese, quindi, un po' meno del 10% del numero delle famiglie dell'istanza, per quelle più piccole, mentre per le istanze più grandi viene usato circa il 6%.

I risultati ottenuti sono riportati nella tabella seguente.

Istanza	Kernel Search ibrida (1)	Kernel Search ibrida (2)
	Obj	Obj
instance01	83735	88898
instance02	256161	255323
instance03	644411	663260
instance04	75888	80384
instance05	242984	236138
instance06	609109	636953
instance07	217280	223145
instance08	630474	655294
instance09	1647542	1740078
instance10	458750	460615
instance11	1316737	1365507
instance12	3198382	3552813

Si nota subito che i risultati ottenuti con i parametri della Kernel Search standard sono peggiori rispetto a quelli ottenuti con i parametri specifici per la Kernel Search ibrida.

6.5.3 Analisi computazionale dell'algoritmo Kernel Search

L'implementazione di questo metodo ha ottenuto risultati migliori rispetto a VNDS nelle istanze più piccole, in alcuni casi anche migliori di Gurobi.

Anche in questo caso, il numero di famiglie per bucket viene impostato in base al numero delle famiglie dell'istanza. Tuttavia, invece di usare un valore percentuale, è stata definita un'equazione lineare che avesse come risultato circa il 10% delle famiglie per bucket nell'istanza più piccola (5 famiglie nei risultati migliori) e circa il 6% delle famiglie per bucket nell'istanza più grande (60 famiglie nei risultati migliori). La stessa cosa è stata fatta anche per la dimensione del kernel, che viene impostata al 15% delle famiglie dell'istanza. Inoltre, il tempo limite per la risoluzione di ogni sottoproblema è stato impostato a 350 secondi.

I migliori risultati ottenuti sono riportati nella tabella seguente. Vengono comparati i valori delle funzioni obiettivo per le migliori soluzioni trovate da Gurobi e dalla Kernel Search. La colonna % indica il rapporto percentuale tra la soluzione trovata dalla Kernel Search e quella trovata da Gurobi. In verde sono evidenziate le istanze in cui l'algoritmo euristico è riuscito a ottenere un risultato migliore o uguale a Gurobi, cioè se il rapporto percentuale risulta maggiore o uguale a 100%. Il tempo di esecuzione è stato omesso perché è sempre stato raggiunto il limite massimo (3600s per Gurobi, 600s per l'algoritmo euristico).

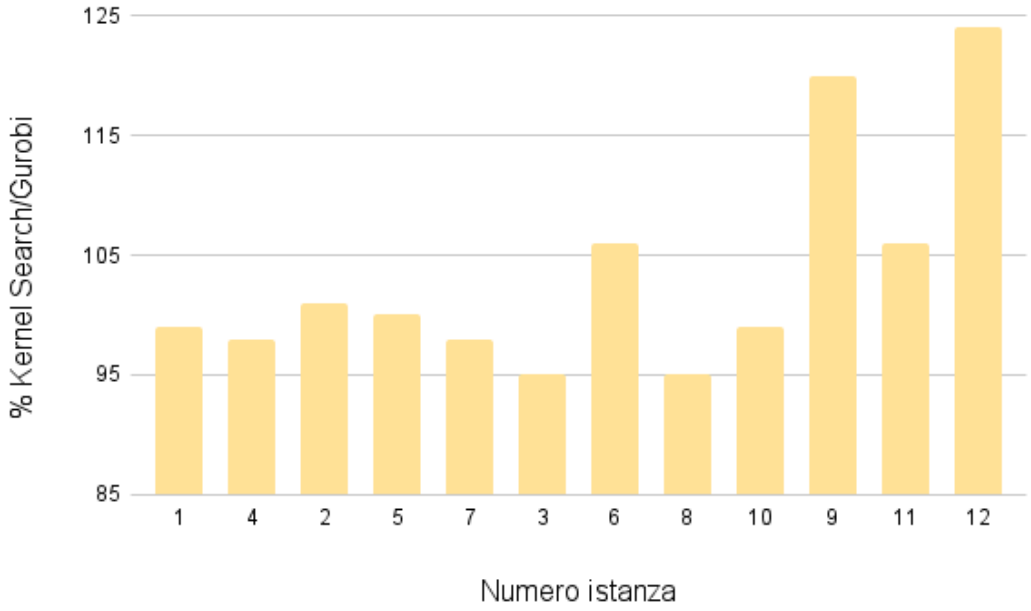
Istanza	Gurobi	Kernel Search	%
	Obj	Obj	
instance01	93103	92375	99%
instance02	272718	278043	101%
instance03	723224	688232	95%
instance04	85036	83545	98%
instance05	259140	259988	100%
instance06	645278	686068	106%
instance07	234578	232145	98%
instance08	696121	666745	95%
instance09	1493885	1796071	120%
instance10	496636	491845	99%
instance11	1256441	1335773	106%
instance12	2929729	3648698	124%

L'algoritmo Kernel Search riesce a ottenere risultati ottimi e a battere Gurobi in metà delle istanze.

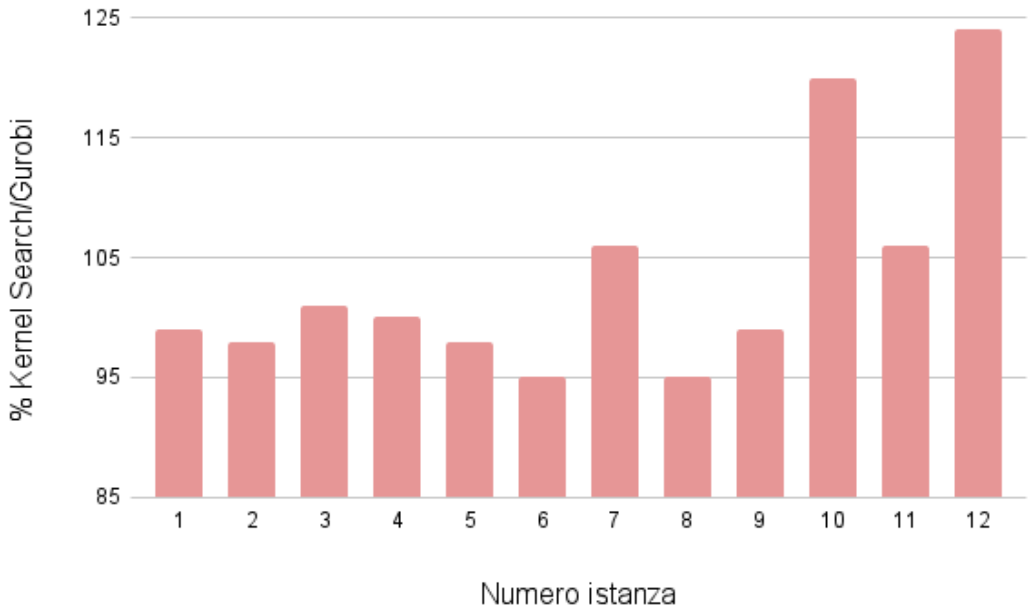
Nei grafici seguenti sono stati riportati i valori della colonna % mostrati nella tabella precedente, per ogni istanza. Sull'asse delle ordinate vengono indicati i valori percentuali del rapporto tra la soluzione trovata dalla Kernel Search e quella trovata da Gurobi per l'istanza corrispondente.

Sull'asse delle ascisse, invece: nel primo grafico vengono indicati i numeri di istanza ordinati per numero crescente di oggetti; nel secondo grafico sono ordinati per numero crescente di risorse.

In questo caso, osservando il seguente grafico, si vede che l'andamento risulta più equilibrato. Infatti, rispetto ai grafici di VNDS e Kernel Search ibrida, c'è un aumento generale di percentuale che risulta più evidente nelle istanze piccole. Lo stesso grafico vale anche per il numero di famiglie, perché l'ordinamento delle istanze coincide.



Ordinando le istanze per numero di risorse crescente, anche in questo caso non compare una differenza evidente tra le prime tre istanze e tutte le altre.



Si possono vedere nella tabella seguente prestazioni migliori rispetto alla Kernel Search ibrida, infatti quest'ultima viene battuto in quasi tutte le istanze.

Istanza	Kernel Search ibrida	Kernel Search	%
	Obj	Obj	
instance01	88898	92375	103%
instance02	255323	278043	108%
instance03	663260	688232	103%
instance04	80384	83545	103%
instance05	236138	259988	110%
instance06	636953	686068	107%
instance07	223145	232145	104%
instance08	655294	666745	101%
instance09	1740078	1796071	103%
instance10	460615	491845	106%
instance11	1365507	1335773	97%
instance12	3552813	3648698	102%

Per motivare i parametri specificati, nella prossima tabella viene mostrata la differenza tra due versioni dell'algoritmo.

Istanza	Gurobi	Kernel Search (1)		Kernel Search (2)	
	Obj	Obj	%	Obj	%
instance01	93103	92320	99%	92375	99%
instance02	272718	264859	97%	278043	101%
instance03	723224	691726	95%	688232	95%
instance04	85036	83160	97%	83545	98%
instance05	259140	248812	96%	259988	100%
instance06	645278	670224	103%	686068	106%
instance07	234578	232351	99%	232145	98%
instance08	696121	682934	98%	666745	95%
instance09	1493885	1782383	119%	1796071	120%
instance10	496636	480003	96%	491845	99%
instance11	1256441	1413173	112%	1335773	106%
instance12	2929729	3552813	121%	3648698	124%

Nella prima versione vengono messe nel kernel iniziale le famiglie con variabile x_j associata che assume valore 1 nell'ottimo del problema rilassato (come è stato fatto nella Kernel Search

ibrida), mentre la seconda è la versione finale, con dimensione del kernel pari al 15% del numero di famiglie dell'istanza.

La scelta di variare la dimensione del kernel e dei bucket in base alla dimensione dell'istanza e di aumentare il tempo limite di esecuzione per i sottoproblemi è stata presa perché si è osservato che le soluzioni migliori tendevano a essere nei primi bucket. Utilizzando questo approccio l'algoritmo riesce a focalizzarsi proprio su questi e a selezionare soltanto le famiglie migliori.

Nella tabella sottostante è possibile vedere il confronto della risoluzione della dodicesima istanza (quella più grande) tra *Kernel Search (1)* e *Kernel Search (2)*. Nel primo, la dimensione del kernel è di 645 (che è il numero famiglie con variabile x_j a valore 1 nel rilassato) e il tempo limite di risoluzione del sottoproblema è di 100 secondi; nel secondo, la dimensione del kernel è al 15% delle famiglie (cioè 150) e il tempo limite per sottoproblema è di 350 secondi.

Nella colonna *Sottoproblema* viene indicato il numero del bucket aggiunto al kernel per quel sottoproblema, mentre nelle colonne *Kernel Search (1)* e *Kernel Search (2)* sono riportati i valori delle funzioni obiettivo per i due algoritmi. Nella prima riga è indicato *kernel* perché si tratta del sottoproblema che comprende solo le famiglie del kernel iniziale.

Sottoproblema	Kernel Search (1)	Kernel Search (2)
	Obj	Obj
kernel	2389527	886370
bucket01	3552813	1236372
bucket02	3552813	1582573
bucket03	3552813	1934591
bucket04	3552813	2278890
bucket05		2637792
bucket06		2995876
bucket07		3346213
bucket08		3648698
bucket09		3648698

Il primo caso trova la soluzione migliore nel primo bucket, poi non riesce più a trovare un miglioramento. Il secondo caso, invece, risolve i primi bucket all'ottimo in pochi secondi, poi trova la soluzione migliore nel penultimo bucket.

6.5.4 Analisi computazionale dell'algoritmo Sequential VNDS Kernel Search

Quest'ultimo algoritmo riesce a ottenere il meglio da entrambi i suoi componenti, riuscendo a battere Gurobi in cinque istanze su dodici e migliorando i risultati della Kernel Search standard in otto istanze.

I parametri che hanno ottenuto i risultati migliori sono:

- 60 secondi di tempo limite per l'esecuzione dell'algoritmo Kernel Search ibrida

- 500 secondi di tempo limite per ogni sottoproblema della Kernel Search standard
- Dimensione iniziale del kernel pari al 15% delle famiglie dell'istanza
- Dimensione dei bucket leggermente minore di quella utilizzata per la Kernel Search standard
- Limite di iterazioni senza miglioramento per i bucket della Kernel Search ibrida pari a 150

I restanti parametri relativi a VNDS, cioè `-k`, `-is`, `-mi`, `-fa`, `-ks`, `-r` sono stati impostati con gli stessi valori utilizzati per la Kernel Search ibrida (*Sezione 6.5.2*).

I migliori risultati ottenuti sono riportati nella tabella seguente. Vengono comparati i valori delle funzioni obiettivo per le migliori soluzioni trovate da Gurobi e dal Sequential VNDS Kernel Search. La colonna % indica il rapporto percentuale tra la soluzione trovata dalla Kernel Search e quella trovata da Gurobi. In verde sono evidenziate le istanze in cui l'algoritmo euristico è riuscito a ottenere un risultato migliore o uguale a Gurobi, cioè se il rapporto percentuale risulta maggiore o uguale a 100%.

Il tempo di esecuzione è stato omesso perché è sempre stato raggiunto il limite massimo (3600s per Gurobi, 600s per l'algoritmo euristico).

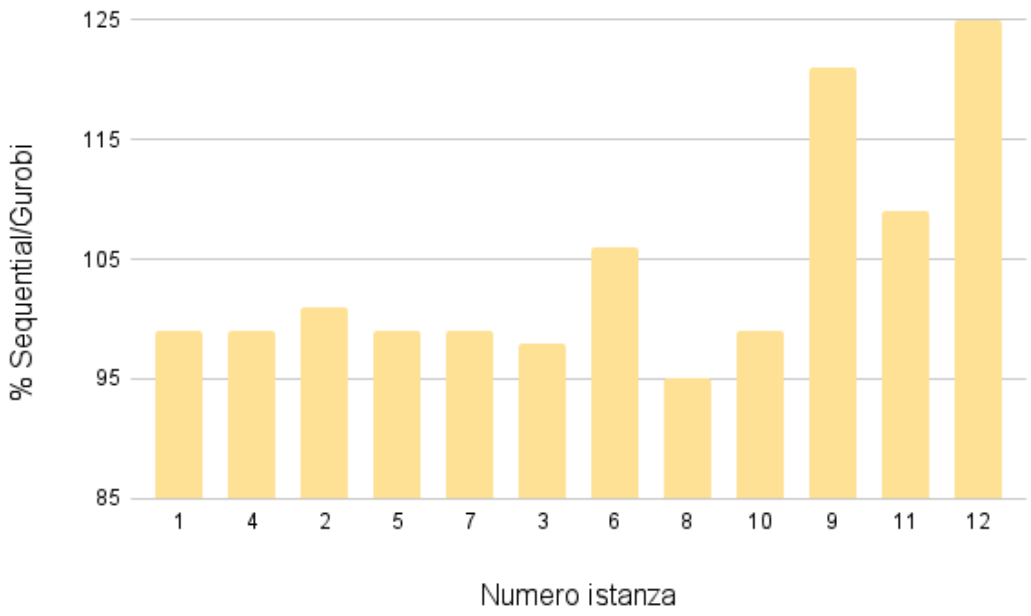
Istanza	Gurobi	Sequential	%
	Obj	Obj	
instance01	93103	93041	99%
instance02	272718	276500	101%
instance03	723224	712635	98%
instance04	85036	84231	99%
instance05	259140	258883	99%
instance06	645278	684413	106%
instance07	234578	232779	99%
instance08	696121	663129	95%
instance09	1493885	1812022	121%
instance10	496636	492780	99%
instance11	1256441	1374921	109%
instance12	2929729	3684472	125%

Questo algoritmo riesce a battere Gurobi in cinque istanze su dodici, avvicinandosi di molto nelle altre.

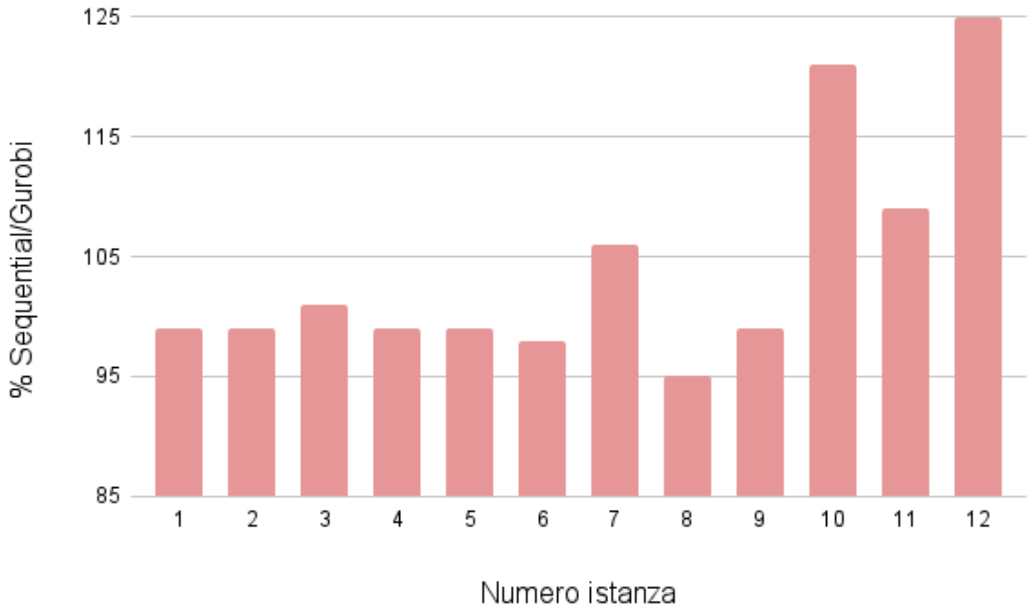
Nei grafici seguenti sono stati riportati i valori della colonna % mostrati nella tabella precedente, per ogni istanza. Sull'asse delle ordinate vengono indicati i valori percentuali del rapporto tra la soluzione trovata dal Sequential VNDS Kernel Search e quella trovata da Gurobi per l'istanza corrispondente.

Sull'asse delle ascisse, invece: nel primo grafico vengono indicati i numeri di istanza ordinati per numero crescente di oggetti; nel secondo grafico sono ordinati per numero crescente di risorse.

Osservando nel grafico seguente l'andamento della percentuale al crescere del numero di oggetti, si vede che mantiene l'equilibrio evidenziato anche nella Kernel Search. Lo stesso grafico vale anche per il numero di famiglie, perché l'ordinamento delle istanze coincide.



Anche in questo caso, ordinando per numero di risorse crescente, non si vede un andamento particolare.



La tabella successiva mostra un paragone con i risultati della Kernel Search standard, la colonna % indica il rapporto percentuale tra il risultato dell'algoritmo Sequential VNDS Kernel e quello della Kernel Search. Si può notare che soltanto in quattro istanze questo algoritmo ottiene un risultato peggiore, ma di molto poco.

Istanza	Kernel Search	Sequential	%
	Obj	Obj	
instance01	92375	93041	100%
instance02	278043	276500	99%
instance03	688232	712635	103%
instance04	83545	84231	100%
instance05	259988	258883	99%
instance06	686068	684413	99%
instance07	232145	232779	100%
instance08	666745	663129	99%
instance09	1796071	1812022	100%
instance10	491845	492780	100%
instance11	1335773	1374921	102%
instance12	3648698	3684472	100%

Per la selezione dei valori migliori per i parametri (tuning) sono stati effettuati diversi tentativi. Nella prossima tabella viene analizzato il confronto tra diverse configurazioni di Sequential VNDS Kernel provate.

Istanza	Sequential (1)	Sequential (2)	Sequential (3)	Sequential (4)
	Obj	Obj	Obj	Obj
instance01	92616	92320	92375	93041
instance02	273901	264859	278043	276500
instance03	717593	691726	688232	712635
instance04	84423	83160	83545	84231
instance05	258253	248812	259988	258883
instance06	685055	670224	686068	684413
instance07	234226	232351	232145	232779
instance08	659565	682934	666745	663129
instance09	1794639	1782383	1796071	1812022
instance10	491880	480003	491845	492780
instance11	1326745	1413173	1335773	1374921
instance12	3652847	3552813	3648698	3684472

La colonna *Sequential (4)* corrisponde alla configurazione migliore, di cui i parametri sono stati indicati sopra. Rispetto a questa, le altre presentano delle differenze.

Nella prima è stato dato un limite di tempo di 350 secondi ai sottoproblemi della Kernel Search e sono state usate 5 famiglie per bucket in entrambi gli algoritmi.

Nella seconda è stato dato un limite di tempo alla Kernel Search ibrida pari a 100 secondi, mentre ai sottoproblemi della Kernel Search 400 secondi.

Nella terza il limite di iterazioni senza miglioramento per i bucket della Kernel Search ibrida è stato impostato a 50.

I parametri che sono stati omessi in queste differenze sono stati impostati con gli stessi valori della configurazione migliore.

Capitolo 7: Conclusioni

In questa tesi viene affrontato il problema dello zaino multidimensionale con penalità family split (MMdKFSP), il quale è caratterizzato da un insieme di oggetti, divisi in famiglie, a cui può essere assegnato uno zaino, con l'obiettivo di inserire negli zaini più famiglie possibili, ottenendo un profitto per ciascuna di quelle allocate. Le famiglie non possono essere inserite parzialmente, ma possono avere oggetti in zaini diversi e l'utilizzo di più zaini da parte di una famiglia comporta una penalità.

Vengono viste due varianti del problema, che si differenziano per come viene gestita la penalità per le famiglie che abbiano subito delle divisioni (split): la prima variante, definita da Mancini et al. (2021 *[1]*), prevede che venga applicata una penalità per ogni famiglia che abbia subito almeno uno split; la seconda variante, che viene proposta per la prima volta in questa tesi, prevede che venga applicata una penalità per ognuno degli split subiti da una famiglia, moltiplicando la penalità di ognuna per il numero di split ottenuti dalla stessa.

È stato mostrato, tramite un applicazione del problema in un contesto reale, che in diversi casi è fondamentale considerare il numero di split effettuati e non dare una sola penalità nel momento in cui avviene una divisione.

Dopo la definizione delle due varianti, vengono proposti e analizzati degli algoritmi euristici per la loro risoluzione. Il primo algoritmo è un'implementazione di VNDS, che inizialmente viene usato per risolvere la prima variante di MMdKFSP, utilizzando le istanze prese dalla letteratura.

Successivamente viene modificato per risolvere la seconda variante di MMdKFSP, introducendo anche delle nuove istanze benchmark più complesse, perché quelle precedenti venivano risolte molto velocemente da Gurobi. Vengono, inoltre, introdotti ulteriori algoritmi per risolvere la seconda variante, basandosi sull'implementazione di VNDS e sulla Kernel Search.

Tutti i risultati vengono confrontati con quelli ottenuti dal risolutore MIP commerciale Gurobi, dando un limite di tempo di un'ora a quest'ultimo e di dieci minuti agli algoritmi, per ogni istanza. Il confronto avviene basandosi sul valore della funzione obiettivo ottenuto.

I risultati ottenuti hanno inizialmente mostrato che VNDS ottiene buoni valori rispetto a quelli di Gurobi, nel poco tempo a disposizione. È stato mostrato che il metodo di costruzione della soluzione iniziale fornisce un buon punto di partenza per l'algoritmo.

Per migliorare ulteriormente la qualità delle soluzioni, è stata introdotta la scomposizione in sottoproblemi, come definita nella Kernel Search, per poter concentrare l'algoritmo sulle famiglie più promettenti. Questo ha portato all'implementazione della Kernel Search ibrida.

È stata poi introdotta un'implementazione della Kernel Search standard, che utilizza il risolutore Gurobi per risolvere i sottoproblemi, con lo scopo di avere un ulteriore punto di riferimento.

Infine, è stata introdotta una combinazione degli algoritmi: la Sequential VNDS Kernel Search. Essa utilizza sia la Kernel Search ibridizzata con VNDS che quella standard per poter ottenere il meglio da entrambe.

È stato mostrato che i risultati ottenuti da quest'ultimo algoritmo riescono a migliorare i risultati di Gurobi in cinque istanze su dodici e si avvicinano di molto nelle altre. Paragonato alla Kernel Search, si mostra che i risultati sono migliori in otto istanze, ma comunque molto vicini anche nelle quattro rimanenti.

Capitolo 8: Bibliografia

[1] Simona Mancini, Michele Ciavotta, e Carlo Meloni, «The Multiple Multidimensional Knapsack with Family-Split Penalties», *European Journal of Operational Research*, vol. 289, fasc. 3, pp. 987–998, mar. 2021, doi: 10.1016/j.ejor.2019.07.052.

[2] Simona Mancini, Michele Ciavotta, e Carlo Meloni, «A decomposition approach for multidimensional knapsacks with family-split penalties», *International Transactions in Operational Research*, vol. 31, fasc. 4, pp. 2247–2271, set. 2022, doi: 10.1111/itor.13207.

[3] Enrico Angelelli, Renata Mansini, e M. Grazia Speranza, «Kernel search: A general heuristic for the multi-dimensional knapsack problem». 6 febbraio 2010. doi: 10.116/j.cor.2010.02.002.

[4] Silvano Martello e Paolo Toth, *Knapsack Problems: Algorithms and Computer Implementations*.

[5] Hans Kellerer, Ulrich Pferschy, e David Pisinger, *Knapsack Problems*.

[6] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, e Silvano Martello, «Knapsack problems — An overview of recent advances. Part I: Single knapsack problems», *Computers & Operations Research*, vol. 143, lug. 2022, doi: 10.1016/j.cor.2021.105692.

[7] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, e Silvano Martello, «Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems», *Computers & Operations Research*, vol. 143, lug. 2022, doi: 10.1016/j.cor.2021.105693.

[8] Lin Chen e Guochuan Zhang, «Packing Groups of Items into Multiple Knapsacks», *Symposium on Theoretical Aspects of Computer Science (STACS 2016)*, vol. 33, feb. 2016, doi: 10.4230/LIPIcs.STACS.2016.28.

[9] Chandra Chekuri e Sanjeev Khanna, «A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem». 3 febbraio 2006. doi: 10.1137/S0097539700382820.

[10] Mauro Dell'Amico, Maxence Delorme, Manuel Iori, e Silvano Martello, «Mathematical models and decomposition methods for the multiple knapsack problem», *European Journal of Operational Research*, vol. 274, fasc. 3, mag. 2019, doi: 10.1016/j.ejor.2018.10.043.

[11] C. E. Ferreira, A. Martin, e R. Weismantel, «Solving Multiple Knapsack Problems by Cutting Planes», *SIAM Journal on Discrete Mathematics*, vol. 6, fasc. 3, 1996, doi: 10.1137/S1052623493254455.

[12] Arnaud Fréville, «The multidimensional 0-1 knapsack problem: An overview», *European Journal of Operational Research*, vol. 155, fasc. 1, mag. 2004, doi: 10.1016/S0377-2217(03)00274-1.

[13] David Pisinger, «An exact algorithm for large multiple knapsack problems», *European Journal of Operational Research*, vol. 114, fasc. 3, mag. 1999, doi: 10.1016/S0377-2217(98)00120-9.

[14] Silvano Martello e Paolo Toth, «An Exact Algorithm for the Two-Constraint 0-1 Knapsack Problem», *Operations Research*, vol. 51, fasc. 5, ott. 2003, doi: 10.1287/opre.51.5.826.16757.

[15] Seiji Kataoka e Takeo Yamada, «Upper and lower bounding procedures for the multiple knapsack assignment problem», *European Journal of Operational Research*, vol. 237, fasc. 2, set. 2014, doi: 10.1016/j.ejor.2014.02.014.

[16] V. C. Li e G. L. Curry, «Solving multidimensional knapsack problems with generalized upper bound constraints using critical event tabu search», *Computers & Operations Research*, vol. 32, fasc. 4, mag. 2005, doi: 10.1016/j.cor.2003.08.021.

[17] Takeo Yamada e Takahiro Takeoka, «An exact algorithm for the fixed-charge multiple knapsack problem», *European Journal of Operational Research*, vol. 192, fasc. 2, 2009, [Online]. Disponibile su: https://econpapers.repec.org/RePEc:eee:ejores:v:192:y:2009:i:2:p:700-705

[18] Hadas Shachnai e Tami Tamir, «Polynomial time approximation schemes for class-constrained packing problems», *Journal of Scheduling*, vol. 4, fasc. 6, ott. 2001, doi: 10.1002/jos.86.

[19] Alberto Ceselli e Giovanni Righini, «An optimization algorithm for a penalized knapsack problem», *Operations Research Letters*, vol. 34, fasc. 4, lug. 2006, doi: 10.1016/j.orl.2005.06.001.

[20] Thibaut Lust e Jacques Teghem, «The multiobjective multidimensional knapsack problem: a survey and a new approach», *International Transactions in Operational Research*, vol. 19, fasc. 4, feb. 2012, doi: 10.1111/j.1475-3995.2011.00840.x.

[21] Taha Ghasemi e Mohammadreza Razzazi, «Development of core to solve the multidimensional multiple-choice knapsack problem», *Computers & Industrial Engineering*, vol. 60, fasc. 2, mar. 2011, doi: 10.1016/j.cie.2010.12.001.

[22] Renata Mansini e Roberto Zanotti, «A core-based exact algorithm for the multidimensional multiple choice knapsack problem», *INFORMS Journal on Computing*, vol. 32, fasc. 4, mar. 2020, doi: doi.org/10.1287/ijoc.2019.0909.

[23] Ron Adany *et al.*, «All-Or-Nothing Generalized Assignment with Application to Scheduling Advertising Campaigns», *ACM Transactions on Algorithms*, vol. 12, fasc. 3, apr. 2016, doi: 10.1145/2843944.

[24] George B. Dantzig, Alex Orden, e Philip Wolfe, «The generalized simplex method for minimizing a linear form under linear inequality restraints», *Pacific Journal of Mathematics*, vol. 5, fasc. 2, ott. 1955.

[25] Pierre Hansen, Nenad Mladenović, Raca Todosijević, e Saïd Hanafi, «Variable neighborhood search: basics and variants», *EURO Journal on Computational Optimization*, vol. 5, fasc. 3, set. 2017, doi: 10.1007/s13675-016-0075-x.

[26] Pierre Hansen, Nenad Mladenović, e Dionisio Perez-Britos, «Variable Neighborhood Decomposition Search», *Journal of Heuristics*, vol. 7, pp. 335–350, lug. 2001.

[27] Matteo Fischietti e Andrea Lodi, «Local branching», *Mathematical Programming*, vol. 98, pp. 23–47, mar. 2003.

[28] «Exploring relaxation induced neighborhoods to improve MIP solutions», *Mathematical Programming*, vol. 102, pp. 71–90, mag. 2004.

[29] «Proximity search for 0-1 mixed-integer convex programming», *Journal of Heuristics*, vol. 20, pp. 709–731, nov. 2014.

[30] J E Beasley, «OR-Library». [Online]. Disponibile su: https://people.brunel.ac.uk/~mastjjb/jeb/info.html

[31] miciav, «instance-generator». [Online]. Disponibile su: https://github.com/miciav/instance-generator

[32] G. Guastaroba, M. Savelsbergh, e M.G. Speranza, «Adaptive Kernel Search: A heuristic for solving Mixed Integer linear Programs», *European Journal of Operational Research*, vol. 263, fasc. 3, dic. 2017, doi: 10.1016/j.ejor.2017.06.005.

[33] Leonardo Lamanna, Renata Mansini, e Roberto Zanotti, «A two-phase kernel search variant for the multidimensional multiple-choice knapsack problem», *European Journal of Operational Research*, vol. 297, fasc. 1, feb. 2022, doi: 10.1016/j.ejor.2021.05.007.

[34] G.Guastaroba e M.G. Speranza, «Kernel Search: An application to the index tracking problem», *European Journal of Operational Research*, vol. 217, fasc. 1, feb. 2012, doi: 10.1016/j.ejor.2011.09.004.

[35] G. Guastaroba e M.G. Speranza, «Kernel search for the capacitated facility location problem», *Journal of Heuristics*, vol. 18, pp. 877–917, set. 2012.

[36] Enrico Angelelli, Renata Mansini, e M. Grazia Speranza, «Kernel Search: a new heuristic framework for portfolio selection», *Computational Optimization and Applications*, vol. 51, pp. 345–361, mar. 2010.

[37] Claudia Archetti, Gianfranco Guastaroba, Diana L. Huerta-Muñoz, e M. Grazia Speranza, «A kernel search heuristic for the multivehicle inventory routing problem», *International Transactions in Operational Research*, vol. 28, fasc. 6, feb. 2021, doi: 10.1111/itor.12945.

[38] Harvey M. Salkin e Cornelis A. De Kluyver, «The knapsack problem: A survey», *Naval Research Logistics Quarterly*, vol. 22, fasc. 1, mar. 1975, doi: 10.1002/nav.3800220110.

[39] Krzysztof Dudziński e Stanisław Walukiewicz, «Exact methods for the knapsack problem and its generalizations», *European Journal of Operational Research*, vol. 28, fasc. 1, gen. 1987, doi: 10.1016/0377-2217(87)90165-2.

[40] Min Kong, Peng Tian, e Yucheng Kao, «A new ant colony optimization algorithm for the multidimensional Knapsack problem», *Computers & Operations Research*, vol. 35, fasc. 8, ago. 2008, doi: 10.1016/j.cor.2006.12.029.

[41] Thibaut Lust e Jacques Teghem, «The multiobjective multidimensional knapsack problem: a survey and a new approach», *International Transactions in Operational Research*, vol. 19, fasc. 4, feb. 2012, doi: 10.1111/j.1475-3995.2011.00840.x.

[42] R. Parra-Hernandez e N.J. Dimopoulos, «A new heuristic for solving the multichoice multidimensional knapsack problem», *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 35, fasc. 5, ago. 2005, doi: 10.1109/TSMCA.2005.851140.

[43] Yalçın Akçay, Haijun Li, e Susan H. Xu, «Greedy algorithm for the general multidimensional knapsack problem», *Annals of Operations Research*, vol. 150, pp. 17–29, dic. 2006.

[44] M. Hifi, M. Michrafy, e A. Sbihi, «Heuristic algorithms for the multiple-choice multidimensional knapsack problem», *Journal of the Operational Research Society*, vol. 55, fasc. 12, dic. 2017, doi: 10.1057/palgrave.jors.2601796.

[45] N. Mladenović e P. Hansen, «Variable neighborhood search», *Computers & Operations Research*, vol. 24, fasc. 11, nov. 1997, doi: 10.1016/S0305-0548(97)00031-2.

[46] Pierre Hansen e Nenad Mladenović, «Variable neighborhood search: Principles and applications», *European Journal of Operational Research*, vol. 130, fasc. 3, mag. 2001, doi: 10.1016/S0377-2217(00)00100-4.

[47] Bassem Jarboui, Houda Derbel, Saïd Hanafi, e Nenad Mladenović, «Variable neighborhood search for location routing», *Computers & Operations Research*, vol. 40, fasc. 1, gen. 2013, doi: 10.1016/j.cor.2012.05.009.

[48] M.A. Lejeune, «A variable neighborhood decomposition search method for supply chain management planning problems», *European Journal of Operational Research*, vol. 175, fasc. 2, dic. 2006, doi: 10.1016/j.ejor.2005.05.021.

[49] Xiangjing Lai, Jin-Kao Hao, Zhang-Hua Fu, e Dong Yue, «Neighborhood decomposition-driven variable neighborhood search for capacitated clustering», *Computers & Operations Research*, vol. 134, ott. 2021, doi: 10.1016/j.cor.2021.105362.

[50] M.-C. Costa, F.-R. Monclar, e M. Zrikem, «Variable neighborhood decomposition search for the optimization of power plant cable layout», *Journal of Intelligent Manufacturing*, vol. 13, pp. 353–365, ott. 2002.

[51] Abraham Duarte, Nenad Mladenovic, Jesús Sánchez-Oro, e Raca Todosijević, «Variable Neighborhood Descent». 20 maggio 2022. doi: 10.1007/978-3-319-07124-4_9.

[52] Jie Gao, Linyan Sun, e Mitsuo Gen, «A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems», *Computers & Operations Research*, vol. 35, fasc. 9, set. 2008, doi: 10.1016/j.cor.2007.01.001.

[53] Ping Chen, Hou-kuan Huang, e Xing-Ye Dong, «Iterated variable neighborhood descent algorithm for the capacitated vehicle routing problem», *Expert Systems with Applications*, vol. 37, fasc. 2, mar. 2010, doi: 10.1016/j.eswa.2009.06.047.