****

Politecnico di Torino

Corso di Laurea in Ingegneria Gestionale L-8

A.a. 2022/2023

Sessione di Laurea Ottobre 2023

**Metodi di risoluzione per il min-Knapsack Problem con vincoli di compattezza**

|  |
| --- |
| **Relatore:**  Prof. Rosario Scatamacchia |
|  |
|  |
|  | **Candidata:** |
|  | Ludovica Marino |

**Indice**

[**Introduzione** 2](#_Toc145697280)

[Introduzione alla Ricerca Operativa 2](#_Toc145697281)

[Introduzione al problema 5](#_Toc145697282)

[**Modello lineare** 9](#_Toc145697283)

[Introduzione alla programmazione lineare 9](#_Toc145697284)

[Modello matematico 9](#_Toc145697285)

[Generazione delle istanze 12](#_Toc145697286)

[**Algoritmo euristico** 15](#_Toc145697287)

[Teoria dell’euristica greedy 15](#_Toc145697288)

[Algoritmo sviluppato 16](#_Toc145697289)

[**Analisi computazionale** 20](#_Toc145697290)

[**Conclusioni** 29](#_Toc145697291)

[**Bibliografia** 31](#_Toc145697292)

[**Appendice** 32](#_Toc145697293)

# **Introduzione**

## Introduzione alla Ricerca Operativa

La Ricerca Operativa è una disciplina il cui concetto non può essere definito in modo univoco. Viene infatti considerata sia come l’insieme di problemi, metodologie e soluzioni, sia come un processo applicativo caratterizzato da un approccio sistematico.

Lo scopo, ed in ugual modo il ruolo fondamentale di questa tipologia di scienza, è definito nell’attività di supporto che la ricerca operativa svolge, aiutando a prendere decisioni scientificamente fondate per risolvere problemi complessi del mondo reale. L’approccio teorico è infatti finalizzato ad essere applicato ad un modello pratico, ovvero una forma di rappresentazione semplificata di un sistema complesso del mondo reale. La rappresentazione assume la forma di equazioni, disequazioni e funzioni matematiche. Il principale obiettivo della creazione di un modello matematico è identificare soluzioni ottimali per problemi che coinvolgono il sistema reale, evitando di interagire direttamente con quest’ultimo.

Il processo di modellazione comprende tre fasi chiave, che possono essere schematizzate come in Fig. 0.

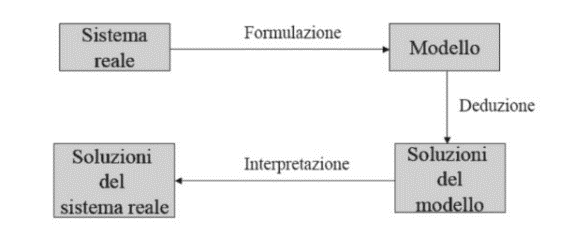


Fig. 0: Modellazione di un sistema lineare

La prima fase, detta formulazione, implica la creazione del modello e richiede creatività ed esperienza per selezionare gli aspetti rilevanti del sistema da rappresentare. Questa fase è soggettiva poiché non esiste un’unica formulazione “corretta” universale. Nella fase di deduzione vengono sviluppate le soluzioni del modello mediante l’applicazione di tecniche risolutive basate su algoritmi, adatti alla complessità del modello stesso. Questa fase garantisce quindi precisione e oggettività nell’ottenimento delle soluzioni, in quanto caratterizzata da una rigorosa applicazione matematica. Nella fase finale di interpretazione, si richiede la partecipazione umana per tradurre le soluzioni del modello in azioni pratiche per il sistema reale. Questo ultimo passaggio richiede particolare attenzione, poiché aspetti trascurati nelle fasi precedenti possono emergere come rilevanti, rendendo necessarie alcune correzioni o adattamenti alle soluzioni proposte dal modello.

Questi modelli sono quindi strumenti potenti per risolvere problemi complessi, che richiedono però un equilibrio stabile tra creatività e rigore matematico ed un’accurata precisione nell’interpretazione delle soluzioni ottenute. Lo studio dei modelli matematici permette infatti ad una figura gestionale di essere supportata e guidata da dati certi, nelle decisioni aziendali e nell’ottimizzazione di varie situazioni, lavorative e non, che deve però apportare la propria conoscenza personale per assumere decisioni funzionali e vantaggiose.

La parola “ottimizzazione” richiede un’attenzione particolare, sia all’interno della presente tesi, sia nelle vite quotidiane di questo secolo. La società infatti frenetica in cui viviamo, è causata dai processi ottimizzati che utilizziamo nella quotidianità, ed è allo stesso tempo una concausa dello sviluppo e della ricerca necessaria per il campo della ricerca operativa, in quanto il continuo miglioramento della qualità di vita e della tecnologia, richiede una costante crescita, e quindi ottimizzazione, degli strumenti che occorrono alla popolazione.

In egual modo anche la globalizzazione ha contribuito agli studi dei metodi matematici, cercando i migliori approcci e i punti strategici per permettere la connessione sempre più agevole di varie zone geografiche. Alcune delle sfaccettature più peculiari della ricerca operativa sono infatti la scelta della posizione ideale per un punto vendita in una specifica area geografica o la definizione di percorsi minimi, usati per la pianificazione degli ormai comuni navigatori, il cui accesso è garantito ad una grande parte della popolazione, grazie all’integrazione delle macchine di calcolo digitali alle mere formule matematiche.

In ambito aziendale troviamo tante altre applicazioni utili alla gestione operativa, come per esempio la logistica, la gestione dei flussi di cassa, fino all’ottimizzazione di intere catene e flussi di produzione. Tutto ciò avviene in modo rapido ed agevole grazie all’implementazione dei modelli matematici attraverso sistemi informatici avanzati.

Lo spettro di fattori analizzati permette di constatare che la ricerca operativa promuove lo sviluppo del sapere e quindi del potere umano. La sua particolarità risiede infatti nella capacità di affrontare problemi apparentemente insuperabili, spingendo l’intelligenza umana a cercare soluzioni innovative e di conseguenza contribuendo al progresso della società moderna.

Inizio modulo

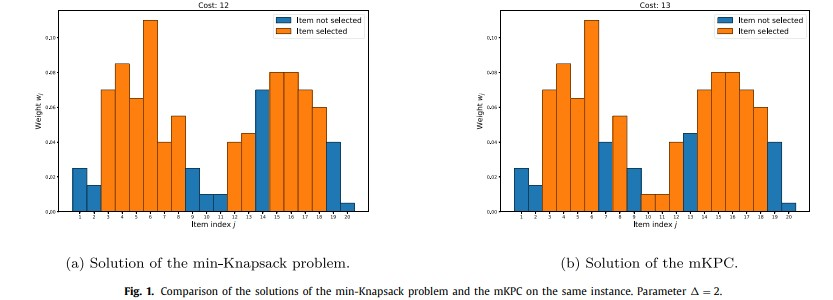
## Introduzione al problema

Il problema che verrà analizzato nella presente tesi è chiamato min-Knapsack Problem con vincoli di compattezza (mKPC). Il problema è stato introdotto in letteratura nel 2023, e di conseguenza analizzato in un articolo del Luglio 2023, pubblicato sulla rivista “European Journal of Operational Research” (A. Santini ed E. Malaguti). Il problema in analisi è dunque una particolare estensione del generico min-Knapsack problem, che fa parte a sua volta della famiglia del Knapsack Problem (KP).

Alla base del “problema dello zaino” è presente un modello decisionale lineare binario. L’esito del processo decisionale lineare, e dunque della soluzione finale, è valutato tramite una combinazione lineare dei valori associati a ciascuna delle decisioni binarie. Formalmente, il modello decisionale lineare è definito da n variabili binarie xi ∈ {0,1}, che corrispondono alla selezione nella i-esima decisione binaria, e da valori di profitto pi. Possiamo quindi assumere che in qualsiasi dei due casi xi=1 e xi=0, abbiamo sempre pi ≥ 0. Il profitto totale associato a tutte le n decisioni binarie è quindi dato dalla somma di tutti i valori pi per tutte le decisioni in cui è stata selezionata la prima alternativa. Il problema dello zaino è infine ottenuto aggiungendo una restrizione di capacità in modo che la somma dei pesi di tutte le decisioni binarie non superi un dato valore c, che concettualmente equivale alla capacità del contenitore, ovvero dello zaino (knapsack). L’obiettivo è selezionare un sottoinsieme degli n elementi, tale che il profitto totale degli oggetti selezionati sia massimizzato e il peso totale non superi c.

Il min-Knapsack problem è una variante del sopra citato problema dello zaino, che viene in questa casistica trasformato in un problema di minimizzazione. Da un numero finito di oggetti n, si deve selezionare un sottoinsieme di elementi con un peso totale almeno pari a c, tale che il profitto totale, in questo caso considerato come costo, degli oggetti sia minimizzato.

Come anticipato all’inizio di questa sezione, il problema in analisi per questa tesi, è un’estensione di quest’ultimo, in quanto aggiunge al comune vincolo sul peso, un altro vincolo di compattezza, che opera a livello spaziale, dichiarando che gli elementi selezionati nel sottoinsieme non possono essere tra loro troppo distanti. Immaginando infatti che i vari oggetti giacciano in una sequenza ordinata, con questo vincolo aggiuntivo l’obiettivo è quello di selezionare un insieme di elementi quanto più compatto. Il vincolo sulla distanza utilizza come numeri di riferimento il valore numerico degli indici degli elementi. Calcolando infatti la differenza fra i due indici, viene definita la distanza tra i due. Usando un valore positivo intero Δ come parametro della distanza, se la distanza tra gli indici dei due oggetti considerati è maggiore di Δ, allora viene richiesto che sia selezionato almeno un altro elemento tra i due. Analizzando l’esempio in Fig.1, proposto dall’articolo (A. Santini ed E. Malaguti), gli oggetti sono ordinati sull’asse x in modo crescente in base al proprio indice, mentre l’altezza delle barre indica il proprio peso. Il costo è unitario per tutti gli elementi ed il parametro Δ per il mKPC è pari a 2.



La versione (a), ovvero senza i vincoli di compattezza, ha un costo totale di 12, ma considerano il parametro Δ, esso viene violato tra gli elementi con indice 8 e 12, che presentano una distanza di 4>2. Nella versione (b), è presentata una soluzione ottima che presenta un costo totale pari a 13, che non viola il vincolo spaziale. Si nota quindi che il mKPC presenta una soluzione con un costo maggiore di un normale min-Knapsack, ma selezionando un sottoinsieme compatto di elementi.

Lo studio di questa variante del problema, come viene sottolineato nell’articolo precedentemente citato, è motivato dalle sue applicazioni nell’analisi delle serie temporali e nella statistica con grandi dimensioni. Una serie temporale è una sequenza di valori numerici indicizzati da punti temporali discreti (Hamilton, 1994). Data una serie temporale y1, …, yn, l’obiettivo del rilevamento dei cambiamenti è identificare se la distribuzione di probabilità sottostante di y cambia, quante volte lo fa e in quali punti temporali. I tipici punti di cambio per le serie temporali si verificano quando la serie temporale cambia il suo valore atteso, la sua varianza o entrambi. Alcune tra le applicazioni più importanti sono per esempio nel settore sanitario, per rilevare i cambiamenti nelle condizioni nei pazienti, o in econometria, per avvertire i segnali di una crisi.

# **Modello lineare**

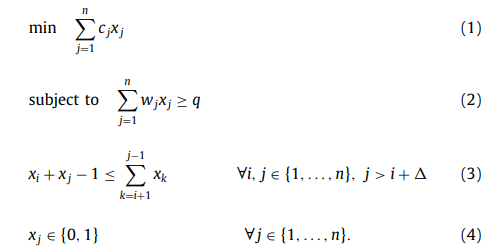
## Introduzione alla programmazione lineare

I modelli di programmazione lineare sono modelli matematici di problemi di ottimizzazione di tipo lineare (continuo e intero), ampiamente impiegati nell’ambito della ricerca operativa. Essi sono costituiti da una “funzione obiettivo”, rappresentata da un’espressione lineare delle incognite del problema da massimizzare o minimizzare, ovvero il nostro obiettivo ultimo. Accanto a questa funzione è presente un sistema di vincoli, che delimitano geometricamente la regione di ammissibilità del problema, attraverso un insieme di equazioni e disequazioni lineari che le incognite del problema devono soddisfare, ovvero per i valori in cui la funzione obiettivo è ammissibile. Le incognite sono delle variabili decisionali o di controllo, delle quali viene specificato l’intervallo di valori consentito, consentendo quindi la presenza di diverse tipologie, che vengono generalmente suddivise in variabili reali non negative, intere non negative o binarie.

## Modello matematico

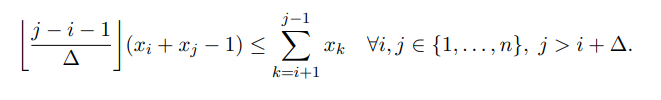
Presenteremo ora il modello lineare elaborato per affrontare il problema in questione.

Possiamo formulare il problema mKPC con il seguente modello, in cui la variabile binaria xj assume il valore 1 se l’elemento j-esimo è selezionato.



Proseguendo per gradi, viene presentata prima la funzione obiettivo (1), la quale ha l’obiettivo appunto, di minimizzare la sommatoria del valore dei costi c di tutti gli elementi selezionati, che presentano quindi un valore pari a 1 per la variabile xj. Nella seconda linea del modello troviamo il primo vincolo sul peso (2), il quale dichiara che la sommatoria di tutti i valori dei pesi w degli elementi selezionati, dev’essere almeno pari al valore di una costante q ≥ 0. Procedendo troviamo il vincolo di compattezza (3), caratteristico del problema in analisi, in quanto lo differenzia da un semplice min-Knapsack problem; Il vincolo richiede che, nell’istante in cui ho selezionato due variabili xi e xj e viene rispettata la disequazione degli indici j > i + Δ, devo avere un minimo di altre variabili xk selezionate fra loro, tali che la disequazione venga rispettata e non ci siano spazi maggiori di Δ tra le variabili che compongono la soluzione finale. Vediamo inoltre che sia i che j appartengono ad un intervallo di valori che va da 1 a n, ovvero il numero totale degli oggetti analizzati. Nell’ultima riga (4) viene infine dichiarata la natura binaria delle variabili xj.

Tuttavia il vincolo di compattezza (3) può essere rafforzato come segue:



Strutturando (5) come illustrato, quando per esempio due degli elementi selezionati giacciono in posizioni con distanza fra loro pari a 2Δ, tramite questo rinforzo possiamo selezionare direttamente almeno altri due elementi contenuti tra i primi due.

Con queste definizioni il modello risulta completo e quindi adatto per risolvere qualsiasi istanza del problema, che contenga *n* oggetti, un costo *c* e un peso *p* assegnato ad ogni oggetto, oltre ad un valore reale positivo per la costante *q* ed uno intero positivo per Δ.

Il modello appena presentato, ai fini di questa tesi, è stato implementato su una macchina virtuale con linguaggio di programmazione python, attraverso un codice che utilizza la libreria MIP (Mixed-Integer-Programming), peculiare della programmazione lineare.

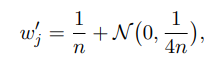
## Generazione delle istanze

All’interno dell’articolo già citato molteplici volte, viene utilizzato un set di istanze, di cui pesi e costi vengono generati con diversi metodi, mentre n, Δ e q sono selezionate in modo arbitrario.

Per lo scopo delle analisi di questa tesi sono state generate 92 istanze, attraverso un solo metodo di generazione dei costi ed invece due metodi per i pesi.

Le istanze possono essere raggruppate in diversi sottoinsiemi, che si differenziano per il metodo usato per la generazione dei pesi e per i valori attribuiti a n, Δ e q.

Il metodo che è stato scelto per la generazione dei costi, nell’articolo di Santini e Malaguti viene presentato e denominato come “Random”, poiché esso assegna ad ogni oggetto j-esimo del problema un valore di costo uniformemente distribuito in un intervallo [1, 10].

Per la generazione dei pesi invece, il primo metodo utilizzato è denominato “Noise”. Questa procedura in primo luogo assegna ad ogni elemento j-esimo un peso definito come segue:

dove N(λ, σ) denota una distribuzione normale. Poiché la somma di tutti i pesi generati in tal modo potrebbe non essere necessariamente uguale a uno, normalizziamo tutti i valori. Le istanze generate da questo metodo generalmente richiedono un maggior numero di elementi da selezionare per raggiungere il peso minimo richiesto da q.

Il secondo metodo che è stato scelto è invece chiamato “One Peak”. La procedura di quest’ultimo consiste nel scegliere una media λ tra 1 e *n*, campionando da una distribuzione normale troncata con media *n/2* e deviazione standard *n/4*, arrotondando all’intero più vicino. Procede quindi nella generazione di un’istanza i cui pesi presentano un picco vicino a λ. A questo scopo viene considerata un'altra distribuzione normale troncata tra 1 e *n*, con media λ e deviazione standard *n/k*, dove *k* è un parametro della generazione dell’istanza. Seguendo le regole della statistica, i pesi saranno più strettamente distribuiti attorno al picco con valori di *k* più alti. Dal campionamento viene poi costruito un istogramma con n barre. Ogni j-esima barra conta quanti campioni ricadono nell’intervallo [j, j+1). In questo modo viene assegnato ad ogni elemento j un peso w pari all’altezza della j-esima barra del grafico. I pesi sono poi anche in questo caso normalizzati.

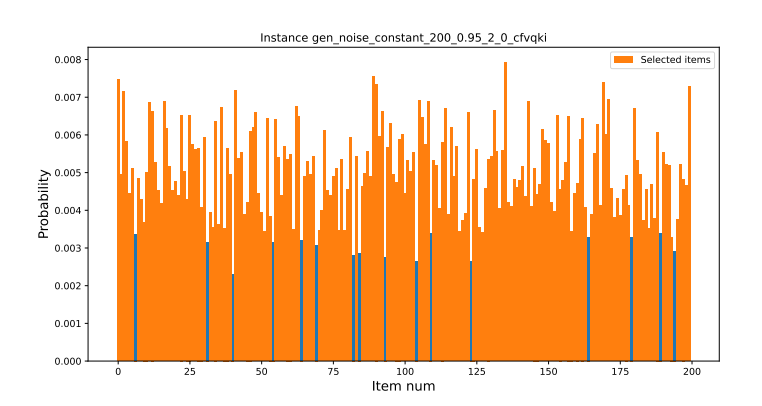


Fig. 2: Esempio di un istanza NOISE con la sua soluzione ottimale

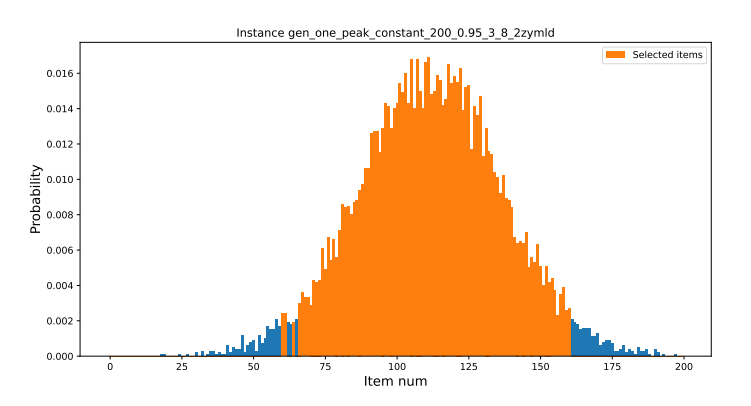
****

Fig. 3: Esempio di un istanza ONE PEAK con la sua soluzione ottimale

Sono quindi stati usati tre valori per *n* {100, 200, 400} e tre valori per Δ {3, 5, 10}, con un valore di q = 0.95. Generando le istanze con i due metodi differenti (Noise e OnePeak) si generano una combinazione di 18 istanze. Ogni combinazione viene testata attraverso 4 tentativi. Sono poi state testate altre 5 combinazioni di istanze, utilizzando un valore di q = 0.5 e il metodo NOISE per la generazione dei pesi, testate anche queste ultime 4 volte l’una. Attraverso questa tecnica sono state generate quindi 92 diverse istanze.

# **Algoritmo euristico**

## Teoria dell’euristica greedy

L’euristica è un metodo di risoluzione dei problemi che si focalizza sulla ricerca di soluzioni praticabili in tempi ragionevoli, anche se non necessariamente ottimali. Spesso è proficuo utilizzare questo approccio per gestire efficacemente situazioni complesse, in cui le risorse sono limitate o ristrette e di conseguenza la ricerca di una soluzione ottimale richiederebbe un notevole sforzo computazionale o temporale. Gli algoritmi euristici mirano quindi a fornire, attraverso una sequenza di operazioni, una soluzione che si avvicini il più possibile a quella ottima, ma in un tempo accettabile e con risorse limitate, come suggerito dall’etimologia del termine stesso “euristica”, che deriva dal termine greco ԑυρίσκω (scoprire, trovare), associato quindi in uso moderno alla risoluzione di problemi tramite l’uso dell’intuizione e dell’esperienza.

La costruzione di questa classe di algoritmi richiede un’analisi puntigliosa del problema, tale da permettere di individuarne lo scheletro, con le relative caratteristiche specifiche utili alla risoluzione. Lo studio deve inoltre comprendere anche la scelta della tipologia di algoritmo da usare.

Così come esplicitato nello studio dell’articolo già più volte citato, utilizzeremo la struttura di un algoritmo “greedy” (vorace). Questi particolari algoritmi operano attraverso una sequenza di scelte “localmente ottime”, ossia decisioni che sembrano le migliori nel contesto immediato. Ciò implica che tali algoritmi procedono unicamente sulla base delle informazioni disponibili all’istante, senza mai riconsiderare o modificare le decisioni già prese. La strategia greedy viene apprezzata per la sua implementazione relativamente semplice e per le performance computazionalmente efficienti che possono offrire.

## Algoritmo sviluppato

Il principio fondamentale dell’utilizzo di un algoritmo greedy per un generico knapsack problem è quello di dare precedenza nell’inserimento nello zaino agli oggetti “più promettenti”, ossia quelli che, seguendo un determinato criterio euristico, hanno una maggiore probabilità di far parte di una soluzione ottimale.

La procedura greedy sviluppata per il mKPC ha l’obiettivo di identificare un sottoinsieme di oggetti P ⊆ {1,...,n} , mantenendo l’imposizione di avere un peso totale che sia almeno pari a q e che ovviamente soddisfi i vincoli di compattezza. L’algoritmo comincia ordinando gli n elementi in una sequenza ordinata in base al rapporto crescente costo/peso del singolo oggetto, in quanto l’obiettivo del nostro problema è quello di minimizzare il costo e massimizzare il peso. Seguendo questo procedimento i primi numeri saranno quelli con costo minore al denominatore e peso maggiore al denominatore, che consente di abbassare il valore del rapporto. Dopo questo passaggio P viene inizializzato con un solo elemento, che corrisponde al primo elemento della nuova sequenza ordinata rispetto al rapporto scelto. In seguito, ad ogni iterazione l’insieme P continuerà ad aumentare, poiché verranno aggiunti progressivamente gli elementi “più promettenti” non ancora appartenenti a P, seguendo la sequenza ordinata, ma che contemporaneamente rispettino i vincoli di compattezza. Infine l’algoritmo si ferma quando raggiunge la condizione tale che la somma dei pesi degli elementi appartenenti all’insieme P sia maggiore o uguale a q. Come evidenziato dal criterio di conclusione scelto per questo algoritmo euristico, esso si ferma anche senza aver necessariamente trovato una soluzione ottima.

Nell’implementazione in codice sviluppata su python di questo algoritmo, il primo passaggio è quello di leggere le istanze generate secondo i metodi illustrati sopra nel modello lineare. Successivamente viene creato l’array che conterrà i rapporti costo/peso ordinati in ordine crescente:

for i in range(n):

array.append(c[i] / w[i])

array.sort()

Di seguito lo scheletro del cuore dell’algoritmo.

# chiavi è una lista di indici di posizione corrispondenti al valore passato

# indici è una lista con tutti gli indici, che vengono tolti una volta selezionato l’item corrispondente

# aggiungo a 'selected' solo gli indici di posizione degli item che seleziono

selected.append(chiavi[0])

indici.pop(chiavi[0])

# aggiungo al peso totale il peso del primo elemento selezionato

w\_tot = w[chiavi[0]]

while len(indici) != 0 and w\_tot < q:

# itero su tutti gli n items della mia istanza

for iterazione in range(1, n):

trovato = False

if w\_tot >= q: # sono arrivata alla soluzione ed esco dall’algoritmo

break

else:

chiavi = trova\_chiavi\_da\_valore(indici, arr[iterazione])

for z in range(len(chiavi)):

index2 = chiavi[z] # indice dell'elemento che sto considerando

for itt in range(len(selected)):

index1 = selected[itt] # indice degli elementi già selezionati

if abs(index2 - index1) <= delta:

# se entra nell'IF posso selezionare il valore e andare avanti, basta che il vincolo di compattezza sia rispettato per UNO degli elementi già selezionati. Quindi aggiungo a selected l’indice dell’elemento che sto considerando {index2}

selected.append(index2)

# aggiorno anche il peso totale

w\_tot = w\_tot + w[index2]

indici.pop(chiavi[z]) # rimuovo l’indice dell’item selezionato

trovato = True

break # esco dal ciclo che itera sugli elementi già selezionati

break

if trovato == True:

break

Come evidenziato dalla riga di codice che indica la condizione di uscita **if w\_tot >= q**, vediamo che l’algoritmo prima di terminare non procede ad un controllo sull’ottimalità della soluzione. È inoltre importante evidenziare come l’euristica sia efficace in quanto i vincoli di compattezza vengono verificati ad ogni iterazione per la scelta del nuovo elemento da selezionare, come espresso dalla condizione rappresentata da **if abs(index2 - index1) <= delta**, poiché appunto se l’indice dell’elemento su cui è in atto la decisione di selezione non rispetta il criterio di compattezza con gli indici degli altri elementi già selezionati, l’algoritmo non entra in questo ramo decisionale e scorre all’elemento successivo su cui attuare la decisione binaria.

# **Analisi computazionale**

Nel campo dell’analisi computazionale, entriamo in un territorio teorico fondamentale per comprendere quanto siano complessi i problemi e come possano essere risolti tramite algoritmi. L’obiettivo principale di questa analisi è infatti quello di individuare l’algoritmo più efficiente per risolvere determinati problemi. Gli studi computazionali richiedono l’indagine di vari parametri, come per esempio il tempo di esecuzione, fondamentale per determinare la praticità di un algoritmo. L’analisi computazionale valuta la quantità di memoria necessaria per far funzionare un algoritmo al crescere delle dimensioni dell’istanza del problema, oltre a classificare i problemi in base alla loro difficoltà intrinseca. Esistono infatti suddivisioni in classi che ci permettono di stabilire se un problema può essere risolto efficientemente o se è intrinsecamente complesso. I parametri citati fino ad ora definiscono nel loro insieme ciò che è il costo computazionale di un algoritmo. Esso rappresenta un aspetto critico da considerare quando si sviluppano o si scelgono algoritmi, poiché un alto costo computazionale può rendere impraticabile l’uso dell’algoritmo in alcune situazioni o su determinati hardware e software.

Lo scopo dell’analisi computazionale svolta in questa tesi è di estrarre un sistema di risoluzione più efficiente, dopo aver testato l’euristica e la struttura del modello lineare. L’efficacia verrà valutata quantitativamente tramite il confronto dei tempi di elaborazione in base alle specifiche del problema e delle diverse istanze utilizzate. Useremo principalmente il confronto tra due parametri: il tempo di risoluzione del problema e il valore, ovvero il costo minimizzato, della funzione obiettivo. La risoluzione tramite la struttura del modello lineare attraverso l’utilizzo di Gurobi 10.0.1 come MIP solver. Per tutte le istanze è stato posto un tempo limite di esecuzione di 300 secondi, ovvero di 5 minuti. La scelta di impostare un time limit ha garantito la risoluzione in tempi accettabili anche per le istanze che richiedono iterazioni elaborate.

Di seguito la prima valutazione fatta è stata il confronto tra le 23 diverse combinazioni di istanze, già citate in precedenza, testate ognuna 4 differenti volte, quindi con valori di costi e pesi differenti. Vengono, nella prima tabella, analizzate in particolare le due caratteristiche principali. La caratteristica 1 rappresenta la media del rapporto della differenza tra la funzione obiettivo dell’euristica e quella del solver e la funzione obiettivo del solver, che viene considerata come soluzione ottimale. La percentuale indicata in tabella identifica dunque la differenza in eccesso (quindi non minimizzata) del valore della funzione obiettivo dell’algoritmo euristico rispetto alla soluzione ottima. La caratteristica 2 invece indica la media tra il tempo di elaborazione dell’euristica e quello del modello lineare. Il valore espresso quindi in questa casistica indica la porzione percentuale di tempo corrispondente al modello lineare che l’euristica impiega ad arrivare ad una soluzione.

|  |  |  |
| --- | --- | --- |
| Gruppo di Test | Caratteristica 1 | Caratteristica 2 |
| Test 1 | 0,62% | 4,16% |
| Test 2 | 0,47% | 1,84% |
| Test 3 | 0,07% | 1,87% |
| Test 4 | 0,58% | 2,88% |
| Test 5 | 0,36% | 2,01% |
| Test 6 | 0,32% | 1,57% |
| Test 7 | 0,32% | 1,88% |
| Test 8 | 0,12% | 1,26% |
| Test 9 | 0,19% | 0,95% |
| Test 10 | 0,50% | 0,85% |
| Test 11 | 0,82% | 0,93% |
| Test 12 | 0,66% | 0,98% |
| Test 13 | 0,71% | 0,87% |
| Test 14 | 0,45% | 0,84% |
| Test 15 | 0,41% | 0,80% |
| Test 16 | 1,70% | 0,34% |
| Test 17 | 1,27% | 0,40% |
| Test 18 | 0,28% | 0,37% |
| Test 19 | 0,65% | 0,93% |
| Test 20 | 13,91% | 0,05% |
| Test 21 | 8,12% | 0,52% |
| Test 22 | 0,72% | 0,77% |
| Test 23 | 12,41% | 0,34% |
| Totale | 1,98% | 1,19% |

Tabella 1: analisi globale delle istanze

Dai risultati che emergono in tabella, è evidente notare che le differenze tra l’euristica e il modello lineare sono minime, inferiori al 2%. Nella maggior parte delle istanze notiamo che il valore della funzione obiettivo dell’euristica è maggiore di una percentuale inferiore a 1. Nei test 16 e 17, le cui istanze presentano un n pari a 400 e Δ rispettivamente pari a 3 e 5 ed i pesi vengono generati con il metodo ONE PEAK, notiamo che l’euristica si differenzia dal valore trovato dal solver, per un valore maggiore di 1. Ciò si verifica perché grazie all’utilizzo del metodo ONE PEAK, gli elementi con valori di peso maggiore si concentrano e compattano automaticamente, rendendo più efficace la gestione dei vincoli di compattezza nel modello lineare. Necessitano invece di un altro discorso le ultime istanze, che presentano un valore percentuale della prima caratteristica altamente maggiore di 1. Queste istanze (20, 23) presentano un valore di q pari a 0,5, i pesi generati attraverso il metodo NOISE e valori di Δ bassi (3). Questo risultato, che si discosta nettamente dall’andamento generale della totalità dei test, cela dietro di sé anche la non ottimalità della soluzione del solver, che si ferma a una soluzione feasible a causa del limite imposto sul tempo. Per chiarità del presente testo dichiaro che sono stati considerati come valori della funzione obiettivo del modello lineare per l’analisi delle caratteristiche i risultati feasible, tralasciando i valori di best bound presentati dal solver. Essendo infatti la capacità che la sommatoria dei pesi deve raggiungere inferiore, i due algoritmi devono analizzare molte più combinazioni di soluzioni rispetto ai casi precedenti. Proprio in queste casistiche emerge la potenza del modello lineare che, nonostante non trovi una soluzione ottimale causata dai limiti del solver, essendo la complessità computazionale elevata arriva comunque a buone soluzioni, migliori di quelle raggiunte dall’euristica. Da questa valutazione capiamo che l’aumento di complessità computazionale rappresenta un simil punto di svolta, per cui risulta vantaggioso utilizzare il modello lineare, a discapito del tempo di elaborazione, poiché garantisce un risultato nettamente migliore dell’euristica. Nelle istanze precedenti, che presentano un ridotto costo computazionale, notiamo infatti che per istanze con valore di n maggiori di 100, l’euristica è sempre vantaggiosa a livello di tempo di elaborazione oltre ad essere il suo valore di soluzione quasi trascurabilmente peggiore di quello del solver.

Raggruppando invece le diverse istanze in tre grandi gruppi, possiamo analizzare caratteristiche più generali. I tre gruppi si dividono in istanze generate da metodo NOISE e q=0,95, metodo ONE PEAK e q=0,95 e infine metodo NOISE e q=0,5, ignorando quindi i diversi valori di n e di Δ.

|  |  |  |
| --- | --- | --- |
| Gruppo | Caratteristica 1 | Caratteristica 2 |
| Gruppo 1 | 0,34% | 2,05% |
| Gruppo 2 | 0,76% | 0,71% |
| Gruppo 3 | 7,16% | 0,52% |

Tabella 2: analisi dei 3 gruppi principali

Come quindi già evidenziato dall’analisi generali svolta sopra vediamo che l’euristica risulta di molto svantaggiosa rispetto al solver per valori minori di q. Non si notano invece rilevanti differenze, per lo stesso valore di q, tra le istanze generate dai metodi NOISE e ONE PEAK. Per quanto riguarda la caratteristica 2, l’unica osservazione rilevante da fare è che con alti valori di q il metodo NOISE tramite solver risulta più rapido da elaborare, apportando un valore di percentuale più alto alla caratteristica.

Viene osservato poi il gruppo di istanze relativo ai Test 16, 17 e 18, che presentano i pesi generati con il metodo ONE PEAK, un valore di n=400 e q=0,95 e rispettivamente valori di Δ pari a 3, 5 e 10, in cui vengono riportati i valori di funzione obiettivo e del tempo di elaborazione per entrambi gli approcci alle soluzioni.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | fObb\_solver | fObb euristica | T\_solver | T\_euristica |
| Test16\_1 | 1229,15 | 1252,12 | 123,415 | 0,635 |
| Test16\_2 | 1010,63 | 1029,39 | 135,506 | 0,417 |
| Test16\_3 | 1022,44 | 1042,07 | 159,935 | 0,427 |
| Test16\_4 | 1096,29 | 1109,18 | 161,606 | 0,476 |
| Test17\_1 | 1077,49 | 1106,23 | 121,701 | 0,437 |
| Test17\_2 | 1166,26 | 1173,07 | 109,267 | 0,459 |
| Test17\_3 | 1231,16 | 1234,81 | 158,831 | 0,545 |
| Test17\_4 | 1126,82 | 1143,93 | 91,678 | 0,456 |
| Test18\_1 | 1239,22 | 1241,32 | 110,6 | 0,508 |
| Test18\_2 | 1105,11 | 1107,7 | 122,23 | 0,41 |
| Test18\_3 | 992,12 | 997,31 | 119,469 | 0,352 |
| Test18\_4 | 1108,09 | 1110,42 | 110,125 | 0,416 |

Tabella 3: risultati computazionali dei Test 16, 17, 18

Dall’analisi di questa tabella risalta che la scelta di utilizzare un approccio euristico piuttosto che lineare dipende ampiamente dal contesto decisionale in cui ci troviamo e dagli obiettivi preposti da raggiungere tramite l’elaborazione degli algoritmi. Questa scelta non è mai quindi una decisione oggettiva e universale, ma dipende da una serie di fattori e variabili specifiche. Vediamo infatti ancora una volta che il tempo di risoluzione dell’euristica è nettamente più rapido di quello del solver, ma l’approccio euristico non consente di arrivare allo stesso stato di ottimizzazione della soluzione con la stessa frequenza. Quindi è evidente che la decisione di utilizzare un metodo piuttosto che l’altro dipenderà da quanto sia cruciale ottenere una soluzione di alta qualità rispetto a quanto sia importante risolvere rapidamente il problema. Se pensiamo infatti ad un contesto aziendale, in cui va acquistata della merce mensilmente da tenere in magazzino, ma si hanno dei vincoli di compattezza sulla frequenza dell’invio di un ordine, un costo fisso che viene ammortizzato raggiungendo almeno una capacità q di magazzino e un numero di elementi molto alti (es. n > 100000), potrebbe essere più conveniente, con gli strumenti utilizzati nella presente tesi, ricorrere all’algoritmo euristico, in quanto aiuterebbe ad avere una soluzione veloce con una spesa di poco maggiore, garantendo un uso di risorse limitato (es. potenza di calcolo). Al contrario, per altre esigenze in cui il tempo non sia un vincolo, è più conveniente utilizzare il modello lineare, in quanto assicura il risparmio massimo.

Come velatamente esplicato dalle considerazioni sopra, anche la tolleranza all’errore svolge un ruolo cruciale nella decisione. In alcuni contesti, una soluzione approssimata ottenuta con un approccio euristico potrebbe essere accettabile, mentre in altri casi è necessaria un’ottimizzazione precisa. Oltre a questo fattore, anche l’esperienza e la competenza dell’utente che utilizza gli algoritmi possono influenzare notevolmente la preferenza a diversi metodi.

Ritengo necessario anche analizzare visualmente i valori della caratteristica 2, per i diversi valori di n (100, 200, 400) per le istanze che prevedono un valore di q=0,95, riportati in Tabella 4.

Tabella 4: istogramma della caratteristica 2

Come già sommariamente evidenziato in precedenza, si nota che per valori di n bassi (100, 200), il valore percentuale della caratteristica 2, ovvero quale percentuale del tempo del solver equivale al tempo di elaborazione dell’euristica, è più alto, poiché per istanze con quantità minori il solver arriva alla soluzione ottima in un tempo relativamente limitato e comunque non ampiamente maggiore di quello dell’algoritmo euristico. Contrariamente, per n = 400, è evidente che il valore percentuale sia nettamente più basso, in quanto il solver inizia ad essere particolarmente lento, mentre l’euristica continua ad avere un tempo di elaborazione in media pari a 1 secondo.

# **Conclusioni**

Da ciò che emerge nelle analisi svolte in questa tesi e dai ragionamenti effettuati sui modelli matematici utilizzati, è importante sottolineare che la scelta tra un approccio euristico e uno lineare è altamente soggettiva e dipende dalle circostanze specifiche di ogni problema. Non esiste infatti una risposta universale corretta, ma la decisione dovrebbe essere basata su una valutazione attenta delle esigenze, delle risorse disponibili e degli obiettivi da raggiungere. Come evidenziato grazie agli studi dell’algoritmo del mKPC, la complessità intrinseca delle istanze del problema da risolvere riveste un ruolo molto importante nella scelta dell’approccio.

L’euristica dimostra infatti di essere un efficace strumento risolutivo in quasi tutte le situazioni, mentre il modello lineare si dimostra particolarmente necessaria e ottima in contesti di bassa complessità o in situazioni che richiedono maggiore precisione nella determinazione della soluzione ottimale.

Le deduzioni rese possibili da questa tesi rimangono comunque limitate agli strumenti utilizzati e agli studi ristretti effettuati. Esse possono infatti essere migliorate proseguendo la ricerca con strumenti statistici più avanzati, così da individuare maggiori correlazioni tra le caratteristiche di ciascun gruppo di istanze ed i valori della funzione obiettivo ottenuti dai due approcci. Una ricerca più avanzate potrebbe rendersi utile nel definire con maggiore oggettività le situazioni ottimali in cui usare l’uno o l’altro metodo, per garantire una risoluzione altamente efficace del problema.

In generale tuttavia, l’analisi computazionale conferma l’importanza della ricerca operativa e dei propri strumenti nella risoluzione di problemi complessi del mondo reale, offrendo mezzi potenti per prendere decisioni scientificamente fondate. L’ottimizzazione è infatti una componente cruciale nella gestione delle risorse e nell’affrontare sfide sempre più complesse nella società moderna. Avere a disposizione dunque una combinazione di modelli matematici, algoritmi e strumenti computazionali continua e continuerà a contribuire al progresso in primis della conoscenza scientifica in letteratura oltre al progresso della società tramite la risoluzione di problemi reali. Il vero progresso non può quindi in alcun modo basarsi sulla scelta “bianco o nero” di uno specifico settore di ricerca, ma deve sfruttare la combinazioni di tutti i rami e ambiti della scienza.

# **Bibliografia**

Tadei R., Della Croce F., (2010). *Elementi di Ricerca Operativa*. Esculapio.

Santini A., Malaguti E., (2023). The min-Knapsack problem with compactness constraints and application in statistics, *European Journal of Operational Research*. <https://doi.org/10.1016/j.ejor.2023.07.020>

Kellerer H., Pferschy U. & Pisinger D., (2004). *Knapsack problems*. Springer

# **Appendice**

In appendice viene inserito il link ad una repository pubblica di GitHub, che contiente il codice Python integrale usato per gli scopi di questa tesi, ovvero sia per l’implementazione del modello matematico lineare sia per la logica dell’algoritmo euristico. Il codice generato viene riportato anche per iscritto nelle seguenti pagine, a scopo informativo. La cartella online riporta anche i documenti su cui sono riportate le istanze generate dal codice ed il foglio di calcolo su cui sono analizzate e di cui vengono riportati i risultati.

Repository GitHub: <https://github.com/LudovicaM/Tesi_RicercaOperativa>

Modello lineare

from mip import Model, xsum, minimize, BINARY, GRB  
import numpy as np  
import random  
import time  
  
# 4 tentativi  
tentativi = 0  
# n = 100, 200, 400  
n = 200  
# costante q -> peso minimo degli oggetti  
# q = 0.95, 0.5 (x20)  
q = 0.5  
# distanza max  
# delta = 3, 5, 10  
delta = 5  
  
  
# generazione dei costi -> metodo RANDOM:  
  
outfile = open("test\_finale.txt", 'a', encoding="utf-8")  
while tentativi < 4:  
 c = [random.uniform(1, 10) for x in range(n)]  
 t\_c = 0.00  
 for cc in range(n):  
 c[cc] = round(c[cc], 2)  
 t\_c = t\_c + c[cc]  
 print("costi {}\n".format(c))  
 print("total cost: %f\n" % t\_c)  
  
 # generazione dei pesi  
 # METODO NOISE  
  
 w\_i = np.zeros(n)  
 w = np.zeros(n)  
 # inizio il processo di generazione dei pesi  
 mean = 0  
 std\_dev = 1/(4\*n)\*\*1/2  
 # normalizzazione  
 arr = np.random.normal(loc=mean, scale=std\_dev, size=n)  
 for i in range(n):  
 w\_i[i] = 1/n + arr[i]  
 if w\_i[i] < 10 \*\* -12:  
 w\_i[i] = 10 \*\* -12  
 tot\_w\_i = 0.0  
 for s in range(n):  
 tot\_w\_i = tot\_w\_i + w\_i[s]  
 for z in range(n):  
 w[z] = w\_i[z]/tot\_w\_i  
  
 t\_w = 0.0  
 for j in range(len(w)):  
 t\_w = t\_w + w[j]  
 print("Total weight: %f\n" % t\_w)  
  
 '''   
 # METODO ONE PEAK  
  
 lab = np.random.normal(n/2, n/4)  
 lab = round(lab)  
 k = 32  
 arr = np.random.normal(loc=lab, scale=n/k, size=5000)  
 hist, bins = np.histogram(arr, bins=n)  
 w\_i = np.zeros(n)  
 w = np.zeros(n)  
 for i in range(n):  
 if hist[i] < 10 \*\* -12:  
 w\_i[i] = 10 \*\* -12  
 else:  
 w\_i[i] = hist[i]  
 print("totale pesi {}\n".format(w\_i))  
 tot\_w\_i = 0.0  
 for s in range(n):  
 tot\_w\_i = tot\_w\_i + w\_i[s]  
 for z in range(n):  
 w[z] = w\_i[z]/tot\_w\_i  
 print("pesi finali {}\n".format(w))  
   
 t\_w = 0.0  
 for j in range(len(w)):  
 t\_w = t\_w + w[j]  
 print("Total weight: %f\n" % t\_w)  
 '''  
  
 # Modello lineare tramite solver GUROBI  
 m = Model("Knapsack", solver\_name=GRB)  
  
 # creo le variabili binarie x  
 x = [m.add\_var(var\_type=BINARY) for j in range(n)]  
  
 # funzione obiettivo -> minimizzo la sommatoria dei costi degli elementi presi (x=1)  
 m.objective = minimize(xsum(c[j]\*x[j] for j in range(n)))  
  
 # vincolo sul peso (la sommatoria dei pesi degli elementi presi dev'essere >= alla costante q)  
 m += xsum(w[j]\*x[j] for j in range(n)) >= q  
  
 for i in range(n):  
 for j in range(i + 1, n):  
 if j > i + delta:  
 m += ((j - i - 1)/delta)\*(x[i] + x[j] - 1) <= xsum(x[k] for k in range(i+1, j))  
  
 time\_limit\_seconds = 300  
 start\_time = time.time()  
 # ottimizzo  
 status = m.optimize(max\_seconds=time\_limit\_seconds)  
  
 end\_time = time.time()  
 # Calcolo del tempo di esecuzione  
 execution\_time = end\_time - start\_time  
  
 m.write('model.lp')  
  
 # stampa su console  
 print("sample size (n): %d" % n)  
 # lista degli elementi selezionati per cui x vale più di 0.99  
 selected = [j for j in range(n) if x[j].x >= 0.99]  
 print("selected items : {}".format(selected))  
 tot\_cost = m.objective\_value  
 print("Funzione obiettivo: {}\n".format(tot\_cost))  
 print("Tempo di risoluzione: {}\n".format(execution\_time))  
  
 # stampa su file di testo  
 # per ogni istanza ho nella prima riga i dati in input e nella seconda i risultati  
 # prima del PL e poi dell'euristica  
 # legenda input: n; delta; q; [c]; [w]\n  
 # legenda output: fObiettivo, time, status, best bound; fObiettivo; time; status; best bound  
  
 outfile.write("%d; %d; %f; " % (n, delta, q))  
 outfile.write("{}; ".format(c))  
 formatted\_data = " ".join(map(str, w))  
 outfile.write(formatted\_data)  
 outfile.write("\n")  
 outfile.write("f obiettivo: {}, ".format(m.objective\_value))  
 outfile.write("Time: {}, ".format(execution\_time))  
 # verifica dello stato della soluzione  
 if execution\_time < time\_limit\_seconds:  
 outfile.write("Status: optimal, ")  
 else:  
 outfile.write("Status: feasible, ")  
 outfile.write("Best bound: {}; \n".format(m.objective\_bound))  
 tentativi = tentativi + 1  
  
outfile.close()

Algoritmo euristico

import ast  
import time  
# questa funzione mi ritorna una lista di chiavi associate al valore passato  
# quindi anche 2 indici associati allo stesso rapporto  
  
  
def trova\_chiavi\_da\_valore(dizionario, valore):  
 chiavi\_corrispondenti = []  
 for chiave, valore\_attuale in dizionario.items():  
 if valore\_attuale == valore:  
 chiavi\_corrispondenti.append(chiave)  
 return chiavi\_corrispondenti  
  
  
with open("test\_finale.txt", "r", encoding="utf-8") as infile, open("test\_completi.txt", "a", encoding="utf-8") as outfile:  
  
 #prendo i dati in input dal file di lettura  
 for line in infile:  
 if line != '':  
 #per ogni istanza ho nella prima riga i dati in input e nella ……………………………..seconda i risultati

#prima del PL e poi dell'euristica  
 #legenda input: n; delta; q; [c]; [w]\n  
 #legenda output: fObiettivo, time, status, best bound; ………………………………fObiettivo, time, status, best bound\n  
 outfile.write(line)  
 dati = line.split('; ')  
 n = int(dati[0])  
 delta = int(dati[1])  
 q = float(dati[2])  
 c = ast.literal\_eval(dati[3])  
 pesi = dati[4].strip("[]").split()  
 w = [float(x) for x in pesi]  
 #qua inizia l'algoritmo dopo aver letto tutti i dati in input  
 start\_time = time.time()  
 #ALGORITMO EURISTICO  
 # creo un array ordinato in modo crescente che contiene i ……………………………..rapporti costo/peso  
 arr = []  
 index\_dic = {} # mi salvo gli indici per ogni rapporto. Key: ……………………………………………………………………….indice, Value: rapporto  
 indici = {} # uguale al precedente ma da qua li tolgo man …………………………………………………………………mano che seleziono gli items  
 for i in range(n):  
 arr.append(c[i] / w[i])  
 index\_dic[i] = arr[i]  
 indici[i] = arr[i]  
 arr.sort() # arr è ordinato in modo crescente in base al …………………………………………………………….rapporto costo/peso  
 # scorro l'array dall'inizio alla fine perchè i primi valori …………………………………saranno quelli con costo minimo e peso maggiore  
 # array vuoto 'selected' in cui vado ad aggiungere man mano i ……………………………valori che seleziono  
 # ovvero quelli che rispettano il vincolo di compattezza  
 selected = []  
 # aggiungo a 'selected' solo gli indici di posizione  
 # aggiungo il primo elemento (costo minore e peso maggiore)  
 chiavi = trova\_chiavi\_da\_valore(index\_dic, arr[0])  
 # chiavi è una lista di indici di posizione corrispondenti al …………………………….valore passato  
 selected.append(chiavi[0])  
 indici.pop(chiavi[0])  
 w\_tot = w[chiavi[0]] # aggiungo al peso totale il peso del ………………………………………………………………………………………..primo elemento selezionato  
 while len(indici) != 0 and w\_tot < q:  
 for iterazione in range(1, n): # continuo a iterare …………………………………………………………………………………………………………………………finchè non finiscono gli elementi  
 trovato = False  
 if w\_tot >= q: # sono arrivata alla soluzione ed esco ……………………………………………………………………………………………….dalla iterazione del for  
 break  
 else:  
 chiavi=trova\_chiavi\_da\_valore(indici, arr[iterazione])  
 for z in range(len(chiavi)):  
 index2 = chiavi[z] # indice dell'elemento che sto ………………………………………………………………………………………………………………………………considerando  
 for itt in range(len(selected)):  
 index1 = selected[itt] # indice degli elementi ………………………………………………………………………………………………………………………………………………………….già selezionati  
 if abs(index2 - index1) <= delta:  
 # se entra nell'if posso selezionare il valore ………………………………………………………………………………………………e andare avanti, basta che il vincolo  
 # di compattezza sia rispettato per UNO degli ………………………………………………………………………………………………elementi già selezionati  
 # QUINDI aggiungo a selected il rapporto …………………………………………………………………………………………….dell'elemento che sto considerando {index2}  
 selected.append(index2)  
 # aggiorno anche il peso totale  
 w\_tot = w\_tot + w[index2]  
 indici.pop(chiavi[z])  
 trovato = True  
 break # esco dal ciclo che itera sugli …………………………………………………………………………………………………………………elementi già selezionati  
 break  
 if trovato == True:  
 break  
 #calcolo il valore della soluzione  
 fObb = 0  
 for i in range(len(selected)):  
 fObb = fObb + c[selected[i]]  
 # rilevo il tempo di esecuzione  
 end\_time = time.time()  
 execution\_time = end\_time - start\_time  
 selected.sort()  
 print("Selected: {}\n".format(selected))  
 #da qui in poi scrivo solo gli output  
 riga2 = infile.readline()  
 riga2 = riga2.strip()  
 outfile.write(riga2 + " " + "f obiettivo: %f, " % fObb + "Time: {}, Status: /, Best Bound: /\n".format(execution\_time))  
  
 elif line == '\n':  
 continue  
 else:  
 break  
  
infile.close()  
outfile.close()