Algoritmo Greedy, Local Search e Metaeuristico per la risoluzione del problema MMKP

Diego Moranda ([diego.moranda@supsi](mailto:diego.moranda@supsi))

Andrea Riccardi ([andrea.riccardi@student.supsi.ch](mailto:andrea.riccardi@student.supsi.ch))

SUPSI DTI

Ingegneria Informatica

Algoritmi Avanzati - Ottimizzazione

Summary

L’obbiettivo del progetto di questo corso è quello di sviluppare tre algoritmi in C++ che possano andare a risolvere il problema dell’MMKP, con diverse configurazioni possibili.

I tre algoritmi da sviluppare sono:

Algoritmo Greedy, Algoritmo di Local Search e Algortimo Metaeuristico.

L’obbiettivo è quello di massimizzare il valore ottenuto per ottenere una buona risoluzione del problema, rispettando le tempistiche richieste.

Keywords: MMKP, C++, Greedy, Local Search, Metaeuristico

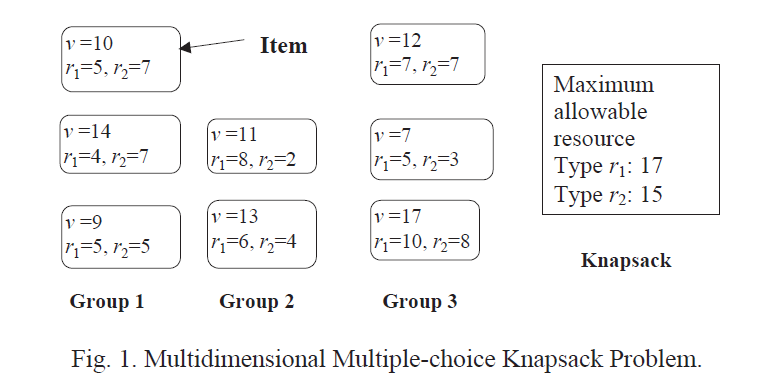
# Report progetto

## Modalità operativa

In questa sezione mostreremo la modalità operativa e lo sviluppo dei vari algoritmi

### **Introduzione Problema MMKP**

Il problema mostrato nella figura sottostante è quello del Multidimensional Multiple-choice Knapsack problem (MMKP). Si ha uno zaino con multiple capacità, si ha ogni gruppo che è composto da n elementi, e tali elementi hanno un valore e multiple dimensioni di risorse occupate. L’obbiettivo è quello di andare a selezionare per ogni gruppo l’elemento migliore che permetta alla fine di avere nello zaino uno valore alto rispettando il consumo delle capacità.



**Analisi file**

Nell’immagine sottostante abbiamo un esempio di che tipo di istanza del problema dobbiamo andare a leggere da file. Qui sotto una breve descrizione dei vari campi:

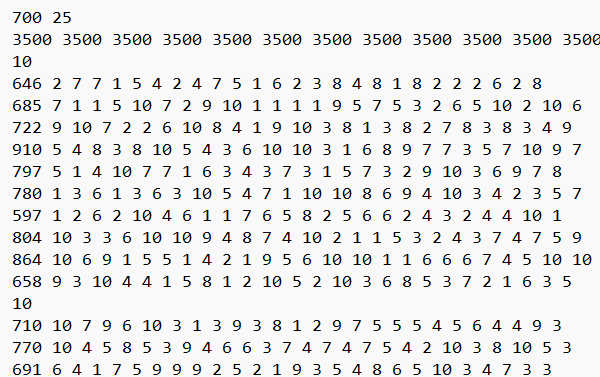
Il primo numero(700): Rappresenta il numero di classi(Group) presenti nel problema

Il secondo numero(25): Rappresenta il numero di risorse che ogni item consuma, e quante tasche ha il nostro Knapsack

La seconda riga rappresenta la capacità di ogni tasca dello zaino, possono essere diverse tra di loro, e il numero fa riferimento al 25 di sopra.

Nella terza riga abbiamo l’inizio della prima classe, il cui il numero 10 sta a rappresentare il numero di Item presenti nella classe.

La quarta riga, il primo numero rappresenta il valore dell’item, mentre gli altri numeri sono quanto risorsa esso occupa per sacca.



**Struttura codice per lettura istanze**

Per la lettura di un instanza del problema MMKP abbiamo fatto questa struttura di classi:

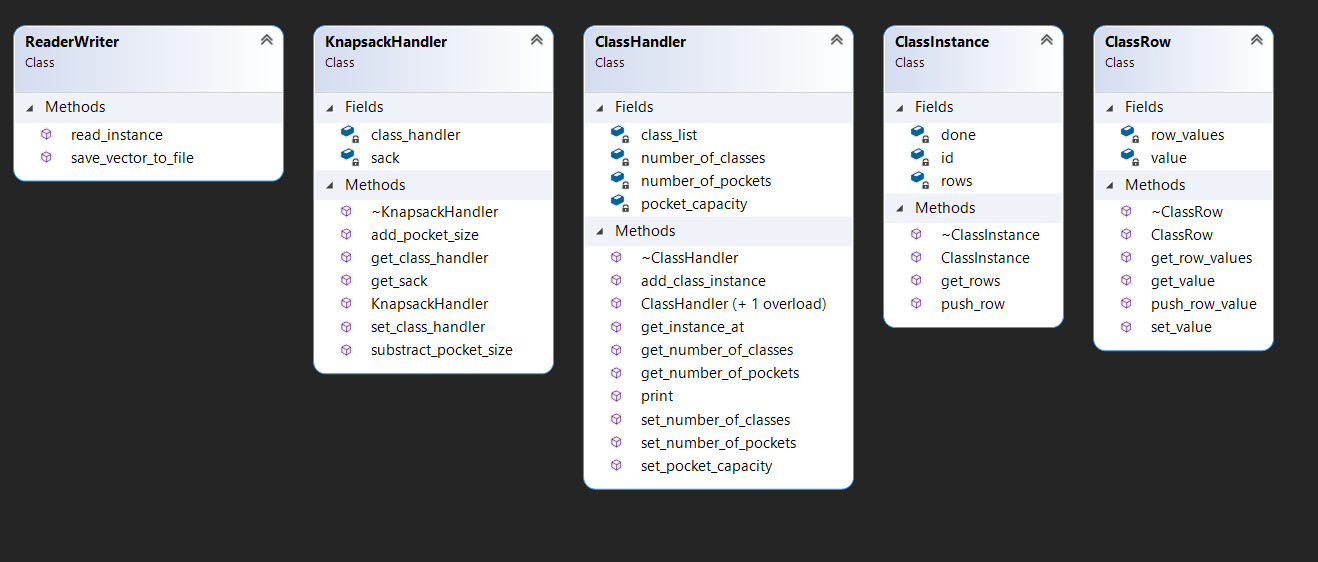
***KnapsackHandler***: Gestisce le risorse rimanenti nel vettore sack, e gestisce anche un ClassHandler dentro di se.

***ClassHandler***: Gestisce tutte le instanze delle Classi (ClassInstance)

***ClassInstance***: Gestisce la singola instanza di classe con dentro tutte le ClassRow ovvero le righe del problema.

***ClassRow***: Gestisce la singola Row del problema, guardando il value e tutte le risorse occupate

***ReaderWriter***: Gestisce le operazioni da file, e ritorna con il metodo read\_instance un KnapsackHandler con tutti i dati e le strutture già configurate



### **Risoluzione Greedy**

Per la risoluzione con l’algortimo Greedy ci siamo basati quindi sulla struttura della lettura di instanza di prima.

Su ogni classe del problema, andiamo a prendere il best\_of\_instance e poi andiamo ad inserirlo nella sacca.

Per trovare la best\_of\_instance, Andiamo a ciclare su tutte le rows della classe ed andiamo a prendere l’average minimo che risulta. Per scegliere l’average, avviene un for su ogni elemento della riga  
average += (float) row\_values[i]/pocket\_sizes[i]\*pocket\_weight[i];

Dove va a prendere il valore della riga, diviso per la capacità della sacca moltiplicato per un peso dato alla sacca. Inizialmente il pocket\_weight è pari a uno e viene aumentato nella procedura di sottrazione dopo.

Dopo aver preso il best\_value della classe lo si va a sottrarre alle sacche. Dentro questo metodo si va a vedere se il valore rimanente diviso il valore iniziale della capacita è minore di un tune value scelto dopo test, se è minore si aumenta il peso della sacca cosi che influenzerà la scelta della prossima best\_instance.

### **Risoluzione Local Search**

Per l’algortimo di Local Search, partiamo da una soluzione generata dall’algoritmo greedy. Dopo di che nella procedura di local search andiamo per ogni classe a sortare ogni row in base al value. Poi per un numero di step andiamo a guardare avanti confrontando se la soluzione attuale sia minore o migliore rispetto ad una con il valore più alto, se minore andiamo ad effettuare uno scambio.

Alla fine della local search, la rieffettuiamo altre due volte sulla soluzione generata, per andare ad ottenere una soluzione migliore

### **Risoluzione Metaeuristica**

We followed the engine structure that was given by the professor, TODO INSERISCI DETTAGLI\_\_. We had a problem with Windows support, so we had to create a class called Utils.h which had some declarations to get over the problem and keep going. After the engine structure was built, we talked about the develop strategy, and so we saw that everything revolved around the OVO Reader and so we focussed on getting the file import working.

### **Fase di test**

At first, we began by reading the Doxygen documentation present in the Zip file. We studied the example of the implementation and translated it in to our graphical engine, we started adding the missing elements that we needed in the various classes, such as some information’s about the node, the mesh, the material. After we added all the extra fields, we spent a bit of time on the recursive function to build the tree because it was giving us a hard time to understand why it wasn’t working. We then figured out the main issue, it was that even if the element didn’t have any child, it also called a recursive load, so creative excessive calls and so it was messing up the returned value. After we gathered all the information’s, we built the scene graph.

### **Conclusioni**

After we successfully implemented the OVO Reader we then got working on the theory of what was needed to start rendering the scene. To do so, we implemented as suggested a class called List, in which we used a Node\* vector to store the Meshes, and the lights, as suggested by the theory notions, we put the lights in the beginning of the list, while we pushed back the meshes. Because the main goal of the list was to use it to call a renderList method that would loop through the list and call the render method on each Node, so we needed to render the lights first.