ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Visualizzatore 3D per Soluzioni di un Problema di Caricamento

Tesi di laurea in:
Ricerca Operativa

Relatore
Prof. Marco Antonio Boschetti

Candidato
Mattia Forti

III Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Nei tempi moderni gli esseri umani devono affrontare diversi ostacoli, alcuni semplici come viaggiare da casa propria al luogo di lavoro, altri invece più complessi, come sviluppare una tecnologia efficiente per la produzione di energie rinnovabili. Per alcuni problemi è stato trovato un modo per calcolare la soluzione migliore, per altri invece si cerca di minimizzare le approssimazioni che vengono applicate per trovarla. In questo documento verrà analizzato uno di questi problemi, per il quale è tutt'ora difficile trovare una soluzione ottima in ogni situazione. Il problema in questione è il Knapsack Problem (KP), uno dei problemi classici della Ricerca Operativa e che copre una moltitudine di casistiche. Uno dei casi più comuni è il classico problema di caricamento di pacchi in un vano carico di un mezzo di trasporto. Si tratta di una casistica molto comune, basti pensare a un semplice trasporto di merci per il quale si vogliono trasportare prima gli oggetti di valore o priorità più alta (gioielli, dispositivi elettronici, attrezzi da lavoro, etc.), per poi passare a quelli con valore più basso (alimenti, vestiti, libri, etc.). L'obbiettivo di questo lavoro di tesi sarà quindi quello di trovare una soluzione a questo problema, ma non è realistico aspettarsi di ottenere quella ottima, questo perché per la tipologia di problemi di cui fa parte il KP ancora non sono stati proposti in letteratura degli algoritmi efficienti per risolverlo a causa della sua estrema difficoltà. Per trovare una soluzione ottima si potrebbe essere costretti a eseguire un numero molto elevato di tentativi, tale che anche con un computer dalle prestazioni notevoli potrebbe essere impossibile trovarla in tempi di calcolo "ragionevoli", perciò questo documento si concentrerà sulla ricerca di una soluzione di buona qualità (ma non necessariamente ottima) applicabile in un contesto reale.

Indice

Elenco delle figure

Elenco dei Code Snippets

Capitolo 1

Introduzione

Uno dei più famosi problemi di ottimizzazione conosciuti e studiati è il **Problema dello Zaino**, o **Knapsack Problem (KP)**.

Il KP viene studiato da più di cento anni, con i primi studi risalenti alla fine del XIX secolo. Si tratta di uno degli argomenti cardini della Ricerca Operativa, in quanto è un problema molto comune e di grande interesse per diversi settori lavorativi.

Uno dei casi più interessanti di questo argomento è quello tridimensionale, o 3-Dimensional Knapsack Problem (3DKP), il quale rappresenta una delle casistiche più comuni, ovvero quella del caricamento di colli in un vano carico di un mezzo di trasporto.

Il bisogno di algoritmi per calcolare come posizionare dei pacchi all'interno di un contenitore, che può variare da un bagagliaio di una macchina a un container per il trasporto navale, è sempre più alto. Al momento della stesura di questa tesi gli utilizzatori principali di questo tipo di applicazione sono gli operatori manuali, ma si sta anche cercando di creare un processo interamente automatizzato con l'utilizzo di braccia meccaniche per le grandi aziende che devono caricare dei container o dei camion.

Essendo un problema strettamente di carattere matematico e algoritmico, questo lavoro di tesi si pone l'obbiettivo di sviluppare un'applicazione open-source che servirà non solo ad applicare gli algoritmi di risoluzione, ma anche a visualizzare i risultati con un'interfaccia grafica per mostrare all'utente una possibile soluzione, utilizzando anche delle animazioni per indicare l'ordine di inserimento dei pacchi.

Capitolo 2

Descrizione del Problema

2.1 Definizione

Prima di poter analizzare in modo più esaustivo e descrittivo il problema, vediamo la sua definizione più comune della versione unidimensionale:

Definizione 1. *Sia dato uno zaino che possa sopportare un determinato peso e siano dati N oggetti, ognuno dei quali caratterizzato da un peso e un valore. Il problema si propone di scegliere quali di questi oggetti mettere nello zaino per ottenere il maggiore valore senza eccedere il peso sostenibile dallo zaino stesso. [MT90]*

2.2 Varianti

Esistono quattro varianti del KP [var24]:

- 0/1
- Con Limiti
- Senza Limiti
- Frazionario

Nelle sezioni successive verranno descritte le varianti appena menzionate, ma quella che verrà analizzata e discussa nel dettaglio in questo documento sarà la 0/1, essendo una delle più interessanti e molto comune.

2.2.1 0/1

In questa variante esiste solo una copia di ogni oggetto, perciò non ci possono essere ripetizioni. Possiamo descrivere questa versione nel seguente modo:

Enunciato 1. *[Nag24] Sia dato un insieme di n coppie:*

$$c_i = (v_i, w_i) \quad con \quad 0 \leq i \leq n-1 \tag{2.1}$$

dove v_i indica il valore (value) e w_i indica il peso (weight) della coppia.

Si vuole massimizzare:

$$\sum_{i=0}^{n-1} v_i \cdot x_i = P \tag{2.2}$$

rispettando la seguente restrizione:

$$\sum_{i=0}^{n-1} w_i \cdot x_i \leq W \tag{2.3}$$

dove:

- $x_i \rightarrow$ *parametri binari* $(0, 1)$ *decisionali che definiscono se la coppia corrispondente deve essere considerata*
- $W \rightarrow$ *peso massimo sopportabile*
- $P \rightarrow$ *guadagno totale (profit)*

Si vogliono trovare tutti i valori x_i per i quali P è massima e W non viene superata.

2.2.2 Con Limiti

La variante Bounded Knapsack Problem (o BKP) rimuove la restrizione della variante 0/1 riguardante l'esistenza di una sola copia per ogni oggetto, definendo però un limite massimo di copie disponibili per ogni oggetto [Jun23].

Enunciato 2. *[Jun23] Sia dato un insieme di coppie come definito in 2.1, si vuole massimizzare la sommatoria 2.2 con la seguente restrizione:*

$$\sum_{i=0}^{n-1} w_i \cdot x_i \leq W \ \textit{con}\ 0 \leq x_i \leq C \tag{2.4}$$

dove x_i è il numero di copie che si vogliono prendere della coppia c_i e C (cap) è il numero massimo di copie che si possono prendere per ogni coppia c_i.

2.2.3 Senza Limiti

La variante Unbounded Knapsack Problem (o UKP) rimuove ogni limite, rendendo possibile prendere un quasiasi numero di copie per ogni oggetto [PYA09].

Enunciato 3. *[PYA09] Sia dato un unsieme di coppie come definito in 2.1, si vuole massimizzare la sommatoria 2.2 con la seguente restrizione:*

$$\sum_{i=0}^{n-1} w_i \cdot x_i \leq W \ \textit{con}\ x \in \mathbb{N} \tag{2.5}$$

dove x_i è il numero di copie che si vogliono prendere della coppia c_i.

2.2.4 Frazionario

La variante frazionaria permette di utilizzare una frazione di ogni oggetto ed è definita in questo modo:

Enunciato 4. *[GT01] Sia dato un insieme di coppie come definito in 2.1, si vuole massimizzare la sommatoria 2.2 con la seguente restrizione:*

$$\sum_{i=0}^{n-1} w_i \cdot x_i \leq W \ \textit{con}\ 0 \leq x \leq 1 \tag{2.6}$$

dove x_i è la porzione che si vuole prendere della coppia c_i.

2.3 3DKP

Quando si pensa a un problema di caricamento uno degli ambiti più conosciuti è quello del trasporto merci, o più nello specifico il caricamento di colli in un vano carico di un mezzo di trasporto. La nostra attenzione passa quindi dal caso base unidimensionale, al caso più pratico in tre dimensioni.

In questo documento verrà analizzato il 3DKP, ovvero verrà considerato il caso in cui gli oggetti da caricare nello zaino esistano in uno spazio tridimensionale. Perciò oltre al peso e al valore di ogni oggetto, dovremo considerare anche lunghezza, altezza e spessore.

Enunciato 5. *Sia dato uno spazio tridimensionale:*

$$S = (W, H, D) \tag{2.7}$$

Dove W è la larghezza (width), H è l'altezza (height) e D è la lunghezza (depth). Consideriamo inoltre il limite massimo di peso che lo spazio S può sopportare e chiamiamolo W_l (weight limit).

Sia dato un insieme di n parallelepipedi rettangoli:

$$K = \{k_1, k_2, ..., k_n\} \text{ dove ogni } k_i = (w_i, h_i, d_i) \tag{2.8}$$

Ogni parallelepipedo k_i è quindi caratterizzato da:

- $w_i \rightarrow$ *larghezza (width)*
- $h_i \rightarrow$ *altezza (height)*
- $d_i \rightarrow$ *spessore (depth)*

Inoltre per ogni parallelepipedo consideriamo anche il peso W_i (weight), il valore V_i (value) e la posizione (x_i, y_i, z_i) nello spazio S che assumerà nella soluzione.

[EP07] Siano $s_i \in \{0, 1\}$ delle variabili decisionali binarie che indicano se il parallelepipedo i deve essere inserito nello spazio S e siano l_{ij} (left), r_{ij} (right), u_{ij} (under), o_{ij} (over), b_{ij} (behind), f_{ij} (front) delle variabili decisionali binarie che indicano la posizione di un parallelepipedo k_i rispetto a un parallelepipedo k_j.

Siccome un parallelepipedo non può fuoriuscire dallo spazio S e due parallelepipedi non possono sovrapporsi, definiamo alcune condizioni:

$$0 \leq x_i \leq W - w_i \ , \ 0 \leq y_i \leq H - h_i \ , \ 0 \leq z_i \leq D - d_i \tag{2.9}$$

$$l_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} = 1 \ \textit{quando} \ s_i = s_j = 1 \tag{2.10}$$

Inoltre in base al posizionamento dei parallelepipedi k_i e k_j, le seguenti disuguaglianze devono essere soddisfatte:

- $l_{ij} = 1 \implies x_i + w_i \leq x_j$
- $r_{ij} = 1 \implies x_i + w_i \leq x_i$
- $u_{ij} = 1 \implies y_i + h_i \leq y_j$
- $o_{ij} = 1 \implies y_i + h_i \leq y_i$
- $b_{ij} = 1 \implies z_i + d_i \leq z_j$
- $f_{ij} = 1 \implies z_i + d_i \leq z_i$

Riformuliamo il 3DKP:

$$max \sum_{i=0}^{n-1} V_i \cdot s_i \tag{2.11}$$

tale che:

- $l_{ij} + r_{ij} + o_{ij} + u_{ij} + f_{ij} + b_{ij} = s_i + s_j - 1$
- $x_i - x_j + W \cdot l_{ij} \leq W - w_j$
- $x_j - x_i + W \cdot r_{ij} \leq W - w_i$
- $y_i - y_j + H \cdot u_{ij} \leq H - h_j$
- $y_j - y_i + H \cdot o_{ij} \leq H - h_i$
- $z_i - z_i + D \cdot b_{ij} \leq D - d_j$
- $z_j - z_j + D \cdot f_{ij} \leq D - d_i$

- $0 \leq x_i \leq W - w_i$
- $0 \leq y_i \leq H - h_i$
- $0 \leq z_i \leq D - d_i$
- $\sum_{i=1}^{n} W_i \cdot s_i \leq W_l$

Capitolo 3

Soluzione Proposta

3.1 Posizionamento dei Pacchi

Un algoritmo molto intuitivo per posizionare i pacchi può inserire i pacchi provando ogni singola posizione nel contenitore, passando in rassegna tutta la larghezza, altezza e lunghezza a determinati intervalli (step). Questa tecnica però è computazionalmente inefficiente per due motivi:

1. Dimensioni del Contenitore → per contenitori molto grandi servirebbero molte iterazioni; basti pensare a una semplice macchina con un bagagliaio di dimensioni $2m \times 1m \times 1m$ (in ordine larghezza, altezza e lunghezza), utilizzando uno step di 1cm bisognerebbe controllare $200 \times 100 \times 100 = 2.000.000$ posizioni diverse. Per casistiche più professionali, come il caricamento un container per il trasporto navale, le iterazioni raggiungerebbero le decine di milioni.

2. Spazio tra i pacchi → per ottenere un buon posizionamento bisognerebbe scegliere uno step molto piccolo in modo da avvicinare i pacchi il più possibile e sprecare meno spazio, così facendo però aumenta il numero di iterazioni e si ritorna al problema 1.

Per minimizzare le iterazioni e la distanza tra i pacchi viene proposto il seguente algoritmo, che prevede un inserimento di pacchi progressivo partendo dall'angolo in basso, in fondo a sinistra, fino ad arrivare all'angolo di fronte, in alto a destra.

Consideriamo un insieme P dove verranno inseriti i pacchi uno alla volta:

1. Vengono calcolate tutte le posizioni possibili relativamente ai pacchi già inseriti in P (se P è vuoto, quello corrente viene inserito in fondo a sinistra e si passa a quello successivo). Considerando n come il numero di pacchi in P, al più avremo $3n$ possibili posizioni, poiché per ogni pacco già inserito, quello successivo potrà essere a destra, sopra o davanti ad esso.

2. Vengono scartate tutte le posizioni nelle quali il pacco si sovrappone ad un altro oppure esce dal contenitore, eseguendo quindi un massimo di $3n^2$ iterazioni (per ognuna delle $3n$ posizioni dobbiamo confrontarla con tutti i pacchi posizionati in precedenza).

3. Siccome vogliamo posizionare i pacchi partendo dal fondo, ordiniamo le posizioni in ordine di profondità (quelle più in fondo hanno priorità più alta), in caso di parità prendiamo quelle più in basso e, analogamente, prendiamo quelle più a sinistra in caso di parità di altezza.

4. Ricaviamo la prima posizione in base alle priorità definite e la assegniamo al pacco, inserendolo in P.

5. Ripartiamo dal punto 1 passando al pacco successivo.

Creiamo lo pseudo-codice di un'iterazione dell'algoritmo appena definito:

function CalculateCoordinates(P, p_i)
  if $P \equiv \emptyset$ **then**
    return (0, 0, 0)
  else
    $C \leftarrow \emptyset$
    for $p_j \in P$ **do**
      $c_1 \leftarrow (x_j + w_j, y_j, z_j)$
      $c_2 \leftarrow (x_j, y_j + h_j, z_j)$
      $c_3 \leftarrow (x_j, y_j, z_j + d_j)$
      $C \leftarrow C \cup \{c_1\}$
      $C \leftarrow C \cup \{c_2\}$
      $C \leftarrow C \cup \{c_3\}$

end for
for $c \in C$ **do**
$(x_i, y_i, z_i) \leftarrow c$
if $Collides(P, p_i)$ **then**
$C \leftarrow C \setminus \{c\}$
end if
end for
if $C \equiv \emptyset$ **then**
return -1
end if
Order C by depth position
if $\exists\ C' : \{c_1, c_2, ..., c_k\} \subseteq C \mid z_1 = z_2 = ... = z_k = min_z(C)$ **then**
$C \leftarrow C'$
Order C by vertical position
if $\exists\ C' : \{c_1, c_2, ..., c_k\} \subseteq C \mid y_1 = y_2 = ... = y_k = min_y(C)$ **then**
$C \leftarrow C'$
Order C by horizontal position
end if
end if
end if
return $(C[1]_x, C[1]_y, C[1]_z)$
end function

Dove la funzione `Collides` è fatta in questo modo:

function COLLIDES(P, p_i)
for $p_j \in P$ **do**
$xCheck \leftarrow x_j + w_j \leq x_i \vee x_i + w_i \leq x_j$
$yCheck \leftarrow y_j + h_j \leq y_i \vee y_i + h_i \leq y_j$
$zCheck \leftarrow z_j + d_j \leq z_i \vee z_i + d_i \leq z_j$
$containerCheck \leftarrow x_i + w_i \leq W \wedge y_i + h_i \leq H \wedge z_i + d_i \leq D$
$collisioncCheck \leftarrow \neg(xCheck \vee yCheck \vee zCheck)$
if $collisionCheck \vee containerCheck$ **then**
return true

```
        end if
    end for
    return false
end function
```

Con questi due metodi si riesce a calcolare una serie di posizioni nelle quali il pacco p_i non si sovrappone a nessun altro pacco precedentemente posizionato e non fuoriesce dal contenitore. Ricordiamo che siccome vogliamo inserire i pacchi partendo dal fondo ordiniamo le posizioni disponibili in base alla profondità, quindi quelle più in fondo saranno le prime nella lista, fino ad arrivare a quelle più frontali che saranno le ultime. Nel caso in cui ci siano due o più posizioni con la stessa profondità, si separano dalle altre e si ordinano per altezza. Analogamente se ce ne sono due o più con la stessa altezza, si separano dalle altre e si ordinano per posizione orizzontale. Al termine degli ordinamenti rimarrà un vettore con una sola posizione o più posizioni che rappresentano le stesse coordinate, perciò basta prendere la prima e restituirla.

Utilizziamo questo algoritmo per ogni pacco che dobbiamo posizionare. Sia I l'insieme dei pacchi da inserire:

```
function InsertPackages(I)
    P ← ∅
    for p_i ∈ I do
        pos ← CalculateCoordiantes(P, p_i)
        if pos ≠ −1 then
            (x_i, y_i, z_i) ← pos
            P ← P ∪ {p_i}
        else
            return ∅
        end if
    end for
    return P
end function
```

In questo modo però si inseriscono i pacchi ignorando quelli di fronte. Se, ad esempio, avessimo un pacco molto grande di fronte ma ci fosse abbastanza spazio dietro di esso per metterne uno più piccolo, quest'ultimo verrebbe giustamente inserito, ma nella

realtà un operatore manuale o un braccio meccanico che segue le istruzioni dell'algoritmo non potrebbe metterlo dietro a causa della presenza del pacco grande posizionato in precedenza.

Per risolvere questo problema basterebbe, alla fine dell'algoritmo, ordinare i pacchi per la posizione (in profondità), ma questa operazione può essere eseguita soltanto nel caso in cui i pacchi vengono inseriti tutti nello stesso momento; nei casi delle grandi aziende di autotrasporti spesso i pacchi vengono caricati in diverse sedi, perciò non è possibile calcolare sin dall'inizio l'ordinamento giusto.

3.2 Calcolo dei Pacchi da Inserire

Per risolvere il problema della scelta dei pacchi viene proposto di utilizzare il *Simulated Annealing*, una tecnica di carattere probabilistico basata sull'*annealing*:

> *"L'annealing è il processo con il quale un solido, portato allo stato fluido, mediante riscaldamento ad alte temperature, viene riportato poi di nuovo allo stato solido o cristallino, a temperature basse, controllando e riducendo gradualmente la temperatura. Ad alte temperature, gli atomi nel sistema si trovano in uno stato altamente disordinato e quindi l'energia del sistema è elevata. Per portare tali atomi in una configurazione cristallina altamente ordinata (statisticamente), deve essere abbassata la temperatura del sistema. Riduzioni veloci della temperatura possono causare difettosità nel reticolo cristallino con conseguente metastabilità, con fessurazioni e fratture del reticolo stesso (stress termico). L'annealing evita questo fenomeno procedendo ad un graduale raffreddamento del sistema, portandolo ad una struttura globalmente ottima stabile."*[Lac19]

Nel Simulated Annealing viene fatto un parallelismo rispetto all'annealing originale; la temperatura diventa il numero di iterazioni e gli atomi diventano gli stati (valori) del modello; di conseguenza quando il numero delle iterazioni è alto, inizialmente i valori saranno molto variabili, mentre verso la fine saranno più consistenti.

3.2.1 Simulated Annealing

Vediamo innanzitutto come funziona questo metodo con un esempio pratico:

Esempio 1. *Immaginiamo di controllare il guadagno annuo di un'azienda per un determinato periodo di tempo. Vogliamo sapere quale anno è stato il più redditizio senza sapere preventivamente tutti i guadagni. Si può ricavare il guadagno di un anno alla volta, "scoprendoli" gradualmente.*

In questo esempio verranno utilizzati i guadagni annui della Nintendo, i dati sono stati presi da https://companiesmarketcap.com/nintendo/revenue/.

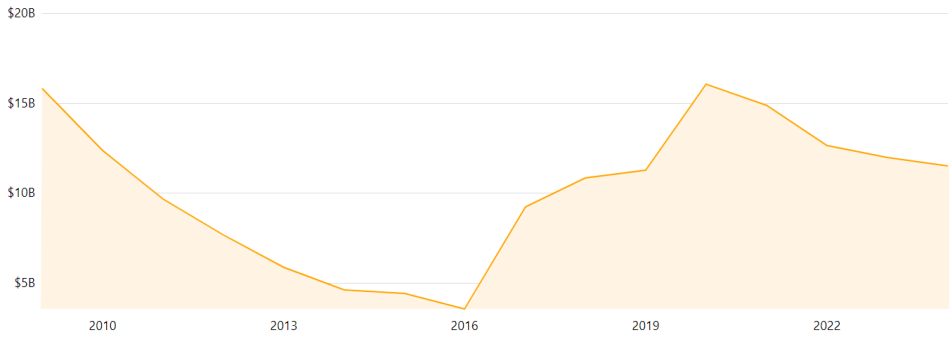


Figura 3.1: Guadagni annui della Nintendo dal 2009 al 2024

Definiamo innanzitutto i parametri iniziali:

- $Y = \{2009, 2010, ..., 2024\} \rightarrow$ insieme degli anni (Y = years)
- $P_y \subset \mathbb{R} \rightarrow$ insieme dei guadagni annui (P = profit)
- $f : Y \rightarrow P \rightarrow$ funzione che lega gli anni ai guadagni
- $I_{max} \rightarrow$ numero massimo di iterazioni
- $S_0 \in Y \rightarrow$ stato (soluzione) iniziale preso casualmente

Per ogni iterazione bisogna scegliere uno stato "vicino" alla soluzione attuale, nel caso di questo esempio prendiamo l'anno precedente o successivo di quello corrente. Successivamente "scopriamo" il valore del nuovo anno, se il guadagno è maggiore allora lo sostituiamo alla soluzione precedente e passiamo all'iterazione successiva.

Se il guadagno è minore non bisogna scartare immediatamente il tentativo, questo perché, nonostante è stato trovato un valore più basso, esso potrebbe essere più vicino al massimo globale rispetto a quello precedente.

In questo esempio l'anno più redditizio è stato il 2016 (16.03B), perciò se il valore iniziale fosse il 2012 e il prossimo valore scelto fosse il 2013, noteremmo che il guadagno annuo è sceso, ma ci siamo avvicinati al massimo globale, perciò non dovremmo scartarlo.

Per fare queste scelte viene utilizzato un algoritmo che decide casualmente se tenere in considerazione il valore con guadagno inferiore rispetto a quello precedente; con ogni iterazione la probabilità di accettare una soluzione inferiore rispetto a quella precedente diminuisce con l'avanzamento delle iterazioni.

Definiamo uno pseudo-codice per questo esempio:

$S \leftarrow \emptyset$
$S_0 \leftarrow random(Y)$
$i \leftarrow I_{max}$
while $i > 0$ **do**
    $S' \leftarrow random(\{S_0 - 1, S_0 + 1\})$
    if $f(S') > f(S_0)$ **then**
        $S_0 \leftarrow S'$
        $S \leftarrow S \cup \{S'\}$
    else
        $p = randomBetween(0, 1)$
        $\Delta P \leftarrow f(S') - f(S_0)$
        if $e^{\frac{-\Delta P}{i}} > p$ **then**
            $S_0 \leftarrow S'$
            $S \leftarrow S \cup \{S'\}$
        end if
    end if
    $i \leftarrow i - 1$
end while

Con questo algoritmo otteniamo una lista S di possibili soluzioni S_i, tra queste ricaviamo quella con $f(S_i)$ maggiore, che dovrebbe essere il massimo globale o un massimo locale. Aumentando il numero di iterazioni aumenta la probabilità di trovare il massimo

globale, in cambio di un algoritmo computazionalmente più pesante.

3.2.2 Simulated Annealing nel 3DKP

Nel caso del 3DKP non abbiamo un grafico lineare come in fig. 3.1, nel quale gli stati sono dei valori interi, ma abbiamo delle permutazioni di pacchi dove l'ordine di inserimento influenza il risultato. Di conseguenza lo spazio di ricerca non è monodimensionale, ma multidimensionale, dove le dimensioni sono date dalle variabili del modello.

Dopo aver scelto dei pacchi casuali (che rispettano i limiti di peso e spazio) come soluzione iniziale abbiamo 3 possibilità:

- Aggiungere un pacco tra quelli non ancora scelti
- Rimuovere un pacco tra quelli scelti
- Sostituire un pacco scelto con uno non scelto

Chiaramente rimuovere un pacco diminuirebbe il guadagno totale, ma ricordiamo che un'iterazione del genere potrebbe avvicinarci alla soluzione ottima, ad esempio nel caso in cui il pacco rimosso non sia presente in essa.

Siccome i pacchi vengono scelti casualmente, ad ogni iterazione dobbiamo controllare che il peso limite e lo spazio disponibile non vengano superati, in caso contrario si passa direttamente all'iterazione successiva. Dunque una soluzione che non supera i controlli di peso massimo o di spazio occupato viene trattata come una soluzione con guadagno inferiore e che non ha superato il test di probabilità del Simulated Annealing.

3.2.3 Costo Computazionale

Il KP presenta una complessità NP-Difficile:

Definizione 2. *[Hos24] Nella teoria della complessità computazionale un problema viene definito NP-Difficile se un algoritmo per la sua soluzione può essere modificato per risolvere qualsiasi problema di categoria NP (Nondeterministic Polynomial time). Non tutti i problemi NP-Difficili sono membri della categoria NP; un problema che è sia NP sia NP-Difficile viene detto NP-Completo.*

Vediamo anche il costo computazionale di tutte le funzioni definite in questa sezione:

1. `Collides`$(P, p_i) \rightarrow \mathrm{O}(|P|)$
2. `CalculateCoordiantes`$(P, p_i) \rightarrow \mathrm{O}(|P|^2)$
3. `InsertPackages`$(S) \rightarrow \mathrm{O}(|S|^3)$

A causa della difficoltà del 3DKP l'algoritmo proposto risulta avere una complessità polinomiale.

Capitolo 4

Tecnologie

4.1 Tecnologie Proposte

Per lo sviluppo di un'applicazione con un'interfaccia grafica sono disponibili diverse librerie per numerosi linguaggi di programmazione, di seguito vengono presentate quelle prese in considerazione per la creazione di un progetto in grado di fornire una soluzione del 3DKP mostrando anche il procedimento.

4.1.1 OpenGL e GLUT

OpenGL (Open Graphics Library) è un'API molto popolare utilizzata per sviluppare applicazioni grafiche in 2D e 3D. È una libreria flessibile che contiene un loop pre-implementato e che permette una forte personalizzazione delle componenti grafiche.

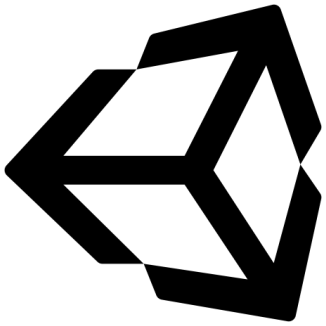
Questa sua caratteristica è data dal fatto che il programmatore può definire manualmente con dei file di codice come devono essere visualizzati gli oggetti dal punto di vista geometrico, come devono essere calcolati i colori (se devono rimanere fissi, se devono sfumare, etc.) e come applicare eventuali texture, fonti di luce e superfici riflettenti.

4.1.2 JavaSwing e JavaFX

JavaSwing e JavaFX sono delle librerie grafiche per Java; contengono delle componenti grafiche (pulsanti, pannelli, etc.) già implementate, permettendo un facile utilizzo delle applicazioni. In questo caso però è più difficile personalizzare il lato grafico, inoltre bisogna sviluppare manualmente un loop per il mantenimento dell'applicazione.

Java è molto popolare per la sua inclinazione all'OOP (Object Oriented Programming), che facilita lo sviluppo di applicazioni complicate che devono gestire un numero elevato di oggetti. Dal punto di vista grafico può risultare più difficile sviluppare un'applicazione 3D, inoltre Java utilizza molte risorse (in termini di memoria) rispetto ad altri linguaggi.

4.1.3 Unity



Una delle piattaforme più popolari per lo sviluppo di videogiochi; contiene un loop logico e delle componenti grafiche (pulsanti, pannelli, animazioni, etc.) già implementate. Tramite Unity è possibile creare delle interfacce grafiche estremamente fluide e *user-friendly*, ma a causa di queste sue caratteristiche le applicazioni risultano pesanti e dipendenti dalla versione utilizzata.

4.2 Tecnologie Scelte

Per lo sviluppo dell'applicazione è stato scelto di utilizzare C++ con le librerie OpenGL e GLUT. La scelta è dovuta all'elevata capacità di personalizzazione dell'interfaccia grafica e all'ottima gestione della memoria di C++. Dal punto di vista progettuale potrebbe sembrare più complicato rispetto alle altre opzioni, questo per il fatto che ogni singola interazione dell'utente con l'interfaccia e ogni metodo di visualizzazione e memorizzazione dei dati deve essere definito manualmente. D'altra parte questa caratteristica può essere un vantaggio perché permette una facile distinzione dei metodi, dei dati e del funzio-

namento generale dell'applicazione, evitando quindi di "danneggiare" una determinata operazione o funzionamento quando si modifica un'altra componente.

Java e Unity mettono a disposizione delle librerie e componenti graficamente appaganti e facili da utilizzare, ma con un grado di personalizzazione inferiore; inoltre per raggiungere lo scopo del progetto è preferibile concentrarsi sulla complessità computazionale dell'algoritmo, scegliendo C++ infatti è possibile ottimizzare il consumo di risorse dell'applicazione, sia dal punto di vista grafico che algoritmico.

Capitolo 5

Sviluppo dell'Applicazione

5.1 Setup dell'Ambiente di Sviluppo

Per lo sviluppo del progetto è stata creata una Console Application con Visual Studio 2022. Il file di esecuzione del programma è `main.cpp`, nel quale verranno inizializzati i dati, generata una soluzione e infine aperta una finestra per mostrare all'utente il risultato.

Siccome dovranno essere gestiti diversi oggetti con strutture piuttosto complicate, lo sviluppo seguirà in parte il paradigma OOP (Object Oriented Programming); con "in parte" si intende che le figure da visualizzare, il risolutore e altri elementi per la gestione delle componenti grafiche saranno modellati da delle classi, mentre i metodi statici di utilità verranno definiti separatamente.

5.1.1 Librerie e Costanti Globali

Come detto nella sezione 4.1.3, verranno utilizzate le librerie OpenGL e GLUT per lo sviluppo dell'interfaccia grafica. In combinazione ad esse verranno utilizzate le librerie `glm` e `vector` per la gestione dei vettori e l'esecuzione di operazioni su matrici.

Inoltre sarà necessario definire alcuni parametri, tra i quali la dimensione dello schermo e la dimensione della finestra dell'applicazione, che serviranno alla creazione dell'interfaccia. Siccome verranno utilizzati frequentemente, vengono definiti in un contesto globale e vengono resi costanti, in modo da evitare delle modifiche non volute.

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <vector>
#include <iostream>
#include <time.h>
#include <direct.h>
#include <fstream>
#include <chrono>
#include <algorithm>

using namespace glm;
using namespace std;
```

```
constexpr auto SCREEN_WIDTH = 1920;
constexpr auto SCREEN_HEIGHT = 1080;

constexpr auto APP_WIDTH = 600;
constexpr auto APP_HEIGHT = 600;

constexpr auto APP_NAME = "3DKP Solver";

constexpr auto CAMERA_SPEED = 0.2f;
constexpr auto CAMERA_ZOOM = 2.0f;

constexpr auto FRAME_LENGTH = 17;
```

Da notare il parametro `FRAME_LENGTH`, si tratta della lunghezza in millisecondi di una schermata (frame); per ottenere delle animazioni fluide sarebbe meglio avere tra i 30 e i 60 FPS (Frames Per Second), infatti il valore presente è stato calcolato dividendo un secondo per 60 frames, ottenendo quindi che ogni frame dovrebbe durare circa 17 millisecondi.

5.1.2 Strutture Dati

Per prima cosa definiamo la struttura di un vertice; i vertici sono la base di ogni figura che dovrà essere visualizzata, infatti ogni pacco e il contenitore saranno caratterizzati da 8 vertici.

```
class Vertex {
  private:
    vec3 coordinates;
    vec4 color;

  public:
    Vertex(vec3 coordinates, vec4 color);
    vec3 getCoordinates() const;
    void setCoordinates(vec3 coordinates);
    vec4 getColor() const;
};
```

Successivamente definiamo una classe per modellare le figure che verranno visualizzate, che nell'esempio dei pacchi saranno dei parallelepipedi. Per ogni oggetto si vuole tenere traccia della posizione nello spazio, la lista dei vertici che lo compongono, le dimensioni, il peso e il valore:

```
class Box {
  private:
    vector<Vertex*> vertices;
    float x, y, z;
    float width, height, depth;
    float value;
    float weight;

  public:
    Box(vector<Vertex*> vertices, float value, float weight);
    vector<Vertex*> getVertices() const;
    vec3 getPosition() const;
    vec3 getSize() const;
    void move(float x, float y, float z);
    float getWeight() const;
    float getValue() const;
};
```

5.2 Implementazione dell'algoritmo

Il prossimo passo è quello di programmare l'algoritmo descritto nel Capitolo 3. Siccome si tratta di un algoritmo che richiede di eseguire molte operazioni ripetitive, cominciamo creando funzioni di utilità che permetteranno di rendere il codice più conciso e leggibile.

```
int getWeight(vector<Box*> boxes) {
  int weight = 0;
  for (Box* box : boxes)
    weight += box->getWeight();
  return weight;
}

int getProfit(vector<Box*> boxes) {
  int profit = 0;
  for (Box* box : boxes)
    profit += box->getValue();
  return profit;
}

vector<Box*> getDifference(vector<Box*> v1, vector<Box*> v2) {
  sort(v1.begin(), v1.end());
  sort(v2.begin(), v2.end());
  vector<Box*> diff;
  set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(diff));
  return diff;
}
```

Questi tre metodi servono a:

- `getWeight()` → calcolare il peso totale di una lista di pacchi
- `getValue()` → calcolare il guadagno totale di una lista di pacchi
- `getDifference()` → calcolare la differenza tra due liste di pacchi, il risultato sarà una lista contenente i pacchi in v_1 che non sono presenti in $v2$

I seguenti metodi verranno utilizzati per il calcolo delle coordinate. Da notare che ne è stato aggiunto uno non menzionato in precedenza per verificare se un pacco sarebbe "volante" in una determinata posizione, ovvero se sotto al suo centro di massa non c'è nessun altro pacco e non si trova a terra. Si tratta di un calcolo non necessario, poiché in un caso reale, seppure sotto ad un pacco non ci fosse un appoggio diretto, potrebbe comunque essere posizionato diagonalmente o ruotandolo, ma siccome in questa applicazione le rotazioni non vengono considerate il metodo è stato incluso per un eventuale utilizzo.

```
1  static bool collides(vector<Box*> placedBoxes, Box* box, vec3 boxPosition, vec3
       containerSize) {
2    for (Box* placedBox : placedBoxes) {
3      vec3 pbPos = placedBox->getPosition();
4      vec3 pbSize = placedBox->getSize();
5      bool xCheck = pbPos.x + pbSize.x <= boxPosition.x || boxPosition.x + box->getSize().
           x <= pbPos.x;
6      bool yCheck = pbPos.y + pbSize.y <= boxPosition.y || boxPosition.y + box->getSize().
           y <= pbPos.y;
7      bool zCheck = pbPos.z + pbSize.z <= boxPosition.z || boxPosition.z + box->getSize().
           z <= pbPos.z;
8      bool collision = !(xCheck || yCheck || zCheck);
9      if (collision || boxPosition.x + box->getSize().x > containerSize.x || boxPosition.y
            + box->getSize().y > containerSize.y || boxPosition.z + box->getSize().z >
           containerSize.z)
10       return true;
11   }
12   return false;
13 }
14
15 bool isFlying(vector<Box*> placedBoxes, Box* box, vec3 position) {
16   if (position.y == 0)
17     return false;
18   vec3 size = box->getSize();
19   vec3 midPoint = vec3(position.x + size.x / 2, position.y, position.z + size.z / 2);
20   for (Box* placedBox : placedBoxes) {
21     vec3 pbPos = placedBox->getPosition();
22     vec3 pbSize = placedBox->getSize();
23     if (pbPos.y + pbSize.y == midPoint.y
24       && pbPos.x <= midPoint.x && midPoint.x <= pbPos.x + pbSize.x
25       && pbPos.z <= midPoint.z && midPoint.z <= pbPos.z + pbSize.z) {
26       return false;
27     }
28   }
29   return true;
30 }
```

```
int getMaxX(vector<Box*> placedBoxes, Box* box, vec3 containerSize) {
  vector<Box*> px;
  std::copy_if(placedBoxes.begin(), placedBoxes.end(), std::back_inserter(px), [&
      containerSize](Box* pBox) {
    return pBox->getPosition().x + pBox->getSize().x <= containerSize.x - pBox->getSize
        ().x;
  });
  int maxX = 0;
  for (int i = 0; i < px.size(); i++) {
    int x = px[i]->getPosition().x + px[i]->getSize().x;
    if (x > maxX)
      maxX = x;
  }
  return maxX;
}

int getMaxY(vector<Box*> placedBoxes, Box* box, vec3 containerSize) {
  vector<Box*> py;
  std::copy_if(placedBoxes.begin(), placedBoxes.end(), std::back_inserter(py), [&
      containerSize](Box* pBox) {
    return pBox->getPosition().y + pBox->getSize().y <= containerSize.y - pBox->getSize
        ().y;
  });
  int maxY = 0;
  for (int i = 0; i < py.size(); i++) {
    int y = py[i]->getPosition().y + py[i]->getSize().y;
    if (y > maxY)
      maxY = y;
  }
  return maxY;
}

int getMaxZ(vector<Box*> placedBoxes, Box* box, vec3 containerSize) {
  vector<Box*> pz;
  std::copy_if(placedBoxes.begin(), placedBoxes.end(), std::back_inserter(pz), [&
      containerSize](Box* pBox) {
    return pBox->getPosition().z + pBox->getSize().z <= containerSize.z - pBox->getSize
        ().z;
  });
  int maxZ = 0;
  for (int i = 0; i < pz.size(); i++) {
    int z = pz[i]->getPosition().z + pz[i]->getSize().z;
    if (z > maxZ)
      maxZ = z;
  }
  return maxZ;
}
```

I metodi `getMaxX`, `getMaxY`, `getMaxZ` verranno utilizzati per calcolare delle eventuali coordinate alternative, ovvero quelle più a destra, in alto e di fronte possibili rispetto ai pacchi inseriti in precedenza. Combinando tutti i frammenti di codice sviluppati finora otteniamo il metodo per calcolare tutte le possibili posizioni in cui il pacco può essere inserito.

```
vector<vec3> getAvailablePositions(vector<Box*> placedBoxes, Box* box, vec3
    containerSize) {
  vector<vec3> positions;
  for (Box* placedBox : placedBoxes) {
    int xi = placedBox->getPosition().x;
    int xRight = xi + placedBox->getSize().x;
    int yi = placedBox->getPosition().y;
    int yUp = yi + placedBox->getSize().y;
    int zi = placedBox->getPosition().z;
    int zFront = zi + placedBox->getSize().z;
    if (!collides(placedBoxes, box, vec3(xRight, yi, zi), containerSize) && !isFlying(
        placedBoxes, box, vec3(xRight, yi, zi)))
      positions.push_back(vec3(xRight, yi, zi));
    if (!collides(placedBoxes, box, vec3(xi, yUp, zi), containerSize) && !isFlying(
        placedBoxes, box, vec3(xRight, yi, zi)))
      positions.push_back(vec3(xi, yUp, zi));
    if (!collides(placedBoxes, box, vec3(xi, yi, zFront), containerSize) && !isFlying(
        placedBoxes, box, vec3(xi, yi, zFront)))
      positions.push_back(vec3(xi, yi, zFront));
  }
  int xValues[] = { 0, getMaxX(placedBoxes, box, containerSize) };
  int yValues[] = { 0, getMaxY(placedBoxes, box, containerSize) };
  int zValues[] = { 0, getMaxZ(placedBoxes, box, containerSize) };
  for (int j = 0; j < 2; ++j) {
    for (int k = 0; k < 2; ++k) {
      for (int l = 0; l < 2; ++l) {
        int x = xValues[j];
        int y = yValues[k];
        int z = zValues[l];
        if (!collides(placedBoxes, box, vec3(x, y, z), containerSize) && !isFlying(
            placedBoxes, box, vec3(x, y, z)))
          positions.push_back(vec3(x, y, z));
      }
    }
  }
  return positions;
}
```

Possiamo ora creare la funzione `CalculateCoordinates` per il calcolo delle coordinate per ogni pacco descritto nella sezione 3.1:

```
vec3 getCoordinates(vector<Box*> placedBoxes, Box* box, vec3 containerSize) {
  if (placedBoxes.size() == 0)
    return vec3(0);

  vector<vec3> positions = getAvailablePositions(placedBoxes, box, containerSize);
  if (positions.size() == 0)
    return vec3(-1);

  // Sort by depth
  sort(positions.begin(), positions.end(), [](vec3 a, vec3 b) { return a.z < b.z; });
  map<int, vector<vec3>> coordinatesMap;
  for (vec3 position : positions)
    coordinatesMap[position.z].push_back(position);
  vector<vec3> backPositions = coordinatesMap[positions[0].z];
  if (backPositions.size() == 1)
    return backPositions[0];

  // Sort by vertical position
  sort(backPositions.begin(), backPositions.end(), [](vec3 a, vec3 b) { return a.y < b.y
      ; });
  coordinatesMap = {};
  for (vec3 position : backPositions)
    coordinatesMap[position.y].push_back(position);
  vector<vec3> bottomPositions = coordinatesMap[backPositions[0].y];
  if (bottomPositions.size() == 1)
    return bottomPositions[0];

  // Sort by horizontal position
  sort(bottomPositions.begin(), bottomPositions.end(), [](vec3 a, vec3 b) { return a.x <
       b.x; });
  coordinatesMap = {};
  for (vec3 position : bottomPositions)
    coordinatesMap[position.x].push_back(position);
  vector<vec3> leftPositions = coordinatesMap[bottomPositions[0].x];
  return leftPositions[0];
}
```

Infine rimane soltanto da creare dei metodi utili alla gestione delle soluzioni, in particolare per creare una soluzione iniziale e una soluzione vicina. Per la soluzione iniziale è stato scelto di utilizzare un ciclo che crea delle permutazioni dei pacchi finché non ne viene generata una che rispetti i limiti di spazio e di peso. Per quanto riguarda le soluzioni "vicine" invece è sufficiente inserire, rimuovere o cambiare uno dei pacchi senza controllare i limiti, questo perché le permutazioni non valide verranno trattate come se avessero un guadagno inferiore, con la differenza che verrà saltata la parte probabilistica.

```
vector<Box*> createInitialSolution(vector<Box*> boxes, int maxWeight, vec3 containerSize
    ) {
  vector<Box*> solution;
  do {
    solution.clear();
    for (Box* box : boxes) {
      if (rand() % 2 == 1)
        solution.push_back(box);
    }
  } while (getWeight(solution) > maxWeight || !fits(solution, containerSize));
  return solution;
}

vector<Box*> createNeighborSolution(vector<Box*> currentSolution, vector<Box*>
    remainingBoxes) {
  if (currentSolution.size() == 0)
    return { remainingBoxes[rand() % remainingBoxes.size()] };

  vector<Box*> neighbor = currentSolution;
  switch (rand() % 3) {
    case 0:
      neighbor.erase(neighbor.begin() + rand() % neighbor.size());
      break;
    case 1:
      neighbor.push_back(remainingBoxes[rand() % remainingBoxes.size()]);
      break;
    case 2:
      neighbor[rand() % neighbor.size()] = remainingBoxes[rand() % remainingBoxes.size()
          ];
      break;
  }
  return neighbor;
}
```

Altri metodi che saranno utili sono i seguenti:

- `getBoxesCoordinates` → ottiene un vettore di coordinate corrispondenti ai pacchi in una lista
- `createSolutionFromBoxes` → crea una soluzione, ovvero una lista di coppie pacco-coordinate

Verrebbe naturale chiedersi se non fosse più comodo utilizzare direttamente le coordinate salvate in ogni oggetto (pacco) piuttosto che salvare delle coppie in questo modo. Il motivo di questa scelta è che, usando le coordinate interne, bisognerebbe obbligatoriamente creare una copia di ogni oggetto per ogni soluzione, copiando quindi diversi altri parametri che non ci interessano e occupando memoria inutilmente. In questo modo invece viene salvato soltanto il puntatore all'oggetto e un `vec3` che rappresenta le sue coordinate in una determinata soluzione.

```
vector<vec3> getBoxesCoordinates(vector<Box*> boxes) {
  vector<vec3> coords;
  for (Box* box : boxes)
    coords.push_back(box->getPosition());
  return coords;
}

vector<pair<Box*, vec3>> createSolutionFromBoxes(vector<Box*> boxes) {
  vector<pair<Box*, vec3>> solution;
  for (Box* box : boxes)
    solution.push_back({ box, box->getPosition() });
  return solution;
}
```

Inserendo i metodi precedenti nell'algoritmo di Simulated Annealing otteniamo:

```
vector<pair<Box*, vec3>> solve3D(vector<Box*> boxes) {
  vector<Box*> currentSolution = createInitialSolution(boxes, this->maxWeight, this->
      containerSize);
  if (currentSolution.size() == boxes.size())
    return createSolutionFromBoxes(currentSolution);
  vector<vector<Box*>> solutionsBoxes = { currentSolution };
  vector<vector<vec3>> solutionsPositions = { getBoxesCoordinates(currentSolution) };

  for (int i = MAX_ITERATIONS; i > 0; i--) {
    vector<Box*> boxesLeft = getDifference(boxes, currentSolution);
    vector<Box*> neighbor = createNeighborSolution(currentSolution, boxesLeft);
    if (getWeight(neighbor) > this->maxWeight || !fits(neighbor, this->containerSize))
      continue;

    int currentProfit = getProfit(currentSolution);
    int neighborProfit = getProfit(neighbor);
    bool accept = false;
    if (neighborProfit >= currentProfit) {
      accept = true;
    } else {
      float p = rand() % 1001 / 1000.0f;
      int delta = neighborProfit - currentProfit;
      if (exp(delta / i) > p)
        accept = true;
    }
    if (accept) {
      currentSolution = neighbor;
      auto it = find(solutionsBoxes.begin(), solutionsBoxes.end(), neighbor);
      if (it == solutionsBoxes.end()) {
        solutionsBoxes.push_back(neighbor);
        solutionsPositions.push_back(getBoxesCoordinates(neighbor));
      }
    }
  }
  vector<Box*> maxBoxes = {};
  vector<vec3> maxPositions = {};
  for (int i = 0; i < solutionsBoxes.size(); i++) {
    if (getProfit(solutionsBoxes[i]) > getProfit(maxBoxes)) {
      maxBoxes = solutionsBoxes[i];
      maxPositions = solutionsPositions[i];
    }
  }
  vector<pair<Box*, vec3>> finalSolution;
  for (int i = 0; i < maxBoxes.size(); i++)
    finalSolution.push_back({ maxBoxes[i], maxPositions[i] });
  return finalSolution;
}
```

5.3 Interfaccia Grafica

L'interfaccia grafica deve essere intuitiva e facile da utilizzare, l'obbiettivo è mostrare un'animazione fluida della soluzione del 3DKP a livello grafico, ovvero visualizzando un contenitore che viene gradualmente riempito con degli oggetti.

Inizialmente si dovrà vedere un contenitore vuoto e degli oggetti di forma rettangolare vicino ad esso. La soluzione ottenuta con l'algoritmo implementato nella sezione precedente verrà visualizzata inserendo uno alla volta i pacchi dentro al contenitore.

5.3.1 Shaders

Il primo passo è la programmazione delle shaders, ovvero dei file di codice che detteranno come i vertici dovranno essere posizionati sullo schermo e colorati:

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec4 aColor;

uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;
uniform vec3 viewPos;

out vec4 ourColor;

void main()
{
  gl_Position = projection*view*model*vec4(aPos, 1.0);
    ourColor = aColor;
}
```

```
#version 330 core

in vec4 ourColor;
out vec4 FragColor;

void main()
{
    FragColor = ourColor;
}
```

Nel primo file (vertex shader) viene ricevuto in input la posizione del vertice, il colore e le matrici di modellazione, proiezione e vista. La posizione del vertice e le matrici verranno utilizzati per calcolare in quale posizione dello schermo dovranno essere colorati i relativi pixel e con quale colore; quest'ultimo viene gestito nel secondo file (fragment shader), che determinerà il colore sfumandolo in base ai colori degli altri vertici dell'oggetto. Se ad esempio assegnassimo il colore blu ai vertici della faccia anteriore di un pacco e rosso a quelli della faccia posteriore, vedremmo il colore delle facce laterali cambiare gradualmente da blu a rosso.

I file di Shader verranno utilizzati per creare l'id del programma e, trattandosi di file di codice, è necessario compilarli prima di poterli utilizzare:

```
char* readShaderSource(char* shaderFile) {
  char cwd[1024];
    if (_getcwd(cwd, sizeof(cwd)) == nullptr) {
        fprintf(stderr, "ERROR - LOADING CURRENT DIRECTORY FAILED\n");
        exit(-1);
    }
    strcat_s(cwd, sizeof(cwd), shaderFile);
    FILE* file;
    errno_t err = fopen_s(&file, cwd, "rb");
    if (err != 0 || file == nullptr) {
        fprintf(stderr, "ERROR %d - FILE LOADING FAILED (%s)\n", err, cwd);
        exit(-1);
    }
    if (fseek(file, 0L, SEEK_END) != 0) {
        fprintf(stderr, "ERROR - SEEKING END OF FILE FAILED (%s)\n", cwd);
        fclose(file);
        exit(-1);
    }
    long size = ftell(file);
    if (size == -1L) {
        fprintf(stderr, "ERROR - FILE SIZE (%s)\n", cwd);
        fclose(file);
        exit(-1);
    }
    if (fseek(file, 0L, SEEK_SET) != 0) {
        fprintf(stderr, "ERROR - SEEKING START OF FILE FAILED (%s)\n", cwd);
        fclose(file);
        exit(-1);
    }
    char* buf = new(std::nothrow) char[size + 1];
    if (buf == nullptr) {
        fprintf(stderr, "ERROR - ALLOCATING FRAME BUFFER FAILED (%s)\n", cwd);
        fclose(file);
        exit(-1);
```

```
    }
    size_t bytesRead = fread(buf, 1, size, file);
    if (bytesRead != size) {
        fprintf(stderr, "ERROR - READING FILE FAILED (%s)\n", cwd);
        delete[] buf;
        fclose(file);
        exit(-1);
    }
    buf[size] = '\0';
    fclose(file);
    return buf;
}
```

```
GLuint createProgram(char* vertexShaderFile char* fragmentShaderFile) {
  int success;
  char infoLog[512];
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

  GLchar* vertexShader = readShaderSource(vertexShaderFile);
  GLuint vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShaderId, 1, (const char**)&vertexShader, NULL);
  glCompileShader(vertexShaderId);

  glGetShaderiv(vertexShaderId, GL_COMPILE_STATUS, &success);
  if (!success) {
    glGetShaderInfoLog(vertexShaderId, 512, NULL, infoLog);
    fprintf(stderr, "ERROR - VERTEX SHADER COMPILATION FAILED\n %s \n", infoLog);
  }

  GLchar* fragmentShader = readShaderSource(fragmentShaderFile);
  GLuint fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShaderId, 1, (const char**)&fragmentShader, NULL);
  glCompileShader(fragmentShaderId);

  glGetShaderiv(fragmentShaderId, GL_COMPILE_STATUS, &success);
  if (!success) {
    glGetShaderInfoLog(fragmentShaderId, 512, NULL, infoLog);
    fprintf(stderr, "ERROR - FRAGMENT SHADER COMPILATION FAILED\n %s \n", infoLog);
  }

  GLuint programId = glCreateProgram();
  glAttachShader(programId, vertexShaderId);
  glAttachShader(programId, fragmentShaderId);
  glLinkProgram(programId);

  return programId;
}
```

5.3.2 Interfaccia e Input Utente

Definiamo una classe per poter creare e gestire l'interfaccia effettiva:

```
class GlutManager {
  private:
    static GlutManager* instance;

    GLuint programId;
    unsigned int projectionMatrixUniform;
    unsigned int viewMatrixUniform;
    unsigned int modelMatrixUniform;
    unsigned int viewPositionUniform;
    mat4 projectionMatrix;
    mat4 viewMatrix;
    vector<Box*> boxes;
    Camera* camera;
    ShadersManager* shadersManager;
    bool insert;
    bool pause;

    static void drawScene(void);
    static void moveCamera(unsigned char key, int x, int y);
    static void zoomCamera(int wheel, int direction, int x, int y);
    static void lookAround(int x, int z);
    static void update(int value);

  public:
    GlutManager(vector<Box*> boxes);
    void openWindow(int argc, char** argv);
};
```

Da notare che ogni oggetto `GlutManager` contiene un'istanza di se stesso, essa è necessaria per l'utilizzo dei metodi di GLUT poiché devono essere obbligatoriamente statici, quindi per accedere ai campi privati bisognerà utilizzare l'istanza invece che il riferimento `this`.

I metodi GLUT utilizzati sono i seguenti:

- Caricamento della scena (`drawScene`)
- Pressione di un tasto della tastiera (`moveCamera`)
- Utilizzo della rotellina del mouse (`zoomCamera`)
- Movimento del cursore all'interno della schermata (`lookAround`)

- Aggiornamento della scena (`update`)

Analizziamo i parametri utilizzati dal `GlutManager`:

- `projectionMatrix` → utilizzata per creare la visuale tramite campo visivo (FOV), rapporto tra larghezza e altezza della finestra (aspectRatio) e i piani visivi (vicino e lontano) che limitano la vista.
- `viewMatrix` → utilizzata per la gestione della telecamera, contiene 3 vettori: uno per la posizione, uno per il punto verso il quale la telecamera guarda e uno per l'UpVector (vettore che indica l'orientamento verticale della scena).
- variabili `uniform` → vengono utilizzate come riferimenti in memoria per accedere ai dati delle relative matrici salvati nella memoria della GPU ed eseguire operazioni su di esse.
- `shadersManager` → utilizzato per la creazione dell'ID del programma e la compilazione delle shaders, contiene i metodi descritti nella sezione 5.3.1.
- `pause` → utilizzato per interrompere e far ripartire l'animazione.
- `insert` → utilizzato per differenziare l'animazione di inserimento da quella di rimozione dei pacchi dal contenitore.

Vediamo ora come avviene l'inizializzazione dell'interfaccia:

```
void GlutManager::openWindow(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitContextVersion(4, 0);
  glutInitContextProfile(GLUT_CORE_PROFILE);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowSize(APP_WIDTH, APP_HEIGHT);
  glutInitWindowPosition(50, 50);
  glutCreateWindow(APP_NAME);

  glutDisplayFunc(drawSceneAccessor);
  glutKeyboardFunc(movementAccessor);
  glutMouseWheelFunc(zoomAccessor);
  glutPassiveMotionFunc(lookAroundAccessor);
  glutTimerFunc(FRAME_LENGTH, update, 0);

  glewExperimental = GL_TRUE;
  glewInit();
  GLuint programId = createProgram(
    (char*)"\\src\\shaderFiles\\vertexShader.glsl",
    (char*)"\\src\\shaderFiles\\fragmentShader.glsl"
  );

  for (Box* shape : this->boxes)
    shape->init();

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_BLEND);
  glEnable(GL_ALPHA_TEST);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

  this->projectionMatrixUniform = glGetUniformLocation(programId, "projection");
  this->modelMatrixUniform = glGetUniformLocation(programId, "model");
  this->viewMatrixUniform = glGetUniformLocation(programId, "view");
  this->viewPositionUniform = glGetUniformLocation(programId, "viewPos");

  glutMainLoop();
}
```

- Righe 2-8 → inizializzazione della finestra utilizzando i parametri definiti in `utils.hpp`
- Righe 10-14 → binding dei metodi statici per gli input utente e l'aggiornamento della scena
- Righe 16-21 → inizializzazione di GLUT e creazione dell'ID del programma utilizzando i file di shaders
- Righe 23-24 → inizializzazione degli oggetti da mostrare nella scena
- Righe 26-29 → abilitazione della profondità e delle miscele di colori
- Righe 31-36 → binding delle variabili `uniform` e avvio del main loop

Per poter visualizzare una figura sullo schermo bisogna anche aggiungere dei parametri alla classe `Box`:

- `VAO` → Vertex Array Object: contiene il formato dei dati dei vertici e i VBO
- `verticesVBO` e `colorsVBO` → Vertex Buffer Object: contiene i dati (posizione e colore) dei vertici
- `EBO` → Element Buffer Object: contiene gli indici di riferimento per i vertici; ogni faccia di un oggetto 3D viene creata utilizzando dei triangoli, perciò nella creazione della faccia di frontale di un cubo bisogna definire 6 vertici, di cui 2 di questi sarebbero ripetuti. Per risparmiare memoria viene utilizzato un vettore di indici nel quale vengono definiti, in triplette, i vertici che devono comporre ogni triangolo.
- `model` → matrice di modellazione, utilizzata per la creazione dei modelli; all'interno di essa sono presenti le informazioni riguardo la traslazione, rotazione e moltiplicatori di dimensioni (scala) dell'oggetto

```
class Box {
  private:
    GLuint VAO;
    GLuint verticesVBO;
    GLuint colorsVBO;
    GLuint EBO;
    mat4 model;
    vector<Vertex*> vertices;
    vector<GLuint> indices;
    int id;
    float x, y, z;
    float xTarget, yTarget, zTarget;
    float xStart, yStart, zStart;
    float width, height, depth;
    float value;
    float weight;
    vec4 anchorWorld;
    vec4 anchorObj;

  public:
    Box(vector<Vertex*> vertices, vector<GLuint> indices, int id, float value, float
        weight);
    int getId() const;
    GLuint* getVAO();
    GLuint* getVerticesVBO();
    GLuint* getColorsVBO();
    GLuint* getEBO();
    vector<Vertex*> getVertices() const;
    vector<vec3> getVerticesCoordinates() const;
    vector<vec4> getVerticesColors() const;
    vector<GLuint> getIndices() const;
    mat4 getModel() const;
    void setModel(mat4 model);
    vec3 getPosition() const;
    void setPosition(vec3 position);
    vec3 getSize() const;
    void move(float x, float y, float z);
    float getWeight() const;
    float getValue() const;
    void init();
    bool targetReached() const;
    void moveTowardsTarget();
    bool startReached() const;
    void moveTowardsStart();
};
```

In particolare l'inizializzazione di un oggetto da visualizzare avviene in questo modo:

```
void initBox(Box* box) {
    glGenVertexArrays(1, box->getVAO());
    glBindVertexArray(*box->getVAO());

    glGenBuffers(1, box->getVerticesVBO());
    glBindBuffer(GL_ARRAY_BUFFER, *box->getVerticesVBO());
    glBufferData(
        GL_ARRAY_BUFFER,
        box->getVerticesCoordinates().size() * sizeof(vec3),
        box->getVerticesCoordinates().data(),
        GL_STATIC_DRAW
    );
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
    glEnableVertexAttribArray(0);

    glGenBuffers(1, box->getColorsVBO());
    glBindBuffer(GL_ARRAY_BUFFER, *box->getColorsVBO());
    glBufferData(
        GL_ARRAY_BUFFER,
        box->getVerticesColors().size() * sizeof(vec4),
        box->getVerticesColors().data(),
        GL_STATIC_DRAW
    );
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)0);
    glEnableVertexAttribArray(1);

    glGenBuffers(1, box->getEBO());
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *box->getEBO());
    glBufferData(
        GL_ELEMENT_ARRAY_BUFFER,
        box->getIndices().size() * sizeof(GLuint),
        box->getIndices().data(),
        GL_STATIC_DRAW
    );
}
```

Il primo parametro di `glVertexAttribPointer` è l'indice dei dati, dove 0 sono le coordinate e 1 sono i colori. I VBO contengono il riferimento in memoria dove è salvato il primo vertice della lista; ognuno di essi è caratterizzato da 7 valori divisi in coordinate (x, y, z) e colore (r, g, b, a). Per le coordinate deve quindi essere definita una "lunghezza" dei dati pari a 3 byte, perciò per ogni 7 byte i primi 3 vengono attribuiti a (x, y, z), di conseguenza i 4 successivi saranno attribuiti (r, g, b, a).

In `drawScene` avviene la renderizzazione della scena. Per ogni frame viene pulito lo schermo, impostato l'ID del programma da utilizzare e vengono disegnati gli oggetti. Da notare che la visualizzazione del contenitore è diversa da quella degli altri oggetti, infatti è stato impostato in modo da potergli vedere attraverso, mentre le facce dei pacchi vengono riempite.

```
void GlutManager::drawScene(void) {
  this->projectionMatrix = perspective(
    radians(this->camera->getPerspective()->getFOV()),
    this->camera->getPerspective()->getAspectRatio(),
    this->camera->getPerspective()->getNearPlane(),
    this->camera->getPerspective()->getFarPlane()
  );
  this->viewMatrix = lookAt(
    vec3(this->camera->getView()->getPosition()),
    vec3(this->camera->getView()->getTarget()),
    vec3(this->camera->getView()->getUpvector())
  );

  glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Background Color
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glUseProgram(this->shadersManager->getProgramId());
  glUniformMatrix4fv(this->projectionMatrixUniform, 1, GL_FALSE, value_ptr(this->
      projectionMatrix));
  glUniformMatrix4fv(this->viewMatrixUniform, 1, GL_FALSE, value_ptr(this->viewMatrix));
  glPointSize(10.0f);

  Box* container = this->boxes[0];
  glUniformMatrix4fv(this->modelMatrixUniform, 1, GL_FALSE, value_ptr(container->
      getModel()));
  glBindVertexArray(*container->getVAO());
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  glDrawElements(GL_LINES, (container->getIndices().size()) * sizeof(GLuint),
      GL_UNSIGNED_INT, 0);
  glBindVertexArray(0);

  for (int i = 1; i < this->boxes.size(); i++) {
    Box* box = this->boxes[i];
    glUniformMatrix4fv(this->modelMatrixUniform, 1, GL_FALSE, value_ptr(box->getModel())
        );
    glBindVertexArray(*box->getVAO());
    glPolygonMode(GL_FRONT_AND_BACK, this->polygonMode);
    glDrawElements(this->elementsMode, (box->getIndices().size()) * sizeof(GLuint),
        GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
  }
  glutSwapBuffers();
}
```

In `moveCamera` viene catturato l'evento di pressione di un tasto della tastiera per il movimento della telecamera:

- `W S` → movimento lungo l'asse Z
- `A D` → movimento lungo l'asse X
- `Q E` → movimento lungo l'asse Y

Inoltre sarebbe comodo avere dei comandi per mettere in pausa l'animazione o tornare indietro nel procedimento di inserimento, in modo tale che l'utente possa riguardare i passaggi a proprio piacimento:

- `P` → attiva/disattiva la pausa
- `C` → attiva l'inserimento dei pacchi nel contenitore
- `B` → attiva la rimozione dei pacchi dal contenitore

Infine premendo il pulsante `ESC` si interrompe l'esecuzione del programma.

```
void GlutManager::moveCamera(unsigned char key, int x, int y) {
  vec4 direction = instance->camera->getView()->getDirection();
  vec4 upVector = instance->camera->getView()->getUpvector();

  vec3 horizontalMovement = cross(vec3(direction), vec3(upVector)) * CAMERA_SPEED;
  vec3 verticalMovement = upVector * CAMERA_SPEED;
  vec3 depthMovement = vec3(direction) * CAMERA_SPEED;

  switch (key)
  {
    case 'a': case 'A': // Left
      instance->camera->getView()->move(vec4(-horizontalMovement, 1.0f));
      break;
    case 'd': case 'D': // Right
      instance->camera->getView()->move(vec4(horizontalMovement, 1.0f));
      break;
    case 'e': case 'E': // Up
      instance->camera->getView()->move(vec4(verticalMovement, 1.0f));
      break;
    case 'q': case 'Q': // Down
      instance->camera->getView()->move(vec4(-verticalMovement, 1.0f));
      break;
    case 'w': case 'W': // Forward
      instance->camera->getView()->move(vec4(depthMovement, 1.0f));
      break;
    case 's': case 'S': // Backward
      instance->camera->getView()->move(vec4(-depthMovement, 1.0f));
      break;
    case 'p': case 'P': // Pause/Unpause
      instance->pause = !instance->pause;
      break;
    case 'c': case 'C': // Start placing in the container
      instance->pause = false;
      instance->insert = true;
      break;
    case 'b': case 'B': // Start removing from the container
      instance->pause = false;
      instance->insert = false;
      break;
    case 27:      // Exit
      glutLeaveMainLoop();
      break;
    default:
      break;
  }
}
```

In `lookAround` e `zoomCamera` vengono gestiti il movimento e lo zoom della telecamera tramite, rispettivamente, movimento del cursore e utilizzo della rotellina del mouse. Per evitare che il cursore esca dalla schermata, viene tenuto sempre al centro tramite `glutWarpPointer`.

```
void GlutManager::lookAround(int x, int y) {
  // Initialization
  static int mouseX = x;
  static int mouseY = y;
  static float theta = -90.0f;
  static float phi = 0.0f;

  float centerX = APP_WIDTH / 2.0f;
  float centerY = APP_HEIGHT / 2.0f;
  mouseX = x;
  mouseY = y;

  float xoffset = (x - centerX) * 0.05f;
  float yoffset = (APP_HEIGHT - y - centerY) * 0.05f;
  theta += xoffset;
  phi += yoffset;

  if (phi > 90.0f) phi = 90.0f;
  if (phi < -90.0f) phi = -90.0f;
  vec3 front = vec3(
    cos(radians(theta)) * cos(radians(phi)),
    sin(radians(phi)),
    sin(radians(theta)) * cos(radians(phi))
  );
  vec4 position = instance->camera->getView()->getPosition();
  vec4 direction = vec4(normalize(front), 0.0f);
  instance->camera->getView()->setTarget(vec3(position + direction));

  // Keep cursor on the center of the window
  glutWarpPointer(centerX, centerY);
}
```

Similmente la prospettiva viene resettata se con lo zoom si eccede un angolo eccessivo, che causerebbe la distorsione dell'immagine.

```
void GlutManager::zoomCamera(int wheel, int direction, int x, int y) {
  instance->camera->getPerspective()->zoom(CAMERA_ZOOM * -direction);
}

void Perspective::zoom(float zoom) {
  this->fov += zoom;

  if (this->fov < 1.0f)
    this->fov = 1.0f;
  if (this->fov > 120.0f)
    this->fov = 120.0f;
}
```

Infine in `update` viene gestito l'aggiornamento continuo della schermata. Tra un frame e l'altro vengono aggiornate le posizioni dei pacchi per creare un'animazione di movimento da fuori dal contenitore a dentro (nella posizione calcolata).

```
void GlutManager::update(int value) {
  if (!instance->pause) {
    if (instance->insert) {
      for (int i = 0; i < instance->boxes.size(); i++) {
        if (!instance->boxes[i]->targetReached()) {
          instance->boxes[i]->moveTowardsTarget();
          break;
        }
      }
    } else {
      for (int i = instance->boxes.size() - 1; i >= 0; i--) {
        if (!instance->boxes[i]->startReached()) {
          instance->boxes[i]->moveTowardsStart();
          break;
        }
      }
    }
  }
  glutTimerFunc(FRAME_LENGTH, update, 0);
  glutPostRedisplay();
}

bool Box::targetReached() const {
  return abs(this->xTarget - this->x) <= 0.001 && abs(this->yTarget - this->y) <= 0.001
      && abs(this->zTarget - this->z) <= 0.001;
}
```

```
26
27 void Box::moveTowardsTarget() {
28   float xStep = (this->xTarget - this->xStart) / (FRAME_LENGTH * 3);
29   float yStep = (this->yTarget - this->yStart) / (FRAME_LENGTH * 3);
30   float zStep = (this->zTarget - this->zStart) / (FRAME_LENGTH * 3);
31   this->move(xStep, yStep, zStep);
32 }
33
34 bool Box::startReached() const {
35   return abs(this->xStart - this->x) <= 0.001 && abs(this->yStart - this->y) <= 0.001 &&
         abs(this->zStart - this->z) <= 0.001;
36 }
37
38 void Box::moveTowardsStart() {
39   float xStep = (this->xStart - this->xTarget) / (FRAME_LENGTH * 3);
40   float yStep = (this->yStart - this->yTarget) / (FRAME_LENGTH * 3);
41   float zStep = (this->zStart - this->zTarget) / (FRAME_LENGTH * 3);
42   this->move(xStep, yStep, zStep);
43 }
```

5.4 Esempio di Utilizzo

In questa sezione verrà mostrato un esempio di come può essere utilizzata l'applicazione. La prima operazione eseguita è la lettura di un file nel quale sono presenti i dati dei pacchi e del contenitore.

I dati nel file devono essere in ordine:

1. Dimensioni del contenitore
2. Peso massimo del contenitore
3. Dati dei pacchi, dove ogni numero corrisponde rispettivamente a:
   (a) Larghezza
   (b) Altezza
   (c) Lunghezza
   (d) Valore
   (e) Peso
   (f) Colore → i 3 valori finali corrispondono a (r, g, b)

```
9 9 9
30
2 4 9 3 5 1 0 0
3 7 3 5 3 0 1 0
3 7 6 4 1 0 0 1
5 2 3 7 8 1 1 0
5 2 6 9 9 1 0 1
6 3 3 3 4 0 1 1
6 3 6 5 4 0.7 0.2 0.2
4 6 3 6 7 0.2 0.7 0.2
4 6 6 6 4 0.2 0.2 0.7
```

Figura 5.1: Esempio di file (.txt) per l'esecuzione del programma

```
char cwd[1024];
if (_getcwd(cwd, sizeof(cwd)) == nullptr) {
  fprintf(stderr, "ERROR LOADING CURRENT DIRECTORY\n");
  exit(-1);
}
strcat_s(cwd, sizeof(cwd), "\\src\\main\\kp.txt");

ifstream file(cwd);
if (!file) {
  std::cerr << "ERROR LOADING FILE" << endl;
  exit(-1);
}

vec3 containerSize = vec3(0);
int maxWeight;
file >> containerSize.x >> containerSize.y >> containerSize.z;
file >> maxWeight;

vector<Box*> boxes;
int width, height, depth, profit, weight;
float r, g, b;
int i = 1;
while (file >> width >> height >> depth >> profit >> weight >> r >> g >> b) {
  auto bVal = ShapeBuilder::createBox(width, height, depth, vec4(r, g, b, 1));
  Box* shape = new Box(bVal.first, bVal.second, i, profit, weight);
  i++;
  boxes.push_back(shape);
}
```

Tramite `_getcwd` possiamo ottenere il percorso della directory corrente, ad esso aggiungiamo il percorso del file di testo con i dati (in questo esempio si trova in `src/main/`). Successivamente con un ciclo `while` possiamo leggere tutti i dati nel file e creare i pacchi con cui verranno eseguiti i calcoli e che verranno visualizzati alla fine.

Successivamente si procede alla soluzione del 3DKP con i dati forniti, si stampa il risultato e si apre l'interfaccia grafica per mostrare il risultato:

```
srand(time(NULL));
KnapsackSolver* ks = new KnapsackSolver(containerSize, maxWeight);

auto start = std::chrono::high_resolution_clock::now();
vector<pair<Box*, vec3>> solution = ks->solve3D(boxes);
auto end = std::chrono::high_resolution_clock::now();

chrono::duration<double> elapsed = end - start;
cout << "Elapsed Time: " << elapsed.count() << " seconds" << endl;

vector<Box*> scene;
auto containerValues = ShapeBuilder::createBox(containerSize.x, containerSize.y,
    containerSize.z, vec4(1.0f));
Box* container = new Box(containerValues.first, containerValues.second, 0, 0.0f, 0.0f);
scene.push_back(container);

cout << "Boxes: ";
int totalProfit = 0, totalWeight = 0;
for (pair<Box*, vec3> box : solution) {
  cout << box.first->getId() << " ";
  totalProfit += box.first->getValue();
  totalWeight += box.first->getWeight();

  box.first->setPosition(box.second);
  box.first->setTarget(box.second);
  box.first->setStartPosition(vec3(box.second.x, box.second.y, rand() % 20 + 10));
  scene.push_back(box.first);
}
cout << "\nProfit: " << totalProfit << "\tWeight: " << totalWeight << "/" << maxWeight
    << endl;

GlutManager* glutManager = new GlutManager(scene);
glutManager->openWindow(argc, argv);
```

Da notare l'utilizzo di `srand(time(NULL))`, che serve a creare un *seed* per la generazione di valori casuali. Questo frammento di codice è di fondamentale importanza perché assicura che ad ogni esecuzione del programma tutti i numeri generati casualmente siano diversi dalle esecuzioni precedenti.

Capitolo 6

Testing dell'Applicazione

Per verificare al meglio l'efficienza dell'algoritmo verranno prima utilizzati dei dati "base", a cui verranno poi applicate delle modifiche per verificare come esso si comporta. Queste modifiche riguarderanno:

- Numero di iterazioni massime
- Dimensioni e peso limite del contenitore
- Numero dei pacchi

Per ottenere più risultati l'algoritmo è stato ripetuto 100 volte per ognuno dei test che verranno mostrati in seguito. Per motivi di spazio e per semplificare la comprensione dei dati, in ogni riga delle tabelle verranno inseriti:

- ID dei pacchi selezionati; ad esempio se viene scritto $1, 2, 3, 4$ vuol dire che sono stati scelti i pacchi con quegli ID, includendo tutte le loro permutazioni
- Numero di volte in cui una permutazione dei pacchi compare tra i 100 risultati
- Guadagno totale, ovvero la somma dei valori dei pacchi selezionati
- Peso totale, ovvero la somma dei pesi dei pacchi selezionati

Il motivo dietro la scelta di mostrare una combinazione di pacchi per rappresentare anche tutte le sue permutazioni risiede nell'eccessivo numero di permutazioni. Nel caso peggiore, ovvero quello in cui tutti gli n pacchi possono essere inseriti nel contenitore,

esistono $n!$ permutazioni se nel risultato vengono scelti tutti i pacchi o tutti tranne uno, $\frac{n!}{2!}$ se vengono scelti tutti tranne 2, proseguendo fino ad arrivare a un solo pacco per il quale avremo n opzioni. Perciò in totale avremmo $\sum_{i=0}^{n-1} \frac{n!}{i!}$ permutazioni, rendendo le tabelle molto difficili da leggere.

L'esecuzione dei test proposti in questa sezione sono stati eseguiti su un computer con le seguenti componenti:

- CPU → Intel(R) Core(TM) i5-10500 3.10GHz
- GPU → NVIDIA GeForce GTX 1050 Ti
- RAM → 16 GB

6.1 Test Base

Il seguente set di dati verrà considerato il set base, ovvero quello che verrà utilizzato per effettuare gli altri test applicando le modifiche proposte in precedenza.

PACCHI					
ID	Larghezza	Altezza	Lunghezza	Valore	Peso
1	2	4	9	3	5
2	3	7	3	5	3
3	3	7	6	4	1
4	5	2	3	7	8
5	5	2	6	9	9
6	6	3	3	3	4
7	6	3	6	5	4
8	4	6	3	6	7
9	4	6	6	6	4

Tabella 6.1: Dataset base

CONTENITORE		
Larghezza	Altezza	Lunghezza
9	9	9

Con questi dati, dal punto di vista di spazio, tutti i pacchi possono essere inseriti nel contenitore, perciò per il test base la soluzione ottima sarà la scelta di tutti i pacchi disponibili.

Vediamo quindi un primo test senza considerare un peso limite per il contenitore e utilizzando un massimo di 1.000 iterazioni per il Simulated Annealing.

Pacchi	Risultati	Guadagno	Peso Totale
2, 4, 5, 6, 7, 8, 9	54/100	39	41
1, 4, 5, 6, 7, 8, 9	13/100	41	39
1, 2, 3, 4, 5, 8, 9	11/100	37	40
2, 3, 4, 5, 6, 7, 8	7/100	40	38
1, 2, 3, 4, 5, 6, 7, 8	5/100	41	42
1, 2, 4, 5, 7, 8, 9	3/100	40	41
1, 2, 4, 5, 6, 8. 9	3/100	40	39
1, 3, 4, 5, 6, 8, 9	2/100	38	38
1, 4, 5, 6, 8, 9	2/100	37	34
TEMPO DI ESECUZIONE: 9,808 secondi			

Tabella 6.2: 9 Pacchi - Contenitore 9x9x9 - Nessun Peso Massimo - 1.000 Iterazioni

Notiamo che otteniamo diversi risultati, ma nessuno di questi è la soluzione ottima. La varietà di soluzioni indica che il numero di iterazioni massime non è sufficiente all'algoritmo per trovare la soluzione ottima, perciò proviamo ad aumentarle a 10.000.

Pacchi	Risultati	Guadagno	Peso Totale
1, 2, 3, 4, 5, 6, 7, 8, 9	99/100	45	48
1, 2, 4, 5, 6, 7, 8, 9	1/100	44	44
TEMPO DI ESECUZIONE: 22,609 secondi			

Tabella 6.3: 9 Pacchi - Contenitore 9x9x9 - Nessun Peso Massimo - 10.000 Iterazioni

Questa volta tutti i risultati tranne uno sono stati una permutazione di tutti i pacchi, quindi abbiamo ottenuto la soluzione ottima per questo test nel 99% dei casi.

6.2 Test con Peso Limitato

Proviamo ora ad assegnare un peso massimo al contenitore e ripetiamo il procedimento appena visto. In questo caso non sappiamo quale sia la soluzione ottima, ma possiamo comunque controllare i risultati per vedere se convergono verso una particolare soluzione. Nella colonna del peso totale verrà indicato il peso totale della soluzione rispetto al peso massimo.

Pacchi	Risultati	Guadagno	Peso Totale
1, 4, 5, 6, 8	42/100	33	28/30
1, 4, 5, 7, 8	37/100	33	30/30
4, 5, 6, 7, 8	8/100	32	30/30
1, 2, 4. 5, 8	7/100	32	30/30
1, 4, 6, 7, 8. 9	3/100	32	30/30
1, 2, 4, 6, 7, 8	1/100	31	29/30
1, 2, 4, 6, 8, 9	1/100	31	30/30
2, 4, 5, 6, 8	1/100	31	30/30
TEMPO DI ESECUZIONE: 8,628 secondi			

Tabella 6.4: 9 Pacchi - Contenitore 9x9x9 - Peso Massimo 30 - 1.000 Iterazioni

I risultati sono ancora divisi tra diverse combinazioni, proviamo quindi ad aumentare il numero di iterazioni a 10.000 per verificare se possono migliorare e/o omogeneizzarsi.

Pacchi	Risultati	Guadagno	Peso Totale
1, 4, 5, 6, 8	61/100	33	28/30
1, 4, 5, 7, 8	39/100	33	30/30
TEMPO DI ESECUZIONE: 84,134 secondi			

Tabella 6.5: 9 Pacchi - Contenitore 9x9x9 - Peso Massimo 30 - 10.000 Iterazioni

Questa volta i risultati si sono raggruppati in due combinazioni con lo stesso guadagno ma peso totale diverso. Da notare che le due soluzioni di questo test sono quelle più frequenti nella tabella 6.4, ciò vuol dire che l'algoritmo riusciva a calcolare delle soluzioni buone anche con meno iterazioni, ma a volte non ci riusciva perché non erano sufficienti; con 10.000 invece sembrerebber riuscire più facilmente ad ottenere quelle soluzioni. Prima

di passare alla prossima tipologia di test proviamo anche ad utilizzare 50.000 iterazioni massime per vedere se i risultati cambiano.

Pacchi	Risultati	Guadagno	Peso Totale
1, 4, 5, 6, 8	62/100	33	28/30
1, 4, 5, 7, 8	38/100	33	30/30
TEMPO DI ESECUZIONE: 494,155 secondi			

Tabella 6.6: 9 Pacchi - Contenitore 9x9x9 - Peso Massimo 30 - 50.000 Iterazioni

I risultati sono rimasti pressoché gli stessi, perciò ci fermiamo con l'aumento del numero di iterazioni e passiamo al test successivo.

6.3 Test con Contenitore più grande

Proviamo ora ad eseguire un test aumentando le dimensioni del contenitore di 3 unità, portandolo a 12x12x12. Con questo test possiamo verificare se un contenitore più grande facilita la ricerca di una soluzione grazie al maggiore spazio o se la intralcia a causa delle maggiori possibilità di inserimenti. Ovviamente nei test senza peso massimo la soluzione ottima sarà uguale al test base, quindi contenente tutti i pacchi.

Pacchi	Risultati	Guadagno	Peso Totale
1, 2, 3, 4, 5, 6, 7, 8, 9	100/100	45	48
TEMPO DI ESECUZIONE: 1,451 secondi			

Tabella 6.7: Contenitore 12x12x12 - Nessun Peso Massimo - 1.000 Iterazioni

Come potevamo aspettarci, il test ha prodotto soltanto soluzioni ottime riducendo drasticamente il tempo di esecuzione grazie al fatto che il contenitore più grande facilita l'inserimento dei pacchi, nel senso che possono essere scelte più permutazioni rispetto al test base.

Proseguiamo con i test e utilizziamo un peso massimo di 30.

Pacchi	Risultati	Guadagno	Peso Totale
1, 4, 5, 6, 8	39/100	33	28/30
1, 4, 5, 7, 8	39/100	33	30/30
1, 2, 4, 5, 8	10/100	32	30/30
1, 4, 6, 7, 8, 9	5/100	32	30/30
4, 5, 6, 7, 8	5/100	32	30/30
1, 2, 4, 6, 8, 9	2/100	31	30/30
TEMPO DI ESECUZIONE: 10,732 secondi			

Tabella 6.8: Contenitore 12x12x12 - Peso Massimo 30 - 1.000 Iterazioni

Pacchi	Risultati	Guadagno	Peso Totale
1, 4, 5, 7, 8	58/100	33	30/30
1, 4, 5, 6, 8	42/100	33	28/30
TEMPO DI ESECUZIONE: 102,589 secondi			

Tabella 6.9: Contenitore 12x12x12 - Peso Massimo 30 - 10.000 Iterazioni

I risultati di tutti i test sono molto simili a quelli con contenitore 9x9x9, notiamo però che in quelli senza peso massimo il tempo di esecuzione è diminuito, mentre negli altri è aumentato. Il motivo di questo fenomeno è dovuto al fatto che, senza un peso massimo, appena si trova una permutazione di tutti i pacchi che non esce dal contenitore l'algoritmo si interrompe e si passa all'esecuzione successiva. Nel caso di un peso massimo invece il tempo aumenta perché ci sono molte più permutazioni di pacchi che possono stare dentro al contenitore, forzando l'algoritmo ad eseguire più controlli per le collisioni. Per contenitori più piccoli infatti le permutazioni di pacchi che possono essere inserite sono di meno e vengono scartate subito, riducendo il tempo di esecuzione.

Da questa tipologia di test possiamo quindi concludere che aumentare le dimensioni del contenitore facilita la ricerca della soluzione quando non si considera un peso massimo, mentre quando quest'ultimo viene impostato per l'algoritmo diventa più difficile trovare una buona soluzione a causa delle troppe possibilità per l'inserimento dei pacchi.

Nel caso del peso massimo si potrebbero provare diverse dimensioni di contenitore per trovare quello ideale, ovvero il contenitore più piccolo nel quale si possono inserire i

pacchi della soluzione "migliore". Ovviamente nei casi pratici un test del genere è difficile da realizzare in quanto furgoni più grandi spesso portano ad avere un carico massimo trasportabile più alto, perciò questa proposta non verrà analizzata.

6.4 Test con Numero di Pacchi Maggiore

Ritorniamo al contenitore originale 9x9x9 e proviamo ad utilizzare più pacchi per verificare le prestazioni dell'algoritmo. Aumentando il numero di pacchi aumentano anche le possibili permutazioni e di conseguenza dovrebbe aumentare il tempo di esecuzione.
Nella tabella seguente vengono riportati i pacchi utilizzati in precedenza (1-9) e alcuni nuovi (10-12).

PACCHI					
ID	Larghezza	Altezza	Lunghezza	Valore	Peso
1	2	4	9	3	5
2	3	7	3	5	3
3	3	7	6	4	1
4	5	2	3	7	8
5	5	2	6	9	9
6	6	3	3	3	4
7	6	3	6	5	4
8	4	6	3	6	7
9	4	6	6	6	4
10	2	2	3	4	6
11	6	2	7	7	5
12	1	8	2	4	6

Tabella 6.10: Dataset con più pacchi

Ripetiamo i test svolti nelle sezioni precedenti.

Pacchi	Risultati	Guadagno	Peso Totale
3, 8, 2, 10, 12, 6, 5, 4, 1	6/100	49	45
7, 10, 8, 1, 6, 5, 4, 12	4/100	49	41
4, 1, 10, 2, 5, 11, 8, 12	3/100	49	45
7, 10, 9, 8, 3, 6, 5, 12, 4	1/100	49	48
4, 6, 1, 8, 12, 5, 10, 2	11/100	48	41
4, 10, 6, 12, 5, 11, 8, 2	6/100	48	45
1, 8, 9, 4, 12, 10, 5, 2	2/100	48	44
7, 8, 1, 9, 10, 5, 4, 6	2/100	47	43
8, 6, 4, 9, 1, 12, 2, 7, 10	1/100	47	43
10, 4, 12, 8, 2, 6, 7, 5	7/100	47	43
4, 10, 6, 12, 8, 5, 9, 2	5/100	47	44
7, 12, 10, 11, 1, 5, 6, 4	2/100	47	42
1, 8, 10, 3, 7, 5, 4, 12	2/100	46	42
10, 11, 1, 5, 12, 7, 2, 4	2/100	46	44
4, 5, 1, 10, 8, 11, 12	4/100	46	40
1, 7, 5, 2, 4, 6, 10, 8	1/100	46	42
6, 12, 2, 8, 4, 9, 1, 5	1/100	46	43
1, 8, 2, 3, 10, 12, 4, 5	2/100	45	42
6, 12, 9, 7, 1, 5, 8, 10	1/100	45	40
10, 5, 7, 8, 11, 4, 12	4/100	45	42
12, 2, 1, 9, 7, 4, 5, 10	2/100	45	43
1, 11, 5, 12, 6, 10, 2, 8	1/100	45	41
1, 5, 4, 10, 6, 9, 12, 2	1/100	45	41
12, 8, 1, 5, 6, 4, 10	6/100	45	36
12, 2, 10, 6, 7, 4, 1, 5	1/100	45	40
5, 4, 8, 10, 11, 12, 9	1/100	45	43
3, 12, 10, 9, 8, 6, 5, 4	1/100	45	43
4, 11, 2, 10, 6, 12, 5, 7	1/100	45	44
1, 12, 10, 2, 4, 8, 5	1/100	44	38
6, 1, 10, 2, 4, 12, 8, 11	1/100	44	39
5, 7, 10, 8, 4, 12, 2	7/100	43	40
8, 5, 12, 2, 9, 10, 4	2/100	43	41
4, 8, 10, 5, 3, 11, 12	3/100	42	41
5, 2, 3, 7, 12, 11, 4, 10	1/100	42	45
4, 5, 1, 12, 10, 7, 6	1/100	42	35
11, 1, 8, 10, 5, 4	3/100	40	36
TEMPO DI ESECUZIONE: 13,087 secondi			

Tabella 6.11: 12 Pacchi - Contenitore 9x9x9 - Nessun Peso Massimo - 1.000 Iterazioni

Pacchi	Risultati	Guadagno	Peso
11, 12, 10, 5, 1, 2, 4, 7, 8	18/100	53	50
8, 6, 5, 1, 10, 9, 2, 4, 12	5/100	52	47
12, 6, 7, 10, 5, 8, 1, 2, 4	19/100	52	46
5, 6, 9, 4, 12, 10, 2, 11, 8	2/100	52	51
6, 7, 12, 11, 5, 8, 2, 4, 10	1/100	52	50
5, 9, 6, 12, 10, 2, 4, 7, 8	5/100	51	49
11, 12, 5, 8, 6, 1, 4, 10	25/100	50	43
11, 10, 4, 5, 1, 8, 7, 12	3/100	50	45
12, 2, 4, 11, 5, 7, 1, 6, 10	4/100	50	47
9, 10, 11, 5, 1, 4, 8, 12	1/100	50	46
8, 4, 6, 9, 7, 1, 5, 10, 2	1/100	50	48
8, 10, 12, 9, 4, 5, 1, 3, 6	1/100	50	46
11, 6, 4, 5, 8, 10, 3, 12, 2	1/100	49	49
11, 1, 8, 5, 10, 2, 4, 12	1/100	49	45
4, 1, 6, 5, 8, 10, 7, 12	11/100	49	41
3, 1, 4, 8, 12, 10, 5, 2, 6	1/100	49	45
5, 1, 12, 4, 6, 3, 10, 11, 7	1/100	48	46
TEMPO DI ESECUZIONE: 164,692 secondi			

Tabella 6.12: 12 Pacchi - Contenitore 9x9x9 - Nessun Peso Massimo - 10.000 Iterazioni

Come ci aspettavamo il tempo di esecuzione è aumentato, soprattutto nel test con 10.000 iterazioni. Questo è dovuto al fatto che avendo più pacchi abbiamo non solo più permutazioni, ma di conseguenza anche più soluzioni diverse.

Proviamo con un numero elevato di iterazioni:

Pacchi	Risultati	Guadagno	Peso Totale
11, 7, 6, 4, 8, 5, 12, 10, 1	7/100	54	48
11, 1, 10, 2, 8, 4, 6, 5, 12	41/100	53	48
7, 5, 4, 12, 10, 2, 8, 11, 1	33/100	53	50
6, 8, 1, 4, 5, 7, 9, 10, 12	8/100	32	47
12, 7, 4, 2, 6, 1, 10, 5, 8	9/100	52	46
8, 6, 5, 2, 1, 9, 10, 4, 12	1/100	52	47
11, 9, 8, 6, 4, 2, 12, 10, 5	1/100	52	51
TEMPO DI ESECUZIONE: 1.105,14 secondi			

Tabella 6.13: 12 Pacchi - Contenitore 9x9x9 - Nessun Peso Massimo - 50.000 Iterazioni

Notiamo che i risultati sono un po' più omogenei, ma quelli con un guadagno di 54 compaiono solo 7 volte, mentre quelli con 53 sono molto più frequenti. Possiamo dedurre che esistono dei massimi locali dai quali l'algoritmo difficilmente si sposta, ma in alcuni casi ci riesce e raggiunge il risultato con 54 di guadagno. Anche in questo caso però non sappiamo se 54 è la soluzione ottima, ma sicuramente 53 è un massimo locale, perciò l'algoritmo riesce ad arrivare ad una soluzione buona.

Proviamo un ultimo test impostando un peso massimo per vedere se anche in questo caso il tempo di esecuzione aumenta di molto.

Pacchi	Risultati	Guadagno	Peso Totale
6, 4, 1, 10, 5, 12	58/100	38	30/30
8, 10, 12, 6, 5, 1	30/100	37	29/30
10, 12, 6, 4, 1, 8	5/100	36	27/30
1, 4, 10, 9, 8, 12	3/100	36	30/30
10, 8, 12, 5, 4	3/100	36	30/30
8, 7, 4, 10, 1, 12	1/100	36	29/30
TEMPO DI ESECUZIONE: 79,495 secondi			

Tabella 6.14: 12 Pacchi - Contenitore 9x9x9 - Peso Massimo 30 - 10.000 Iterazioni

Rispetto al test con 9 pacchi il tempo di esecuzione è molto simile; in questo caso però la maggioranza dei risultati convergono in una soluzione con guadagno 38.

Per i test senza peso limite l'algoritmo produce soprattutto dei risultati buoni, ma la maggior parte di essi non sono la soluzione migliore (tra quelle buone, non ottime). Invece per i test con peso limite le prestazioni rimangono molto simili, questo perché il peso massimo limita le permutazioni che vengono analizzate; infatti nell'algoritmo, prima di controllare se si possono inserire i pacchi in una permutazione, viene controllato se il peso limite è stato superato, in questo modo si evita di fare i controlli di collisioni che sono i più dispendiosi.

6.5 Risultato dell'Interfaccia Grafica

Abbiamo raccolto una vasta gamma di dati tramite l'utilizzo di alcuni test, adesso vediamo come essi vengono visualizzati con l'interfaccia grafica.

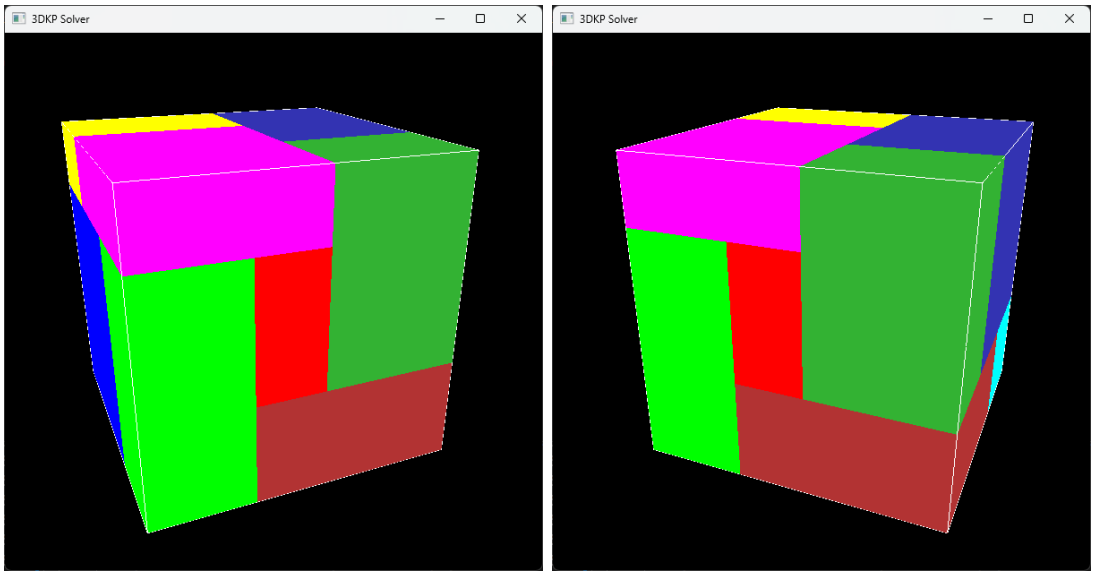


Figura 6.1: Risultato del test relativo alla tabella 6.3

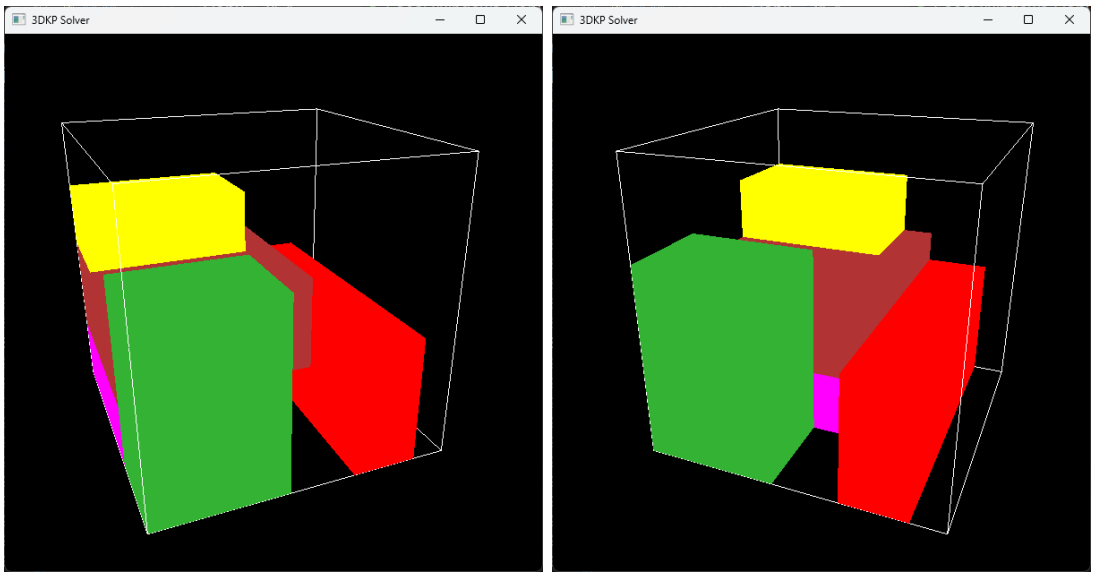


Figura 6.2: Uno dei risultati della tabella 6.5

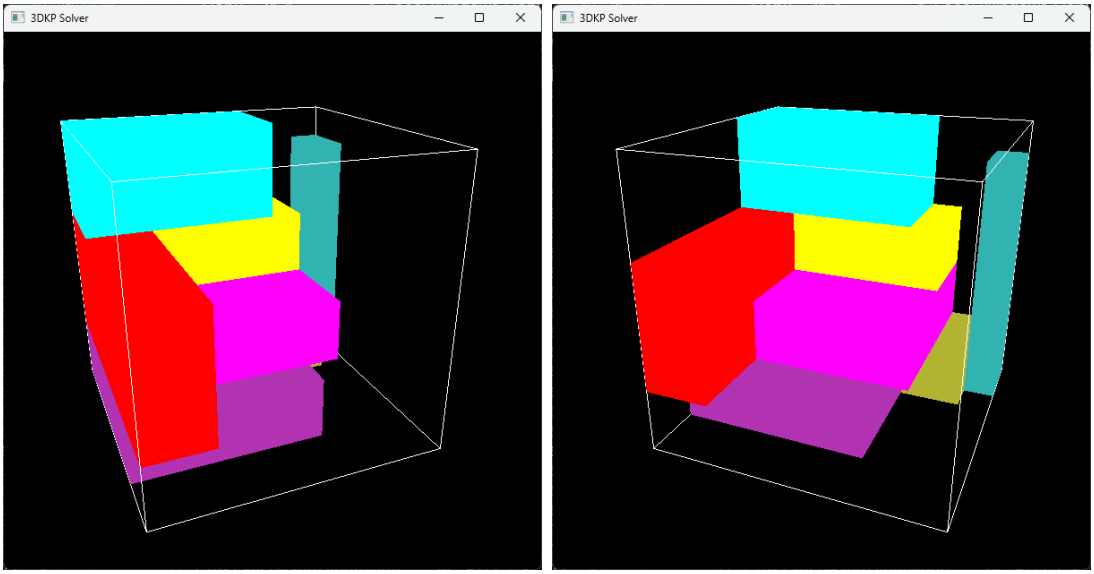


Figura 6.3: Uno dei risultati della tabella 6.9

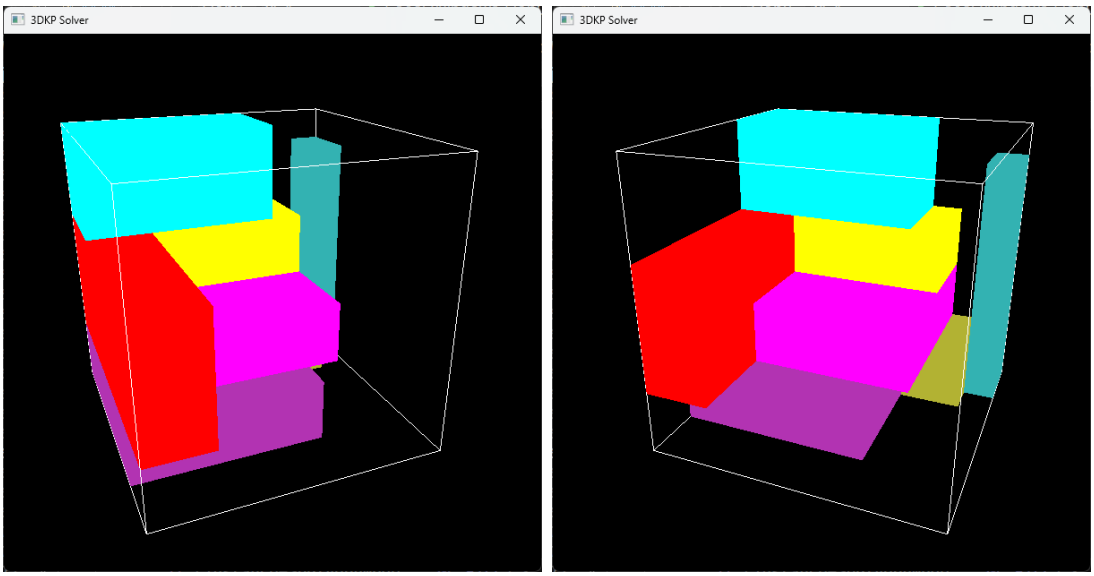


Figura 6.4: Uno dei risultati della tabella 6.12

Capitolo 7

Conclusioni

In questo lavoro di tesi è stato analizzato nel dettaglio il 3DKP, valutando anche la sua complessità e l'opportunità di costruire un algoritmo che consenta di ottenere una soluzione di buona qualità invece di determinare quella ottima. L'algoritmo sviluppato ha dato dei buoni risultati, chiarendo come e perché le prestazioni cambiano in base ai parametri utilizzati. Soprattutto per i contenitori più grandi l'algoritmo richiede più tempo per trovare una soluzione buona a causa delle numerose possibili permutazioni di pacchi che riescono ad essere inseriti nel contenitore, mentre per un numero elevato di pacchi i risultati diventano più variabili ma il tempo di esecuzione rimane piuttosto consistente. È stato quindi raggiunto l'obbiettivo prefissato, ovvero quello di sviluppare un algoritmo risolutivo che potesse fornire delle soluzioni buone in tempi di esecuzione ragionevoli, seppure con un algoritmo alquanto costoso a causa della difficoltà del problema. Questo lavoro di tesi è stato un buon tentativo di implementare un algoritmo semplice in grado di trovare una soluzione di buona qualità che potrebbe in futuro supportare anche la ricerca della soluzione ottima, o quanto meno che riesca a trovare consistentemente delle soluzioni buone. A questo scopo il progetto e questo documento rimarranno pubblici nel repository GitHub al link https://github.com/tiaMat101/3DKP-Solver, affinché chiunque possa consultare, utilizzare e anche migliorare il progetto.

Bibliografia

[EP07] Jens Egeblad and David Pisinger. Heuristic approaches for the two- and three-dimensional knapsack packing problem. *Computers & Operations Research*, pages 1032-1034, 2007.

[GT01] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design*. John Wiley & Sons, Chichester, UK, 2001.

[Hos24] William L. Hosch. P versus np problem. *Encyclopedia Britannica*, 2024.

[Jun23] Florian June. Unveiling the bounded knapsack problem. *Medium*, 2023.

[Lac19] Ing. Valerio Lacagnina. Simulated annealing. *Università degli Studi di Trieste*, pages 81-85, 2018/2019.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, UK, 1990.

[Nag24] Akshat Nagpal. Knapsack problem. *Algorithms for Competitive Programming*, 2024.

[PYA09] Vincent Poirriez, Nicola Yanev, and Rumen Andonov. A hybrid algorithm for the unbounded knapsack problem. *Discreet Journal*, pages 110-111, 2009.

[var24] Knapsack problem and its variations. *fiveable*, 2024.