POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Gestionale Classe L8

Prova Finale
Gestione Magazzino Lego

Relatore prof. Fulvio Corno
Candidato Salvatore Dipace

Dicembre 2020

Indice

Proposta Progetto 4
- Studente proponente . . . 4
- Titolo della proposta . . . 4
- Descrizione del problema proposto . . . 4
- Descrizione della rilevanza gestionale del problema . . . 4
- Descrizione dei data-set per la valutazione . . . 5

Problema affrontato 6

Descrizione del data-set 8

Descrizione delle strutture dati e degli algoritmi 11
- Organizzazione delle classi . . . 11
- Ricorsione . . . 11
  - Ricorsione per l'analisi delle potenzialità del magazzino . . . 12
  - Ricorsione per l'analisi del gap . . . 15
- Grafo . . . 16

Diagramma delle classi 19

Interfaccia grafica 20
- Gestione magazzino . . . 21
  - Caricamento di un set in magazzino . . . 21
  - Indicazione di quanto manca in magazzino per costruire un set . . . 21
- Analisi potenzialità del magazzino . . . 22
- Valutazioni per acquisti futuri . . . 22
  - Creazione grafo e albero con set di interesse . . . 22
  - Pezzi mancanti in magazzino . . . 23
  - Cosa conviene acquistare . . . 23
- Video dimostrativo del software . . . 23

Risultati sperimentali 24
- Caso di studio . . . 24

Conclusioni 26

Proposta progetto

Di seguito si riporta la proposta integrale del progetto con alcune modifiche introdotte durante lo svolgimento del lavoro

Studente proponente

s232047 - Salvatore Dipace

Titolo della proposta

Collezionisti LEGO - analisi del magazzino dei pezzi

Descrizione del problema proposto

Considerato un magazzino di pezzi LEGO ampliato negli anni attraverso l'acquisto di set o di mattoncini sfusi, si vuole sviluppare un'applicazione che permetta di analizzare le potenzialità dello stesso in termini di numero massimo di set (ufficiali o mock proposti da appassionati) costruibili contemporaneamente con i pezzi a disposizione. L'applicazione permette inoltre di capire quali sono i set che conviene acquistare per colmare il gap tra il magazzino a disposizione e un insieme di set a cui si è interessati per minimizzare le spese e ottimizzare l'utilizzo delle risorse a disposizione.

Descrizione della rilevanza gestionale del problema

Si tratta di un problema di gestione di risorse a disposizione per ottimizzare una funzione obiettivo. L'interesse verso questo gioco è rimasto immutato negli anni e coinvolge non solo bambini, ma anche adulti collezionisti. Per alcune persone si tratta anche una forma di investimento che si rivaluta negli anni dopo che un set viene messo fuori produzione. Potrebbe quindi essere interessante fornire uno strumento che permetta di capire come ottimizzare il magazzino a disposizione.

Descrizione dei data-set per la valutazione

Il data-set è reperibile qui: `https://rebrickable.com/downloads/` Inizialmente si è lavorato con quello pubblicato su
`https://www.kaggle.com/rtatman/lego-database/`, ma il primo mette a disposizione dati aggiornati quotidianamente su una base dati strutturata nello stesso modo. Per ogni set commercializzato (tabella `sets`) sono elencati tutti i pezzi necessari (tabella `inventory_parts`). Serve aggiungere una tabella nuova per gestire il magazzino dei pezzi sfusi a disposizione del collezionista.

Problema affrontato

L'applicazione proposta è un semplice prototipo di strumento che i collezionisti del gioco LEGO potrebbero utilizzare per poter valutare le potenzialità dei pezzi a disposizione in un ipotetico magazzino. I problemi affrontati sono due:

1. quali set si possono costruire (interamente o parzialmente) con i pezzi a disposizione;
2. stabilito un obiettivo espresso attraverso un insieme di set che si vorrebbe costruire, capire
   - quali e quanti pezzi mancano in magazzino;
   - quali set conviene acquistare per colmare il gap.

Per il primo problema l'utente può scegliere una o più serie di interesse. Scelta una percentuale di completamento del set, si mostrano a video le combinazioni possibili con il numero più alto di set. Più la percentuale si abbassa, più alta sarà l'insieme dei set costruibili. Si utilizza l'algoritmo della ricorsione in quanto si tratta di un caso di studio simile al problema dello zaino.

Il secondo problema è affrontato per passi successivi:

- scelta delle serie da analizzare
- creazione di un grafo pesato non orientato con le seguenti caratteristiche:
  - i nodi sono costituiti dai set
  - esiste un arco tra due nodi solo se il coefficiente di accoppiamento tra i due set rappresentati dai nodi è superiore a una soglia scelta da interfaccia. Il coefficiente corrisponderà al peso dell'arco.

  Il coefficiente di accoppiamento x tra il nodo A e quello B è definito come

$$\mathrm{x} = \frac{(\text{numero pezzi in comune})^2}{(\text{numero pezzi set A})\,(\text{numero pezzi set B})}$$

- si sceglie un set tra quelli del grafo e si mostra l'albero di visita

- si calcola la lista di pezzi mancanti in magazzino per poter costruire i set che compongono l'albero di visita
- si calcola l'insieme più piccolo di set che conviene acquistare per colmare il gap in base a una percentuale di completamento scelta da interfaccia. Questo perché l'utente potrebbe decidere di coprire parte del gap acquistando pezzi sfusi secondo considerazioni economiche.

L'interfaccia, senz'altro migliorabile, cerca di guidare le scelte dell'utente attivando i bottoni delle varie funzionalità solo quando si sono completati i passi precedenti.

Descrizione del data-set utilizzato per l'analisi

Il data-set considerato è pubblicato su `https://rebrickable.com/downloads/`. La tabelle sono state create con il tool HeidiSQL a partire dal diagramma relazionale a disposizione. I dati, aggiornati ogni giorno, si possono caricare a partire dai file csv di ogni tabella con la funzionalità di import a partire da file csv.

Non tutte le tabelle della base dati sono servite per le funzionalità implementate: quelle utilizzate sono schematizzate nella figura 1. In grigio è evidenziata una tabella non presente nella base dati pubblicata e introdotta per modellare il magazzino del collezionista. Il pezzo è univocamente identificato attraverso un codice, il colore e il materiale. Ogni set è composto da uno o più pezzi e appartiene a una serie. Una breve descrizione di ciascuna tabella è indicata nella tabella 1

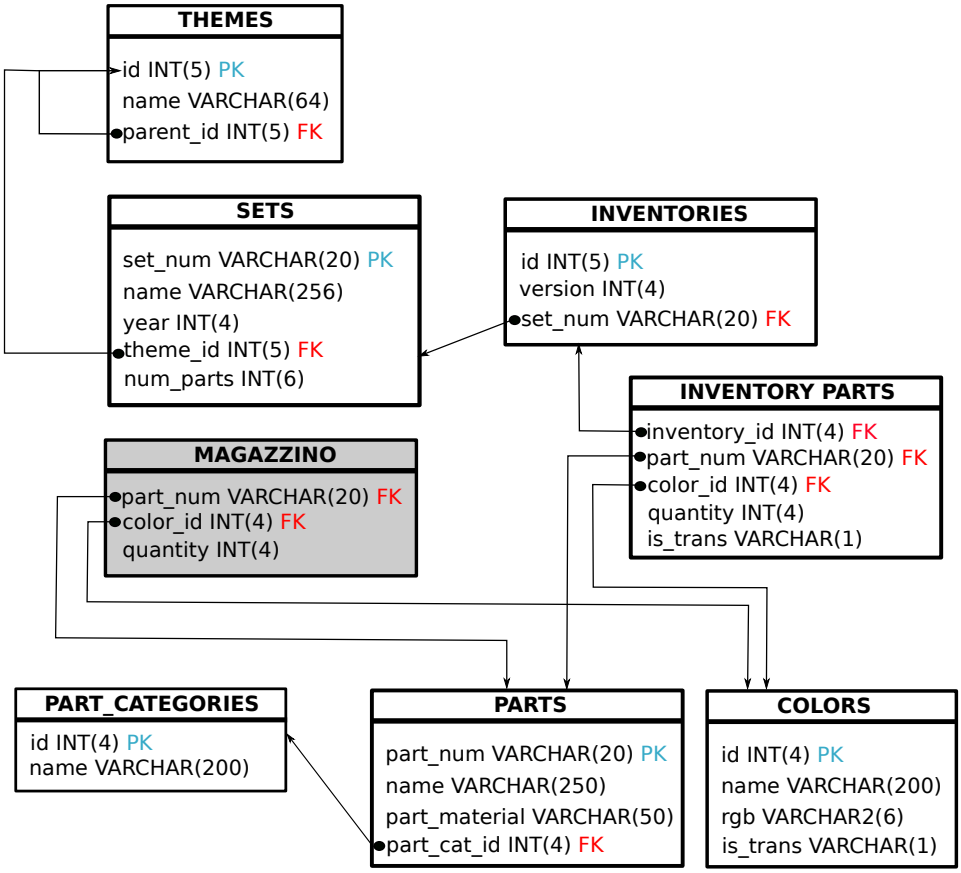


Figura 1. Base dati considerata dall'applicazione di gestione/analisi magazzino Lego

tabella	scopo	numero record	chiave primaria PK	chiavi esterne FK
`themes`	elenco di tutte le serie in commercio	circa 500	campo id	il campo parent_id assume solo valori contenuti nel campo id. Se valorizzato, indica che la serie un filone di un'altra principale
`sets`	elenco di tutti i set in commercio	circa 15000	campo `set_num`	il campo `theme_id` assume solo valori contenuti nel campo id della tabella theme. Ogni set è quindi associato a una serie.
`parts`	elenco di tutti i pezzi in commercio	circa 35000	campo `part_num`	
`colors`	elenco di tutti i colori dei pezzi	circa 200	campo `id`	
`inventories` e `inventory_parts`	relazionano i set e i pezzi che lo compongono	circa 750000		campi `part_num`, `color_id` e `inventory_id`
`colors`	elenco di tutti i colori dei pezzi	circa 200	id	
`magazzino`	contiene i pezzi a disposizione del collezionista. In continuo aggiornamento			campi `part_num` e `color_id`

Tabella 1. Descrizione e caratteristiche delle tabelle utilizzate

Descrizione delle strutture dati e degli algoritmi utilizzati

L'applicazione è stata sviluppata seguendo il pattern MVC. Di seguito si descriveranno le strutture dati utilizzate e gli algoritmi implementati.

Organizzazione delle classi

Il package `model` contiene la classe Model che espone i metodi per il Controller. Per una più semplice leggibilità e manutenzione del codice, si sono creati più package interni:

- `bean`: ospita tutte le classi che modellano gli oggetti;
- `exception`: raccoglie le eccezioni. Si è lavorato con una sola eccezione generica, ma se ne potrebbero creare diverse per ogni situazione specifica.
- `ricorsione`: qui si trovano le classi che si occupano di risolvere i problemi con l'algortimo della ricorsione. Sono due perché l'algoritmo di ricorsione seguito per il problema di come colmare il gap del magazzino è leggermente diversa. Si è sviluppata anche una classe di test
- `graph`: contiene la classe che espone i metodi per creare il grafo e le relative visite. E' accompagnata da una classe di test

Infine c'è il package `db` con la classe DAO e quelle di test e utility.

Ricorsione

L'algoritmo di ricorsione è stato utilizzato per trovare le sequenze più lunghe di set costruibili con il magazzino a disposizione e per determinare l'insieme più piccolo di set da acquistare per colmare il gap del magazzino, per raggiungere un determinato obiettivo.

Ricorsione per l'analisi delle potenzialità del magazzino

La soluzione parziale è rappresentata da una lista di Set. E' una soluzione valida del problema se non sono state trovate fino a quel momento liste di Set con dimensione maggiore o equivalenti. Due liste di set sono equivalenti se hanno stessa dimensione e contengono gli stessi set anche in ordine diverso. Ci possono essere più soluzioni valide: quindi la struttura dati considerata sarà una lista di liste di Set. Il codice dell'algoritmo implementato è il seguente

```
1     protected void scegli(List<Set> parziale, List<List<Set>>
          best, int percentualeCompletamento) {
2
3         if (!parziale.isEmpty() && parziale
4                 .size() >= ((best == null || best.isEmpty()
                      || best.get(0) == null) ? 0 : best.get(0).
                      size())) {
5             // trovato soluzione migliore
6
7             if (parziale.size() > ((best == null || best.
                  isEmpty() || best.get(0) == null) ? 0 : best.
                  get(0).size())) {
8                 best.clear();
9             }
10
11            if (!areSoluzioniEquivalenti(parziale, best)) {
12                java.util.List<Set> temp = new ArrayList<Set
                      >();
13                temp.addAll(parziale);
14                best.add(temp);
15            }
16
17        }
18
19        for (Set s : getSets()) {
20            if (!parziale.contains(s)) {
21                // il set non c'è ancora in parziale e provo
                      ad aggiungerlo
22                if (
                      areThePartsToBuildTheSetWithTheRequiredPercentageInStoc
                      (s, percentualeCompletamento)) {
23                    //nuova soluzione parziale
24                    parziale.add(s);
25                    updateMagazzino(s, "ADD");
26                    // si delega la ricerca al livello
                          successivo
```

```
                    scegli(parziale, best,
                        percentualeCompletamento);
                    //backtracking
                    parziale.remove(s);
                    updateMagazzino(s, "REMOVE");
                }

            }

        }

    }
```

Quando si considera una nuova soluzione parziale si deve aggiornare il magazzino rimuovendo i pezzi che costituiscono il set scelto. In fase di backtracking si devono invece rimettere in magazzino i pezzi del set rimosso dalla soluzione parziale.

```
protected void updateMagazzino(Set s, String operation) {
        Map<String,Part> parts = s.getParts();

        for (String keyPart : parts.keySet()) {
            if (getMagazzino().containsKey(keyPart)) {
                Part magazzinoPart = getMagazzino().get(
                    keyPart);
                if ("ADD".equals(operation)) {
                    magazzinoPart.decrementQuantity(parts.get
                        (keyPart).getQuantity());
                } else {
                    magazzinoPart.incrementQuantity(parts.get
                        (keyPart).getQuantity());
                }

            }

        }
    }
```

La struttura dati utilizzata per rappresentare il magazzino è una mappa dove la chiave è l'identificativo del pezzo e il valore è l'oggetto pezzo. Per capire se una soluzione parziale è equivalente con una delle soluzioni in quel momento salvate come soluzioni definitive, si usa il metodo

```
    protected boolean areSoluzioniEquivalenti(List<Set>
        parziale, List<List<Set>> best) {
        if (parziale == null || parziale.isEmpty()) {
            if (best == null || best.isEmpty()) {
```

```
4                 return true;
5             } else {
6                 return false;
7             }
8         }
9
10        if (best == null || best.isEmpty()) {
11            if (parziale == null || parziale.isEmpty()) {
12                return true;
13            } else {
14                return false;
15            }
16        }
17
18        if (best.get(0).size() != parziale.size()) {
19            return false;
20        }
21
22        for (List<Set> soluzione : best) {
23            /*
24             * //to avoid messing the order of the lists we
                  will use a copy
25             */
26            List<Set> soluzioneTemp = new ArrayList<Set>(
                  soluzione);
27            List<Set> parzialeTemp = new ArrayList<Set>(
                  parziale);
28
29            Collections.sort(soluzioneTemp);
30            Collections.sort(parzialeTemp);
31
32            if (parzialeTemp.equals(soluzioneTemp)) {
33                return true;
34            }
35
36        }
37
38        return false;
39    }
```

Un set può essere aggiunto nella soluzione parziale solo se in magazzino ci sono i pezzi necessari per costruirlo completamente o in parte in base alla percentuale scelta dall'utente. Il metodo implementato è il seguente

```
1   protected boolean
       areThePartsToBuildTheSetWithTheRequiredPercentageInStock
       (Set s, int requiredPercentage) {
2       Map<String, Part> parts = s.getParts();
3       Map<String, Part> magazzinoTemp = new HashMap<String,
            Part>();
4       magazzinoTemp.putAll(getMagazzino());
5
6       int availablePartsNumber = 0;
7       for (String keyPart : parts.keySet()) {
8
9           if (magazzinoTemp.containsKey(keyPart)) {
10              Part magazzinoPart = magazzinoTemp.get(
                    keyPart);
11
12              if (magazzinoPart.getQuantity() < parts.get(
                    keyPart).getQuantity()) {
13                  availablePartsNumber += magazzinoPart.
                        getQuantity();
14
15              } else {
16                  availablePartsNumber += parts.get(keyPart
                        ).getQuantity();
17              }
18
19          }
20
21      }
22      int percentage = (availablePartsNumber * 100) / s.
           getPartsNumber();
23      return percentage >= requiredPercentage;
24
25  }
```

I metodi sono protetti e non privati per poter essere testati nella classe di test.

Ricorsione per l'analisi del gap

La logica è molto simile alla soluzione precedente. Non si riporta il codice, ma si elencano solo le differenze:

- la nuova soluzione parziale si genera rimuovendo un set e non aggiungendolo perché l'obiettivo è trovare l'insieme di set più piccolo che colma il gap con la percentuale scelta

- la nuova soluzione parziale è tale se permette di colmare la parte di gap impostata dall'utente. Se non lo è, il set non viene preso in considerazione e se ne rimuovo un altro

- in questo caso non deve essere aggiornato il magazzino perché questo è virtualmente costituito dalla soluzione parziale che si sta considerando in quel momento e che varia a seconda dei sei aggiunti e tolti.

Grafo

Per analizzare gli accoppiamenti tra un insieme di set avendo evidenza di quelli con più pezzi in comune, si è costruito un grafo non orientato e pesato. L'arco tra due set esiste se l'accoppiamento tra loro è maggiore di una soglia scelta dall'utente. Per costruire il grafo si è sviluppato il metodo

```
1  public Graph<Set, DefaultEdge> creaGrafo(List<Theme>
       themes, double threshold) throws LegoException {
2
3      Graph<Set, DefaultEdge> graph;
4
5      List<Set> sets = init(themes);
6
7      graph = new SimpleWeightedGraph<Set, DefaultEdge>(
           DefaultEdge.class);
8      Graphs.addAllVertices(graph, sets);
9
10     // aggiungo un arco tra tutti i set, con peso x =
           coefficiente di accoppiamento
11     for (Set s1 : graph.vertexSet()) {
12         for (Set s2 : graph.vertexSet()) {
13             if (!s1.equals(s2)) {
14                 Float accoppiamento =
                       calcolaAccoppiamento(s1, s2);
15                 if (accoppiamento * 100 >= threshold) {
16                     Graphs.addEdgeWithVertices(graph, s1,
                           s2, accoppiamento);
17                 }
18             }
19         }
20     }
```

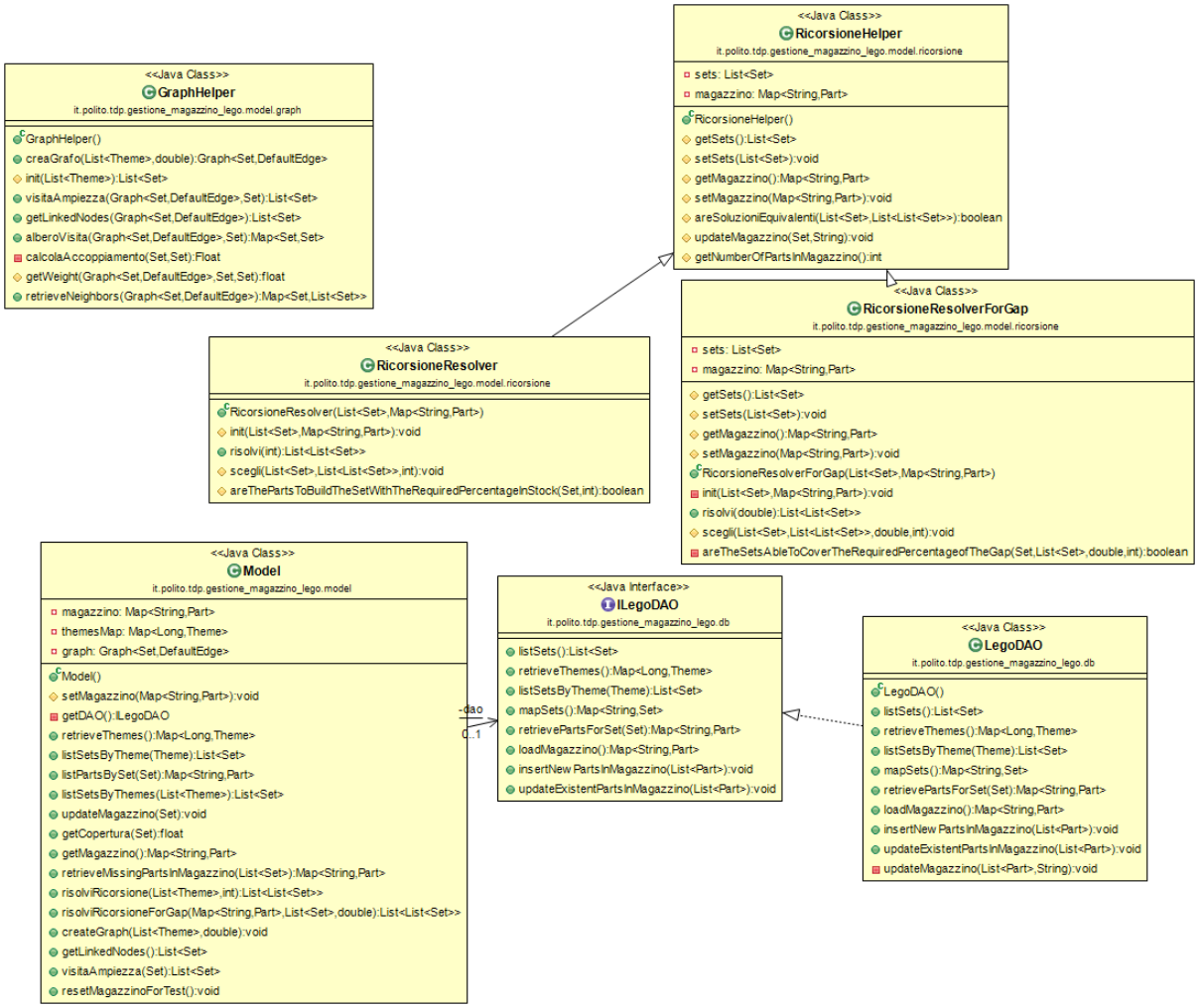
```
21
22          return graph;
23
24      }
```

mentre il calcolo del coefficiente di accoppiamento tra due set avviene con il metodo

```
1       private Float calcolaAccoppiamento(Set s1, Set s2) {
2
3           float coefficienteAccoppiamento;
4           // se uno dei due set non ha pezzi -->0
5           if (s1.getParts().isEmpty() || s1.getParts().isEmpty
                ()) {
6               coefficienteAccoppiamento = 0;
7
8           } else {
9               int numeroPezziInComune = 0;
10              if (s1.getPartsNumber() >= s2.getPartsNumber()) {
11                  for (String keyPart : s2.getParts().keySet())
                        {
12                      if (s1.getParts().containsKey(keyPart)) {
13                          if (s1.getParts().get(keyPart).
                                getQuantity() >= s2.getParts().get
                                (keyPart).getQuantity()) {
14                              numeroPezziInComune += s2.
                                    getParts().get(keyPart).
                                    getQuantity();
15                          } else {
16                              numeroPezziInComune += s1.
                                    getParts().get(keyPart).
                                    getQuantity();
17                          }
18                      }
19                  }
20
21              } else {
22                  for (String keyPart : s1.getParts().keySet())
                        {
23                      if (s2.getParts().containsKey(keyPart)) {
24                          if (s2.getParts().get(keyPart).
                                getQuantity() >= s1.getParts().get
                                (keyPart).getQuantity()) {
25                              numeroPezziInComune += s1.
                                    getParts().get(keyPart).
                                    getQuantity();
```

```
26                          } else {
27                              numeroPezziInComune += s2.
                                    getParts().get(keyPart).
                                    getQuantity();
28                          }
29                      }
30                  }
31
32              }
33              coefficienteAccoppiamento = (float) (
                    numeroPezziInComune * numeroPezziInComune)
34                      / ((s1.getPartsNumber() * s2.
                            getPartsNumber()));
35          }
36
37          return coefficienteAccoppiamento;
38      }
```

Diagramma delle classi delle parti principali



Interfaccia grafica

L'interfaccia utente proposta presenta tre pannelli:

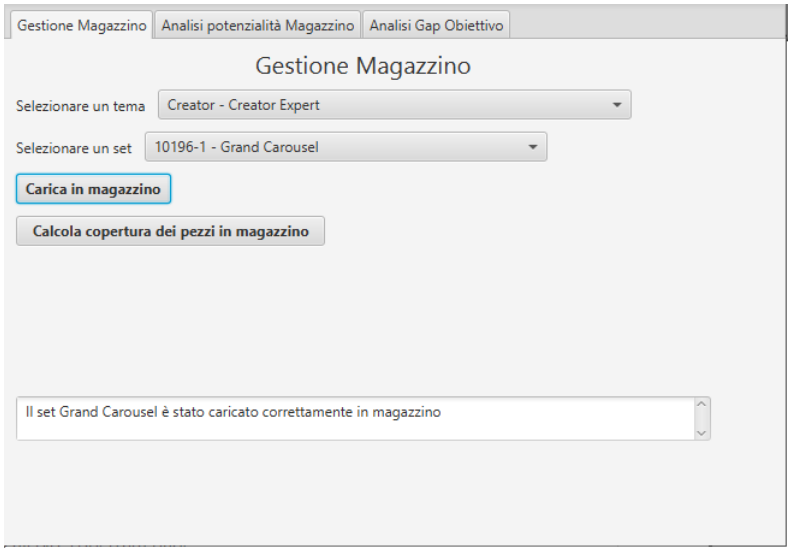
- la prima maschera offre semplici funzionalità per riempire il magazzino simulando l'acquisto di un set. E' prevista la possibilità di acquistare un set già esistente o uno nuovo con pezzi già presenti in magazzino. In questo caso si devono aggiornare le quantità in magazzino con delle `UPDATE` e non delle `INSERT`. E' possibile anche visualizzare un semplice grafico a torta per avere un'indicazione di quanti pezzi ci sono in magazzino per costruire un determinato set

- il secondo pannello offre la possibilità di scegliere alcuni set e capire quali di questi possono essere costruiti con i pezzi in magazzino. Si ipotizza anche che l'utente possa decidere di accontentarsi di costruire parzialmente i set perché poi acquisterà i pezzi mancati singolarmente. Quindi è possibile impostare una percentuale di completamento.

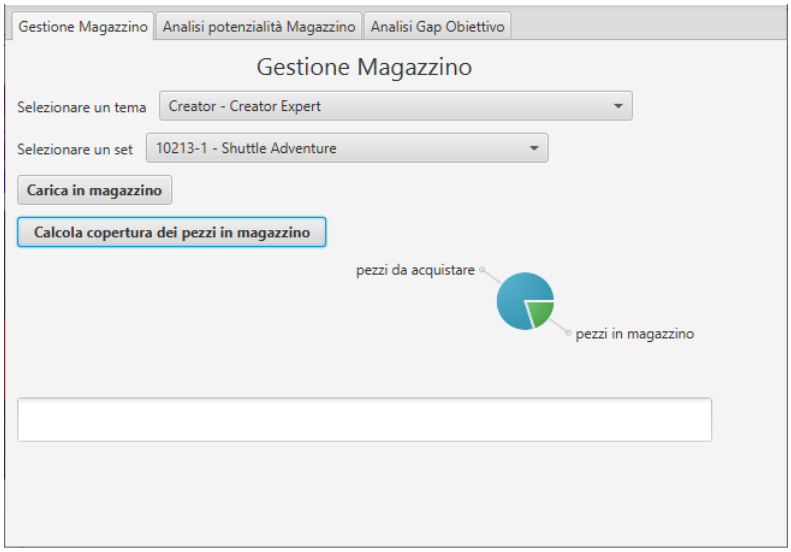
- l'ultima maschera infine dovrà elencare i pezzi mancanti in magazzino per costruire totalmente o in parte un insieme di set e se interessa, cosa conviene acquistare per coprire le mancanze del magazzino.

Gestione magazzino

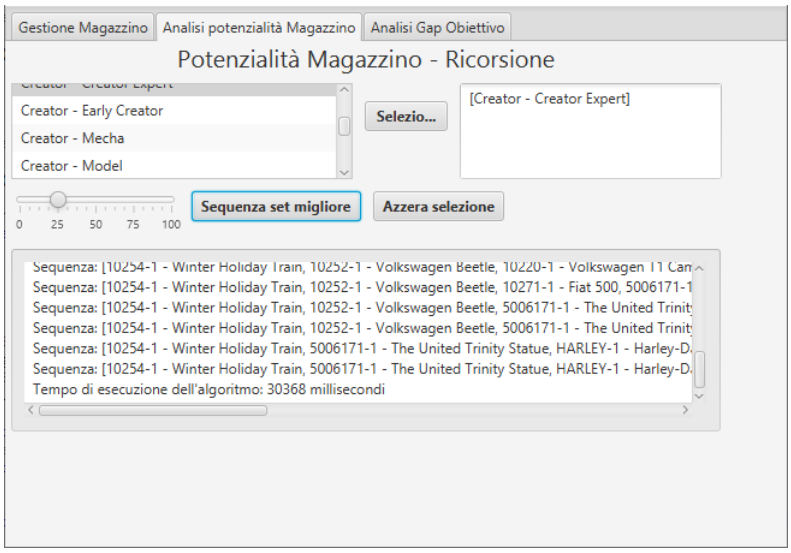
Caricamento di un set in magazzino



Indicazione di quanto manca in magazzino per costruire un set

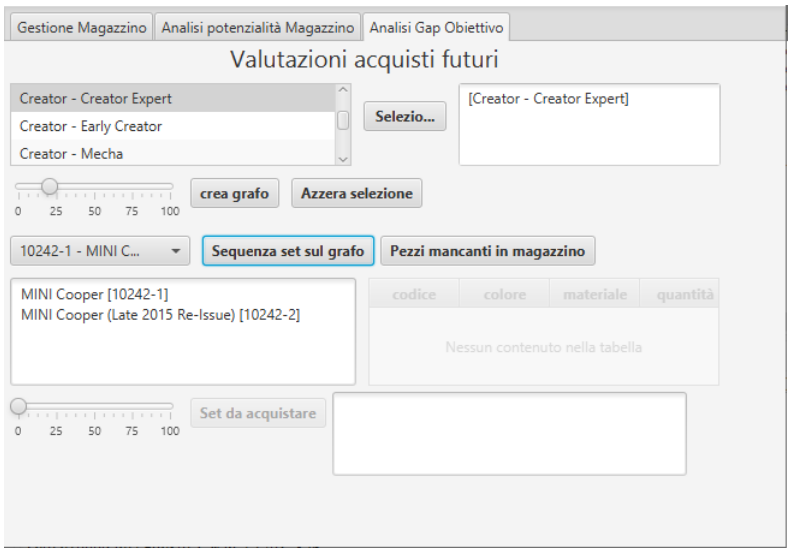


Analisi potenzialità del magazzino

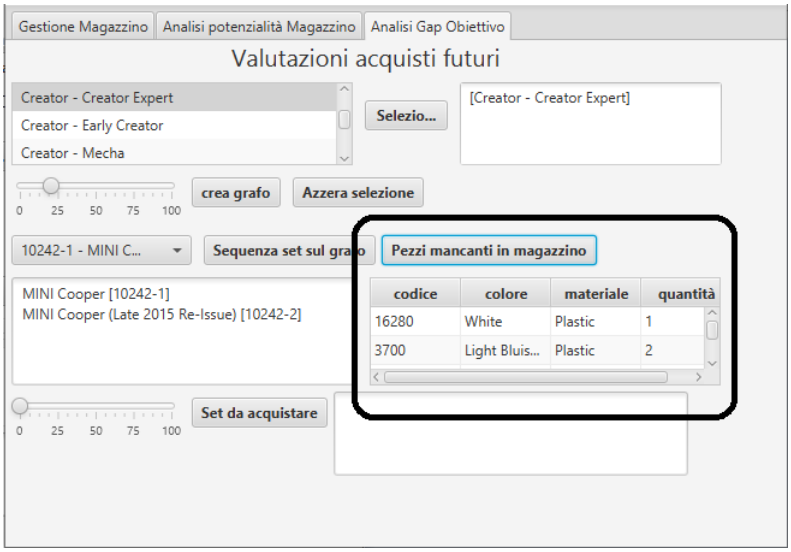


Valutazioni per acquisti futuri

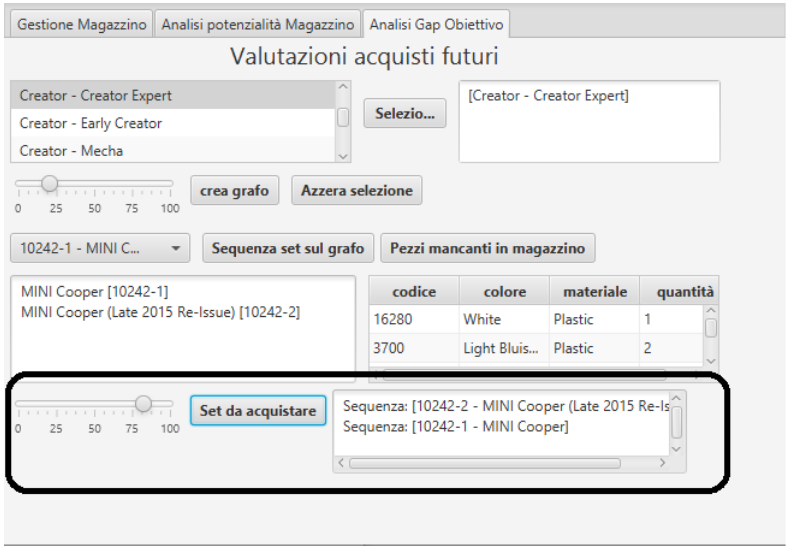
Creazione grafo e albero con set di interesse



Pezzi mancanti in magazzino



Cosa conviene acquistare



Video dimostrativo del software

`https://youtu.be/JzkvN3o1dbM`

Risultati sperimentali

Sia per l'analisi delle potenzialità del magazzino sia per la ricerca dell'insieme di set più conveniente per colmare il gap per raggiungere un determinato obiettivo, si utilizza l'algoritmo della ricorsione. Si è quindi ritenuto utile valutare i tempi di esecuzione per un caso di studio semplice, ma indicativo della crescita esponenziale della complessità del problema.

Caso di studio

Si suppone di partire da un magazzino vuoto e acquistare i seguenti tre set della serie Creator Export:

- Fiat 500 (10271-1)
- London Bus (10258-1)
- NASA Apollo 11 (10266-1)

Dopodiché si calcolano le sequenze di set che si possono costruire cambiando la percentuale di completamento a ogni prova. Si ottengono i risultati di seguito riportati nella tabella 2 e nel grafico 2. Si osserva come nel momento in cui la percentuale di completamento dei set scende sotto un valore sufficientemente basso a seconda della disponibilità di pezzi in magazzino, i tempi aumentano in quanto il numero di soluzioni parziali valide aumenta considerevolmente.

percentuale di completamento dei set	tempi in secondi
90%	0.15
80%	0.15
70%	0.15
60%	0.15
50%	0.15
25%	30
20%	20

Tabella 2. Tempi di esecuzione dell'algoritmo di ricorsione per diverse percentuali di completamento dei set

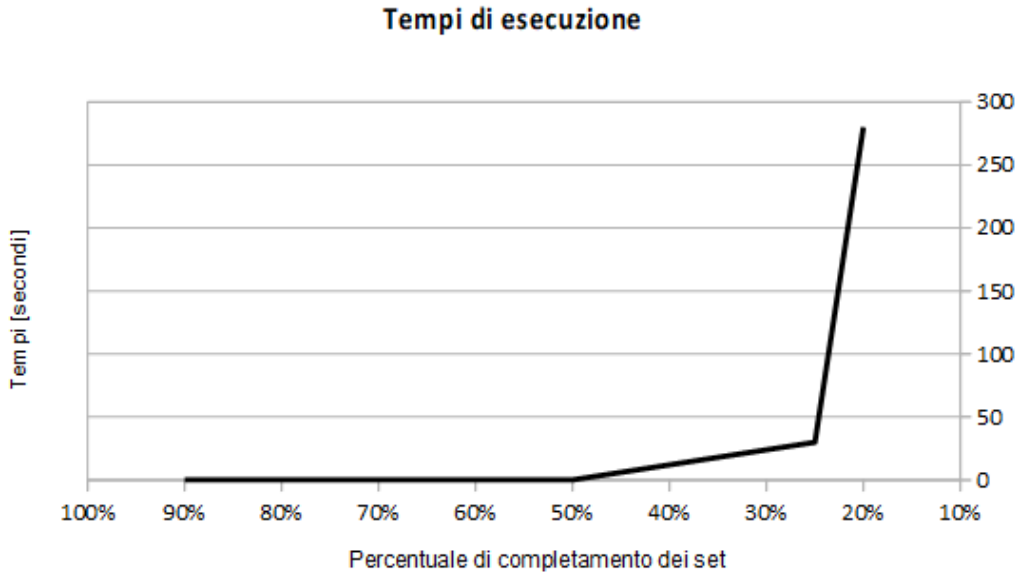


Figura 2. Tempi dell'algoritmo di ricorsione

Conclusioni

La complessità della base dati considerata in termini di numero di tabelle presenti e volume dei dati influisce molto sui tempi di esecuzione delle operazioni implementate. Gli sviluppi sono stati quindi svolti utilizzando una classe DAO di test con pochi dati e semplici. Solo dopo aver verificato il corretto funzionamento degli algoritmi si è passati a lavorare sulla base dati vera.

Tra i punti di debolezza dell'applicazione si indica un'interfaccia grafica che deve essere migliorata dal punto di vista dell'usabilità. Per esempio

- si dovrebbe poter scegliere un insieme di set non necessariamente collegati alla stessa serie;
- manca la funzionalità per poter aggiungere in magazzino pezzi acquistati singolarmente.

Senz'altro alcune parti del codice possono essere migliorate e ottimizzate. In molti punti si devono confrontare tra loro collezioni di oggetti con dimensioni non trascurabili. Un refactoring potrebbe influire sui tempi di esecuzione.

Alcuni risultati non totalmente corretti derivano da

- aver considerato solo i pezzi necessari per la costruzione del set e non anche quelli di riserva (attributo `is_spare` della tabella `inventory_parts`)
- non aver tenuto conto delle minifigure (tabelle `minifigs` e `inventory_minifigs`)
- in alcuni casi il numero di pezzi dichiarato nella tabella dei set (attributo `num_parts`) non sempre corrisponde ai pezzi collegati al set nella tabella `inventory_parts`. Quindi si è preferito calcolarlo ogni volta piuttosto che affidarsi a questa informazione

Infine un punto importante che potrebbe rappresentare un'importante miglioramento delle indicazioni ottenute dall'applicazione consiste nel assegnare un valore/peso a ogni pezzo. La base dati non fornisce informazioni a riguardo e quindi un semplice mattoncino 1x1 ha il valore di una ruota o di un motorino per un modello di veicolo. Se ci fossero queste informazioni, non sarebbero necessari molti

interventi sul codice in quanto già ora per esempio la ricorsione è adatta a risolvere il problema dello zaino (rappresentato o dal magazzino oppure dal gap di pezzi che si vuole colmare).