Relazione Progetto PCD - Parte 1

Giacomo Manzoli 1049820

9 dicembre 2014

Indice

1 Introduzione

Si vuole creare un algoritmo che, partendo da un file contenete la descrizione di tutte le tessere di un puzzle sia in grado di ricostruirlo.
L'algoritmo ideato ricompone il puzzle combinado a due a due i blocchi di pezzi del puzzle finché non si crea un unico blocco che racchiude tutti i pezzi nella loro posizione corretta, dove con blocco di pezzi si intende un'insieme di tessere del puzzle collegate tra loro.
Per evitare complicazioni si sono imposti dei vincoli riguardo a come i pezzi possano essere collegati tra loro, in particolare le tessere del puzzle possono essere combinate in modo che formino solo righe, e solamente le righe complete (le righe che vanno da un bordo all'altro del puzzle) possono essere attaccate verticalemente.
Con queste ipotesi si possono creare solamente blocchi di puzzle di forma regolare che sono più semplici da gestire.

1.1 Guardando al parallelismo

Dato che non vengono fatte ipotesi sul pezzo di partenza è possibile parallelizzare il caricamento dei dati dal file con l'eleborazione. Inoltre l'algoritmo risolutivo può essere eseguito contemporaneamete su più thread dato che consiste nel prendere un'elemento casuale da una lista e ne cerca uno compatibile stando però attenti a gestire correttamente l'accesso concorrente ai dati.

2 Parte ad oggetti

Si è cercato di realizzare una soluzione che sfrutti al massimo le potenzialità del polimorfismo. In particolare tutte le classi che rappresentano le componenti del puzzle implementano la stessa inferfaccia con lo scopo di rendere il codice dell'algoritmo il più lineare possibile.
Per limitare i casi possibili si è scelto di modellare solo alcune delle combinazioni: `PuzzleItem`, `PuzzleRow` e `PuzzleBlock`, di conseguenza i modi in cui questi pezzi possono essere combinati tra loro è limitato, in particolare:

- La combinazione di due `PuzzleItem` può generare solo una `PuzzleRow`
- La combinazione di un `PuzzleItem` e una `PuzzleRow` genera una `PuzzleRow`
- La combinazione di due `PuzzleRow` genera un `PuzzleBlock` e può avvenire solamente se le due `PuzzleRow` sono complete
- Due `PuzzleBlock`, essendo composti solamente da righe complete, possono essere combinati tra loro solamente in modo verticale

In questo modo risulta più semplice combinare tra loro i vari pezzi dato che se non ci fossero questi vincoli si potrebbero generare blocchi di pezzi irregolari che sono molto più difficili da gestire.
Tutte le classi sono raccolte nel package `Puzzle` con le seguenti visibilità:

- **public:** `Puzzle` e `PuzzleElement`
- **protected:** `PuzzleItem`, `PuzzleRow` e `PuzzleBlock`

Il programma eseguibile è stato inserito in un package separato.

2.1 (Interfaccia) PuzzleElement

Rappresenta un elemento generico del puzzle, il quale è identificato da un ID e che può confinare con altri elementi delle stesso tipo che hanno un ID opportuno. Un elemento del puzzle può essere una cosa atomica oppure composta da altri elementi del puzzle.

```
public interface PuzzleElement {
    public String getTop();
    public String getRight();
    public String getBottom();
    public String getLeft();

    public String getID();

    public boolean canAttach(PuzzleElement item);

    public PuzzleElement attach(PuzzleElement item);

    public boolean isComposed();
    public boolean isComplete();

    public int getRowCount();
    public int getColCount();

    public List<PuzzleElement> getComponents();

    public String print();
}
```

2.2 (Classe) PuzzleItem

Questa classe rappresenta una singola tessera del puzzle e implementa l'interfaccia `PuzzleElement`. Di seguito venogno riportate e commentate alcune parti significative del codice.

```
static final public String EMPTY_ID="VUOTO";
```

Rappresenta l'id di una tessera che non appartiene al puzzle, le tessere che confinano con quest ID sono quelle che compongono il bordo del puzzle.

```
private boolean canAttachHorizzontally(PuzzleElement item) {
  return this.getID().equals(item.getRight()) ||
        this.getID().equals(item.getLeft());
}
public boolean canAttach(PuzzleElement item) {
  return canAttachHorizzontally(item);
}
```

Per scelte progettuali due tessere del puzzle si possono attaccare solo orizzontalmente qundi non viene fatto il controllo se si possono attaccare verticalmente o meno.

```
public PuzzleElement attach(PuzzleElement item) {
    List<PuzzleElement> newItem = item.getComponents();
    if (this.canAttachHorizzontally(item)){
      if (this.getID().equals(item.getLeft()))
        { newItem.add(0,this); }
      else
        { newItem.add(this); }
      return new PuzzleRow(newItem);
```

```
    }
    return null;
  }
```

Questo metodo crea una nuova riga sfruttando il costruttore di `PuzzleRow` che accetta una lista di `PuzzleElement` che rappresenta i componenti della riga.

```
  public boolean isComposed(){ return false; }
  public boolean isComplete() {
    return  top.equals(PuzzleItem.EMPTY_ID) &&
        right.equals(PuzzleItem.EMPTY_ID) &&
        bottom.equals(PuzzleItem.EMPTY_ID) &&
        left.equals(PuzzleItem.EMPTY_ID);
  }

  public int getColCount() { return 1; }
  public int getRowCount() { return 1; }
```

Per definizione un `PuzzleItem` è un elemento non composto e quindi è rappresentato in una riga e una colonna. Mentre è completo solo quando un il puzzle è composto da un solo elemento.

2.3 (Classe) PuzzleRow

Questa classe rappresenta una riga del puzzle memorizzata utilizzando una lista di `PuzzleElement`. Anche questa classe implementa l'interfaccia `PuzzleElement`. Nella classe sono presenti due campi dati, `firstItem` e `lastItem` che riferiscono il primo e ultimo elemento della riga. Di seguito vengono riportate e commentate alcune parti significative del codice.

```
public String getID(){ return firstItem.getID();}
```

Per identificare una riga si è scelto l'ID del primo elemento che la compone, questo perché un elemento che compone la riga appartiene solamente a quella riga, quindi ogni elemento di una riga identifica la riga in cui si trova. Per praticità è stato scelto il primo elemento.

```
private boolean canAttachHorizzontally(PuzzleElement item){
    return firstItem.getID().equals(item.getRight()) ||
           lastItem.getID().equals(item.getLeft());
}
private boolean canAttachVertically(PuzzleElement item){
  return (firstItem.getID().equals(item.getTop()) ||
           firstItem.getID().equals(item.getBottom())) && isComplete() && (getColCount() ==
               item.getColCount());
}

public boolean canAttach(PuzzleElement item){
    return canAttachHorizzontally(item) ||
           canAttachVertically(item);
}
```

Ad una riga si può attaccare orizzontalmente un `item` quando l'id dell'elemento della riga coincide con l'id dell'elemento che deve confinare a destra o sinistra del `PuzzleElement` passato per parametro. Lo stesso ragionamento vale anche per `canAttachVertically` con l'aggiunta del fatto che una riga può essere combinata con un altro `item` solo se la riga è completa e l'altro elemento ha lo stesso numero di colonne. Non si può usare `item.isComplete()` al posto del test sulle colonne perché `item` può essere un blocco e si genererebbe un'inconsistenza dato che la chiamata a `canAttachVertically` ritornerebbe `false`, questo perché `item` non può essere

completo se gli manca una riga. Invece usando il numero di colonne ottengo il risultato corretto perché se `item` è di tipo `PuzzleRow` o `PuzzleItem` e la riga corrente è completa allora `item` è completo se ha lo stesso numero di colonne di una riga completa, quindi `isComplete() && (getColCount() == item.getColCount())` viene valutata `true` solo quando entrambe le righe sono complete. Nel caso `item` sia di tipo `PuzzleBlock` allora, per definizione, `item` è composto solo da righe complete quindi `isComplete() && (getColCount() == item.getColCount())` fornisce il risultato corretto.

```
public PuzzleElement attach(PuzzleElement item) {
    List<PuzzleElement> newItem = new ArrayList<>();
    if (this.canAttachHorizzontally(item)){
        newItem.addAll(item.getComponents());
        if (firstItem.getID().equals(item.getRight())){
            newItem.addAll(row);
        }else{
            newItem.addAll(0, row);
        }
        return new PuzzleRow(newItem);
    }
    if (this.canAttachVertically(item)){
        if (item.getRowCount()==1) {
            newItem.add(item);
        }else{
            newItem.addAll(item.getComponents());
        }
        if (this.getID().equals(item.getTop())){
            newItem.add(0, this);
        }else{
            newItem.add(this);
        }
        return new PuzzleBlock(newItem);
    }
    return null;
}
```

Nel metodo `attach` viene costruita una nuova riga o un blocco di righe in base al tipo di match. Se la riga corrente e l'item passato per parametro devono essere attaccati verticalmente è necessario discriminare se l'item è una riga o un blocco di righe, viene quindi usato `getRowCount()` per differenziare i due casi.

```
public boolean isComplete(){
    return firstItem.getLeft().equals(PuzzleItem.EMPTY_ID) &&
         lastItem.getRight().equals(PuzzleItem.EMPTY_ID);
}
```

Una riga è completa quando sia l'elemento a sinistra di `firstItem` sia l'emento a destra di `lastItem` è il bordo, cioè ha id *vuoto*.

2.4 (Classe) PuzzleBlock

Questa classe rappresenta un blocco di righe del puzzle memorizzate utilizzando una lista di `PuzzleElement`. Anche questa classe implementa l'interfaccia `PuzzleElement`. Nella classe sono presenti due campi dati, `firstItem` e `lastItem` che riferiscono la prima e ultima riga del blocco. L'implementazione di `PuzzleBlock` è molto simile a quella di `PuzzleRow`, nonostante ciò si è scelto di non ereditare da `PuzzleRow` perché il significato logico delle due classi è diverso

e in ogni caso la maggior parte dei metodi di `PuzzleRow` sarebbero stati ridefiniti in `PuzzleBox`. Di seguito vengono riportate e commentate alcune parti significative del codice.

```
public String getID(){ return firstItem.getID();}
```

Per identificare un blocco si è scelto l'id della prima riga che lo compone, la motivazione di questa scelta è analoga a quella per `PuzzleRow`.

```
public boolean canAttach(PuzzleElement item){
    return canAttachOnTop(item);
}
private boolean canAttachVertically(PuzzleElement item){
    return canAttachOnTop(item) || canAttachOnBottom(item);
}
private boolean canAttachOnTop(PuzzleElement item){
    return item.getBottom().equals(firstItem.getID()) && getColCount() == item.getColCount();
}
private boolean canAttachOnBottom(PuzzleElement item){
    return item.getTop().equals(lastItem.getID()) && getColCount() == item.getColCount();
}
```

Per definizione, due blocchi di righe non possono essere combinati orizzontalmente, mentre possono essere combinati sopra o sotto se c'è il match degli id. Il controllo su `getColCount` serve per evitare di attaccare al blocco corrente righe incomplete o pezzi singoli, questo perché una riga incompleta avrà sicuramente un numero diverso di colonne rispetto a quelle del blocco perché un blocco viene creato dall'unione di righe complete.

2.5 (Classe) Puzzle

Questa classe racchiude l'algoritmo risulutivo, la parte riguardante alla creazione degli oggetti necessari per far funzionare tale algoritmo e alcuni metodi di utilità. Un oggetto di tipo `Puzzle` viene creato usando i dati contenuti nel file passatto come parametro al costruttore mediante il metodo `loadData`. In particolare per creare ogni elemento viene usato un metodo ausiliario `createElement` il quale data una stringa verifica che sia corretta e nel caso ritorna il `PuzzleElement` corrispondente.
Può capitare che il puzzle sia composto da n righe e una sola colonna oppure che il puzzle sia composto da un solo pezzo, in entrambi i casi `createElement` crea un `PuzzleBlock` dato che l'unica tessera può essere vista come una riga completa che di per se è gia un `PuzzleBlock`

```
private static PuzzleElement createElement(String[] parts) throws Exception {
  if (!verify(parts[1])){
       Exception e = new Exception("Problema con la tessera '"+parts[1]+"'");
       throw e;
   }
   PuzzleElement item = new PuzzleItem(parts[0],parts[1],parts[2],parts[3],parts[4],parts[5]);
   if (item.getLeft().equals(item.getRight())){
       item = new PuzzleRow(item.getComponents());
       item = new PuzzleBlock(item.getComponents());
   }
   return item;
}
```

Per accere ai vari dati relativi al puzzle sono stati messi a disposizione i seguenti metodi pubblici:

```
public int getColNumber(){
    if (solution != null){
```

```
        return solution.getColCount();
    } else {
        return -1;
    }
}
public int getRowNumber(){
    if (solution != null){
        return solution.getRowCount();
    } else {
        return -1;
    }
}
public String getText(){
    if (solution != null){
        return solution.print().replaceAll("\n","");
    } else {
        return "Soluzione non trovata";
    }
}
public String print(){
    if (solution != null){
        return solution.print();
    } else {
        return "Soluzione non trovata";
    }
}
```

2.5.1 Gestione degli errori

Nel caso l'algoritmo non riesca a risolvere il puzzle, perché manca un pezzo oppure perché alcuni pezzi inseriti sono stati inseriti in modo incorretto, l'acesso ai campi dell'oggetto `Puzzle` produce la stampa:

`Soluzione non trovata`

e i metodi `getColNumber` e `getRowNumber` ritornano -1.
Nel caso si verifichi un errore durante la lettura la creazione del pezzo a causa di un errore sui dati, il metodo `createElement` solleva un'eccezione che blocca la lettura del file.

2.5.2 Verifica dei dati di input

Durante la creazione dei vari pezzi del puzzle viene controllato che il file contenga solamente caratteri, numeri o punteggiatura.
La punteggiatura ammessa viene limitata a: `!?.,:;()` inoltre vengono considerati come caratteri validi anche lo spazio e i doppi apici.

```
private static boolean verify(String s){
  String validChar="!?.,:;\"( )";
        return  s.length()==1 &&
        (Character.isLetterOrDigit(s.charAt(0)) || validChar.contains(s));
}
```

3 Algoritmo risulutivo

```
private boolean solve(List<PuzzleElement> pezzi){
  while (pezzi.size() > 1 && !(pezzi.get(0).isComplete() && pezzi.size()==1)){
    PuzzleElement anItem = pezzi.remove(0);
    int i=0;
    while (i < pezzi.size() && !anItem.canAttach(pezzi.get(i))){
      i++;
    }
    if (i != pezzi.size()){
      PuzzleElement itemToAttach = pezzi.remove(i);
      pezzi.add(anItem.attach(itemToAttach));
    } else { return false; }
  }
  return pezzi.get(0).isComplete();
}
```

3.1 Correttezza

Precondizione: `List<PuzzleElement> pezzi` è una lista di PuzzleItem che contiene tutti i pezzi del puzzle.
Postcondizione: Se l'algoritmo è riuscito a completrare il puzzle viene ritornato `true` e l'unico elemento della lista rappresenta la soluzione. In caso contrario viene ritornato `false` e i valori presenti nella lista non sono significativi.
Il ciclo `while` più esterno viene ripetuto finché c'è più di un pezzo oppure c'è solo un pezzo del puzzle e questo pezzo è completo. Questo ciclo, che può potenzialmente essere infinito, ha una fine perché ad ogni iterazione vengono tolti dalla lista due pezzi, combinati tra loro e il risultato rimesso nella lista, perciò ad ogni iterazione il numero di elementi della lista diminuisce di 1. Quando viene eseguito il corpo del ciclo ci sono almeno due pezzi che devono essere combinati tra loro, ne viene preso uno a caso, per comodità il primo, e si inizia una ricerca sequenziale di un altro pezzo attaccabile. La ricerca, sotto le ipotesi della precondizione, termina sempre con un esito positivo. Nel caso non venga trovato il pezzo, cioè che `i==pezzi.size()`, viene terminato l'algoritmo con esito negativo. Una volta trovato, si combinano i due pezzi e si reinserisce tra i pezzi disponibili il risultato. Quando si esce dal ciclo while viene ritornato `true` se la soluzione è completa oppure se si è nel caso in cui il puzzle è composto da un solo pezzo.

3.2 Complessità

Finché ci sono 2 o più elementi del puzzle, ne viene scelto uno e cercato sequenzialmente un altro compatibile, una voltra trovato si crea un nuovo pezzo, ottenuto combinando i due precedentemente trovati.
La ricerca viene fatta in modo sequenziale con una complessità $O(n)$, dove n è il numero di pezzi del puzzle, mentre la combinazione di due pezzi viene fatta in $O(1)$. Il tutto viene ripetuto n volte perché si parte da n pezzi ed a ogni iterazione si diminuisce di un pezzo, portando la complessità totale dell'algoritmo a $O(n^2)$.

4 Compilazione ed Esecuzione

Per compilare le varie classi è sufficiente invocare il comando `make` all'interno della directory `parte1`.
Le istruzioni di compilazione presenti nel makefile non specificano la versione target, viene

quindi scelta la versione di default che deve essere la 7 o superiore come richiesto dalla specifica.
L'invocazione del domando make produrrà la compilazione di tutte le classi nella directory `bin`.
Per eseguire il programma ci sono due possibilità:

1. Dall'interno della direcotry `bin` invocare `java PuzzleSolver.PuzzleSolver percorsoFileInput percorsoFileOutput`
2. Dalla directory principale, la stessa contenente il makefile eseguire lo script bash `PuzzleSolver.sh` passando come parametri il percorso dei file di input e di output.

Se non viene specifiato alcun parametro verranno usati come input i 10 file di test presenti nella direcotry `bin`.

5 Test

Per testare l'effettiva funzionalità della soluzione sono stati svolti 10 test al fine di verificare il corretto funzionamento dell'algoritmo anche nei casi limite. I test effettuati sono:

1. Puzzle di un elemento solo
2. Puzzle quadrato con numero di righe pari (4x4)
3. Puzzle quadrato con numero di righe dispari (5x5)
4. Puzzle rettangolare con numero di righe maggiore del numero di colonne (6x5)
5. Puzzle rettangolare con numero di righe inferiore al numero di colonne (5x6)
6. Puzzle composto da una sola colonna (8x1)
7. Puzzle composto da una sola riga (1x4)
8. Puzzle di un solo elemento ma con dati errati, gli elementi di confine avevano id diverso da VUOTO
9. Puzzle con un pezzo mancante
10. File vuoto

In tutti i test si è ottenuto il risultato atteso.
Gli ultimi tre test riguardano situazioni che non si possono verificare ma sono stati fatti comunque per verificare la solidità dell'algoritmo.