Relazione PCD - parte 2

Gabriele Pozzan
matricola 1051239

gennaio 2015

Il programma implementa un algoritmo di risoluzione concorrente di un puzzle.

1 - Cambiamenti rispetto alla prima parte

Sono stati effettuati dei cambiamenti importanti nell'organizzazione delle classi e nell'algoritmo di risoluzione, che è stato ampiamente semplificato.
Le cause di questi cambiamenti sono state:

- il rendersi conto dello scarso costo in termini temporali della fase di input da file delle informazioni per costruire i vari pezzi (mentre nella visione precedente era data grande importanza al fatto di poter cominciare a risolvere il puzzle durante il caricamento dei dati);
- la difficoltà a rendere effettivamente concorrenti le operazioni richieste dal precedente algoritmo di risoluzione: il meccanismo di *merging* dei vari *SetOfPieces* e *PuzzlePiece* richiedeva una serie di operazioni che mal si adattavano alla concorrenza (il problema principale era il cambiamento concorrente dell'*idSet* di alcuni pezzi presi in considerazione da più thread contemporaneamente); si era arrivati al punto da dover rendere sincronizzata la quasi totalità del codice di soluzione, cancellando i benefici dell'uso della concorrenza.

1.1 - Cambiamenti nella gerarchia

In conseguenza al cambiamento dell'algoritmo è stata cambiata anche la gerarchia di classi.

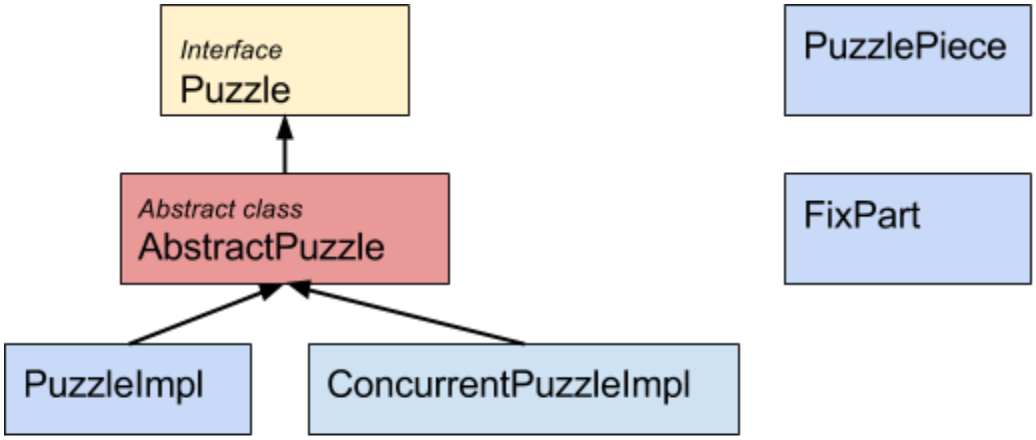

Figura 1 - Gerarchia di classi del package *engine*

- L'interfaccia *Puzzle* rappresenta il contratto pubblico della logica di risoluzione e offre i metodi *initialize* e *solve*
- La classe astratta *AbstractPuzzle* fornisce un certo livello di ereditarietà di implementazione per le concretizzazioni *PuzzleImpl* e *ConcurrentPuzzleImpl*, nello specifico fornisce le strutture dati per rappresentare il puzzle (ovvero l'hashmap *pieceIndex* e l'array doppio di *PuzzlePice matrix*), conserva i riferimenti ad alcuni pezzi particolari (quelli posti negli angoli in alto a sinistra, in alto a destra e in basso a sinistra del puzzle) e implementa il metodo *initialize* che preleva le informazioni per costruire i vari pezzi dal file di input, costruisce i vari pezzi e li salva in *pieceIndex*.
- Le concretizzazioni di *AbstractPuzzle* che implementano il metodo *solve* in una versione sequenziale e una versione concorrente.
- La classe *PuzzlePiece* che rappresenta una tessera di puzzle e fornisce semplicemente un metodo per ottenere le informazioni sui propri vicini e la ridefinizione del metodo *toString* che restituisce il carattere contenuto nella tessera.
- La classe *FixPart* che estende *Runnable* e implementa la logica di risoluzione di una parte di puzzle.

1.2 - Cambiamenti nell'algoritmo di risoluzione (+ versione concorrente)

Una serie di test hanno reso evidente il fatto che la fase di caricamento dei dati di input e di creazione dei vari *PuzzlePiece* non fosse così onerosa come ritenuto.
Si è dunque pensato di cominciare la soluzione del puzzle solo una volta creati tutti i pezzi, ottenendo così i seguenti vantaggi:

- la dimensione del puzzle è nota all'inizio dell'algoritmo di risoluzione;
- c'è un punto di partenza di default (il pezzo in alto a sinistra);
- il puzzle può essere diviso in aree sulle quali diversi thread possono lavorare senza entrare in conflitto.

L’algoritmo dunque parte dal pezzo nell’angolo in alto a sinistra e ha tre possibili versioni:

- **Caso limite**: nel caso il numero di colonne del puzzle fosse minore o uguale a due verranno creati due thread che lo risolveranno a partire dall’angolo in alto a sinistra e dall’angolo in basso a destra.
- **Caso limite**: nel caso il numero di righe fosse minore o uguale a due verranno creati due thread che procederanno in modo simile al punto precedente.
- **Caso normale**: in tutti gli altri casi il puzzle verrà risolto in questo modo: il main thread scorrerà il primo blocco di colonne e ogni tre righe farà partire un thread che risolverà un gruppo di tre righe.

Nella pratica ad ogni passo dell’algoritmo viene invocato il metodo *fixNeighbors* della classe *AbstractPuzzle* su uno degli elementi dell’array doppio di *PuzzlePiece* chiamato *matrix*.
Il metodo *fixNeighbors* controlla che per un dato pezzo ci siano vicini definiti (con carattere diverso da “VUOTO”) e non ancora salvati nell’array e li salva al posto giusto in *matrix*.

La classe *FixPart*, estensione di *Runnable* rappresenta la logica di risoluzione di una parte del puzzle, una sua istanza contiene le informazioni su:

- Un punto di partenza, dato dalle coordinate *i*, *j* del pezzo da cui iniziare a scorrere il blocco di righe o colonne;
- Come spostarsi ad ogni passo della soluzione: questa informazione è salvata nei due campi dati *step_i* e *step_j* che vengono sommati alle coordinate del pezzo corrente ad ogni iterazione;
- Quante iterazioni compiere, questa informazione è salvata nel campo dati *limit*.

Queste informazioni permettono di usare *FixPart* per tutti e tre i casi dell’algoritmo descritti sopra.
Il metodo *run* di *FixPart* semplicemente cicla un numero di volte uguale a *limit* e chiama *fixNeighbors* sui pezzi di puzzle individuati dalle coordinate *i* e *j* aggiornate ad ogni ciclo in base al valore di *step_i* e *step_j*.

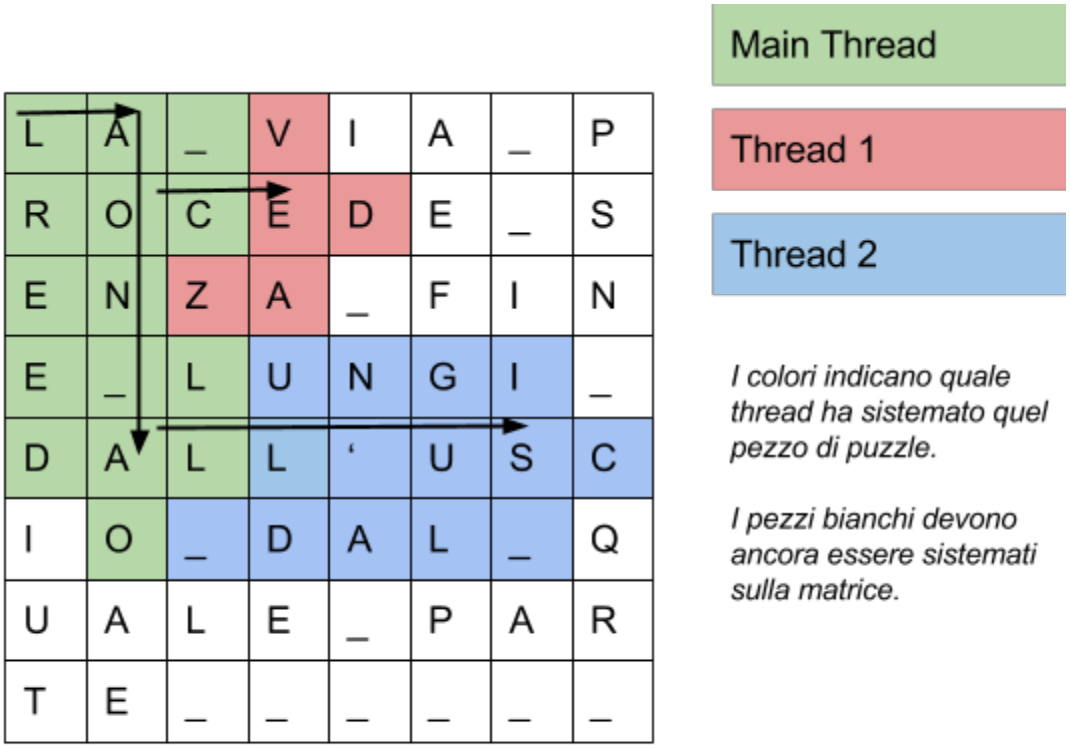

Figura 2 - Esempio potenziale di stato del puzzle durante la risoluzione.

2 - Note sulla versione concorrente dell'algoritmo

2.1 - Thread pool

La logica di risoluzione è stata implementata con un oggetto *Runnable* che rappresenta un *task* che viene eseguito poi da una *thread pool* costruita su un oggetto di tipo *ExecutorService*.
Nei primi due casi dell'algoritmo (numero ristretto di righe o di colonne), la thread pool sarà di tipo *fixed* con due thread, nel terzo caso (più di due righe e più di due colonne) la thread pool sarà di tipo *cached*, ovvero il numero di thread creati e avviati varierà dinamicamente in base al numero di task da completare, si è scelta questa versione perché le dimensioni del puzzle potrebbero essere qualsiasi.

2.2 - Interferenze

Non è stato necessario usare lock o sincronizzazioni perché i vari thread lavorano sempre su aree diverse e ben separate del puzzle e di conseguenza su aree diverse delle strutture dati che lo rappresentano (vengono riempite diverse righe e colonne dell'array, vengono richiesti diversi elementi dell'hashmap).