Informazioni sul documento

Nome documento	Relazione Seconda Parte del progetto PuzzleSolver
Versione documento	v.2.0.0
Data redazione	2015-01-10
Redattore	Tesser Paolo

Sommario

Lo scopo del documento è quello di fornire una presentazione della seconda parte del progetto PuzzleSolver da realizzare, descrivendo e motivando le scelte attuate in questa fase.

Indice

1 Cambiamenti e Aggiunte

In questa sezione verranno descritti i cambiamenti apportati alla precedente versione del programma per permettere all'algoritmo di risoluzione di essere parallelizzato.
In generale non vengono effettuate particolari rivisitazioni della precedente struttura, ma vengono introdotte nuove gerarchie ed estese alcune precedentemente create.
Il client rappresentato dalla classe **PuzzleSolver** andrà modificato solo nella scelta della politica di esecuzione dell'algoritmo, che da sequenziale passerà a parallela utilizzando la nuova classe descritta nella sezione 1.1.1.

1.1 Estensioni

1.1.1 Package solver

Questo package contiene le classi che gestiscono la risoluzione del puzzle.
Viene estesa la gerarchia che comprendeva alla base l'interfaccia **SolverStrategy** e come sottotipo la classe **SolverAlgStrategy** con una nuova classe **SolverParStrategy** responsabile della risoluzione parallela del puzzle. La nuova classe avrà all'interno del metodo **executeSolve** il flusso di quali e quanti thread andranno lanciati per ordinare il puzzle.

1.2 Nuove gerarchie

1.2.1 Package solver

Questo package contiene le classi che gestiscono la risoluzione del puzzle.
In esso vengono aggiunte le classi che contengono l'attività logica che i diversi thread andranno ad eseguire.
La gerarchia creata ha alla base una classe astratta **BasicThread**, che implementa l'interfaccia **Runnable**, utilizzata per memorizzare i membri condivisi da tutti i task come il riferimento all'oggetto condiviso dai Thread per comunicare tra loro o il riferimento all'oggetto Puzzle da risolvere.
Viene estesa da **AngleTileThread**, **FirstColThread**, **RowThread**.
AngleTileThread è il task che ha il compito di cercare e ordinare, nella struttura che contiene il puzzle risolto, il pezzo avente idWest e idNorth uguali a VUOTO e quello avente e idSouth uguali a VUOTO.
FirstColThread è il task che ha il compito di ordinare la prima colonna del puzzle. Per farlo può partire dai precedenti pezzi trovati a seconda di quanto specificato nel paramentro attuale del task durante la sua creazione. I valori che potrà ricevere saranno: "up" per partire dal basso e salire fino alla metà della colonna, mentre "down" per partire dal basso e arrivare sempre fino alla metà.
RowThread è il task che serve per riordinare tutte le righe del puzzle a partire dal primo elemento di ciascuna di essa.

Tutti queste classi svolgono il loro compito nel metodo **run()**.

1.2.2 Package logger

Questo package contiene solamente una classe che mi serve per la gestione di un file testuale di log che viene usato per tracciare l'esecuzione del programma in alcuni punti.

Essendo non importante ai fini delle richieste del proponente non ne verrà fornita una rappresentazione grafica.

In seguito viene illustrato il package solver contenente sia la gerarchia ampliata degli algoritmi di risoluzione, sia quella dei Thread che vengono avviati per ordinare il puzzle.

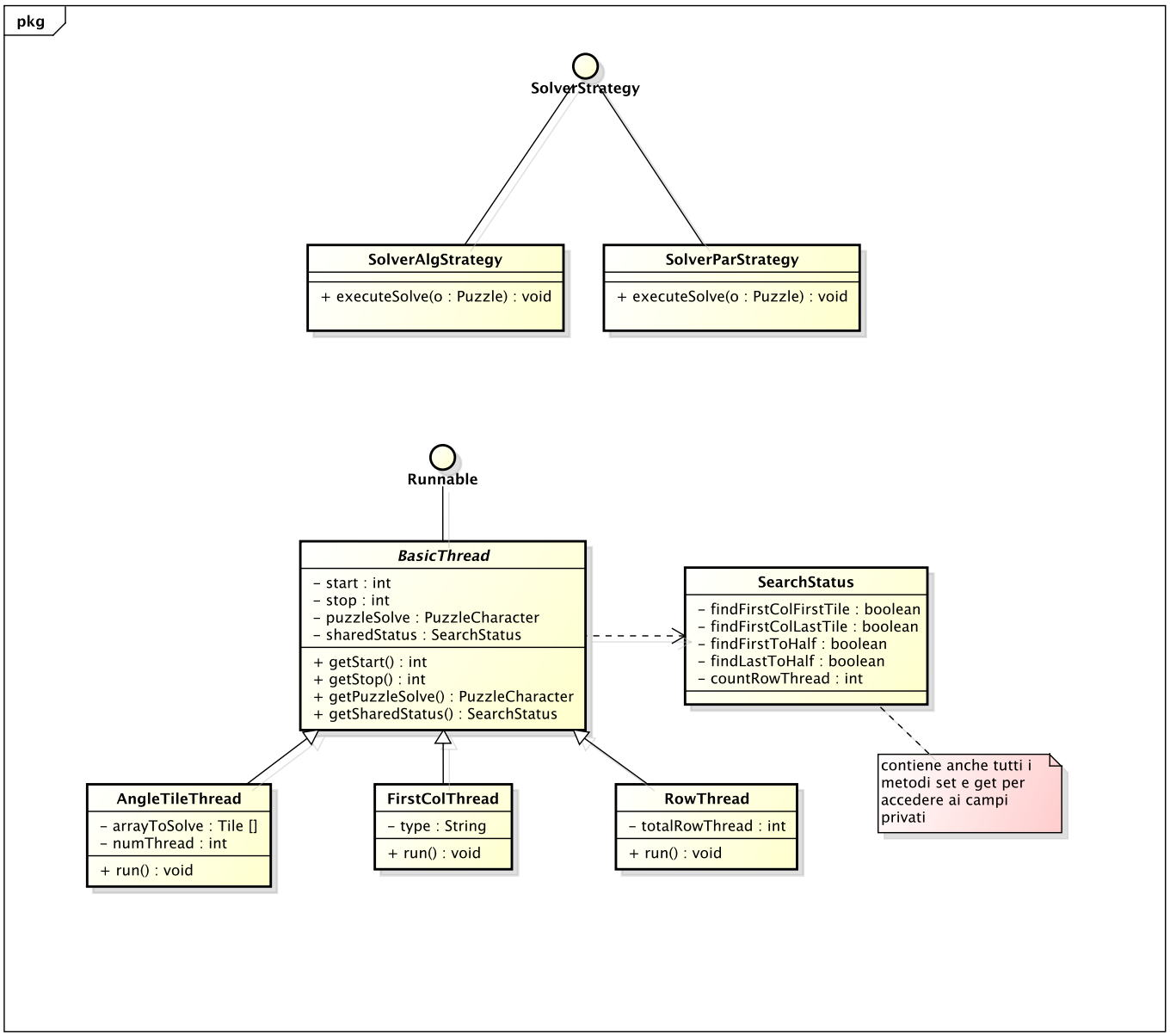

Figura 1: Package Solver

2 Algoritmo di risoluzione (parallelo)

L'algoritmo scelto per la risoluzione del puzzle è tipo di parallelo come richiesto dalla specifica relativa alla seconda parte del progetto.
Per arrivare alla soluzione, le strutture dati introdotte nella prima parte rimangono invariate. Si userà quindi sempre una HashMap per i tasselli del puzzle ancora disordinati e un array bidimensionale per i vari pezzi in ordine corretto.
Se nella precedente versione queste erano le uniche due strutture dati di cui l'algoritmo aveva bisogno, per la versione parallelizzata viene usato anche un array mono dimensionale di oggetti Tile che contiene in ordine casuale i valori della HashMap.
Di seguito vengono esposte le sequenze che vengo eseguite e i thread che vengono lanciati dall'algoritmo, correlate da grafici che mostrano come essi agiscano sulle strutture dati utilizzate.

Nota: nella seguente spiegazione, a differenza della precedente versione della relazione, non verranno introdotti grafici esplicativi della risoluzione del puzzle in quanto molto simili come logica a quelli già inseriti nella parte1.

1. **Ricerco il primo elemento del puzzle (quello in alto a sinistra) e quello in basso a sinistra.**
   Per fare ciò utilizzo diversi thread, in un numero arbitrario a seconda della dimensione del puzzle, che partono da posizioni diverse dell'array mono dimensionale citato precedentemente fino alla posizione meno uno di dove andrà a partire il thread lanciato successivamente.
   Una volta trovati vengono lanciati due thread per ordinare la colonna più a sinistra come vedremo nel passo seguente;

2. **Ordino la colonna più a sinistra (quella con i tasselli aventi id ovest uguale alla stringa VUOTO).**
   Per fare questo, come detto precedentemente, vengono lanciati due thread. In particolare, quando è stato trovato il pezzo con idNorth e idWest uguali a VUOTO può essere avviato il thread che ordina la prima metà della colonna, partendo appunto dall'alto e scendendo verso il basso fino alla metà. Quando invece viene trovato quello con idSouth e idWest uguali a VUOTO può essere avviato il thread che ordina la seconda metà partendo sta volta dal basso e risalendo fino a ordinare la parte mancante.
   La sequenza di avvio di questi due thread non è importante e dipenderà da quando i thread descritti prima troveranno i pezzi necessari.
   La metodologia di ordinamento della colonna è analoga a quella dell'algoritmo sequenziale, utilizzando quindi l'idSouth o l'idNorth per andare di passo in passo a ricavarsi i valori corretti direttamente dall'HashMap e inserirli nell'array bidimensionale finale;

3. **Ordino tutte le righe.**
   Una volta completato il passo precedente, il main thread potrà lanciare l'avvio dei thread che si occupano dell'ordinamento delle righe a partire dal primo elemento di ciascuna di essa, ossia da quei pezzi che formano la prima colonna e che sono già stati ordinati.
   Verranno avviati tanti task quante saranno le righe che compongono il puzzle, ma il numero di thread sarà limitato a un valore arbitrario a seconda del numero di righe per non far aumentare troppo i costi nel caso ci trovassimo di

fronte a puzzle con un elevato numero di righe appunto.
La metodologia di ordinamento della riga è analoga a quella dell'algoritmo, ma invece di scorrere tutte le righe, si limiterà a ordinare quella decisa nel costruttore.
Solo quando tutti questi task saranno completati il main thread potrà continuare e terminare la sua esecuzione, andando ad effettuare le operazione finale richiesta dalla specifica.

3 Gestione dei Thread

Nella seguente sezione vengono descritte le conseguenze che possono avvenire con l'avvio dei thread necessari alla risoluzione in parallelo del puzzle.

3.1 Numero Thread

Si può sempre valutare il caso peggiore e sapere al massimo quanti thread ci saranno in esecuzione nelle diverse fasi dell'esecuzione dell'algoritmo.
Di seguito verrà fornito il numero, illustrandolo in rapporto al flusso dell'algoritmo già descritto nella sezione 2.

1. Durante la ricerca e la sistemazione del primo elemento del puzzle (quello in alto a sinistra e quello in basso a sinistra) il numero massimo di thread attivi nel caso peggiore di dimensione del puzzle in input è di 8:

   - **5**: thread **AngleTileThread**;
   - **2**: thread **FirstRowThread**;
   - **1**: main thread.

   Nota: la successiva fase potrà essere già avviata in questa. Perciò il conteggio dei thread nel caso peggiore per quella, sarà relativo al momento in cui i thread AngleTileThread termineranno la loro esecuzione;

2. Durante l'ordinamento della colonna più a sinistra avrò sempre massimo 3 thread attivi:

   - **2**: thread **FirstColThread**;
   - **1**: main thread.

3. Durante l'ordinamento di tutte le righe il numero massimo di thread attivi nel caso peggiore di numero di righe presenti è di 7:

   - **6**: thread che lanciano i diversi task **RowThread** che possono essere in numero superiore a 6 ed essere messi in coda sui diversi thread;
   - **1**: main thread.

3.2 Interferenze, Deadlock, Attesa attiva

- **Interferenze**: questo fenomeno non può accadere mai sull'oggetto condiviso, in quanto ogni accesso ad esso da parte dei diversi thread in esecuzione, è effettuato dentro a una sezione sicura garantita dal costrutto **synchronized** sull'oggetto appunto condiviso. Per la natura dell'algoritmo parallelo i campi dati della classe **PuzzleCharacter** possono anche non essere sincronizzati;

- **Deadlock**: questo fenomeno non può accadere in quanto il codice è stato progettato per garantire che mai nessun oggetto debba aspettare il lock detenuto da un altro oggetto che a sua volta sta aspettando il lock detenuto dall'oggetto in attesa;

- **Attesa attiva**: questo fenomeno non si verificherà mai, in quanto l'unico thread che cicla su una condizione è il main. Durante le iterazioni però, se la condizione è vera e quindi si continua ad "aspettare" esso viene sospeso

attraverso il metodo **wait()**. Questo garantisce che il thread si metta in attesa passiva e non spreca risorse sul processore fintanto che qualcun altro non lo risvegli tramite una **notifyAll()**.

4 Costrutti di concorrenza

Per come è stata creata la struttura che immagazzina il puzzle e per come si è pensato di risolverlo, l'esecuzione in parallelo non necessita di particolare blocchi sull'oggetto in questione.
I costrutti che vengono utilizzate sono i seguenti.

4.1 Oggetto condiviso

Viene utilizzato un oggetto condiviso tra tutti i thread, compreso il main, per scambiarsi messaggi sullo stato di ordinamento del puzzle e decidere quando procedere o meno alla fase successiva di risoluzione.
Per farlo utilizziamo la classe **SearchStatus** che contiene le diverse condizioni che i thread vanno a controllare per chiamare o meno i metodi **wait()** o **notifyAll()**, descritti successivamente.

4.1.1 Synchronized

Il metodo **synchronized** viene utilizzato per permettere ai thread di eseguire in maniera atomica delle determinate operazioni che rientrano in una sezione critica e cioè una parte di programma che usa l'oggetto condiviso **SearchStatus** dai diversi thread. Questo garantirà che gli altri thread avviati non modifichino lo stato di quello oggetto fintanto che il primo thread che ha preso il lock sull'oggetto non abbia finito.

4.2 wait()

Il metodo **wait()** viene invocato sull'oggetto condiviso solo dentro al main thread quando la condizione che si sta controllando è vera. Questo significa nel nostro caso che i thread lanciati, per un determinato ordinamento, non sono ancora terminati e il main dovrà aspettare che finiscano per proseguire il suo flusso.

4.3 notifyAll()

Il metodo **notifyAll()** viene invocato sull'oggetto condiviso quando viene soddisfatta una particolare condizione. Questo vuol dire e il main thread deve essere appunto risvegliato e controllare la sua condizione d'attesa. Nel caso fosse falsa vorrà dire che il compito dei thread che hanno invocato la notifyAll sarà terminato e l'esecuzione potrà continuare.

5 Note

5.1 Versione JVM

La versione di Java presente nella macchina utilizzata per l'implementazione del codice è quella 1.8.0_20.
Non sono stati usati però costrutti particolari di questa versione e per l'utilizzo di API si è sempre fatto riferimento alla documentazione ufficiale della 1.7.
Per essere sicuri di rispettare la specifica che richiedeva al massimo la versione 1.7, si è testata la compilazione e l'esecuzione dell'applicativo sulle macchine di laboratorio dopo avere eseguito da terminale le istruzioni presenti al seguente link: http://www.studenti.math.unipd.it .

5.2 Compilazione

Dalla root principale consegnata è possibile avviare il comando per la compilazione attraverso l'istruzione **make**.
Se si vuole lanciare il programma, testandolo con degli input definiti dal fornitore per provare l'applicativo, è possibile eseguire sempre da root il comando **make load** il quale lancerà il programma prendendo due file fissi come input e output.
Se si vuole invece lanciare il programma con dei file di test personalizzati è possibile eseguire il seguente script bash che compilerà automaticamente e lancerà l'applicativo:
bash puzzlesolver.sh input output, questo riceve i due file e lancia l'avvio del programma.

Attenzione: lo script bash che lancia l'esecuzione del comando *java* sul main principale del programma va ad eseguire il comando in un livello inferiori rispetto ad esso. Bisogna fare quindi attenzione al percorso del file in input che potrebbe generare un'eccezione qualora non fosse corretto.