

PROGETTO EDUGREP

1. INTRODUZIONE

Edugrep è una reimplementazione della principale funzionalità della famosa utility Unix *grep*, ovvero la ricerca di espressioni regolari in testi. Il funzionamento del programma si basa sulla simulazione di automi a stati finiti non deterministici, modello che ben si adatta allo scopo, ottenuta tramite un interprete (o *parser*) di espressioni regolari (classe *Automaton*) e un'unità di ricerca in base ad automi (classe *Matcher*).

Il software lavora sull'alfabeto ["a-zA-Z"] e, per la formulazione di espressioni regolari, accetta i simboli:

- "+" per la ricerca in OR, cioè a fronte di una espressione regolare del tipo *espressione1 + espressione2*, verranno cercati match per *espressione1* o *espressione2*;
- "*" per la ripetizione di un carattere o di un'espressione, cioè a fronte di una espressione regolare del tipo *a** saranno rilevate da 0 a infinite ripetizioni del carattere *a*;
- "(" e ")" per specificare le priorità al parser e per includere intere espressioni sotto una *Greene star*. Esempio *(a+c)** rileverà da 0 a infinite ripetizioni dell'espressione tra parentesi, cioè *a* o *c*, quindi saranno rilevate tutte le stringhe del tipo: "", *a*, *c*, *aa*, *ac*, *cc*,

2. REQUISITI

I requisiti per utilizzare *edugrep* sono :

- distribuzione del sistema operativo Linux che supporti l'installazione di pacchetti software in formato *deb* (Debian);
- corretta installazione dell'interprete Ruby, versione 1.8 o successiva (normalmente il pacchetto per Linux è chiamato:

```
ruby.<tipo_architettura>(.numero_versione)
```

dove *tipo_architettura* è la sigla che identifica l'architettura hardware del sistema operativo e *numero_versione* è il codice della versione del programma, tra parentesi poiché spesso visualizzato solo nel dettaglio da parte dei gestori di pacchetti;

- una shell in grado di eseguire script compatibili con *sh* .

Essendo Ruby un linguaggio interpretato, esso è molto portabile e il codice scritto in questo linguaggio può essere eseguito su qualsiasi architettura (hardware e software) per la quale sia disponibile l'interprete Ruby; per poterne però apprezzare tutte le funzionalità, è necessaria una shell in grado di gestire il *piping* (es. *sh*, *bash*), ovvero la ridirezione dell'output di un processo all'input di un altro.

3. DESIGN

Per quanto riguarda il design, si è cercato di seguire le regole di buona programmazione, in modo da ottenere un codice fortemente modulare, robusto e il più possibile chiaro.

Si è anche scelto di renderlo il più possibile simile al *grep* originale per quanto riguarda il comportamento: *edugrep*, infatti, accetta input solo tramite argomenti da linea di comando e da standard input all'avvio (per realizzare il funzionamento tramite *pipe*) e non richiede quindi particolare interazione con l'utente, permettendo quindi l'esecuzione "batch". Questa scelta ne semplifica l'uso nel caso di pattern molto complessi e/o di file voluminosi (anche se occorre ricordare che, a causa della natura del linguaggio interpretato e della scarsa ottimizzazione, tale utilizzo sia sconsigliato poiché poco performante).

Scomposizione moduli e implementazione

Le varie funzionalità sono state implementate in 8 classi:

1. **edugrep.rb**

La vera e propria main class del programma, che, oltre a gestire l'input (da file o da *stdin*, cioè da *pipe*) e una parte delle stampe di output, è lo scheletro del programma, da cui vengono istanziate e chiamate al momento opportuno tutte le altre classi, passando loro gli eventuali dati richiesti.

Questa classe offre anche la possibilità di stampare a schermo gli automi risultanti, prima e dopo l'ottimizzazione. Questa funzionalità è attivabile con l'opzione a linea di comando *verbose* (vedi *man page*).

Per un periodo dello sviluppo si era pensato di procedere ad un refactoring, suddividendo alcuni compiti di questa classe tra le altre ed, eventualmente, aggiungendone una per la gestione del file; si è concluso in seguito di non procedere a tale refactoring, convinti che mantenendo il codice sufficientemente ordinato e commentato, la classe potesse rimanere comunque leggibile e manutenibile.

2. **Automaton.rb**

I metodi di questa classe sono quelli che svolgono tutta la prima parte del "lavoro" del programma, cioè si occupano, in modo ricorsivo, di compilare il pattern della espressione regolare inserita simulando un ASFND (automa non deterministico a stati finiti), rappresentato da tre array paralleli, su cui poi si baserà il brute-force matching applicato al testo. Naturalmente in questa classe è presente anche un minimale controllo di errore, che, in caso di pattern mal formulati, solleva un eccezione. Tale eccezione causerà poi la stampa di un messaggio su *stderr* e l'interruzione dell'esecuzione del programma.

Per mantenere una certa semplicità di implementazione, si accetta che l'automa risultante sia fortemente ridondante e contenga stati inutili.

3. **Optimizer.rb**

Questa classe, tramite il suo metodo *ottimizza()*, ovvia per quanto possibile alla forte ridondanza introdotta dalla classe Automaton, per alleggerire il carico di lavoro sul Matcher. Il suo funzionamento consiste in una scansione degli array, durante la quale si eliminano gli stati inutili (sequenze di transizioni non deterministiche con un solo stato di uscita), occupandosi anche però, ad ogni cancellazione di uno stato, di riportare gli altri in uno stato coerente (solitamente decrementando l'indice a cui puntano) e di tenere traccia della posizione finale dello stato iniziale.

E' possibile disabilitare l'ottimizzazione tramite l'opzione a linea di comando *nooptimize*.

4. **ParseException.rb**

Questa classe implementa semplicemente una estensione della classe Exception, a cui non aggiunge in effetti nulla, ma consente di avere una nostra eccezione da intercettare in caso di espressioni non riconosciute dall'automa, rendendo più leggibile il codice.

5. **Matcher.rb**

Questa classe contiene i metodi che si occupano della ricerca su testo. Il metodo *ricerca()* effettua una ricerca sul testo ricevuto come parametro, iniziando dall'indice dato, fino ad un match o ad un fallimento e restituisce un flag. Il metodo *ricerca_scan()*, che simula il *brute-force matching*, chiamando in loop *ricerca()*, passandogli il testo (nel nostro caso abbiamo scelto di passare una riga del testo alla volta, sempre cercando una somiglianza con *grep*) e incrementando ogni volta il contatore. Il metodo *ricerca_scan()* tiene traccia dei match trovati e restituisce il loro numero alla classe principale che, in caso di almeno un match, stampa la riga a schermo.

La classe Matcher offre anche la possibilità di stampe di dettaglio aggiuntive (da attivare con l'opzione *verbose*) e di debug (attivabili con l'opzione *debug*).

La chiamata a questa classe può essere esclusa inserendo l'opzione *pretend*. Escludendo la chiamata alla classe Matcher, verrà soltanto eseguito il parsing dell'espressione regolare (utile se si desidera soltanto visualizzare la configurazione dell'automa per una determinata espressione).

6. **Util.rb**

Questa classe è destinata a contenere metodi di utilità utilizzabili dalle altre classi. Al momento, contiene unicamente il metodo utilizzato per effettuare le stampe di stato degli automi.

7. **ParseOpt.rb**

Questa classe implementa lettura e gestione dei parametri da riga di comando e la stampa di aiuto invocata lanciando *edugrep* con l'opzione -h (o --help).

4. **UNIT TEST**

Per assicurare il corretto comportamento di edugrep durante lo sviluppo (e in previsione di sviluppi futuri) sono stati realizzati due unit test:

- **test_automaton.rb**

Comprende due tipologie di test sulla costruzione dell'automa:

- `test_simpleregexp()` , `test_optimized_simpleregexp()`

verificano la costruzione dell'automa sull'espressione regolare `simple`, in forma sia ottimizzata che non, confrontando dimensione e contenuti dei tre array dell'automa con i risultati previsti dalle specifiche.

- `test_star_regexp()`

verifica la costruzione dell'automa del semplice pattern contenente la Greene star `a*`, confrontando dimensione e contenuti dei tre array dell'automa con i risultati previsti dalle specifiche.

- **test_global.rb**

Intende controllare il comportamento globale dell'applicazione, cioè di Automaton, Optimizer e Matcher, nel loro complesso. Per questione di modularità, e per permettere una semplice modifica ed estensione dei test effettuati, si è deciso di utilizzare un file di configurazione da cui attingere i parametri con cui testare il programma. Il file utilizzato è `testlist.txt`, che si trova all'interno della cartella `test`, e può essere modificato/ampliato rispettando poche e semplici regole, cioè:

- si può inserire un test per riga, secondo la sintassi:
`pattern@testo@risultato_aspettato`;
- si possono inserire commenti, iniziando la riga col carattere '#';
- non si devono lasciare righe vuote, perché ciò causa un errore durante la lettura (un problema che deve essere ancora risolto).

Samuele Baisi `ciccio87 <at> gmail <dot> com`

Giulio Vezzelli `giulio.vezzelli <at> gmail <dot> com`

Antonio Radogna `slatislao <at> gmail <dot> com`

<http://edugrep.googlecode.com>