Valutazione Numerica della Complessità

Esempio degli algoritmi di ordinamento

ASDL - Luca Tesei

Valutazione numerica delle prestazioni

- Validazione sperimentale della valutazione analitica della complessità in tempo degli algoritmi
- L'idea è molto semplice

Sia A un algoritmo implementato

- si creano K input randomici diversi, di dimensione crescente, per A. Ad esempio N_1, N_2, ..., N_m diverse dimensioni crescenti dell'input
- si esegue A sui K * m input e per ogni run si registra:
  - il tempo impiegato per l'esecuzione
  - eventuali altri indicatori numerici che caratterizzano la complessità di A
- si aggregano i dati calcolando quantità statistiche opportune
- si visualizzano i risultati con degli opportuni grafici

Esempio: algoritmi di ordinamento

- Consideriamo gli algoritmi di ordinamento che si basano su confronti
- Esempi: BubbleSort, HeapSort, QuickSort, MergeSort, InsertionSort, eccetera
- L'input per uno qualunque di loro è una sequenza di elementi da ordinare lunga un certo numero, diciamo N_i (i = 1, 2, ..., m)
- L'output è la sequenza ordinata
- Il **numero di confronti** effettuati per fare l'ordinamento può essere usato come indicatore per caratterizzare la complessità in tempo dell'esecuzione dell'algoritmo
- Oltre al numero di confronti si può misurare (questo è sempre possibile) il **tempo che ha impiegato l'algoritmo** per fare l'ordinamento (è bene usare la scala dei nanosecondi)
- Abbiamo bisogno di un **framework** che esegua gli algoritmi che vogliamo analizzare su sequenze di lunghezza crescente e che rilevi i dati che ci servono

Evaluation Framework

Un framework che fa quello che ci serve è implementato dalle classi:

- SortingAlgorithmEvaluationFramework.java
- SortingAlgorithmEvaluationFrameworkParameters.java
- SortingException.java
- SortingAlgorithm.java
- SortingAlgorithmResult.java

Più le varie classi che implementano l'interfaccia SortingAlgorithm, cioè i vari tipi di algoritmi di ordinamento

Si veda il codice allegato

Esecuzione del framework (1)

Per minimizzare le interferenze di altri programmi in esecuzione che possono influenzare il conteggio del tempo impiegato per l'esecuzione:

- chiudere tutte le applicazioni e tutti i servizi del sistema operativo non necessari
- staccare/disattivare la rete
- assicurarsi di aver implementato correttamente le classi degli algoritmi di ordinamento da testare
- eseguire la classe main del framework da riga di comando invece che dall'interno di Eclipse

NOTA: in ogni caso alcune interferenze saranno inevitabili perché il sistema operativo e i suoi servizi di base saranno sempre attivi e potranno essere schedulati nel processore durante l'esecuzione degli algoritmi di ordinamento in qualunque momento. Vedremo dopo un modo per abbassare eventuali picchi di tempo di esecuzione che derivano chiaramente da queste interferenze.

Esecuzione del framework(2)

1. salvare tutto e chiudere il proprio IDE
2. aprire una finestra del terminale (dipende dal sistema operativo, ad esempio su Windows bisogna eseguire cmd.exe oppure Prompt dei comandi)
3. `$> cd <path del workspace>/bin`
4. adattare il path e la sottocartella **`bin`** all'organizzazione del proprio IDE. La directory in cui si deve andare è quella in cui inizia la gerarchia di directory **`it/unicam/cs/asdl2021/es8` che contiene i file .class** delle classi .java
5. `$> mkdir CSVData`
6. oppure creare una directory alternativa del nome che si vuole nel posto che si vuole per far scrivere i dati in output
7. `$> java -cp . it.unicam.cs.asdl2021.es8.SortingAlgorithmEvaluationFramework CSVData`
8. Inserire al posto di **`CSVData`** il path relativo alla directory alternativa eventualmente creata al punto 6.
9. Attendere che siano state generate tutte le sequenze e che il programma esca

Dati prodotti in output

Nella cartella **`CSVData`** (o in quella alternativa creata al punto 6 della slide precedente) saranno presenti due file CSV (Comma Separated Values):

1. **`evalfram.csv`** - ogni riga, corrispondente a una certa sequenza randomica, contiene il numero di confronti e il tempo di esecuzione in nanosecondi per ogni algoritmo inserito nella classe **`SortingAlgorithmEvalutationFramework.java`** (dalla riga 50 in poi)
2. **`sequences.csv`** - contiene tutte le sequenze generate randomicamente e passate agli algoritmi. Ogni sequenza ha un codice, riportato nella prima colonna di ogni riga del file **`evalfram.csv`**

NOTA: i file CSV sono il formato standard minimo per lo scambio di dati, sono file di testo in cui sono riportate le righe e le colonne di una tabella di dati. Le righe sono le righe del file di testo, mentre le colonne sono riconoscibili all'interno di ogni riga tramite un carattere separatore, di solito la virgola o il punto e virgola. La presenza della riga di intestazione della tabella è opzionale. Il numero di colonne per riga può essere diverso.

Importazione dei dati

Per poter fare le nostre analisi dobbiamo importare il file **`evalfram.csv`** in un'applicazione che ci permette di gestire un foglio elettronico e creare dei grafici. Ad esempio:

- Microsoft Excel
- OpenOffice Calc
- Apple Numbers
- Google Fogli (in questa lezione useremo questo)

Utilizzare la funzione di importazione dei file CSV dell'applicazione scelta e importare l'intero file. In alternativa a un foglio elettronico si possono usare software specializzati per l'analisi statistica o matematica in generale, come R (ambiente RStudio) oppure MatLab. Infine si possono anche scrivere degli script in Python o altri linguaggi che mettono a disposizione librerie adatte per la statistica e i grafici.

Il file **`sequences.csv`** serve solo come registro delle sequenze per fare eventualmente dei controlli successivi su certe sequenze che hanno valori strani o particolari. I dati che ci servono per l'analisi sono solo su **`evalfram.csv`**

Aggregazione e pulitura dei dati (1)

Come dicevamo prima dobbiamo prima pulire i dati da eventuali interferenze dovute all'esecuzione di processi del sistema operativo in parallelo al framework di valutazione

Per far questo dobbiamo eliminare i valori esageratamente alti (picchi) dei tempi di esecuzione dovuti sicuramente a interferenze. Tali valori in statistica sono chiamati **outliers** e il metodo standard per determinare i valori di outlier è il seguente.

Data un insieme di valori numerici, **un valore v è un picco outlier** se

$$v >= Q3 + r * (Q3 - Q1)$$

dove Q3 è il valore del **3° quartile** dell'insieme e Q1 è il valore del **1° quartile** dell'insieme. **r** è una costante che può essere variata per rendere la soglia più o meno selettiva. Noi useremo r = 3.

Noi dovremo considerare 30 insiemi di K = 100 valori ciascuno, cioè i tempi di esecuzione relativi alle sequenze che hanno una stessa lunghezza: 50, 100, 150, 200, ..., 1500.

Aggregazione e pulitura dei dati (2)

I dati che dobbiamo pulire sono solo quelli in nanosecondi che riguardano i tempi di esecuzione. Il numero di confronti effettuati da ogni algoritmo non risente delle interferenze di processi in parallelo del sistema operativo!

Quindi da una parte conserveremo i dati relativi al numero di confronti così come sono, dall'altra elimineremo i picchi outlier dei tempi di esecuzione.

Si veda il foglio google allegato

Aggregazione dei dati

Seguendo le linee della valutazione analitica, cercheremo di stimare numericamente la complessità in tempo nel caso ottimo, medio e pessimo:

- Per stimare il **caso ottimo** cercheremo il **valore minimo** tra i K=100 valori (eventualmente meno perché alcuni sono stati tolti perché outliers) di ogni lunghezza di sequenza
- Per stimare il **caso medio** calcoleremo il **valore medio**, la **deviazione standard**, il **valore medio + la deviazione standard** e il **valore medio - la deviazione standard** tra i K=100 valori (eventualmente meno perché alcuni sono stati tolti perché outliers) di ogni lunghezza di sequenza
- Per stimare il **caso pessimo** cercheremo il **valore massimo** tra i K=100 valori (eventualmente meno perché alcuni sono stati tolti perché outliers) di ogni lunghezza di sequenza

Si veda il foglio google allegato

Realizzazione dei grafici

Per ogni algoritmo, nel nostro caso BubbleSort e HeapSort - Per ogni misura, nel nostro caso numero di confronti e tempo di esecuzione in nanosecondi

Costruiamo un grafico g dove riportiamo i valori per ogni valutazione, cioè caso ottimo, medio e pessimo

Nello stesso grafico disegnamo anche le funzioni $c * n^2$ e $c * n * \log_2 n$ dove c è una costante di scala opportunamente scelta per scalare le due funzioni più o meno agli stessi valori del grafico g.

Sono state scelte proprio queste due funzioni perché sappiamo che analiticamente esse sono un limite superiore e un limite inferiore per i casi medio e pessimo dei due algoritmi che abbiamo preso in considerazione. L'andamento del grafico g rispetto a queste due funzioni (opportunamente scalate) ci dà l'informazione visiva della complessità stimata numericamente

Si veda il foglio google allegato

Grafici di confronto

Oltre a mostrare le stime dei casi ottimo, medio e pessimo per ogni algoritmo e per ogni misura possiamo anche confrontare due o più algoritmi sullo stesso caso e sulla stessa misura per vedere le loro prestazioni relative

Anche qui possiamo riportare (nello stesso grafico) una o più funzioni di base (come n, n^2 e n * $\log_2$ n), opportunamente scalate, per confrontarle con gli andamenti delle curve disegnate

Si veda il foglio google allegato