DETERMINAZIONE DEL VINCITORE IN UN'ASTA COMBINATORIALE

Federica Zampedri e Sara Sogari

23 marzo 2010

Indice

1 Introduzione

Tale progetto si pone l'obiettivo di implementare e valutare un algoritmo completo per risolvere il problema di determinazione del vincitore in un'asta combinatoriale attraverso l'approccio della Bucket Elimination.

Un'asta combinatoriale è usata quando si vogliono vendere più oggetti contemporaneamente; gli offerenti possono sottoporre un'offerta su un sottoinsieme di beni. Ogni offerta sarà quindi rappresentata da:

- un sottoinsieme di beni
- il valore dell'offerta

Le offerte vincenti verranno scelte in modo da perseguire uno specifico obiettivo (nel nostro caso la massimizzazione del guadagno) e in maniera tale che ogni sottoinsieme assegnato sia disgiunto.

Con tale programma si è quindi cercato di simulare uno scenario simile a quello descritto sopra, dove a fronte di un'insieme di proposte, verrà determinato il sottoinsieme ottimale di offerte disgiunte.

2 Elenco delle classi

- **Bid.java**

  Classe contenente le proprietà (il prezzo proposto e l'insieme degli oggetti interessati) e i metodi (come la funzione che determina se due offerte distinte hanno oggetti in comune) dell'oggetto offerta.

- **Bucket.java**

  Classe che rappresenta il bucket vero e proprio. Contiene l'insieme delle relazioni presenti nel Bucket e i metodi necessari a processarlo e massimizzare la funzione costo.

- **BucketElimination.java**

  È la classe principale, in cui si creano le offerte, si assegnano i vincoli ai bucket corrispondenti, si massimizzano le funzioni costo e si propagano i valori per calcolare la combinazione di offerte che ottimizzi il guadagno.

- **Relazione.java**

  Rappresenta il vincolo fra due offerte distinte. Contiene tutte le possibili combinazioni valide che le variabili (offerte) in questione possono assumere.

- **SavitchIn.java**

  Classe Java usata per l'input da tastiera.

3 Struttura del programma

Il programma è stato scritto in JAVA e permette di testare l'algoritmo sia con istanze generate casualmente che con istanze create dall'utente; in entrambi i casi si procede con una lettura da file.

L'esecuzione ritorna un sottoinsieme di offerte tale per cui nessuna di queste condivida oggetti e il guadagno sia massimo. Inoltre calcola la complessità e il tempo di esecuzione, utili ai fini di studiare il comportamento dell'algoritmo.

Nota : si sono utilizzati come struttura dati principale gli ArrayList, che consentono di lavorare con liste di dimensione variabile.

3.1 Input

In ingresso il programma prende un insieme di offerte, ognuna consistente di un insieme di oggetti e di un prezzo associato.

L'istanza viene passata al programma tramite un file .txt in cui si elencano tutte le offerte e i relativi costi.

Esempio

85 1 3 15 4 6
99 8 6 2 3 6 7

dove ogni riga rappresenta un'offerta, il primo valore identifica il prezzo e i restanti numeri costituiscono il sottoinsieme degli oggetti.

Tale file può essere generato dall'utente o direttamente dal programma. Nel primo caso si dovrà solamente indicare il nome del file su cui è memorizzata l'istanza; nel secondo caso invece si procede con un opportuno metodo di generazione delle offerte random.

Più precisamente l'utente dovrà solamente inserire il numero di oggetti che desidera vengano messi in vendita. A partire da tale valore il programma genererà un numero casuale di offerte (entro un intervallo fissato a priori). Lo schema per calcolare un numero random nell'intervallo [min, max) è

$$Random\_number = (int)((max - (min + 1)) \cdot Math.random()) + min$$

Nel nostro caso l'intervallo è stato fissato a [2, 20). Lo stesso procedimento è stato usato anche per fissare casualmente il valore di ogni offerta nell'intervallo [20, 100).

Il passo successivo è stato quello di creare per ogni bid (offerta) il rispettivo scope, cioè l'insieme di oggetti per cui si propone l'offerta.

```
public ArrayList<Integer> creaScope(int numeroOggetti) {
    ArrayList<Integer> numbers = new ArrayList<Integer>(1);
    int scopeBid = (int) (Math.random()·(numeroOggetti) + 1);
    for (int j = 0; j < scopeBid; j++) {
        int rand = (int) (Math.random()·(numeroOggetti) + 1);
        while (numbers.contains(rand)) {
            rand = (int) (Math.random()·(numeroOggetti) + 1);
        }
        numbers.add(rand);
    }
    return numbers;
}
```

in cui l'intero numeroOggetti è il valore immesso dall'utente.

A questo punto si richiederà di inserire il nome del file .txt in cui memorizzare l'istanza del problema.

3.2 Assegnamento dei vincoli ai bucket

Per eseguire l'algoritmo, si chiederà di inserire il nome del file su cui è memorizzata l'istanza del problema. Da tale file verranno lette tutte le offerte e verranno inserite in un ArrayList di Bid (arrayDiBid).

A questo punto l'utente dovrà immettere l'ordine con cui desidera che le offerte vengano processate; sottolineiamo il fatto che l'ordine viene fatto inserire da tastiera invece di essere generato casualmente; questo per permettere all'utente di testare il programma utilizzando la medesima istanza ma con ordini diversi.

Abbiamo deciso di creare le relazioni solo dopo che è stato inserito l'ordinamento per due motivi: per evitare che la stessa relazione venga creata due volte ($R_{i,j}$ e $R_{j,i}$ rappresentano la medesima relazione, quindi non ha senso crearle entrambe) e per permettere che, nel momento in cui si popolano i bucket, sia sufficiente controllare solamente la prima variabile della relazione per conoscere il bucket che la deve contenere.

- si scorre tutto l'ArrayList contenente i bid in ordine inverso
- per ogni bid si crea la relazione con tutti i bid che lo precedeno nell'ordinamento e si controlla se nel loro scope ci sono elementi in comune

- in caso affermativo i due bid non possono essere selezionati contemporaneamente e quindi le uniche combinazioni possibili sono 00,01,10
- altrimenti posso selezionare indifferentemene l'uno, l'altro, entrambi o nessuno, ovvero 00,01,10,11
- infine si inseriscono le relazioni in un ArrayList di Relazioni

A questo punto si assegnano i vincoli $R_{i,j}$ ai bucket corrispondenti; notiamo che si inseriscono solo le relazioni che rappresentano una intersezione non vuota fra i due bid i e j.

Si procede con un ulteriore controllo per inserire nel bucket anche quei vincoli che saranno generati a runtime dal calcolo della funzione costo:

- si processa ogni bucket secondo l'ordinamento inverso (sia B_i il bucket preso in esame);
- considero tutte le variabili delle relazioni in B_i - la variabile i e le memorizzo in un ArrayList chiamato *variabiliX*;
- scorro tutto l'array di Bucket (in tale ArrayList i bucket sono già inseriti secondo l'ordinamento inverso) e cerco il primo bucket la cui variabile è contenuta in *variabiliX* (supponiamo sia B_j);
- a questo punto scorro tutte le variabili delle relazioni in B_j:
  - se sono presenti tutte le varibili contenute in *variabiliX* allora non devo aggiungere niente
  - altrimenti aggiungo fra le relazioni di B_j anche i vincoloi fra j e le variabili mancanti.

3.3 Join

Continuiamo processando ogni singolo bucket secondo l'ordinamento inverso; il metodo fondamentale in tale contesto è la funzione che implementa il join fra tutte le relazioni nel bucket.

```
//tale metodo implementa il join tra matrici
public static ArrayList<ArrayList<Integer>> join(ArrayList<ArrayList<Integer>> m1,
ArrayList<ArrayList<Integer>> m2)
{
    //matrice risultato
    ArrayList<ArrayList<Integer>> matriceX = new ArrayList<ArrayList<Integer>>(1);
```

```
    int bit;
    //rigaVar contiene tutte le variabili coinvolte nel join
    ArrayList<Integer> rigaVar = new ArrayList<Integer>(1);
    rigaVar = m1.get(0);
    for(int w=1;w<m2.get(0).size();w++)
        rigaVar.add(m2.get(0).get(w));
    // inseriamo rigaVar come prima riga della matrice risultato
    matriceX.add(rigaVar);
    for(int r=1;r<m1.size();r++)
    {
        //bit = primo bit di ogni riga di m1 (tranne la prima poiché contiene le variabili)
        bit = m1.get(r).get(0);
        for (int s=1;s<m2.size();s++)
        {
            //se la riga s di m2 inizia con bit...
            if (m2.get(s).get(0) == bit)
            {
                    ArrayList<Integer> rigai = new ArrayList<Integer>(1);
                    for (int w=0;w<m1.get(r).size();w++)
                        //...copio tutta la riga di m1...
                        rigai.add(m1.get(r).get(w));
                    for(int w=1;w<m2.get(s).size();w++)
                        //...più tutta la riga di m2 senza considerare la prima colonna
                        rigai.add(m2.get(s).get(w));
                    matriceX.add(rigai);
            }
        }
    }
    return matriceX;
}
```

Tale metodo riceve in input due relazioni $R_{i,j}$ e $R_{i,k}$ e restituisce la matrice risultante con variabili i, j, k

Esempio

i	**j**
0	0
0	1
1	0

i	k
0	0
0	1
1	0

e il risultato sarà

i	j	k
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0

3.4 Processamento dei bucket

Per ogni singolo bucket sommiamo tutte le funzioni ed eliminiamo la corrispondente variabile per massimizzazione, questo crea un nuovo vincolo con scope: tutte le variabili menzionate dai vincoli nel bucket in questione − la variabile corrispondente al bucket. La nuova funzione H creata verrà posta nel primo bucket successivo con rispettiva variabile presente nello scope di H.

3.5 Propagazione dei valori

Si propagano i valori per calcolare la tupla ottimale:

- Calcoliamo una tupla (parziale) che massimizzi la somma delle funzioni del primo bucket (secondo l'ordinamento)
- propaghiamo il valore della tupla al bucket successivo
- calcoliamo una tupla (parziale) che massimizzi la somma delle funzioni, dati i valori delle tuple propagate dai bucket precedenti
- continuiamo cos per tutti i bucket.

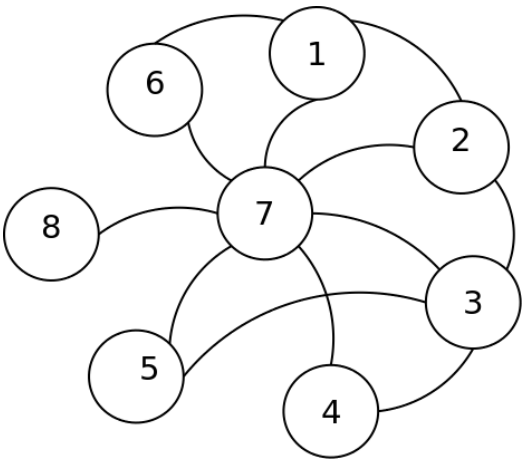
Il programma termina restituendo la tupla completa (che rappresenta la soluzione ottimale al problema), la complessità dell'istanza e il tempo di esecuzione.

4 Caso di studio

Vogliamo ora provare a evidenziare come l'ordinamento abbia un ruolo decisivo per quanto riguarda l'aspetto della complessità algoritmica e del tempo di esecuzione. Per far ciò testiamo l'algoritmo utilizzando un'istanza creata ad hoc e variando l'ordine con cui si processano i bucket.

Consideriamo per prima cosa un'istanza formata da un numero relativamente modesto di offerte. In questo caso il file da cui leggere e' stato creato manualmente, cercando di creare un grafo delle dipendenze che fosse significativo ai fini dello studio. Si sono create otto offerte, ognuna legata alle altre da un numero variabile di relazioni; quindi ad esempio il nodo relativo alll'offerta B7 è stato collegata a tutti gli altri vertici; il nodo rappresentate l'offerta B3 è in relazione con quattro vertici e così via. A questo punto sono stati scelti gli ordinamenti: come caso iniziale si è deciso di processare per primi i bucket più vincolati, successivamente abbiamo invertito l'ordine e infine abbiamo testato l'algoritmo con sequenze random.

L'istanza che abbiamo creato è:



Le relative offerte sono:

B1 : { 1, 8, 12 } offerta: 55
B2 : { 2, 11, 12 } offerta: 89
B3 : { 3, 9, 10, 11 } offerta: 72
B4 : { 4, 10 } offerta: 23
B5 : { 5, 9 } offerta: 18
B6 : { 6, 8 } offerta: 31
B7 : { 1, 2, 3, 4, 5, 6, 7 } offerta: 99
B8 : { 7 } offerta: 15

e il rispettivo file.txt risulta:

55 1 8 12
89 2 11 12
72 3 9 10 11
23 4 10
18 5 9
31 6 8
99 1 2 3 4 5 6 7
15 7

- Primo caso : o = { 8 6 5 4 2 1 3 7 }
  - Soluzione ottimale : 2 4 5 6 8
  - Guadagno : 176
  - Complessità : 7
  - Tempo di esecuzione : 73 ms

  In questo caso si sono ordinate le offerte in base al numero di vincoli, partendo da B8, legata da una sola relazione e terminando con B7. Possiamo notare che questo è il peggiore risultato ottenuto poiche‘ la complessità è la più alta possibile e il tempo di esecuizione è di 73ms.

- Secondo caso : o = { 7 3 1 2 4 5 6 8 }
  - Soluzione ottimale : 2 4 5 6 8
  - Guadagno : 176
  - Complessità : 3
  - Tempo di esecuzione : 3 ms

  Si sono ordinati in questo caso le offerte in maniera tale che prima vengano processati i bucket corrispondenti alle offerte meno vincolate. Come era prevedibile questo risulta essere il migliore ordinamento possibile, avendo complessità pari a 2 e tempo di esecuzione molto minore rispetto al primo caso.

- Terzo caso : o = { 7 1 2 3 4 5 6 8 }
  - Soluzione ottimale : 2 4 5 6 8
  - Guadagno : 176
  - Complessità : 2

- Tempo di esecuzione : 16 ms

Questo è un ordinamento casuale, creato senza usare nessuna euristica; si può notare che la complessità risulta essere la migliore ma rispetto al secondo caso il tempo di esecuzione è molto peggiorato.

- Quarto caso : o = { 5 6 1 4 2 7 3 8 }
  - Soluzione ottimale : 2 4 5 6 8
  - Guadagno : 176
  - Complessità : 5
  - Tempo di esecuzione : 44 ms

Anche questo è un ordinamento casuale e sia la complessità che il tempo di esecuzione sono mediocri, non buoni come il secondo caso ma nemmeno male come il primo.

Consideriamo ora un'istanza più complessa, con ad esempio 15 offerte e anche in questo caso testiamo l'algoritmo ordinamenti diversi. Come primo caso iniziamo con il processare per primi i bucket corrispondenti alle offerte maggiormente vincolate; poi testeremo l'algoritmo con ordinamento inverso rispetto al precedente. Infine proveremo a lanciarlo con sequenze casuali.

L'istanza utilizzata è

B1 : { 1, 8, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 } offerta: 62
B2 : { 2, 19, 20, 21, 22, 23, 24, 25, 26 } offerta: 81
B3 : { 3, 19, 27, 28, 29, 30, 31, 32 } offerta: 73
B4 : { 4, 20, 33, 34, 35, 36 } offerta: 55
B5 : { 5, 21, 37, 38 } offerta: 51
B6 : { 6, 17, 18, 22, 28, 33, 39, 40, 41 } offerta: 34
B7 : { 7, 29, 42 } offerta: 84
B8 : { 8, 36, 37, 50 } offerta: 65
B9 : { 9, 23, 41, 43, 44, 45 } offerta: 86
B10 : { 10, 24, 25, 30, 38, 43 } offerta: 24
B11 : { 11, 16, 40, 46 } offerta: 83
B12 : { 12, 31, 32, 47, 48 } offerta: 73
B13 : { 13, 26, 35, 44, 45, 47 } offerta: 64
B14 : { 14, 39, 46, 49, 50 } offerta: 56
B15 : { 15, 34, 42, 48, 49 } offerta: 30

- Primo caso : o = { 7 5 8 11 15 14 12 4 13 10 9 6 3 2 1 }
  - Soluzione ottimale : 7 5 11 12 4 9

  - Guadagno : 432
  - Complessità : 14
  - Tempo di esecuzione : 354596 ms

- Secondo caso : o = { 1 2 3 6 9 10 13 4 12 14 15 11 8 5 7 }
  - Soluzione ottimale : 7 5 11 12 4 9
  - Guadagno : 432
  - Complessità : 7
  - Tempo di esecuzione : 341 ms

- Terzo caso : o = { 1 7 2 8 3 10 6 11 4 12 5 13 15 14 9 }
  - Soluzione ottimale : 7 5 11 12 4 9
  - Guadagno : 432
  - Complessità : 9
  - Tempo di esecuzione : 801 ms

- Primo caso : o = { 7 4 8 10 1 11 5 12 2 13 15 6 9 14 3 }
  - Soluzione ottimale : 7 5 11 12 4 9
  - Guadagno : 432
  - Complessità : 10
  - Tempo di esecuzione : 1933 ms

5 Conclusioni

Si può notare che l'ordinamento adottato ha un impatto considerevole sulla dimensione delle funzioni create e quindi sulla complessità.

- La dimensione delle funzioni generate equivale al numero di variabili nel bucket meno la variabile del bucket stesso
- Massimizzare la funzione è esponenziale rispetto alla sua dimensione, di conseguenza l'algoritmo Bucket Elimination è esponenziale rispetto alla dimensione del bucket maggiore
- Più precisamente la complessità dell'algoritmo rispetto all'ordinamento o è in tempo $O(e \cdot d^{w(o)+1})$ e in spazio $O(n \cdot d^{w(o)})$ dove e è il numero di funzioni, d la dimensione più grande del dominio, n il numero di variabili e $w(o)$ è l'ampiezza indotta del problema rispetto all'ordinamento o che, in pratica, rappresenta il massimo numero di colonne fra le matrici risultanti dal join di tutti i vincoli presenti per ogni bucket meno la variabile del bucket stesso.

Dai nostri test infatti abbiamo potuto constatare che processando per primi i bucket corrispondenti ai nodi meno connessi, la complessità è molto minore rispetto a quella che si ottiene considerando un ordinamento contrario, ovvero processando per primi i nodi maggiormente connessi.