

1.5 Union Find

Dynamic connectivity

Consideriamo il caso: l'input è una sequenza finita di interi, dove ogni intero rappresenta un oggetto di un certo tipo, ed interpretiamo le coppie p, q come ' p è connesso a q '.

Dico per scontato che la frase 'è connesso a' sia una relazione di equivalenza, ovvero:

- Riflessiva: p è connesso a p
- Simmetrica: $p \rightarrow q \Rightarrow q \rightarrow p$
- Transitiva: $p \rightarrow q, q \rightarrow r \Rightarrow p \rightarrow r$

In questo caso, due oggetti sono nello stesso clade di equivalenza se e solo se entrambi sono connessi.

Il nostro obiettivo è quello di scrivere un programma che filtri le coppie esterne, ovvero le coppie dove entrambi gli elementi sono nello stesso clade di equivalenza.

In altre parole, quando un programma legge una coppia p, q dall'input, dovrebbe stampare la coppia in output solo se le coppie che ha ricevuto fino a quel punto NON implica che p è connesso a q .

Networks

Gli interi potrebbero rappresentare dei computer in una rete estera, e le coppie potrebbero rappresentare le connessioni nella rete.

Successivamente, il nostro programma determina se dobbiamo stabilire una nuova connessione diretta per p e q per poter comunicare, oppure se possiamo utilizzare una connessione già esistente.

Variable - nome equivalenze

In alcuni ambienti di programmazione è possibile dichiarare due nomi di variabile come equivalenti, ovvero che fanno riferimento allo stesso oggetto.

Dopo una serie di dichiarazioni di questo tipo, il sistema ha bisogno di determinare quando due dati nomi sono equivalenti.

Per specificare il problema, sviluppiamo una API che incapsuli le operazioni basilari che ci servono: inizializzare, aggiungere una connessione tra due elementi, identificare il componente contenente un elemento, determinare se due elementi sono nello stesso componente, e contare il numero di componenti.

- UF (int N) // inizializza N spazi con numeri interi (0-N-1)
 - void Union (int p, int q) // add a connection
 - int find (int p) // identificatore da 0 a N-1
 - boolean connected (int p, int q) // check component
 - int count // numeri di componenti
- API Union-Find ↑

Iniziamo con N componenti, e per ogni union() che include due diversi componenti decremente il numero di componenti di 1.

Ogni implementazione di questa API deve:

- Definire una struttura dati per rappresentare le connessioni NOTE:
- Sviluppare dei metodi efficienti per union(), find(), connected(), e count().

Per testare le funzioni delle API, includiamo un client nel main() che lo utilizza per risolvere un problema di connessione dinamico.

Degge il valore di N seguito da una sequenza pari di interi, chiamando find() per ogni coppia: se i due elementi sono già connessi, continua con la coppia successiva; se non lo sono, chiama union() e stampa le coppie.

Implementazioni

Quick find

Un approccio è quello di generare che p e q sono connessi se e solo se $\text{id}[p]$ è uguale a $\text{id}[q]$. In altre parole, tutti gli el. in un componente devono avere lo stesso valore in $\text{id}[i]$. Questo metodo è chiamato quick-find perché $\text{find}(p)$ ritorna semplicemente $\text{id}[p]$, che implica immediatamente che $\text{connected}(p, q)$ contiene solo il test di $\text{id}[p] == \text{id}[q]$, e ritorna True se e solo se p e q sono nello stesso componente.

Sorting

Selection Sort

Uno degli algoritmi più semplici è il sel sort. Per prima operazione, Trova l'elemento più piccolo nella collezione e lo scommia con il primo elemento. Successivamente, Trova il secondo elemento più piccolo e lo scommia con l'el. alla seconda posizione.

Si continua in questo modo finché l'array non è ordinato.

Ogni scambio pone un elemento nella sua posizione finale, quindi il numero di scambi è N . Inoltre, il tempo di esecuzione è dato dal numero di confronti.

Proposizione A: Il sel sort usa $\sim N^2/2$ confronti, ed N scambi per ordinare un array di N elementi.

Dimostrazione: Si può dimostrare questo fatto esaminando la 'TRACE', ovvero una Tabella composta da $N \times N$ elementi.

Più precisamente, l'esame del codice rivelà che, per ogni i da 0 a $N-1$, c'è uno scambio ed $N-1-i$ confronti, così, il totale sono N scambi e $(N-1)+(N-2)+\dots+2+1+0 = \underline{N(N-1)/2} \sim N^2/2$ confronti.

Esempio pg 248

Il tempo di esecuzione non dipende dall'input.
Il processo di ricerca dell'elemento più piccolo
in un paraggio, non ci dà molte informazioni
su dove deve essere l'elemento più piccolo
del prossimo paraggio.
Questo proprietà può essere sventaggiosa in
alcune situazioni.

Data Movement is minimal: Il Sel Sort usa
N scambi, quindi il numero di scambi
all'array è lineare in funzione alla grandezza
dell'array.

Nessuno degli altri algoritmi che consideriamo
hanno queste proprietà.

```
public class Selection {  
    public static void sort (Comparable[] a) {  
        int N = a.length;  
        for (int i = 0; i < N; i++) {  
            int min = i;  
            for (int j = i+1; j < N; j++) {  
                if (less(a[j], a[min])) min = j;  
            }  
            exch(a, i, min);  
        }  
    }  
}
```

Insertion Sort

E' l'algoritmo che spesso le persone utilizzano per ordinare le carte che hanno in mano durante una partita a carte.

In una implementazione informatica, dobbiamo fare spazio per inserire l'elemento corrente, muovendo gli elementi di piu' grande dimensione alle destra, prima di inserire l'elemento corrente nella posizione vuota.

Come succede per il sel-sort, gli elementi a sinistra dell'indice corrente SONO ORDINATI, ma non sono NELLA LORO POSIZIONE FINALE.

L'array e' in uno stato ordinato quando l'indice raggiunge l'estremita' 'destra'.

A differenza del Sel-Sort, il tempo di esecuzione dell'ins-sort dipende dal tipo di input; cio' vuol dire, in questo caso, che se l'array e' di grandi dimensioni e i suoi elementi sono QUASI ordinati, questo algoritmo e' molto piu' veloce rispetto a quando gli elementi sono in posizioni random oppure ordinati al contrario.

Proposizione B: L'insertion Sort utilizza $\sim N^2/4$ confronti e $N^2/4$ scambi per ordinare un array con elementi random di grandezza N con elementi distinti, in media.

Il caso peggiore è $\sim N^2/2$ confronti e $\sim N^3/2$ scambi.

Il caso migliore è $\sim N-1$ confronti e 0 scambi.

Dimostrazione

Nuovamente visualizziamo la dimostrazione con il diagramma $N \times N$.

Contiamo gli elementi al di sotto della diagonale principale: tutti per il caso peggiore e nessuno per il caso migliore.

Per array Random, in media ci aspettiamo che ogni elemento si sposti circa per metà delle posizioni indietro, con contorno $\frac{1}{2}$ degli elementi sotto la diagonale.

Il numero di confronti è il numero di scambi più un termine addizionale uguale ad $N -$ il numero di volte che l'elemento inserito è il più piccolo fino a quel punto.

Nel caso peggiore il termine è trascurabile in relazione al totale. Nel caso migliore è pari $N-1$.

L'insertion Sort, come detto, funziona molto bene su collezioni già parzialmente ordinate; un esempio di collezione p. ordinata può essere:

- Un array dove ogni elemento non è lontano dalla sua posizione finale.
- Un piccolo array 'aggiunto' ad un grande array ordinato.
- Un array con solo pochi elementi che non sono in posizione.

ShellSort

L'insertion sort è lento per grandi array NON ordinati, perché gli unici scambi che comprende sono tra elementi adiacenti, così gli elementi possono muoversi attraverso l'array solo un posto alla volta. La shell sort è una semplice estensione dell'ins. sort, che guadagna velocità permettendo scambi tra elementi che sono lontani tra loro, in modo da produrre array PARZIALMENTE ordinati che possono essere ordinati in modo efficiente, magari con l'insertion sort.

L'idea è quella di riordinare l'array in modo da dargli la proprietà che, prendendo un qualsiasi h -esimo elemento, esso faccia riferimento ad una sotto-sequenza ordinata.

Ordinando in modo ' h -sorting' per grandi valori di h , siamo in grado di muovere gli elementi nell'array per lunghe distanze, ed in questo modo è più semplice ' h -sort' per piccoli valori di h .

Parlandoci chiaro...

L'algoritmo essenzialmente funziona in questo modo: abbiamo un array di m elementi (12). Calcoliamo $h = m/2$ (6); questo indice h ci serve per poter confrontare due elementi che sono h posizioni di distanza:

1 2 3 4 5 6 7 8 9 10 11 12
└─────────┘

$h \quad ? \neq < 1$

↳ NO

Continuiamo in questo modo per tutti gli elementi FINO ALL'ULTIMO elemento dell'indice superiore:

1 2 3 4 5 6 7 8 9 10 11 12



Per il nostro esempio, arriveremo a controllare fino agli elementi 6 - 12.

Durante il controllo, se il secondo elemento è minore del primo si effettua uno scambio e si continua.

Finita la prima iterazione, h diventa $h = h/2 = 3$ e procediamo allo stesso modo, ma ~~non~~ controlliamo anche le posizioni precedenti: (solo se il secondo è minore del primo):

24 27 2 61 109 122 111 119 149 125 34 145



$34 < 119 ?$ SI

↳ Scambio

24 27 2 61 109 122 111 34 149 125 119 145



$34 < 109 ?$ SI

↳ Scambio

24 27 2 61 34 122 111 109 149 125 119 145



$34 < 27 ?$ NO

↳ Proseguo

Continuo in questo modo ~~non~~ aggiornando $h = h/2$ ad ogni iterazione finché $K=1$, farò l'ultima iterazione ed avrò finito.

Shuffling

Shuffling vuol dire mescolare in modo uniforme un array di elementi, ovvero permettendo che TUTTE le permutazioni abbiano pari probabilità. Per avere questo risultato, bisogna consentire a tutte le permutazioni di essere equiprobabili.

Un metodo molto semplice per ottenere questo risultato, è quello di generare un numero reale random per ogni elemento dell'array e poi ordinare l'array in base ai numeri reali.

Il problema che incontriamo in questo caso è che fare il sorting ha una complessità, nel caso peggiore, lineare.

Noi, invece, vogliamo fare lo shuffling in un tempo LINEARE.

Come facciamo fare lo Shuffle in Tempo Lineare?

Poniamo usare una 'variante' degli algoritmi visti in precedenza:

Ad ogni passo I , andiamo a calcolare un numero intero random tra 0 ed I e facciamo lo scambio tra le posizioni I ed r