

## Esercitazione 07

21 novembre 2023

Lo scopo di questa esercitazione è la scrittura di un programma che calcola la “scorciatoia” in grafo pesato. Si supponga di avere una matrice  $d$  di dimensioni  $n \times n$  di valori reali positivi (nel codice saranno rappresentati con il tipo “float”) con la diagonale di zero. Questa può essere interpretata come la matrice dei di un grafo di  $n$  nodi, i.e., la posizione  $d_{i,j}$  indica il peso dell’arco tra il nodo  $i$  e il nodo  $j$  e  $d_{i,i} = 0$ .

Si vuole definire una matrice  $m$  sempre di dimensioni  $n \times n$  nel seguente modo: per ogni  $1 \leq i, j \leq n$  si ha che

$$m_{i,j} = \min_{1 \leq k \leq n} d_{i,k} + d_{k,j}$$

In pratica stiamo chiedendo di trovare il percorso minimo tra i nodi  $i$  e  $j$  passando per esattamente un altro nodo  $k$  (non necessariamente distinto da  $i$  e  $j$ ). Si vuole scrivere una funzione che, data una matrice  $d$ , riempia i valori della matrice  $m$  come da definizione. Per fare questo sarà necessario, per ogni posizione  $m_{i,j}$  di  $m$  confrontare  $n$  diverse somme di valori della matrice  $d$  per trovarne il minimo:  $d_{i,1} + d_{1,j}$ ,  $d_{i,2} + d_{2,j}$ ,  $\dots$ ,  $d_{i,n} + d_{n,j}$

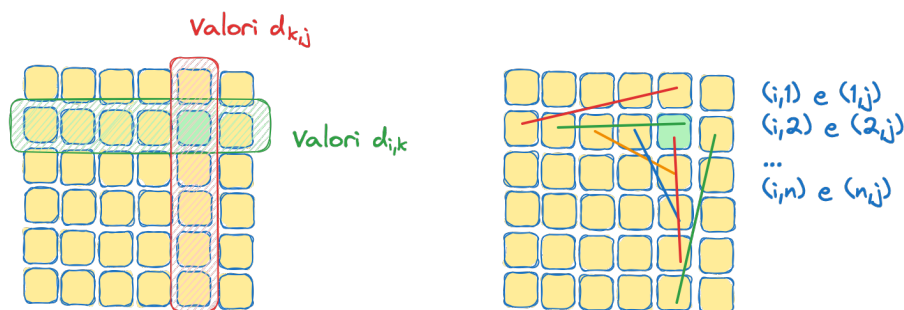


Figure 1: Un esempio dei valori della matrice  $d$  a cui accediamo per la posizione  $d_{i,j}$ . A destra gli elementi a cui accediamo al variare di  $k$

### Funzioni da definire

Nel file `graph.c` sono da definire le seguenti funzioni:

```
void shortcut_std(float * m, float * d, int n);
```

Riempie i valori della matrice  $m$  a partire da quelli contenuti nella matrice  $d$  (le due matrici hanno dimensione  $n * n$ ). In questa implementazione è necessario fare una trasposizione diretta della definizione (i.e., iterare su tutte le posizioni di  $m$  e, per ognuna, iterare sugli  $n$  valori possibili di  $k$  per trovare la somma minima dei valori in  $d$ ).

```
void shortcut_trs(float * m, float * d, int n);
```

Possiamo notare che la matrice `d` viene acceduta sia per righe (efficiente) che per colonne (meno efficiente). Potrebbe essere utile creare una copia trasposta `t` di `d` e computare i valori di `m` accedendo per righe sia in `d` che in `t`. Ricordatevi di deallocare `t` alla fine della chiamata a funzione.

```
void shortcut_omp(float * m, float * d, int n);
```

In questo caso dovete prendere il codice di `shortcut_std` e renderlo parallelo con OpenMP

```
void shortcut(float * m, float * d, int n);
```

Funzione dove ogni ottimizzazione è consentita (ma misurate dopo aver implementato). Potreste iniziare parallelizzando il codice di `shortcut_trs`.

### Funzioni già definite

In `matrix.h`:

```
float * random_matrix(int n);
```

Ritorna una matrice quadrata di lato  $n$  di valori casuali positivi in cui la diagonale principale è composta da zero. La matrice è rappresentata in modo contiguo in memoria.

```
float * zero_matrix(int n);
```

Ritorna una matrice quadrata di lato  $n$  in cui tutti i valori sono zero. La matrice è rappresentata in modo contiguo in memoria.

In `main.c`:

```
double test_shortcut(void (*f) (float *, float *, int), float * m, float * d, float n)
```

Esegue per un certo numero di ripetizioni (di default 5) la funzione per il calcolo degli shortcut

### Note

- La funzione “base” può essere fatta semplicemente innestando tre cicli for (iterando su  $i$ ,  $j$  e  $k$ ). Notate che dovremo stabilire un valore iniziale per  $m_{i,j}$ , alcune scelte possibili sono  $d_{i,j}$  (dato che  $d_{i,j} = d_{i,j} + d_{j,j}$ , quindi è un valido candidato per essere il minimo) oppure  $+\infty$ , che in C è ottenibile importando `math.h` e usando la costante `INFINITY`.
- Nella definizione delle matrici  $d$  e  $m$  sono stati usati gli indici  $1, \dots, n$ . Nel codice sarà più utile usare gli indici  $0, \dots, n - 1$ .
- Stiamo usando la funzione `omp_get_wtime()` che ritorna il tempo passato da un momento iniziale in secondi (rappresentato come `double`, quindi anche le frazioni di secondo sono presenti). Questo è il “wall clock time”,

ovvero il tempo effettivamente passato “guardando un orologio”. Potenzialmente possiamo contare il totale del tempo di ogni thread. Questo risulterà maggiore nel caso di programmi con più thread dato che, per esempio, il wall clock time di 10 thread che, in parallelo, eseguono per un secondo è di 1 secondo, mentre il totale del tempo speso dai thread è 10secondi (i.e., l'equivalente tempo come se eseguiamo i thread uno alla volta). Nel nostro caso ci interessa il tempo effettivamente passato.