
Politecnico di Torino

03AAXOA

# Commento all'appello del 09/02/2022

# Indice

# 1 Traccia da 12 punti

## 1.1 2pt - Generazione vincolata di matrice

### 1.1.1 Sintesi della richiesta

L'esercizio richiede di scrivere una funzione `f` che ricevuta in input una matrice di interi $M$, di dimensioni note $r \times c$, generi una seconda matrice $M'$, di dimensioni da definire $r' \times c'$, derivata da $M$ trascurando tutte le righe e colonne di $M$ a indice dispari.

### 1.1.2 Strategia risolutiva

Occorre identificare le dimensioni della matrice $M'$, per poter allocare opportunamente la nuova matrice, e poi procedere al suo riempimento. In questo caso è possibile calcolare direttamente le dimensioni della nuova matrice, valutando il risultato della divisione intera per 2 delle dimensioni originali e tenendo conto dell'eventuale riga/colonna extra necessaria laddove le dimensioni iniziali fossero dispari.

```
new_r = r / R + r % R;
new_c = c / C + c % C;
```

Identificate le righe/colonne di interesse è possibile procedere con la fase di copiatura, incrementando opportunamente gli indici della matrice di destinazione. $M$ e $M'$ richiedono un set di indici separati per l'accesso alle rispettive celle. Nello snippet di codice riportato $M'$ è chiamata semplicemente `m` per distinguerla dalla variabile `M` della matrice originale.

```
for(i=0, k=0;i<R;i+=2) {
  for(j=0, l=0;j<C;j+=2) {
    (*m)[k][l++] = M[i][j];
  }
  k++;
}
```

Il prototipo della funzione, da completare, rende necessario che gli argomenti $M'$, $r'$ e $c'$ siano passati per riferimento alla funzione stessa, per far sì che i loro dati modificati siano visibili a chiunque invochi la funzione. Notare nello snippet precedente l'accesso con deferenziazione a `m` dove si da esplicitamente precedenza all'operatore `*` con l'uso delle parentesi tonde.

### 1.1.3 Errori comuni

- Mancato passaggio per riferimento, o uso scorretto del passaggio per riferimento, delle variabili usate per $M'$, $r'$ e $c'$;
- Calcolo errato delle dimensioni della nuova matrice $M'$, non tenendo conto dell'eventuale resto della divisione intera;
- Aggiornamento errato dell'indice di riga per la scrittura in $M'$. Sia l'indice di riga sia l'indice di colonna vengono incrementati a ogni scrittura, quando in realtà solo la colonna dovrebbe essere modificata;
- Mancato reset dell'indice di colonna per $M'$ al passaggio alla riga successiva.

## 1.2 4pt - Decodifica con Huffman

### 1.2.1 Sintesi della richiesta

L'esercizio richiede di scrivere una funzione che ricevuto un albero binario rappresentante una possibile codifica di Huffman, e una stringa interpretabile come un codice binario, determini il valore decodificato per la stringa proposta in input.

### 1.2.2 Strategia risolutiva

La specifica prevede di definire il tipo `H` come ADT e il tipo `nodo` come quasi ADT. Si rimanda direttamente alle slide del corso per quanto riguarda la corretta definizione delle strutture dati e la descrizione teorica della codifica.

Si ricorda che ogni nodo dell'albero presenta un valore intero (la frequenza di un singolo carattere, per le foglie, o la somma delle frequenze dei figli per i nodi intermedi). Si ricorda inoltre che solo le foglie riportano uno dei caratteri codificati. I nodi intermedi non rappresentano un carattere nella codifica. Un'eventuale visita che si fermi in un nodo intermedio, è sintomo di una condizione di errore.

Una soluzione ragionevole, in questo caso, prevede di effettuare multiple visite dell'albero, tentando di risolvere un carattere alla volta. La sequenza di visite termina al termine della stringa di input o in caso di errore. Si può adottare un approccio completamente iterativo per decodificare la stringa, operando in un modo simile a quanto riportato a seguire. La strategia proposta assume che l'albero non faccia uso di sentinelle.

```
link iter = h->root;
for(i=0;i<len && iter;i++) {
  if(str[i] == '1')
    iter = iter->r;
```

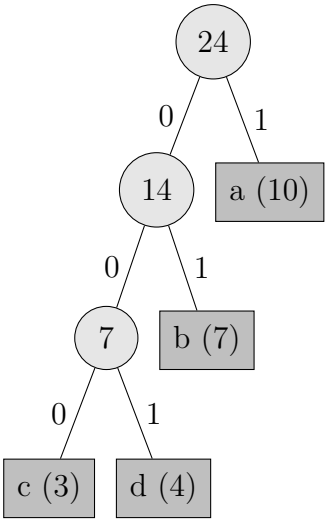


Figura 1: Un albero rappresentante una codifica di Huffman per 4 caratteri $a$ (10), $b$ (7), $c$ (3), $d$ (4), con le relative frequenze tra parentesi.

```
  else if(str[i] == '0') {
    iter = iter->l;
  } else {
    ... // Eventuale gestione di input sbagliato
  }
  if (iter && !iter->l && !iter->r) {
    dec[dec_len++] = iter->val;
    iter = h->root;
  }
}

if (iter != h->root) {
  // Se terminiamo la decodifica senza essere tornati alla radice, errore
}
```

### 1.2.3 Errori comuni

- Mancato allineamento con la specifica per quanto riguarda la definizione dei tipi;
- Limitata conoscenza della teoria su cui si fonda l'esercizio;
- Errori nei criteri di stop della visita, nella gestione dei casi limite e di errore.

## 1.3 6pt - Sottoinsieme vincolato a cardinalità massima

### 1.3.1 Sintesi della richiesta

L'esercizio richiede di scrivere una funzione in grado di individuare un singolo sottoinsieme vincolato, a cardinalità massima, composto da persone tali per cui ognuno sia amico di *almeno* $k$ altri membri del gruppo.

### 1.3.2 Strategia risolutiva

Si tratta di un problema risolvibile sfruttando il modello delle combinazioni semplici, con lo scopo di andare a generare sotto-insiemi delle persone coinvolte. L'ordine con cui le persone sono aggiunte le gruppo è irrilevante. Per quanto riguarda l'approccio alla ricerca della soluzione, conviene seguire un approccio fondato su cardinalità target decrescenti poiché stiamo cercando un massimo: dal caso migliore in cui si possono includere tutte le persone nel gruppo fino al caso peggiore in cui la soluzione non esiste.

Poiché il vincolo di accettabilità è nella forma *almeno* $k$, ma non è richiesta amicizia totale, siamo disposti ad includere nel gruppo in costruzione anche persone non direttamente amiche, se questo permette comunque di rispettare il vincolo e aumentare la cardinalità della soluzione. Evitare di includere coppie di non-amici a priori, nel caso generale sarebbe un errore. È comunque possibile applicare del pruning, sebbene non sia particolarmente agevole da implementare, valutando la capacità residua nel gruppo in costruzione. Possiamo decidere di non includere persone nella condizione di non-amicizia se non rimanessero abbastanza slot liberi per introdurre ulteriori persone in modo tale che il vincolo sia rispettato in terminazione.

Sebbene non sia errato, cercare la soluzione utilizzando il powerset è potenzialmente meno efficiente. Non potendo controllare, di base, il numero di 1 nella soluzione, non è possibile guidare la ricerca per cardinalità target.

### 1.3.3 Errori comuni

- Utilizzare un modello che implichi un criterio di ordine per gli elementi della soluzione;
- Fraintendimento della richiesta del problema o della condizione di accettazione (es: imporre amicizia totale tra i membri del gruppo);
- In generale, mancanza di familiarità con i vari modelli finendo quindi a mischiare approcci diversi con risultati più o meno discutibili.

# 2 Traccia da 18 punti

## 2.1 Inquadramento del problema

Il problema proposto è riconducibile a un problema di *pathfinding* vincolato. Nello specifico si chiede di individuare un cammino Hamiltoniano semplice, poiché si vuole che siano visitate tutte le celle bianche una e una sola volta. In aggiunta alle condizioni di generazione del cammino, si introduce un ulteriore criterio di ottimo tale per cui la soluzione deve effettuare il minor numero possibile di cambi di direzione nell'attraversare la mappa.

Per quanto riguarda la struttura del problema in esame, la griglia di gioco ammette una interpretazione a grafo. In letteratura questa famiglia di grafi può assumere vari nomi, come *lattice graphs*, *mesh graphs* o *grid graphs*. Ogni cella della griglia può essere mappata su un vertice del grafo e la proprietà di adiacenza tra due celle può essere mappata su un arco del grafo, incidente su due nodi-cella vicini. Nel caso in questione abbiamo un *grid graph* bidimensionale $L(R, C)$, in cui ogni vertice ha grado massimo pari a 4, poiché il movimento/condizione di vicinanza è ammesso solo lungo le quattro direzioni principali.

Sebbene abbiano una struttura molto vincolata e particolare, in quanto un generico *grid graph* bidimensionale è planare e bipartito (basta pensarlo colorato come una scacchiera per rendere evidente la possibile bipartizione dei vertici del grafo), la ricerca di un generico cammino di Hamilton tra due vertici rimane comunque un problema NP-completo.

Nel caso in esame, la presenza di celle nere complica ulteriormente la natura del grafo indotto, che potrebbe essere addirittura non connesso, per cui non si possono nemmeno fare considerazioni sull'esistenza del cammino cercato.

## 2.2 Verifica di una soluzione proposta

Per verificare l'accettabilità di una soluzione si deve tenere conto dei seguenti aspetti:

- Tutte le mosse proposte devono rispettare le dimensioni della griglia di partenza. Non sono ammesse mosse che escano al di fuori dei bordi della griglia stessa;
- Tutte le mosse devono interessare solamente porzioni bianche della griglia. Non sono ammesse mosse che attraversino celle nere;
- Tutte le celle bianche della griglia devono essere visitate esattamente una volta.

A seconda del formato scelto, la verifica di validità di una soluzione proposta può risultare più o meno agevole.

È possibile scegliere una rappresentazione esplicita della visita, andando a leggere una matrice coerente con la griglia di partenza dove ogni cella bianca riporta la direzione dei movimento (tra i quattro ammessi) con cui si esce dalla cella stessa. In questo caso è necessario verificare solo la fattibilità della mossa (dentro ai bordi, verso cella bianca). Poiché il cammino cresce di un passo alla volta, la verifica di accettabilità è relativamente agevole da implementare. In questo formato sono esclusi automaticamente movimenti in diagonale.

Una seconda possibile rappresentazione per la visita sarebbe come sequenza di coppie `<direzione> <numero_di_passi>`, che sintetizza la proposta precedente. In questo caso è necessario preoccuparsi, oltre che del rispetto della geometria della mappa, di gestire correttamente l'attraversamento di celle multiple per ogni mossa letta. Anche in questo caso i movimenti in diagonale sono implicitamente esclusi (se `<direzione>` è una di quelle ammesse).

Una terza possibile rappresentazione della visita sarebbe come sequenza di celle visitate, lette come coppia `<riga> <colonna>`. In questo caso si aggiunge la necessità di verificare esplicitamente l'eventuale presenza di movimenti in diagonale non ammessi.

In tutti i casi, è opportuno prevedere un modo di marcare le celle già viste in modo da garantire il rispetto del vincolo di semplicità del cammino.

Per quanto riguarda il conteggio del numero di cambi, le varie rappresentazioni permettono di effettuarlo in maniera più o meno agevole, a seconda del formato scelto. Nella rappresentazione esplicita del cammino passo-passo, il conteggio del numero di cambi è automatico. Nella rappresentazione compatta degli spostamenti occorre tenere conto del fatto che il numero di mosse lette non necessariamente coincide con il numero di cambi: è possibile leggere `SUD 3 SUD 2` anziché `SUD 5` e le due varianti rappresentano il medesimo movimento, potenzialmente valido, senza cambio interno di direzione. Nella terza rappresentazione, data per coppie di coordinate, è necessario valutare esplicitamente i cambi di direzione tenendo conto della differenza tra ascisse o ordinate di movimenti consecutivi. Complessivamente, il terzo formato è quello formalmente più scomodo da usare per rappresentare la visita.

## 2.3 Generazione di una soluzione ottima

La generazione di una soluzione ottima richiede di adattare una classica visita in profondità, introducendo backtrack per ammettere l'esplorazione di tutti i cammini. Non è necessario rappresentare esplicitamente il grafo indotto: è sufficiente trattare

la griglia come se fosse un grafo e applicare l'algoritmo di visita direttamente sulla griglia stessa.

Partendo dalla cella di partenza, si ammette movimento verso tutte le celle adiacenti non visitate e attraversabili, ricorrendo sulla nuova destinazione. Si procede in questo modo finché sono disponibili movimenti validi.

Poiché è noto il numero di celle bianche, è possibile usare tale informazione come condizione esplicita di terminazione. Solo nel caso in cui si siano effettivamente esaurite le celle bianche, siamo in una condizione di successo. Oltre alla condizione di terminazione, è possibile prevedere un criterio generico di pruning che interrompa qualsiasi visita che faccia attualmente uso di un numero di cambi di direzione superiori all'eventuale ottimo già noto.

Una versione sintetica della strategia proposta è assimilabile al listato presentato a seguire.

Il punto di invocazione della ricerca è unico, in quanto si ammette di partire solo dalla cella in alto a sinistra. Dichiariamo di partire da quella posizione marcandola con un carattere particolare `X` per evitare il riconoscimento di un primo cambio di direzione non voluto.

```
int steps = contaBianche(m, R, C);
m[0][0] = directions[MOVES];
solve_r(R, C, m, best, 0, &best_val, 0, steps-1, directions[MOVES]);
```

La funzione ricorsiva, come già detto, è una rivisitazione di un classico algoritmo di visita in profondità con backtrack. La posizione corrente sulla griglia è linearizzata in una singola variabile `pos`. Gli indici di riga e colonna sono calcolati all'occorrenza.

```
void solve_r(int R, int C, int **sol, int **best, int curr_val, int
    *best_val, int pos, int steps, char direction) {
  if (steps == 0) { // Visita di tutto lo spazio bianco riuscita
    if (*best_val > curr_val) {
      *best_val = curr_val;
      copia2d(R,C,sol,best);
    }
    return;
  }
  if (curr_val > *best_val) return; // Soluzione parziale peggiore
     dell'ottimo
  int i, step, r, c;
  for(i=0;i<MOVES;i++) {
    r = pos / C + rShift[i];
```

```
    c = pos % C + cShift[i];
    step = mossaValida(r, c, R, C, sol);
    if (step == WHITE) {
      sol[r][c] = directions[i];
      solve_r(R, C, sol, best, curr_val + changeDir(direction,
          directions[i]), best_val, r*C+c, steps-1, directions[i]);
      sol[r][c] = WHITE;
    }
  }
}
```

Per comodità si scorpora il calcolo dello spostamento ammesso, della fattibilità della mossa e del cambio di direzione.

```
#define MOVES 4
// NORD/SU, OVEST/SX, EST/DX, SUD/GIU'
int rShift[MOVES] = {-1, 0, 0, 1};
int cShift[MOVES] = {0, -1, 1, 0};
char directions[MOVES+1] = "NWESX";
```

```
int mossaValida(int r, int c, int R, int C, int **m) {
  // Se dentro i bordi, ritorna il valore della cella destinazione
  if ((r >= 0) && (r < R) && (c >= 0) && (c < C)) {
    return m[r][c];
  }
  return BLACK; // Tratta out-of-bound come un ostacolo
}
```

```
int changeDir(char prev, char curr) {
  if(prev != directions[MOVES] && prev != curr) return 1;
  return 0;
}
```

## 2.4 Errori comuni

Nel caso dell'esercizio in questione è difficile generalizzare sulle tipologie di errore incontrante.

Per quanto riguarda la verifica, i problemi principali si sono manifestati per una non corretta interpretazione della richiesta, oppure per un approccio alla verifica che non tenesse conto delle varie casistiche da considerare sulla base della rappresentazione

adottata. Sovente si sono incontrate implementazioni in cui non si tenesse nemmeno in conto di dover evitare di uscire dai bordi, sforando inopportunamente gli estremi della matrice/griglia.

Per quanto riguarda la ricerca di una soluzione ottima, l'errore principale riscontrato è stato il non tenere conto della geometria della griglia, ammettendo un'esplorazione che ignorasse i vincoli di vicinanza/raggiungibilità tra celle. Così facendo lo spazio delle soluzioni, già potenzialmente grande di per sé, viene esteso a includere un numero esponenziale di soluzioni completamente inammissibili poiché incompatibili con la topologia del grafo.