Politecnico di Torino

03AAXOA

Commento all'appello del 27/01/2022

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 in cui appaia almeno un elemento nullo.

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. Conviene appoggiarsi a due vettori ausiliari per marcare le righe/colonne da mantenere nella seconda fase. Rispetto a contare solamente il numero di righe/colonne da utilizzare, tracciare effettivamente le righe/colonne di interesse rende più agevole la fase di copiatura dei valori. La marcatura è ottenibile con un doppio-for annidato (per ogni riga/per ogni colonna) iterando sulla matrice M alla ricerca dei valori nulli.

```
for(i=0;i<R;i++) {
  for(j=0;j<C;j++) {
    // riga/colonna da saltare
    if(!M[i][j]) markr[i] = markc[j] = 1;
  }
}
```

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++) {
  // Salta intera riga (non incrementa k)
  if(markr[i]) continue;
  for(j=0, l=0;j<C;j++) {
    // Se la colonna non va saltata, copia e incrementa l
    if(!markc[j]) (*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' poiché sia il numero di righe sia di colonne viene decrementato a ogni valore nullo individuato. Molteplici zeri su una stessa riga causano un decremento del numero di colonne, ma le righe vanno decrementate una sola volta finché non c'è un cambio di riga su M;
- 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 - Verifica proprietà BST in albero binario generico

1.2.1 Sintesi della richiesta

L'esercizio richiede di scrivere una funzione che ricevuto un generico albero binario determini se questo possa essere interpretato come *binary search tree* ossia se questo rispetti la proprietà funzionale degli alberi binari di ricerca.

1.2.2 Strategia risolutiva

La specifica prevede di definire il tipo `BT` 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.

Perché l'albero binario sia interpretabile come albero binario di ricerca deve valere quanto segue:

- Il sottoalbero sinistro di un nodo x contiene soltanto nodi con chiavi minori della chiave del nodo stesso;
- Il sottoalbero destro di un nodo x contiene soltanto nodi con chiavi maggiori della chiave del nodo stesso;

- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca;
- Un albero vuoto o una foglia, soddisfano automaticamente la proprietà.

Nota bene, la relazione minore/maggiore rispetto a una certo nodo radice x deve valere *per tutti i nodi del rispettivo sottoalbero* sinistro/destro, e non solo per il figlio sinistro/destro.

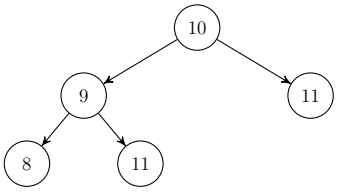

Figura 1: Un albero binario che NON rappresenta un BST.

Si possono seguire strade diverse per valutare l'ammissibilità della rappresentazione, con costi computazionali diversi. Per semplicità, l'albero considerato non fa uso di sentinelle, ma usa direttamente il valore `NULL`.

Una prima variante, con costo lineare nel numero dei nodi, si fonda sull'idea di propagare il range di valori ammissibili per il sottoalbero attualmente in esame. Avviamo l'esplorazione dalla radice dell'albero ammettendo l'intero range dei numeri interi come estremi dell'intervallo.

```
if(!bt || !bt->root) return 1;
return fR(bt->root, INT_MIN, INT_MAX);
```

Valutiamo ricorsivamente la proprietà per tutti i nodi dell'albero.

```
if(!root) return 1;
if(root->val < min || root->val > max) return 0;
return fR(root->l, min, root->val) && fR(root->r, root->val, max);
```

Una seconda variante, con costo al peggio quadratico nel numero dei nodi, si fonda sull'idea di calcolare il valore minimo/massimo in un certo sottoalbero radicato, per ogni nodo radice che andiamo a visitare. La ricerca non può fare assunzione sulla struttura del sottoalbero poiché non si ha alcuna garanzia sull'ordinamento delle chiavi, per cui la visita deve essere esaustiva.

```
if (root == NULL)
  return 1;
```

```
int max_left = max(root->l);
int min_right = min(root->r);
if (max_left > root->val || min_right < root->val)
  return 0;
if (fR(root->l) && fR(root->r))
  return 1;
return 0;
```

Una terza variante applicabile, sebbene non particolarmente conveniente da implementare poiché la dimensione dell'albero è ignota, prevede di materializzare la visita in-order dell'albero binario stesso al fine di valutare l'ordinamento delle chiavi così individuate.

1.2.3 Errori comuni

- Mancato allineamento con la specifica per quanto riguarda la definizione dei tipi;
- Proprietà del BST verificata solo rispetto alla terna *radice*, *figlio sinistro*, *figlio destro*, anziché rispetto all'intero sottoalbero radicato a sinistra/destra. Questo errore è stato riscontrato nella quasi totalità degli elaborati, ed è rimasto tale anche nella maggior parte dei codici "corretti" consegnati con le relazioni.

1.3 6pt - Partizionamento vincolato

1.3.1 Sintesi della richiesta

L'esercizio richiede di scrivere una funzione in grado di suddividere delle persone, tra cui può o meno esistere una relazione simmetrica di amicizia, nel minor numero di gruppi possibile tale per cui tutti i membri di un gruppo siano mutualmente amici.

1.3.2 Strategia risolutiva

Si tratta di un problema di partizionamento, e può essere correttamente affrontato sfruttando l'algoritmo di Er oppure, in maniera meno efficiente, applicando il modello delle disposizioni con ripetizione, dove i valori "ripetuti" sono gli indici dei possibili gruppi. Le disposizioni sono meno efficienti poiché possono generare partizionamenti simmetrici a soluzioni già visitate in precedenza. L'interpretazione più conveniente di una soluzione per il problema in esame è rappresentata da un vettore di interi lungo quanto il numero di persone, in cui in ogni $i-$esima posizione sia memorizzato l'indice del gruppo assegnato alla persona di indice i. Poiché non si chiede di individuare insiemi *massimali* di amici, possiamo trascurare il caso in cui una certa persona possa

essere posizionata in più di un insieme, poiché ciò non va a influire sulla cardinalità della soluzione cercata.

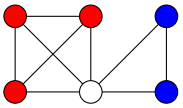

Figura 2: Un partizionamento in cui una persona (nodo bianco) potrebbe indifferentemente fare parte di due gruppi (rosso o blu), senza che la soluzione al problema cambi.

Per quanto riguarda l'approccio alla ricerca della soluzione, se si decide di guidare la cardinalità della soluzione, conviene seguire un approccio fondato su cardinalità target crescenti poiché stiamo cercando un minimo: dal caso migliore in cui si può usare un singolo gruppo (tutti amici con tutti) al caso peggiore in cui è necessario avere tanti gruppi quante sono le persone (ogni persona è amica solo con se stessa).

È possibile tagliare rami di ricorsione evitando di includere una persona in un gruppo già parzialmente formato se questa aggiunta facesse fallire la condizione di "mutua amicizia" per almeno una coppia di amici nel gruppo. Allo stesso modo, è possibile tagliare rami di ricorsione tali per cui la cardinalità della soluzione corrente sia peggiore rispetto al valore migliore noto in un dato momento.

1.3.3 Errori comuni

- Tentare di costruire una soluzione per il problema utilizzando come modello combinatorio l'insieme delle parti (powerset). Tale modello permette di generare, uno alla volta, i possibili sottoinsiemi dato un insieme di partenza, decidendo quali elementi includere/escludere (1/0) nella soluzione. In terminazione si ha un singolo gruppo di persone selezionate dall'insieme di partenza (e implicitamente un singolo insieme di esclusi). Ciò non è sufficiente a rispondere alla domanda, dove il partizionamento desiderato è un insieme di sottoinsiemi (disgiunti) la cui unione copra l'intero insieme di partenza.
- Tentare di costruire una soluzione per il problema utilizzando come modello combinatorio le combinazioni con ripetizione. Utilizzando le combinazioni ripetute, anziché le disposizioni ripetute, è impossibile generare alcune soluzioni potenzialmente valide (e distinte da quanto generato in precedenza) per il problema. Ad esempio, tramite combinazioni non sarebbe possibile generare una soluzione nella forma $(0, 1, 0)$ poiché considerata equivalente a $(0, 0, 1)$ generata in precedenza. Le due però rappresentano soluzioni diverse per il problema in esame: nella prima abbiamo p_0 e p_2 nel gruppo di indice 0, nella seconda i membri del gruppo 0 sono invece p_0 e p_1.

- 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 si posiziona nel contesto della geometria computazionale bidimensionale, e può essere fatto ricadere nella categoria generica dei problemi di "polygon decomposition". In questa categoria troviamo tutti quei problemi che abbiano come obiettivo la decomposizione di un poligono in *unità* più semplici, come triangoli, quadrati, rettangoli o altri gadget considerati elementari.

Le tipologie classiche di decomposizione ricadono genericamente in una delle seguenti casistiche:

Covering la decomposizione *copre* l'intero poligono, ammettendo sovrapposizioni tra i gadget usati

Packing la decomposizione *impacchetta* i gadget nell'area del poligono, senza sovrapposizioni e senza pretesa di produrre una copertura completa

Partitioning le decomposizione *partiziona* il poligono senza ammettere sovrapposizione tra i gadget e garantendo copertura completa

Nel caso del problema in esame il poligono da coprire, rappresentato dall'area bianca della griglia, è un poligono rettilineo ortogonale con buchi, per via della possibile presenza di zone nere al suo interno. A seconda della tipologia di decomposizione desiderata, dei gadget ammessi e della geometria del poligono, il problema risultante può ammettere soluzioni nel polinomiale o ricadere direttamente in classi di complessità superiore.

Nello scenario proposto, il problema è un partizionamento con gadget quadrati, a lato variabile. In questa istanza il problema ricade nella classe NP-hard (ammette una riduzione a 3-SAT planare).

2.2 Verifica di una soluzione proposta

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

- Tutte le regioni proposte devono rispettare le dimensioni della griglia di partenza. Non sono ammesse regioni che escano al di fuori dei bordi della griglia stessa;
- Tutte le regioni devono occupare solamente porzioni bianche della griglia. Non sono ammesse regioni che si sovrappongono, anche solo in parte, con celle nere;

- Tutte le celle bianche della griglia devono essere coperte da esattamente una regione. Non sono ammesse sovrapposizioni tra regioni. Non sono ammesse soluzioni che lascino scoperte celle bianche;
- Tutte le regioni sono quadrate.

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

La rappresentazione più conveniente sarebbe l'acquisizione di regioni rappresentate come coordinata dell'angolo in alto a sinistra e lato del quadrato, che si riduce a leggere terne del tipo `<indice_riga_top> <indice_colonna_left> <lato>`. In questo formato è possibile dare per scontato che le regioni siano quadrate (lo sono per definizione). Rimangono da verificare le restanti condizioni.

Una rappresentazione quasi equivalente alla precedente è data dall'acquisire regioni espresse come coppia di coordinate `<top_left>` e `<bottom_right>`, utilizzando 4 indici, due per le righe e due per le colonne. In questo caso è necessario verificare che la base e l'altezza siano equivalenti, mediante sottrazione, ed eventualmente assicurarsi che la coppia di coordinate non sia proposta al contrario rispetto alle aspettative.

In entrambi questi due casi, la verifica di copertura della nuova regione letta può essere fatta andando a colorare una matrice ausiliaria, inizialmente identica alla griglia letta in input, e fermarsi non appena si vada a tentare di scrivere in una cella nera o già colorata.

Nel caso in cui si opti per andare a leggere una soluzione rappresentata in maniera esplicita come matrice è necessario, oltre ad assicurarsi che la copertura sia compatibile con la griglia originale, farsi carico di verificare che la geometria delle regioni sia effettivamente corretta. La verifica deve essere in grado di riconoscere situazioni come quella presentata in Fig. 3.

Un'altra potenziale situazione di conflitto sarebbe il dover verificare regioni (valide o meno) non contigue caratterizzare dallo stesso identificatore, come rappresentato in Fig. 4. Questa situazione è più gestibile poiché basterebbe accettare di ri-etichettare la regione in conflitto, ma obbliga comunque a tener conto anche di questa evenienza.

2.3 Generazione di una soluzione ottima

La generazione di una soluzione ottima richiede di adottare alcuni accorgimenti per assicurarsi di aprire i sottoproblemi strettamente necessari a ottenere il risultato voluto.

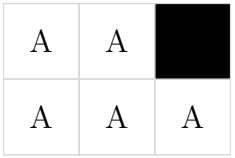

Figura 3: Una regione con geometria non quadrata potenzialmente scomoda da identificare.

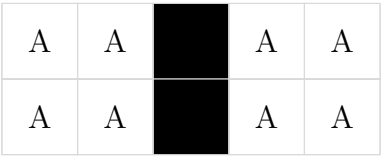

Figura 4: Due regioni non contigue etichettate con lo stesso identificatore.

In questo caso è conveniente applicare una strategia di partizionamento assimilabile alla seguente visione: *"per ogni possibile cella ancora bianca, prova a posizionare (temporaneamente) una regione quadrata di lato massimo (compatibile con la geometria corrente della griglia) e ricorri. In caso di backtrack su questa posizione ritenta con una regione quadrata via via più piccola"*.

Per evitare di visitare più volte del dovuto le celle della griglia, è utile guidare la ricorsione con una visita linearizzata dello spazio bidimensionale: lasciamo che la classica variabile `pos` identifichi la singola posizione di interesse per un certo livello della ricorsione, ricavando indirettamente l'indice di riga e colonna.

Poiché non si controlla direttamente il numero di quadrati che si intende utilizzare, è opportuno introdurre una condizione di pruning che chiuda tutti i rami di ricorsione che tentino di usare più regioni dell'eventuale ottimo attualmente noto.

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

```
void solve_r(int **m, int R, int C, int pos, int **sol, int curr_val, int
    **best, int *best_val) {
  int r, c, ok, l, L;
  if (pos >= R*C) { // terminazione
    if (*best_val > curr_val) { // soluzione migliorante
      *best_val = curr_val;
      copia2d(R, C, sol, best);
    }
    return;
  }
```

```
  // La soluzione corrente e' peggiore dell'ottimo noto
  if (curr_val > *best_val) return;
  // Ricava riga e colonna da pos
  r = pos / C;
  c = pos % C;
  // Se la cella e' libera
  if (sol[r][c] == WHITE) {
    // stima lato massimo (prima riga/colonna del quadrato)
    L = findMinHW(sol, R, C, r, c); // minimo tra altezza e larghezza
    curr_val += 1; // dichiara di usare una nuova regione
    for(l=L;l>0;l--) { // per tutti i lati, da L a decrescere
      // prova a posizionare la nuova regione
      ok = fill(sol, R, C, r, c, l, curr_val);
      // se non ci sono sovrapposizioni, ricorri
      if(ok) solve_r(m, R, C, pos+l, sol, curr_val, best, best_val);
      // pulisci l'area appena occupata
      undo(m, sol, R, C, r, c, l);
    }
    curr_val -= 1;
  } else {
    // ricorri direttamente per saltare regioni colorate o nere
    solve_r(m, R, C, pos+1, sol, curr_val, best, best_val);
  }
}
```

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, la maggior parte delle implementazioni esaminate è caratterizzata da una complessità intrinsecamente più elevata del dovuto poiché non si è prestato attenzione a guidare la ricerca in modo da evitare di esplorare porzioni già viste dello spazio. Questo accade ad esempio quando non si sfrutta `pos` per identificare la cella di lavoro di interesse ma ci si affida

a un doppio-for aggiuntivo dentro alla funzione ricorsiva stessa per aprire ulteriori rami di ricorsione.

Per la natura del problema è poi poco conveniente cercare di fissare a priori il numero di regioni target e/o la loro dimensione, poiché non si ha modo di inferire a priori delle informazioni utili a guidare la ricerca. In assenza di informazioni specifiche, risulta più conveniente lasciare che la ricerca usi un numero libero di regioni a dimensione qualsiasi, e poi fare pruning all'occorrenza.