Appunti senza pretese di P2+Lab:
Il Problema del Labirinto

Alessandro Panconesi
DSI, La Sapienza
via Salaria 113, piano terzo
00198 Roma

In queste dispense affronteremo il seguente problema, detto del labirinto. Abbiamo un labirinto, rappresentato da una struttura dati a puntatori, come in Figura 1.

I nostri labirinti saranno sempre quadrati e denoteremo con m il valore del lato (numero di nodi). In figura $m = 10$. Un nodo bianco è libero e pertanto attraversabile, mentre un nodo nero si considera occupato da un muro. L'entrata è il nodo in alto a sinistra, mentre l'uscita è in basso a destra. Gli spostamenti possono avvenire solo secondo il verso delle freccie, verso il basso o a destra. Vogliamo scrivere un programma C che, data una struttura dati di tipo "labirinto" restituisca vero se esiste un cammino tra l'entrata e l'uscita e falso altrimenti.

Per la discussione che segue è opportuno introdurre la seguente convenzione. Ci riferiremo ai nodi di un labirinto usando due coordinate. Il nodo in basso a destra è il nodo $(0, 0)$, il nodo di entrata ha coordinate $(0, m - 1)$, l'uscita $(m - 1, 0)$, ecc.

Problema 1 *Definire una struttura a puntatori dati* `Maze` *atta a rappresentare un labirinto.*

Il problema ammette una soluzione ricorsiva semplice ed elegante. Sia p (il puntatore alla) posizione corrente, allora:

- (Caso base.) Se p é l'uscita restituisci vero, mentre se p é un muro restituisci falso.
- (Passo induttivo.) Restituisci vero se esiste un cammino che porta all'uscita dal vicino destro oppure dal vicino in basso.

Pertanto, il codice C in Figura 2 risolve il problema. Notare che il codice sfrutta in modo essenziale la lazy evaluation del C. Rispetto alla soluzione appena vista, il codice C deve anche gestire i puntatori `NULL` che rappresentano i bordi del labirinto.

Vediamo adesso ad un esempio di dimostrazione di correttezza del software. Notare che dimostrare la correttezza di un algoritmo è cosa ben diversa e in generale assai più agevole che dimostrare la correttezza del codice.

La dimostrazione seguente è semplice nella sua esecuzione, ma non del tutto nella sua progettazione. La difficoltà consiste nel trovare un enunciato che renda la dimostrazione induttiva semplice.

Teorema 1 *Il codice di Figura 2 termina ed è corretto per ogni posizione del labirinto.*

Dimostrazione: Dire che il codice è corretto significa che esso da la risposta giusta, `True` se il cammino esiste e `False` altrimenti.

Introduciamo una nozione di *distanza*. La distanza tra due nodi del labirinto è il numero minimo di passi che servono per spostarsi dall'uno altro in un labirinto privo di muri. Non

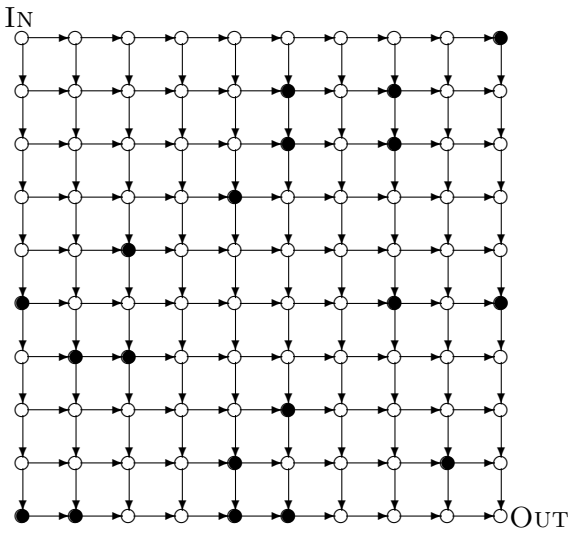

Figure 1: Un labirinto

```
Boolean find(Maze p} {
   if (p == NULL) || (p->type == WALL) return FALSE
   else return (p->type == EXIT) || find(p->right) || find(p->down)
   }
```

Figure 2: Codice C per il Labirinto.

sempre la distanza è definita ma lo è tra un nodo qualsiasi e l'uscita. Ad esempio in Figura 1 il nodo $(5, 2)$, che è un muro, ha distanza 6 dall'uscita, il nodo $(9, 5)$, un nodo dal quale l'uscita non è raggiungibile, ha distanza 5 dall'uscita, ecc.

Definiamo poi il concetto di *livello*. Il livello di un nodo è la sua distanza dall'uscita. Ad esempio, l'uscita è a livello 0 mentre l'entrata è a livello $2m$. In Figura 1 il nodo $(5, 2)$ è a livello 6 ecc.

La dimostrazione dell'enunciato è per induzione sul numero di livello. Nel caso base (livello $= 0$) dobbiamo capire cosa fa il programma se viene invocato sul nodo uscita. Ci sono due casi da considerare. Se l'uscita è libera il programma termina restituendo `True`. Se invece c'è un muro termina resituendo `False`. Quindi in entrambi i casi l'asserto è provato.

Nel caso generale, sappiamo che l'asserto è vero per tutti i nodi di livello minore o uguale a k e dobbiamo considerare un nodo a livello $k + 1$. Consideriamo solamente il caso in cui il nodo a livello $k + 1$, che denominiamo x non è un nodo del bordo, denominando i due vicini d_x (destra) e g_x (giù). Lasciamo l'analisi dei casi mancanti come esercizio per il lettore. Per un nodo interno, i casi da considerare sono questi:

- Il nodo è murato. In questo caso evidentemente il codice termina restituendo `False`, la risposta corretta.
- Se il nodo x è libero l'ispezione del codice rivela che viene effettuata una chiamata ricorsiva sul vicino destro d_x. Per induzione la chiamata termina. Se il valore restituito è `True` anche la chiamata su x restituisce `True`. Questa è la risposta corretta perchè per induzione esiste un cammino tra d_x e l'uscita e quindi anche tra x e l'uscita.

  Se invece la chiamata su d_x restituisce `False`, per induzione sappiamo che non c'è un

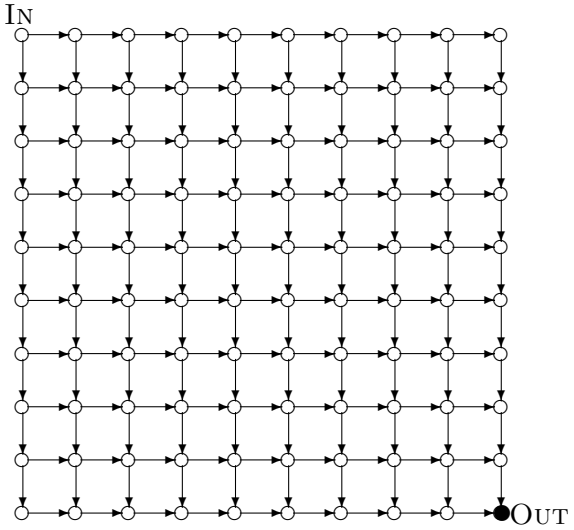

Figure 3: Il caso peggiore.

cammino tra d_x e l'uscita. Il programma prosegue invocando ricorsivamente se stesso su g_x. Per induzione la chiamata termina. Se questa restituisce `True` allora sappiamo che anche la chiamata su x farà altrettanto e che questa è la risposta giusta. Se invece il valore restituito è `False` allora per induzione sappiamo che non c'è cammino tra g_x e l'uscita. In questo caso la chiamata su x restituisce `False` che è la risposta corretta in quanto l'uscita non è raggiungibile da nessuno dei due nodi immediatamente raggiungibili da x.

$\triangledown$

Esercizio 1 *Si supponga di modificare il labirinto collegando il nodo (m, k) al nodo $(0, k)$, per $0 \leq k \leq m-1$, ed il nodo $(0, k)$ al nodo $(m-1, k)$, sempre per ogni $k \in \{0, 1, \ldots, m-1\}$ (il labirinto risultante diverrebbe un toro, cioè una ciambella [sic]). Funzionerebbe ancora il codice di Figura 2? Dare una dimostrazione oppure evidenziare in che punto la dimostrazione per induzione cesserebbe di essere valida.*

Il programma è semplice ed elegante e come abbiamo visto è persino possibile dimostrarne la correttezza, ma ha un problema. Il tempo di calcolo può essere astronomico! Per capirlo effettueremo una semplice ma elegante analisi di tipo combinatorio. Intuitivamente, il caso peggiore è quando il programma visita tutti i cammini possibili, per scoprire solo alla fine che l'uscita è bloccata. Consideriamo quindi il caso di Figura 3, con un solo muro piazzato sull'uscita.

In questo caso il programma effettivamente esplora tutti i possibili cammini tra entrata e uscita. Il tempo di calcolo quindi è almeno proporzionale al numero di cammini, numero che adesso ci accingiamo a calcolare.

I nostri cammini hanno una struttura particolare, possono infatti andare solo in giù o a destra. Il primo passo della soluzione del nostro problema è introdurre una codifica per i cammini. Li rappresenteremo con una fila di 0 e 1, dove 0 vuol dire "giù" e 1 vuol dire "destra". La prima osservazione cruciale è che se il labirinto è un quadrato di lato m allora una stringa rappresentante un cammino deve avere $2m$ bits di cui m 0 e m 1. Questo perchè per raggiungere l'uscita dobbiamo fare almeno m spostamenti verso destra ed altrettanti verso

il basso. Sia

$$\mathcal{C} := \text{insieme dei cammini tra entrata e uscita}$$

e sia inoltre

$$\mathcal{S} := \{s : s \text{ è una stringa di } 2m \text{ bits contenente } m \text{ 0 e } m \text{ 1}\}.$$

Fatto 1 $|\mathcal{C}| = |\mathcal{S}|$.

Dimostrazione: La dimostrazione consiste nelllo stabilire una corrispondenza biunivoca f tra i due insiemi. Come visto, dato un cammino c, la stringa corrispondente $s := f(c)$ la si ottiene scrivendo uno 1 per ogni passo verso destra ed un 0 per ogni passo verso il basso. Dimostriamo innanzitutto che f è iniettiva. Siano dati due cammini diversi c_1 e c_2 e siano s_1 ed s_2 le stringhe corrispondenti. Dato che i cammini sono diversi, partendo dall'origine ad un certo punto deve per forza succedere che uno va verso il basso mentre l'altro va verso destra. Pertanto nella posizione corrispondente una stringa avrà 1 mentre l'altra avrà 0.

Per la suriettività bisogna dimostrare che, data comunque una stringa s di m zeri ed m 1, esiste un cammino c tale che $s = f(c)$. Se interpretiamo s come una lista di istruzioni per andare dall'entrata all'uscita secondo la convenzione 1 = *destra* e 0 = *giù* vediamo che sicuramente questo cammino connette l'entrata all'uscita in quanto (a) non ci sono muri e pertanto tutte le mosse sono eseguibili e (b) il cammino ci sposta di m posizioni in basso e di m a destra. ▽

Il problema quindi si sposta e diventa quello di stimare la cardinalità di $\mathcal{S}$. Definiamo

$$[n] := \{1, 2, \ldots, n\}$$

e

$$\mathcal{A} := \{X : \ X \subset [2m], \ |X| = m\}$$

Fatto 2

$$|\mathcal{S}| = |\mathcal{A}| = \left(\begin{array}{c} 2m \\ m \end{array} \right) > 2^m$$

Dimostrazione: Un sottoinsieme X di $[2m]$ si può rappresentare con un vettore di $2m$ coordinate in questo modo:

$$X_i := \begin{cases} 1 & \text{if } i \in X \\ 0 & \text{if } i \notin X. \end{cases}$$

Ma in questo modo si ottiene una stringa di $\mathcal{S}$. Chiaramente insiemi diversi vengono mappati in stringhe diverse. Viceversa, ogni stringa di $\mathcal{S}$ può essere interpretata come un insieme di $\mathcal{A}$. Quindi esiste una corrispondenza biunivoca tra $\mathcal{A}$ e $\mathcal{S}$. Chiaramente

$$\begin{aligned} |\mathcal{A}| &= \left(\begin{array}{c} 2m \\ m \end{array} \right) \\ &= \frac{2m \cdot (2m-1) \ldots \cdot (m+1)}{m \cdot (m-1) \ldots \cdot 1} \\ &> 2^m. \end{aligned}$$

▽

La complessità è pertanto esponenziale. Denotando con $n := m^2$ la dimensione del labirinto si ha che la complessità è almeno $2^{\sqrt{n}}$.

```
Boolean find(Maze p} {
   if (p == NULL) || (p->type == WALL) || (p->visited) return FALSE
   else {
       p->visited = TRUE;
       return (p->type == EXIT) || find(p->right) || find(p->down)
   }
}
```

Figure 4: Codice C per il Labirinto di complessità lineare.

Vediamo ora come fare per ridurre il tempo di esecuzione dell'algoritmo. La soluzione è basata su un principio semplice e importante: ricordando un po' di informazione in più è spesso possibile risparmiare tempo.

In questo caso l'idea è semplicemente quella di marcare come visitati i nodi che man mano si esplorano. Il codice risultante è illustrato in Figura 4.

Che la complessità sia lineare è evidente in quanto ogni nodo può essere visitato al più due volte in quanto ogni nodo ha al massimo due archi entranti. Questo codice è più efficiente perchè evita di rivisitare cammini già esplorati. La sua correttezza è lasciata come utile esercizio al lettore.

Concludiamo la nostra chiacchierata ponendo all'attenzione del lettore un esercizio di programmazione.

Esercizio 2 *Scrivere una funzione C che restituisca un cammino tra entrata ed uscita se questi esiste e NULL altrimenti.*