Progetto di Algoritmi e Strutture Dati

Diego Rosalio
10441A

10 luglio 2024

Sommario

Questa relazione descrive lo sviluppo e l'implementazione di un programma per la gestione e la manipolazione di un piano di piastrelle, con diverse operazioni come colorazione, spegnimento, gestione di regole di propagazione e calcolo di proprietà di blocchi di piastrelle. Il programma è stato implementato in Go e questa relazione illustra le principali componenti del codice, le strutture dati utilizzate e le funzionalità implementate.

1 Approccio al problema

Il piano è composto da una griglia di piastrelle aventi un colore e un'intensità, se l'intensità è nulla la piastrella è spenta. Ogni piastrella è inoltre associata a una posizione bidimensionale.

La modellazione più banale è quella che usa una matrice M in cui la cella M_{ij} contiene le informazioni della piastrella in posizione (i, j). Questa rappresentazione permette accesso $O(1)$ alle informazioni di ciascuna piastrella e delle sue circonvicine, spegnimento di piastrelle in tempo $O(1)$. Siccome non esiste limitazione al numero di piastrelle, la colorazione di nuove piastrelle risulta critica: è impensabile pre–allocare una matrice grande abbastanza da mantenere tutte le possibili piastrelle. È quindi buona idea mantenere una matrice grande abbastanza da contenere solamente le piastrelle colorate fino a quel punto, e usare la ri–allocazione quando necessario. Questa rappresentazione però spreca molto spazio: per esempio, colorando due piastrelle, una in posizione $(0, 0)$ e l'altra in posizione $(31, 31)$ si memorizzano anche altre 1022 piastrelle spente (matrice da 32×32).

In relazione alle funzionalità descritte nella specifica, ogni piastrella spenta è uguale a tutte le altre spente: l'intensità è nulla e il colore è irrilevante siccome le piastrelle spente non partecipano alla propagazione e l'unico modo per accenderle è quello di assegnarli un nuovo colore ogni volta. Le piastrelle spente non hanno quindi motivo di essere memorizzate, il che permette di risparmiare molta memoria. La modellazione più utile a questo scopo (**e usata in questo progetto**) è quella di grafo non orientato e non connesso in cui ogni vertice rappresenta una piastrella accesa e ogni arco rappresenta la nozione di circonvicinanza.

L'insieme dei vertici può essere rappresentato attraverso un dizionario (mappa di go), indicizzato per posizione della piastrella rappresentata (Listing 1). Grazie al funzionamento del look–up delle mappe in go (paradigma *comma–ok*), risulta molto leggibile il codice: `piastrella, accesa := p.piastrelle[pos]`. Il valore di accesa (che indica l'esistenza dell'elemento nella mappa) indica appunto se la piastrella è accesa o meno.

Listing 1: Struttura dati piano

```
type piano struct {
  piastrelle map[Punto]*Piastrella
  regole     *[]*Regola
}
```

Siccome la circonvicinanza è definita a partire dalle posizioni delle piastrelle gli archi sono già implicitamente definiti: partendo da una piastrella in posizione $p_1 = (x_1, x_2)$ esiste un arco se esiste nella mappa la piastrella in posizione p_2, dove p_2 può essere: (x_1, y_1+1) o (x_1+1, y_1+1) o (x_1+1, y_1) o (x_1+1, y_1-1) o (x_1, y_1-1) o (x_1-1, y_1-1) o (x_1-1, y_1) o (x_1-1, y_1+1). Esplicitando però gli archi in vettori di adiacenza (usando così qualche byte di memoria in più) si possono evitare molti accessi alla mappa, consentendo di trovare le piastrelle circonvicine in tempo costante invece che in tempo ammortizzato costante.

Le piastrelle quindi oltre a intensità e colore memorizzano anche il vettore di adiacenza (Listing 2).

Listing 2: Struttura dati piastrella

```
type Piastrella struct {
  Colore    string
  Intensità int
  adiacente [NumeroDirezioni]*Piastrella
}
```

Usare un vettore al posto di una slice per le adiacenze risulta molto comodo avendo a disposizione un'enumerazione delle possibili direzioni (Listing 3).

Grazie alla possibilità offerta da go di creare tipi *alias* e associare a essi dei metodi ho associato al tipo `Direzione` il metodo che consente di ottenere la direzione opposta (esempio: la direzione opposta di `NordOvest` è `SudEst`).

Se il vettore di adiacenza contiene un riferimento `nil` in posizione i significa che la piastrella circonvicina in direzione $d(i)$ è spenta.

Altra struttura dati di supporto è `Punto` che rappresenta una posizione bidimensionale e ha un metodo di utilità (`PuntoA(Direzione) Punto`) che restituisce il punto spostato della direzione.

Le regole sono rappresentate come strutture ciascuna contenenti i vari requisiti, memorizzati in una slice per consentirne la stampa ordinata secondo le specifiche, il risultato dell'applicazione della regola e il numero di volte in cui questa è stata usata. Nella struttura dati piano presentata nel Listing 1 le regole sono memorizzate in una slice (per essere ordinate facilmente in seguito) di cui è memorizzato il puntatore, questo perché nelle signature delle funzioni date nella traccia, un'istanza di piano è passata sempre per copia.

Listing 3: Enumerazione direzione

```
type Direzione int

const (
  Nord Direzione = iota
  NordEst
  Est
  SudEst
  Sud
  SudOvest
  Ovest
  NordOvest
  NumeroDirezioni
)

func (d Direzione) opposta() Direzione {
  return (d + 4) % NumeroDirezioni
}
```

2 Aggiunta e Rimozione di piastrelle

Quando una piastrella passa dallo stato di spento a quello di acceso (operazioni di colorazione e propagazione), essa deve essere aggiunta come vertice del grafo (le piastrelle spente non ne fanno parte). Quando si aggiunge una piastrella al grafo bisogna modificare i valori delle liste di adiacenza delle piastrelle circonvicine e aggiornare il dizionario dei vertici. A questo scopo esiste il metodo `aggiungiPiastrella(pos Punto, pst Piastrella)`. Sia n il numero di vertici del grafo. Siccome l'accesso alla mappa avviene in tempo ammortizzato costante e il ciclo `for` itera sulle direzioni (che non dipendono da n) l'aggiunta di una piastrella avviene in tempo $O(1)$ e in spazio $O(1)$ (vengono create solo un numero costante di variabili).

Quando invece una piastrella passa dallo stato di acceso a quello di spento (operazione di spegnimento), deve essere eliminata la piastrella dalla dizionario dei vertici e devono essere eliminati i riferimenti dal dizionario dei vertici. A questo scopo esiste il metodo `rimuoviPiastrella(pos Punto)`. Similmente all metodo `aggiungiPiastrella`, la rimozione avviene in tempo $O(1)$ e in spazio $O(1)$.

3 Visita in Ampiezza

La visita in ampiezza dei grafi risulta molto utile per operazioni che richiedono di visitare tutte le piastrelle connesse a partire da una piastrella specifica, come la propagazione di modifiche o il calcolo dell'intensità di un blocco di piastrelle. In questo caso specifico la BFS si può usare anche per calcolare la lunghezza delle piste minime; questo perché ogni arco ha peso unitario e non si ha l'interesse di trovare il cammino minimo esatto ma solo la sua lunghezza: la BFS genera un albero in cui ogni livello h contiene tutti i vertici che sono raggiungibili dalla radice in h passi: in questo modo la lunghezza delle piste minime è facilmente calcolabile.

Visto la sua utilità è necessario generalizzare l'algoritmo di visita in ampiezza in modo tale che sia utilizzabile per le diverse funzionalità.

L'algoritmo astrae l'operazione di visita tramite il passaggio di funzione: in particolare il tipo `Visitatore` indica la funzione che deve essere invocata su ogni vertice durante la visita BFS. A questa funzione verrà passato anche un parametro `profondità` che indica l'attuale distanza tra il vertice da cui è partita la BFS e il vertice su cui e invocato il visitatore, questo parametro sarà pari a 0 per la prima piastrella visitata. Il visitatore deve restituire un valore booleano che indica se la visita BFS deve estendersi anche ai vicini del nodo corrente non ancora visitati (più lontani dalla radice).

Il metodo `visitaInAmpiezza(*Piastrella, Visitatore)` effettua la BFS su un'insieme di piastrelle partendo da una specifica. La visita si propaga a tutte le piastrelle adiacenti, mantenendo traccia della profondità di ciascuna piastrella visitata. Ogni piastrella è visitata una sola volta.

3.1 Dettagli implementativi

Se la piastrella di partenza (partenza) è `nil`, la funzione termina immediatamente.

Per memorizzare le prossime piastrelle da visitare si utilizza una coda rappresentata da una lista (`list.New()`). Ogni elemento nella coda è una struttura (`elementoFrangia`) che contiene un puntatore a una piastrella e la profondità associata alla piastrella nella visita. In ogni momento la coda mantiene due invarianti:

1. tutti gli elementi sono in coda per la prima volta
2. siano e_i ed e_j due elementi nella coda tali che e_i viene prima di e_j, allora vale che $p(e_i) \leq p(e_j)$ dove $p(e)$ denota la profondità associata alla piastrella.

Si tiene traccia delle piastrelle già entrate nella coda tramite un insieme rappresentato da una mappa (`statoEsplorazione`). Inizialmente l'insieme contiene solamente la piastrella di partenza

Il processo di visita avviene in questo modo:

1. La funzione estrae elementi dalla coda fino a quando questa non è vuota.
2. Per ogni piastrella estratta, viene chiamata la funzione di visita (parametro) con la piastrella e la sua profondità.
3. Se visita restituisce `false`, la visita della piastrella corrente viene terminata.
4. Per ogni direzione (`Nord`, `NordEst`, `Est`, `SudEst`, `Sud`, `SudOvest`, `Ovest`, `NordOvest`), la funzione verifica se c'è una piastrella adiacente non ancora esplorata.
5. Se una piastrella adiacente viene trovata e non è stata ancora esplorata, viene aggiunta alla coda con la profondità incrementata di 1, e viene segnata come esplorata nella mappa statoEsplorazione.

3.2 Costi

L'accesso alla mappa avviene in tempo ammortizzato costante. In go l'inserimento in coda è l'eliminazione in testa degli elementi nella lista che rappresenta la coda avviene in tempo costante.

Sia n il numero di vertici del sotto–grafo connesso a cui appartiene il vertice di partenza e v_i un generico vertice del sotto–grafo con $1 \leq i \leq n$. Per ogni elemento nella coda viene:

1. rimosso un elemento dalla coda ($O(1)$);
2. chiamata una funzione f sul vertice v_i che si conclude in tempo $T_f(v_i)$;
3. per ogni direzione vengono fatti inserimenti in mappe e code e siccome il numero di direzioni non dipende da n, il tempo è $O(1)$

Siccome per la prima invariante ogni vertice compare nella coda al massimo una volta, il costo dell'algoritmo nel caso peggiore (cioè vengono visitati tutti i vertici connessi) è:

$$\sum_{i=1}^{n} T_f(v_i) + O(1)$$

Per quanto riguarda lo spazio esso è di $O(n + \max S_f(v_i))$ perché viene crato un dizionario che avrà al massimo dimensione pari a n (alla fine della visita) e una coda che può contenere al massimo n elementi contemporaneamente (prima invariante). $S_f(v_i)$ indica lo spazio occupato dal visitatore f in corrispondenza del generico vertice v_i.

4 Operazioni

4.1 Colora

L'operazione `colora` si basa sul metodo di aggiunta delle piastrelle che ha costo temporale $O(1)$ e spaziale di $O(1)$.

4.2 Spegni

L'operazione `spegni` si basa sul metodo di rimozione piastrelle che ha costo temporale $O(1)$ e spaziale di $O(1)$.

4.3 Regola

Per inserire una nuova regola viene usata la funzione `regola(p piano, r string)`. Essa riceve in input una stringa che rappresenta gli argomenti passati da linea di comando per la creazione della nuova regola.

Per prima cosa la stringa di input di lunghezza n viene suddivisa in base a spazi, al costo temporale di $O(n)$, la dimensione della slice contenente il risultato è lineare alla lunghezza dell'input e quindi questo passo consuma $O(n)$ spazio.

Viene creata una nuova regola contenente una slice di requisiti con la dimensione corretta, in modo tale da evitare costose ri–allocazioni successive.

Viene eseguito il `for` un numero di volte lineare alla lunghezza dell'input, e per ogni iterazione si eseguono operazioni costanti (grazie alla allocazione preventiva della slice `Requisiti`).

Viene infine aggiunta la regola al grafo al costo temporale di $O(1)$ e spaziale di $O(1)$

Il costo temporale di questa operazione è quindi pari a $O(n)$. Il costo spaziale è pari a $O(n)$.

4.4 Stato

Per ottenere lo stato di una piastrella viene usata la funzione di interrogazione `stato(p piano, x int, y int) (string, int)`. Essa crea un numero costanti di variabili e quindi consuma spazio $O(1)$. Viene interrogata il dizionario dei vertici, che avviene in tempo ammortizzato $O(1)$.

4.5 Stampa

La funzione `stampa(p piano)` effettua l'operazione di `stampa`. Essa itera su ogni regola presente nel piano e ne effettua la stampa. Essa crea un numero costanti di variabili e quindi consuma spazio $O(1)$. La funzione itera su tutte le regole perciò, se n indica il numero di regole, la funzione consuma tempo $O(n)$

4.6 Blocco e BloccoOmog

Le operazioni di `blocco` e `bloccoOmog` sono simili. Entrambe si appoggiano al metodo `calcolaIntensitàBlocco(pnt Punto, omogeneo bool) int`.

Esso, dopo aver creato un numero costante di variabili, usa una visita in ampiezza. Il visitatore usato ha tempo di esecuzione $T_f(v) = O(1) \quad \forall v \in V$ e spazio $S_f(v) = O(1) \quad \forall v \in V$.

Segue che tempo e spazio sono lineari alla dimensione del blocco e cioé pari a $O(n)$.

4.7 Propaga

L'operazione `propaga` è implementata dal metodo `propaga(Punto)`.

I metodi coinvolti sono:

- `applicabile(map[string]int) bool`: tempo $O(1)$ siccome ogni regola ha al massimo 8 requisiti;
- `primaRegolaApplicabile(map[string]int) *Regola`: tempo $O(m)$ dove m è il numero di regole totali;
- `statoIntorno(Punto) map[string]int`: tempo $O(1)$ perché al massimo vengono controllate 8 piastrelle e l'accesso a mappe avviene in tempo costante

La propagazione su singola piastrella avviene quindi in tempo dipendente solamente dal numero di regole presenti, ovvero $O(m)$. Tutti i metodi coinvolti usano variabili di appoggio in numero, nel caso peggiore, costante quindi lo spazio è $O(1)$

4.8 PropagaBlocco

L'operazione `propagaBlocco` è implementata dal metodo `propagaBlocco(Punto)`.

Esso usa una visita in ampiezza per ottenere una lista contenente tutte le piastrelle del blocco in questione. Denoto con V l'insieme delle piastrelle del blocco. Il visitatore ha $S_f(v) = O(1) \quad \forall v \in V$. Il visitatore impiega, nel caso peggiore (ovvero quando la lista creata per contenere tutte le piastrelle del blocco è piena), $T_f(v_i) = O(i-1)$. La visita in ampiezza può quindi impiegare tempo pari a $O(n^2)$. Questo succede quando aggiungere un elemento alla lista comporta sempre la ri–allocazione. In go però viene usata la

tecnica del raddoppiamento–dimezzamento che consente in termini di costo ammortizzato $T_f(v_i) = O(1)$. In go quindi questa visita costa temporalmente $O(n)$.

Successivamente per ogni piastrella del blocco viene calcolato il risultato dell'applicazione delle regole, il cui risultato è salvato in una slice esterna per mantenere le proprietà di propagazione definite nella traccia. L'applicazione delle regole avviene in tempo $O(n \cdot m)$ dove m è il numero di regole totali (Vedi sezione 4.7).

Infine vengono aggiornate tutte le piastrelle con i risultati precedentemente calcolati. Il tempo totale è quindi $O(n \cdot m)$.

Vengono usate diverse slice grandi quanto il numero di vertici del blocco. Lo spazio è quindi lineare: $O(n)$

4.9 Ordina

L'ordinamento di regole al piano avviene tramite il metodo `ordinaRegole()`.

Sia n il numero di regole associate al piano. Siccome le regole sono memorizzate in una slice e l'ordinamento di esso viene effettuato dalla funzione di libreria `slices.SortStableFunc()` che ha costo $O(n \log n)$, il costo temporale di ordinamento è $O(n \log n)$.

Tipicamente non viene allocato spazio aggiuntivo durante l'ordinamento, perciò lo spazio occupato è $O(1)$

4.10 Pista

L'operazione di calcolo della pista è implementata dal metodo `pista(Punto, []Direzione) []*Piastrella`. Esso contiene un ciclo `for` che itera sulle direzioni passate come parametro. Con n denoto il numero di direzioni passate come parametro. Il tempo usato dal metodo è $O(n)$.

Il più grande utilizzo dello spazio è rappresentato dalla slice che viene restituita che contiene $n + 1$ elementi. Anche lo spazio è perciò lineare: $O(n)$

4.11 Lung

Per calcolare la lunghezza della pista più breve viene usata una visita in ampiezza (vedi Sezione 3). Il metodo che implementa questa operazione è `lunghezzaPistaBreve(partenza, arrivo Punto) int`.

Denoto con n il numero di vertici del sotto–grafo connesso a cui appartiene la piastrella di partenza.

Nel caso pessimo, la visita si protrae finché vengono visitati tutti i vertici e si scopre che le piastrelle di partenza e arrivo non appartengono allo stesso sotto–grafo connesso. Il visitatore utilizza spazio e tempo costante, perciò la visita costa in totale $O(n)$ sia per lo spazio, sia per il tempo.

Il metodo usa un numero di variabili costanti e la maggior parte del calcolo viene fatto grazie alla visita in ampiezza. Il tempo e lo spazio di esecuzione totale perciò è pari a $O(n)$

5 Esempi

Sono presenti alcuni esempi che sono stati utilizzati per verificare il programma. Ogni file di input è nel formato `10441A_rosalio_diego_input_n.txt`. Ogni file di output è

nel formato `10441A_rosalio_diego_output_n.txt` dove `n` indica il numero del test. Ogni file di output è stato scritto a "mano" e poi confrontato con l'effettivo risultato. Ecco gli aspetti su cui si concentra l'ennesimo test:

1. Nonostante le piastrelle accese siano molto sparse, viene usata la quantità minima di memoria.
2. Si concentra sulla propagazione.
3. Si concentra sull'ordinamento delle regole.
4. Si concentra sul calcolo delle piste date le direzioni.
5. Si concentra sugli spegnimenti.