De Luca Daniele, 01576A

Relazione progetto “Piastrelle Digitali”

Modellazione e implementazione

Analizzando la traccia emergono alcune considerazioni importanti: è necessaria una struttura dati adeguata e efficiente per rappresentare il piano con le piastrelle e i relativi blocchi. Inoltre è necessario usare delle strutture per rappresentare adeguatamente le regole e i suoi campi.

Implementazione del piano

La rappresentazione del **piano** doveva includere l'insieme delle **piastrelle** e delle **regole**. E' stato necessario quindi memorizzare tutte le piastrelle, per fare ciò si è utilizzata una mappa che ha come chiave una singola piastrella e come valore associato un tipo **colore**. Le piastrelle sono salvate nella mappa tramite le coordinate (x, y) nel piano. Ad ogni piastrella sono associati i dati relativi al **colore** e all'**intensità** di essa grazie al tipo colore, che presenta questi dati come campi. Questa rappresentazione è utile perchè permette di verificare velocemente se una piastrella è accesa o meno, e se accesa, di che colore è.

```
type piano struct {
    piastrelle map[piastrella]colore
    regole     *[]regola_
}
type piastrella struct {
    x, y int
}

type colore struct {
    coloree   string
    intensita int
}
```

Le **regole** sono memorizzate tramite un puntatore all'indirizzo di una slice di regole. L' utilizzo del puntatore è utile per gestire dinamicamente la struttura dati, senza dover restituire la slice modificata dalle funzioni che operano su di essa (la slice di regole viene definita solo nel main).

```
type regola_ struct {
    addendi   []colore
```

```
    risultato string
    consumo   int
}
```

Il tipo regola_ ha tre campi:

- **addendi**, rappresentati da una slice di colori, contengono i colori e le intensità associate di una regola
- **risultato**, cioè il colore finale se la regola di propagazione può essere utilizzata, rappresentato tramite una stringa
- **consumo**, un campo intero che viene incrementato all'utilizzo della una regola di propagazione

Implementazione funzioni principali

Colora

```
func colora(p piano, x int, y int, alpha string, i int) {
    p.piastrelle[piastrella{x, y}] = colore{alpha, i}
}
```

La funziona colora ha come parametri il **piano**, le **coordinate** di una piastrella, il **colore** sotto forma di stringa, e l'**intensità**. La piastrella in input viene colorata grazie all'utilizzo della **mappa** che contiene le piastrelle nel piano, a questa vengono passate le coordinate, il colore e l'intensità desiderata.

- **Complessità temporale**: l'accesso a una mappa ha costo **O(1)**
- **Complessità spaziale**: non viene allocato alcuno spazio, quindi abbiamo costo costante di **O(1)**

Spegni

```
func spegni(p piano, x int, y int) {
    delete(p.piastrelle, piastrella{x, y})
}
```

Spegni ha come parametri il **piano** e le **coordinate** di una piastrella. Questa funzione permette di spegnere la piastrella passata in input tramite le coordinate, nel piano. Ciò viene fatto sotto forma di cancellazione della chiave nella mappa che contiene tutte le piastrelle.

- **Complessità temporale**: l'operazione di **delete** ha tempo costante **O(1)**
- **Complessità spaziale**: non viene allocato alcuno spazio, quindi abbiamo costo costante di **O(1)**

Regola

```
func regola(p piano, r string) {
    arr := strings.Split(r, " ")
    var nuovaRegola regola_
    nuovaRegola.risultato = arr[1]
    var addendo colore

    for i := 2; i < len(arr); i++ {
        if i%2 == 0 {
            addendo.intensita, _ = strconv.Atoi(arr[i])
        } else {
            addendo.coloree = arr[i]
            nuovaRegola.addendi = append(nuovaRegola.addendi, addendo)
        }
    }
    *p.regole = append(*p.regole, nuovaRegola)
}
```

La funzione regola permette di aggiungere una regola nel piano, ha come parametri il **piano** e una **stringa**, che contiene i dati della regola da aggiungere. La stringa ha questa forma:

β k1 α1 k2 α2 $\cdot$ $\cdot$ $\cdot$ kn αn

Dove β è il **colore risultato** della regola, ki αi sono gli **addendi** della regola. Viene effettuata una **Split** sulla stringa, i suoi dati vengono salvati su una slice di stringhe, ignorando gli spazi. Viene effettuato **parsing** della slice, e grazie a questa operazione viene creata e aggiunta la regola nella slice di regole nel **piano**.

- **Complessità temporale**: la **Split** ha complessità **O(n)**, dove **n = numero caratteri stringa**. Abbiamo un **ciclo for** che itera sulla **slice**, che ha **k elementi**. Inoltre le restanti operazioni (assegnamenti di variabili e confronti hanno complessità costante **O(1)**). Nè risulta una complessità di **O(n) + O(k) = O(n)**, poichè **k <= n**.
- **Complessità spaziale**: Abbiamo due **variabili** che occupano spazio costante **O(1)**. La slice **addendi** cresce nell'ordine di **O(n) dove n è al max 8** dato che una regola non ha mai più di 8 addendi. L' aggiunta della regola occupa spazio costante **O(1)**. La **Split** ha complessità pari a **O(n)**. Quindi la complessità spaziale è nell'ordine di **O(n)**.

Stato

```
func stato(p piano, x int, y int) (string, int) {
    var piast colore
```

```
    var ok bool
    if piast, ok = p.piastrelle[piastrella{x, y}]; ok {
        fmt.Println(piast.coloree, piast.intensita)
    }
    return piast.coloree, piast.intensita
}
```

Stato prende come parametro il **piano** e le **coordinate** di una piastrella, **restituisce** e **stampa** il **colore** e l'**intensità** della piastrella in input se questa è accesa nel piano, altrimenti non verrà stampato nulla. Stato funziona correttamente grazie a un controllo di esistenza di una chiave in una mappa, in questo caso, passo alla mappa che contiene le piastrelle, le coordinate date come parametro in input e appunto nè controllo l'esistenza.

- **Complessità temporale**: assumiamo che i confronti, gli assegnamenti di variabili e la restituzione di valori abbiano tempo costante, quindi la complessità è **O(1)**
- **Complessità spaziale**: lo spazio utilizzato in questa funzione è nell'ordine di **O(1)**

Stampa

```
func stampa(p piano) {
    if len(*(p).regole) > 0 {
        fmt.Println("(")
        for _, rule := range *p.regole {
            fmt.Print(rule.risultato, ":")
            for i := 0; i < len(rule.addendi); i++ {
                fmt.Print(" ", rule.addendi[i].intensita, " ", rule.addendi[i].coloree)
            }
            fmt.Println()
        }
        fmt.Println(")")
    }
}
```

Questa funzione permette di **stampare** tutte le **regole** nel piano, nell'ordine in cui si trovano. La stampa viene effettuata nel seguente formato:

```
(
r1
r2
..
.
```

```
rm
)
```

La funzione prima controlla se sono presenti delle regole da stampare, in caso affermativo, viene iterata la slice di regole, e per ognuna stampa i diversi addendi, ciò viene fatto con due **cicli for** annidati.

- **Complessità temporale**: il primo ciclo ha complessità **O(n)** dove **n = numero regole nel piano**, il for interno effettua sempre al massimo **8 iterazioni** (perchè una regola ha al max 8 addendi), quindi ha complessità **O(k)** dove **k = 8**. Abbiamo quindi **O(n) * O(k) = O(n * k)**.
- **Complessità spaziale**: assumiamo che i confronti, gli assegnamenti e le operazioni di stampa abbiano complessità costante di **O(1)**

Blocco

```go
func blocco(p piano, x, y int) (int, []piastrella) {
    var inizio colore
    var ok bool
    var intensitaTotale int
    var sliceBlocco []piastrella
    if inizio, ok = p.piastrelle[piastrella{x, y}]; !ok {
        return 0, nil
    }

    intensitaTotale += inizio.intensita
    visitati := make(map[piastrella]bool)

    sliceBlocco = append(sliceBlocco, piastrella{x, y})
    coda := queue{}
    coda.Enqueue(piastrella{x, y})

    visitati[piastrella{x, y}] = true
    for !coda.isEmpty() {
        piast, _ := coda.Dequeue()

        adiacenti := cercaAdiacenti(p, piast)

        for i := 0; i < len(adiacenti); i++ {
            if _, ok := visitati[adiacenti[i]]; !ok {
                val := p.piastrelle[adiacenti[i]]
                intensitaTotale += val.intensita
                sliceBlocco = append(sliceBlocco, adiacenti[i])
                visitati[adiacenti[i]] = true
                coda.Enqueue(adiacenti[i])
```

```
            }
        }
    }
    return intensitaTotale, sliceBlocco
}
```

Blocco ha come parametri il **piano** e le **coordinate** di una piastrella. La funzione deve stampare la **somma delle intensità** del blocco di appartenenza della piastrella passata come parametro. Questa funzione restituisce l'intensità e una **slice di piastrelle** che contiene le piastrelle che fanno parte del blocco rispetto alla piastrella passata in input. Questa slice servirà più avanti per la funzione **propagaBlocco**. La funzione blocco fa uso dell'algoritmo **BFS** e di una funzione **cercaAdiacenti**:

```
func cercaAdiacenti(p piano, piast piastrella) []piastrella {
    var circonvicine []piastrella
    // genera combinazioni di coordinate possibili per la piastrella adiacente a quella in
    for i := -1; i <= 1; i++ {
        for j := -1; j <= 1; j++ {
            // esclude la piastrella in input (i == 0 e j == 0)
            if i != 0 || j != 0 {
                if _, ok := p.piastrelle[piastrella{piast.x + i, piast.y + j}]; ok {
                    circonvicine = append(circonvicine, piastrella{piast.x + i, piast.y + j}
                }
            }
        }
    }
    return circonvicine
}
```

cercaAdiacenti trova le piastrelle circonvicine rispetto a una piastrella in input generando tutte le possibili combinazioni di piastrelle intorno a quella in input, infine restituisce una **slice** contenenti le piastrelle circonvicine. La funzione ha **complessità temporale** costante **O(1)**

La BFS inoltre fa uso di una **coda**, notiamo l'utilizzo del campo **tail** che è utile nel metodo **Enqueue()**, dato che permette di non dover scorrere tutta la coda per aggiungere un elemento in ultima posizione, ma questa aggiunta viene effettuata in tempo costante.

```
type queueNode struct {
    value piastrella
    next  *queueNode
}
```

```
type queue struct {
    head   *queueNode
    tail   *queueNode
    length int
}
```

- **Complessità temporale**: assumiamo che i confronti, assegnamenti, accessi alla mappa, aggiunute di elementi a esse, e le operazioni sulla coda, abbiano tempo costante **O(1)**. La **ricerca degli adiacenti** ha complessità **O(8)**, (infatti il massimo di piastrelle adiacenti è 8). La **BFS** ha due cicli, il primo si ferma quando la **coda** è vuota, il secondo invece itera sulle piastrelle **circonvicine** all'elemento corrente in coda. Abbiamo una complessità **O(n)** dove **n = insieme dei vertici del grafo che rappresentano le piastrelle nel blocco**.
- **Complessità spaziale**: abbiamo la slice **sliceBlocco**, che nel caso peggiore può contenere fino a **n** elementi, quindi ha complessità **O(n)**. La mappa **visitati[piastrelle]bool**, anch'essa nel caso peggiore deve salvare tutte le piastrelle e avere complessità **O(n)**, cosi come la coda. La complessità spaziale totale è quindi **O(n)**.

Blocco omogeneo

Blocco omogeneo ha un comportamento simile a **blocco**, solo che in questo caso calcola e restituisce l'**intensità** di un **blocco omogeneo di appartenenza**. L' implementazione della funzione è pressochè identica a blocco, con una differenza, è presente un **if** che controlla se il **colore della piastrella iniziale** è uguale al **colore della piastrella circonvicina**, per verificare l'**omogeneità del colore**. Le prestazioni e la complessità non variano rispetto a blocco, quindi:

- **Complessità temporale**: **O(n)**
- **Complessità spaziale**: **O(n)**

Propaga

Propaga prende come input una **piastrella** e applica la prima regola di propagazione applicabile dall'elenco delle regole, ricolorando la piastrella. Se nessuna regola è applicabile, non viene eseguita alcuna operazione.

```
func propaga(p piano, x, y int) {
    colori := propagaGenerico(p, x, y)
    coloraPiastrelle(p, colori)
}

func propagaGenerico(p piano, x, y int) map[piastrella]regola_ {
```

```
    quantitaColori := make(map[string]int)
    coloriRisultati := make(map[piastrella]regola_)
    var flag bool
    var i int
    var rule regola_
    adiacenti := cercaAdiacenti(p, piastrella{x, y})

    for _, piastSingola := range adiacenti {
        val := p.piastrelle[piastSingola]
        col := val.coloree
        quantitaColori[col]++
    }
    for i, rule = range *p.regole {
        for _, str := range rule.addendi {
            arr := strings.Split(str.coloree, " ")
            v := str.intensita
            if c, ok := quantitaColori[arr[0]]; ok && c >= v {
                flag = true
            } else {
                flag = false
                break
            }
        }
        if flag {
            coloriRisultati[piastrella{x, y}] = rule
            (*p.regole)[i].consumo++
            flag = false
            break
        }
    }
    return coloriRisultati
}
```

Per questa funzione usiamo due mappe, **quantitaColori[string]int** che conta i colori delle piastrelle adiacenti a quella in input, cosi da verificare se una regola di propagazione è applicabile o meno, e **coloriRisultati[piastrella]regola_** che verrà restituita come risultato della funzione, questa mappa memorizza che regola di propagazaione deve essere applicata a una certa piastrella. Per "riempire" la mappa **quantitaColori** viene usata la funzione cercaAdiacenti. La funzione ha poi **due cicli for** annidati, il ciclo esterno itera sull'elenco delle regole, e quello interno itera sugli addendi delle regole per verificare se la regola è applicabile o meno (per verificare se una regola è applicabile ci aiutiamo con l'utilizzo di una **flag**). Poi se abbiamo trovato una regola, viene aumentato il consumo della regola, e questa viene salvata e restituita sotto forma di mappa. Infine viene chiamata la funzione **coloraPiastrelle** che applica la funzione **colora** e quindi colora la piastrella.

- **Complessità temporale**: il ciclo per popolare la mappa **quantitaColori** ha una complessità **O(1)**, dato le operazioni all'interno del ciclo hanno costo costante e che abbiamo un massimo di 8 piastrelle adiacenti. Per i **due cicli for annidati** invece, quello esterno itera su tutte le **regole**, e quello interno itera sugli **addendi** di essa, quindi supponendo che le altre operazioni hanno tempo costante, sappiamo che gli addendi per una regola sono al massimo 8, quindi abbiamo complessità **O(1)**, il ciclo esterno ha complessità **O(n)** dove **n = numero regole**, quindi la complessità totale è **O(n)**. La funzione **coloraPiastrelle** in questo caso itererà un massimo di una volta, su un'unica piastrella, quindi ha complessità **O(1)**. La **complessità temporale** di propaga è **O(n) + 0(1) = O(n)**.
- **complessità spaziale**: la mappa **quantitaColori** conterrà al massimo i colori di 8 piastrelle, **coloriRisultati** avrà invece al massimo 1 regola di colorazione per una piastrella, entrambe hanno quindi complessità **O(1)**

propagaBlocco

Questa funzione deve propagare il colore **non solo su una piastrella**, ma sull'**intero blocco di appartenenza**.

```
func propagaBlocco(p piano, x, y int) {
    _, slc := blocco(p, x, y)
    colori := make(map[piastrella]regola_)
    var modifiche []map[piastrella]regola_

    for i := range slc {
        colori = propagaGenerico(p, slc[i].x, slc[i].y)
        if len(colori) > 0 {
            modifiche = append(modifiche, colori)
        }
    }

    for i := range modifiche {
        coloraPiastrelle(p, modifiche[i])
    }
}
```

Inanzitutto individuiamo le piastrelle che fanno parte del blocco usando la funzione **blocco**, che restituisce le piastrelle in una slice. Anche qui viene usata una mappa **colori[piastrella]regola_** che associa ad una piastrella la regola da utilizzare. Abbiamo poi **var modifiche []map[piastrella]regola_**, cioè una **slice di modifiche** da applicare a ogni piastrella. La funzione itera sulla slice che contiene le piastrelle del blocco, e vi applica la funzione **propagaGenerico**, che vi restituisce la regola da applicare ad una piastrella (se è presente una

regola applicabile), e la aggiunge alla slice delle modifiche. Infine si itera sulle slice delle modifiche e si chiama la funzione **coloraPiastrelle**.

- **Complessità temporale**: la complessità temporale è pari a **O(P * R)** dove **P = numero piastrelle nel blocco**, **R = numero regole**
- **Complessità spaziale**: abbiamo una slice che contiene le piastrelle nel blocco, quindi con una complessità **O(P)**, una mappa con i colori risultanti che ha complessità **O(n)** dove **n = numero colori**, quindi la complessità è **O(P * n)**.

Ordina

La funzione ordina, ordina l'elenco delle regole di propagazione in base al consumo delle regole stesse in ordine non decrescente, facendo attenzione a mantenere l'ordine relativo delle regole, quindi la funzione che ordina deve essere **stabile**.

```
func ordina(p piano) {
    slices.SortStableFunc(*p.regole, func(a, b regola_) int {
        return a.consumo - b.consumo
    })
}
```

Ho usato la funzione di libreria **SortStableFunc**. Questa funzione usa una variante di **merge sort in loco**.

- **Complessità temporale**: l'ordinamento è basato su dei confronti, nel caso peggiore non si scende al di sotto di **O(n log n)**
- **Complessità spaziale**: è in loco, non alloca spazio aggiuntivo, quindi la complessità è costante **O(1)**.