Relazione “Piastrelle digitali”

Pellegrino Dario 943224

1 Premesse sui file del progetto

Il progetto contiene diversi file tra cui un solo file main **943224_pellegrino_dario.go** eseguibile, sarà sufficiente eseguire il file **943224_pellegrino_dario.go** per eseguire il programma. Non è richiesta nessuna procedura di compilazione del progetto. Runnando il file *main* senza argomenti il programma sarà eseguito come da specifica tuttavia, aggiungere come argomento di input un **file** o una **directory** (es. *“go run 943224_pellesgrino_dario.go input”*), permette di eseguire il contenuto del file riga per riga consegnando su standard output il risultato delle righe (ovviamente il file deve esistere, e deve contenere delle operazioni valide, separate da a capo, assumiamo input corretto). È una utility per testare grossi input scritti in anticipo. È possibile trovare degli *input* già costruiti con i corrispondenti *output*, all’interno delle cartelle *inputs* e *outputs*.

2 Scelta delle strutture

Le piastrelle

Piastrella è il *tipo struttura* più semplice del progetto, contiene solamente le proprietà della piastrella vista singolarmente, quindi una *stringa* che indica il colore e un *intero* che indica la sua intensità.

Il piano

La rappresentazione della griglia del piano è stata implementata utilizzando una *mappa* con come insieme delle chiavi coppie di interi (vettore di interi di 2 dimensioni), e come valore associato un puntatore ad una piastrella: **[2]int –> *Piastrella**

Ritengo che questa fosse la scelta migliore in termini di spazio e con un ottimo compromesso nei tempi di accesso rispetto ad esempio ad una *matrice* dove il tempo di accesso è sempre costante.

Perché l’uso di una mappa

Flessibilità e Sparsità

Il problema richiede la gestione di piastrelle che possono essere posizionate in qualsiasi punto di una griglia molto ampia, con coordinate che possono raggiungere valori molto grandi e distanziati. Una *matrice* richiederebbe l'allocazione di spazio per tutte le posizioni possibili, anche per quelle che non contengono piastrelle, o meglio le cui piastrelle sono spente, risultando in uno **spreco significativo di memoria**. La *mappa* permette di **memorizzare solo le piastrelle effettivamente accese** sul piano, rendendo il programma molto più efficiente in termini di spazio quando la griglia è **popolata in modo sparso.**

Una *matrice* richiederebbe un controllo dei *bounds* ogni qual volta si voglia interagire con il piano per evitare errori di *out of bound.* In una *matrice* bisogna anche memorizzare dei valori di default che indicano una piastrella **spenta**, come il puntatore a **nil** cioè il puntatore “*che non punta a niente*”, anche questo sarebbe una vulnerabilità in più da gestire. Sicuramente dei controlli a tempo costante non appesantirebbero a livello di stime asintotiche dei tempi, ma aumenterebbero le righe di codice e i controlli, rendendo il codice più insidioso e aumentando il rischio di vulnerabilità.

Questi punti in una *mappa* non sarebbero una problematica, perché un accesso con chiave ad un elemento inesistente nella *mappa* si può sempre gestire con un semplice controllo dei valori restituiti.

Inserimento, colorazione e spegnimento di piastrelle

L'accesso agli elementi in una *mappa* ha una complessità media di *O(1)* grazie alla *hashtable* sottostante. Anche se l'accesso non è garantito essere costante nel caso peggiore, nella pratica l'implementazione di una buona *hashtable* rende l'accesso ammortizzato molto vicino a *O(1)*.

L'accesso a un elemento in una *matrice* è sempre *O(1)*, poiché gli elementi sono memorizzati in modo contiguo in memoria e possono essere indirizzati direttamente tramite gli indici. Tuttavia, questo vantaggio diventa irrilevante se si considera la quantità di memoria necessaria per rappresentare una griglia molto grande e sparsa. E anche le buone prestazioni in tempo di accesso delle mappe. Quindi questo vale analogamente per le operazioni di **inserimento**, **colorazione** e **spegnimento**.

Grazie all’efficienza del *re-hashing* nelle *hashtable* il costo ammortizzato di *N inserimenti* in tabella rimane *O(N), O(1) per ogni inserimento* anche effettuando *k re-hashing* per superamento del fattore di carico. Diverso invece è il caso di superamento dei limiti di una *matrice*, dove l’aumento delle dimensioni nel caso peggiore è molto più pesante. Soprattutto considerando che le operazioni di aggiunta e rimozione di piastrelle sono molto frequenti nel progetto.

Sebbene l'accesso alle mappe nel caso peggiore possa teoricamente essere *O(n)*, nella pratica, grazie alle buone proprietà delle *hashtable,* gli accessi e le operazioni sulle mappe sono ammortizzati a *O(1)*. Considerare le stime ammortizzate riflette più accuratamente le prestazioni reali del programma, che nella maggior parte dei casi beneficerà di tempi di accesso costanti e rapidi, garantendo efficienza e reattività anche su input di grandi dimensioni.

Analogamente vale lo stesso ragionamento per *append* sulle slice, nel caso peggiore può richiedere di ridimensionare la slice con tempo *O(n),* ma nella pratica il tempo di append è ammortizzato a *O(1)*.

Sostanzialmente la *mappa* ci permette di simulare una griglia di dimensioni infinite risparmiando molto spazio e tempi per le operazioni ammortizzati costanti.

Il piano contiene anche un puntatore ad una *slice* di *puntatori* a regole ***[]*Regola**, questo per poter utilizzare il *piano* come è stato previsto nelle segnature.

Regola

La regola contiene una mappa **string → int**. Contiene per ogni **colore α** *string* il corrispondente *intero* **k** (numero di vicini di quel colore α per applicare la regola). Assumiamo input corretto quindi non ci sono controlli su eventuali errori (es. somma dei vicini maggiore di 8).
Contiene il **risultato** in *string*, cioè il colore che viene applicato sulla piastrella se la regola è applicabile ed infine il contatore del **consumo** *intero*.
Una variabile di tipo regola ha spazio costante *O(1)* perché la *mappa* dei *colori* dei vicini può avere al massimo 8 elementi (assumendo input corretto).
Contiene anche una *slice* di *string* **ordine** con il solo scopo di permettere di stampare sempre nello stesso ordine la regola, per agevolare il testing.

Direzioni

Esistono due strutture per le parti in cui è necessario esplorare le direzioni dei vicini di una piastrella. Lo spazio occupato è ovviamente costante.

Vettore dirs dei vicini

Contiene tutte le direzioni (praticamente dei versori) verso i vicini di una qualsiasi *piastrella* di coordinate *x y*, tornando molto utile nelle successive funzioni che richiedono di esplorare la *"griglia"* di *piastrelle*.

Mappa degli spostamenti

È una mappa che associa ad ogni direzione con il sistema dei punti cardinali *{NO, NN, NE, EE, SE, SS, SO, OO}* il versore corrispondente. Utile per la funzione *pista* successivamente.

Nota sulle stringhe

Una piccola premessa sulla stima della complessità spaziale e temporale dove erano coinvolte le stringhe, ho considerato le stringhe che rappresentano i colori, come lunghezza “costante”, cioè ragionevolmente non da tenere in considerazione per le stime asintotiche. Per altre stringhe come le piste, ho tenuto in considerazione la lunghezza per le stime

3 Metodi e funzioni

Prima di entrare nei dettagli delle **funzioni richieste** dai **requisiti**, ci sono alcuni *metodi* e *funzioni utility* di cui parlare brevemente. Se le funzioni richiedono un'analisi più approfondita della **complessità temporale e spaziale** sarà precisata, altrimenti si assume sia irrilevante (costante).

Metodi e funzioni utility

Go permette di definire *metodi* sui tipi struttura, e ho deciso di utilizzarli per rendere il codice più leggibile.
Il metodo *String()* definito su *piastrella* e *regola,* funziona in modo simile al *ToString()* di *Java,* Grazie all’interfaccia *fmt.Stringer*.

Il metodo **applicabile** su una **regola,** dato in input una mappa **string → int** (i colori effettivi trovati) restituisce *true* se la regola è applicabile, *false* altrimenti.

La *funzione* **punto** restituisce un vettore di 2 posizioni (punto) contenente e coordinate x e y, principalmente per leggibilità.

La funzione *creaPiano()* permette di creare un *piano vuoto*.

Helper per visita in ampiezza della griglia bfsBlocco

Per la **funzione helper bfsBlocco** serve una spiegazione più dettagliata.
La BFS è ideale per esplorare tutte le piastrelle connesse a partire da una *piastrella* iniziale *(x, y)*. Garantisce che **tutte le piastrelle raggiungibili vengano visitate**, fornendo una **copertura completa del blocco**.

È utile per ridurre la duplicazione del codice in quanto viene chiamata nelle funzioni ***blocco, bloccoOmog*** e ***propagaBlocco***. Ho deciso in tutte e tre di fare una **visita in ampiezza** quindi, le 3 funzioni avevano molte porzioni di codice identiche o al massimo molto simili.

La **funzione bfsBlocco** dato un *piano* e una coordinata *(x, y)* di una piastrella, restituisce una slice di coordinate a piastrelle rappresentante la *visita in ampiezza* (in ordine di visita) del **grafo**

generato a partire da piastrella in posizione *(x, y)*. Praticamente facendo la *visita in ampiezza* a partire da una piastrella si ottengono le coordinate di tutte le piastrelle del suo blocco di appartenenza.
C'è anche la possibilità di visitare solo le piastrelle di colore uguale a piastrella in posizione *(x, y)* ottenendo il *blocco omogeneo* di appartenenza. Purtroppo in *go* non c'è la possibilità di argomenti in input opzionali come ad esempio in *python,* quindi è necessario dare in input all'argomento *bool* ***checkColore*** *true* se si vuole il *bloccoOmogeneo* e *false* se si vuole il blocco di appartenenza della piastrella in posizione *(x, y).*

La scelta di costruire la **coda** con una semplice slice e di utilizzare lo slicing è per una questione di semplicità di utilizzo e per l'efficienza garantita.

Complessità Temporale della bfs

Nel caso di questa funzione la complessità temporale richiede un'analisi più approfondita.
Grazie a come è strutturata la griglia e grazie alle prestazioni della *mappa*, la complessità temporale è migliore di quella di una normale visita in profondità di un grafo.
Ci sono *O(n)* piastrelle da visitare e per ogni piastrella visito 8 vicini, un numero costante quindi in tempo *O(1)*. Tuttavia fa 2 accessi in 2 mappe:

- Dentro il *piano* se ci sono m *piastrelle* il tempo ammortizzato è *O(1)* costante, ma il caso peggiore molto raro è *O(m).*
- Dentro la *mappa dei visitati*, che può arrivare a contenere n piastrelle, (quindi tutto il blocco) anche qui il tempo di accesso ammortizzato è *O(1)* costante, ma il caso peggiore molto raro è *O(n).*

Quindi **il tempo ammortizzato** è ***O(n)*** poiché esegue *O(n)* ripetizioni del ciclo principale in tempo ammortizzato costante *O(1),* con *n = #piastrelle del blocco.*

Mentre **nel caso peggiore molto raro** esegue *O(n)* ripetizioni del ciclo principale, e per ogni ripetizione impiega *O(n + m)*, se il piano è un unico blocco abbiamo il caso peggiore, dove:
n = #piastrelle del blocco = m, ***O(m) * O(2m) = O(m²).***
Caso molto raro perché l'accesso alla *mappa* del *piano* deve collidere tante volte e deve esserci un *unico blocco* nel *piano,* ha senso quindi considerare il **tempo ammortizzato**.

Complessità Spaziale della bfs

La **complessità spaziale** nel caso peggiore è ***O(m)*** cioè quando il *blocco* è delle dimensioni dell'intero *piano*. In questo caso anche la coda raggiunge spazio *O(m).*
O(m) + O(m) = ***O(m)***

Funzione Helper intensitàBlocco

La **funzione helper intensitàBlocco**, data una *slice* di *posizioni* di *piastrelle*, restituisce la somma delle intensità delle piastrelle. Chiaramente non controlla che la *slice* di *piastrelle* sia effettivamente un *blocco,* ma la *funzione* viene usata nel *programma* esclusivamente per calcolare le *intensità* dei *blocchi*, essendo anche l'unico caso in cui è richiesto nei *requisiti*.
La **complessità temporale ammortizzata** è ***O(n)****, n = #piastrelle del blocco,* più interessante per le reali prestazioni del programma.

La **complessità temporale** nel **caso peggiore** (molto raro) è $\boldsymbol{O(m^2)}$*:*

- *n = #piastrelle in input, n iterazioni.*
- *m = #piastrelle nel piano*

Il caso peggiore **non** è sempre $O(m^2)$ perché la *funzione* potrebbe ricevere in *input* n posizioni di piastrelle con $n > m$ visto che non fa controlli (quindi anche piastrelle spente). Ma essendo una funzione helper quindi utilizzata solo "internamente" nei casi in cui è chiamata ha caso peggiore: $O(n * m)$, con $n \le m$. Quindi se $n = m$*, ho caso peggiore* $\boldsymbol{O(m^2)}$ (molto raro).
Lo spazio è costante con una sola variabile intera

Funzioni richieste

colora

La **funzione colora** aggiunge alla *mappa* la coppia *vettore* di 2 dimensioni *(x, y)* rappresentante la posizione della *piastrella* e come *valore* il *puntatore* alla *piastrella* contenente *intensità* e *colore*. Se la piastrella esiste già sovrascrive e se l'intensità in input è zero chiama spegni sulla *piastrella (x, y)*.
Il tempo ammortizzato è costante *O(1)*, ma il caso peggiore (molto raro) è *O(n),* dove n è il numero di elementi presenti nella *mappa* (*hashtable*). Lo spazio è costante, crea una *struttura piastrella* contenente una *stringa* e un *intero*

spegni

La **funzione spegni** chiama *delete* sulla *mappa* di *piastrelle* che la elimina se la chiave esiste (*vettore* di 2 *interi*) e non fa nulla altrimenti.
Il tempo ammortizzato è costante *O(1)*, ma il caso peggiore (molto raro) è *O(n),* dove n è il numero di elementi presenti nella *mappa* (*hashtable*).

stato

La **funzione stato** stampa su *std output* lo stato di una piastrella nella forma *"colore intensità"* e restituisce i due valori *stringa* e *intero* non stampa niente se la piastrella è spenta e restituisce una *stringa* vuota e zero

Il tempo ammortizzato è costante *O(1)*, ma il caso peggiore (molto raro) è *O(n),* dove n è il numero di elementi presenti nella *mappa* (*hashtable*).

regola

La **funzione regola** chiama *append* sulla *slice* di *puntatori a regola* quindi mettendola in fondo alla *slice* come richiesto. Il tempo ammortizzato di *append* è *O(1)* ma il caso peggiore (sempre molto raro) è *O(n)*, per come sono implementate le *slice*. Lo spazio è costante se assumiamo l'input sempre corretto.

stampa

La **funzione stampa,** stampa su *std output* le regole nell'ordine attuale con un *for-each* e nel formato definito dal *metodo String()* di regola. Il tempo è Θ(n) dove n è il numero di regole definite (la lunghezza della *slice*)

blocco

La **funzione blocco,** calcola la somma delle intensità del *blocco* a cui appartiene la *piastrella (x, y)* e la stampa in *std output*, se la *piastrella* è *spenta* stampa zero. La **complessità spaziale e temporale** sono le stesse di *bfsBlocco* (che poi va a influenzare anche *intensitàBlocco*).

bloccoOmog

La **funzione blocco,** calcola la somma delle intensità del *blocco omogeneo* a cui appartiene la *piastrella (x, y)* e la stampa in *std output*, se la *piastrella* è *spenta,* stampa zero. La **complessità spaziale e temporale** sono le stesse di *bfsBlocco* (che poi va a influenzare anche *intensitàBlocco*).

propaga

La **funzione propaga**, salva le caratteristiche dei vicini con il *vettore dirs* e cerca la prima *regola applicabile*. Trovata la *regola* la applica sulla *piastrella (x, y)* e aumenta il *consumo* della *regola,* potrebbe accadere che nessuna *regola* sia applicabile e che la *funzione* debba scorrere tutte le *regole* . Nel caso peggiore, cioè scorrere tutta la slice di regole la **complessità temporale** è ***O(r)***:
r = #regole, O(r)

propagaBlocco

La **funzione propaga,** salva il blocco in una mappa di supporto, successivamente sulla base dei colori reali nel piano, colora la mappa di supporto applicando le regole per ogni piastrella del blocco (se ne esiste una applicabile) e aumentando i consumi. In questo modo si propagano le *piastrelle* con il *blocco* nello **stato iniziale**, **prima della procedura di propagazione sul blocco.** Successivamente colora le piastrelle del blocco con la mappa di supporto.

Il **tempo ammortizzato** è ***O(n*r)*:**
n = #piastrelle del blocco, r = # numero di regole, ***O(n*r).***

Cioè dopo aver fatto la bfs che ha tempo ammortizzato *O(n)*, per ogni piastrella del blocco controlla i vicini in *tempo costante*, verifica che una tra le *regole* sia applicabile, che richiede tempo *O(r).*

Il **tempo** nel **caso peggiore** è molto diverso, anche se estremamente raro questa è l'analisi. Nel caso peggiore la bfs ha tempo $O(m^2)$, quindi il blocco coincide con tutto il piano e ogni accesso alla *mappa* del piano è sempre il caso peggiore.
Se si verificano le stesse condizioni appena viste per la bfs la creazione del supporto richiede tempo $O(m^2)$.
Di nuovo, se l'accesso alla mappa è sempre il caso peggiore e ci sono m piastrelle nel blocco, Impiego $O(m^2 * r)$ perché la funzione scorre anche le regole alla ricerca di una applicabile.

Questo caso è molto raro ed è più sensato considerare il tempo ammortizzato.

La complessità spaziale per propagaBlocco è *O(m)*, vediamo nel dettaglio.
Se il *blocco* coincide con l'intera *pista* di *piastrelle accese* di dimensione *m*, *bfsBlocco* restituisce una slice contenente *m vettori di 2 interi* (posizioni delle piastrelle), *O(m)*.
La mappa di supporto contiene anch'essa esattamente *O(m)* valori (*stringhe* che sarebbero colori, ragionevolmente non enormi*)*.
O(m) + O(m) = O(m)

ordina

Per la **funzione ordina,** ordina le regole usando i **consumi** come chiavi (*interi***,** ho scelto di utilizzare una funzione di libreria *go* del modulo *sort*, *sort.SliceStable*. Questa funzione secondo la documentazione usa una versione modificata di mergeSort, che funziona in loco con l'ausilio di symMerge. Allego link alla documentazione.
https://cs.opensource.google/go/go/+/refs/tags/go1.18.3:src/sort/zsortinterface.go;l=359;drc=635b1244aa7671bcd665613680f527452cac7555

L'algoritmo garantisce **O(n * log(n))** confronti, e **O(n * log(n) * log(n))** scambi.
Grazie al fatto che usa mergeSort l'algoritmo è stabile ed era richiesto per l'ordinamento delle regole.

pista

La **funzione pista** utilizza la *mappa* spostamento per tradurre il percorso in versori, e utilizza la *funzione Split* di *strings* per creare una slice di spostamenti. Poi traduce ogni spostamento *s* con la *mappa* spostamento in una direzione *d*, se trova una piastrella nella direzione d accesa continua e aggiunge la piastrella alla pista, altrimenti si ferma e non stampa nulla. Se riesce a percorrere tutto il percorso lo stampa in *std output*, non stampa niente altrimenti.
Considerando il **tempo ammortizzato** (quindi con l'accesso alla mappa del piano costante e append della pista costante):

- Dato *N = #lunghezza della stringa s (lunghezza del percorso),* la split impiega tempo *O(N)* per splittare s.

- Se *strings.Split di s = seq, S = #lunghezza di seq.*
- Iterate e stampare la pista generata dalla sequenza di spostamenti *seq* impiega tempo *O(S) + O(S) = O(S)*
- Sicuramente *S* < *N* quindi il tempo ammortizzato è *O(N)*.

Quindi il tempo è *O(N)*

È corretto anche considerare il **caso peggiore** anche se come nelle altre funzioni è improbabile:

- Considerando *S* = *lunghezza di seq,* una pista che segue tutte le m piastrelle in piano
- Per ogni spostamento *s* in *seq,* impiega *O(m)* (tante collisioni nella mappa di piastrelle)
- Aggiornare la slice con le coordinate delle piastrelle della pista nel caso peggiore (la slice deve essere ridimensionata frequentemente) può essere *O(m)*.
- Dal calcolo del tempo precedente split impiega *O(N)*
- *O(N) + O(S * m) = O(S * m)*

Quindi il **tempo nel caso peggiore** è ***O(S * m)***

La complessità spaziale per pista è *O(S)*, vediamo i dettagli.
S = lunghezza di seq = #elementi della pista risultante.
O(S) + O(S) = O(S)

lung

La **funzione lung** calcola la lunghezza del percorso più breve tra due piastrelle in una griglia utilizzando di nuovo l'algoritmo BFS.

La BFS è particolarmente adatta per trovare la pista di lunghezza minima, perché nel nostro caso *"il grafo"* non è pesato e quindi è come se ogni arco avesse peso identico. In questi casi esplorare in ampiezza, salvare il vertice v (visitato) da espandere in coda con il peso del percorso fino a quel momento (numero di archi su cui si è passati da s fino a v). Ci permette di trovare il percorso minimo a partire da un vertice s per qualsiasi altro vertice v. Una pista da *piastrella (x, y)* a *piastrella (x, y)* è lunga 0 chiaramente.

Questo è il motivo della scelta di una semplice bfs adattata invece che algoritmi specifici per i cammini minimi come *dijkstra*. *Dijkstra*, pur essendo un algoritmo con l'esatto scopo di trovare il cammino minimo, (pista di lunghezza minima tra due piastrelle), richiede una coda con priorità e ha complessità temporale peggiore maggiore, perchè viene usato quando il grafo è pesato e gli archi hanno pesi differenti.

Utilizzando una *bfs* la complessità temporale e anche quella spaziale nel caso peggiore è la stessa di *bloccoBfs* spiegata sopra.

il tempo ammortizzato è ***O(n) ,*** *n = #piastrelle del blocco*

Mentre **il caso peggiore molto raro**, *n = #piastrelle del blocco = m = piastrelle accese sul piano,* ***$O(m)$ * $O(2m)$ = $O(m^2)$*****.**
La **complessità spaziale** nel caso peggiore è ***$O(m)$*** cioè quando il *blocco* è delle dimensioni dell'intero *piano*.