

# Relazione Progetto Algoritmi (Piastrille Digitali)

Kevin Manca, matricola 978578

10 Luglio 2024

## Contents

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Problema . . . . .	2
1.1.1	Piano e Piastrille . . . . .	2
1.1.2	Blocco . . . . .	2
1.1.3	Propagazione del colore . . . . .	2
<b>2</b>	<b>Modellazione</b>	<b>3</b>
<b>3</b>	<b>Strutture Dati</b>	<b>3</b>
<b>4</b>	<b>Funzioni</b>	<b>4</b>
4.1	colora . . . . .	4
4.2	spegni . . . . .	5
4.3	regola . . . . .	5
4.4	stato . . . . .	5
4.5	stampa . . . . .	5
4.6	blocco e bloccoOmog . . . . .	6
4.7	propaga . . . . .	6
4.8	propagaBlocco . . . . .	6
4.9	ordina . . . . .	6
4.10	pista . . . . .	6
4.11	lung . . . . .	7
<b>5</b>	<b>Funzioni di utilità</b>	<b>7</b>
5.1	calcolaDeltaVertice . . . . .	7
5.2	dfs . . . . .	7
5.3	piastrelleCirconvicine . . . . .	8
5.4	verificaRegola . . . . .	8
5.5	calcolaPista . . . . .	9
5.6	calcolaPistaBreve . . . . .	9
<b>6</b>	<b>Requisiti e Testing</b>	<b>10</b>
6.1	Test . . . . .	10

## 1 Introduzione

Questo documento contiene una panoramica e la documentazione relativa alle strutture dati, gli algoritmi scelti (con i costi a loro associati), come è stato modellato il problema indicato

nella specifica, oltre ad opportune scelte implementative.

Insieme a questo documento, sono presenti all'interno dell'archivio `978578_manca_kevin.zip` altri file richiesti dalla specifica:

- File `go: 978578_manca_kevin.go` contenente il codice sorgente
- File di test `go: 978578_manca_kevin_test.go` che contiene le funzioni utili a testare il programma
- Cartella di test: `test/` che contiene ulteriori sottocartelle con i file utilizzati per il testing del programma
- File `go.mod`: utilizzato principalmente per definire modulo e versione di go e per l'utilizzo di `go test` da linea di comando

## 1.1 Problema

Il problema imposto dalla specifica è quello di studiare le configurazioni di insiemi di piastrelle digitali su un piano, e la loro influenza sulle piastrelle circonvicine.

In questa sezione viene fatta una panoramica relativa alla specifica ed alcuni concetti presenti in questo documento e come sono stati modellati per una possibile soluzione al problema.

### 1.1.1 Piano e Piastrelle

Il **piano** è suddiviso in quadrati di lato unitario, ciascuno dei quali è occupato da una **piastrella**, la quale può essere *accesa* o *spenta*. La *piastrella(a, b)* rappresenta la piastrella di lato unitario con vertici  $(a, b)$ ,  $(a + 1, b)$ ,  $(a + 1, b + 1)$ ,  $(a, b + 1)$ .

Ogni piastrella è inoltre caratterizzata da un *colore*, quando una piastrella è accesa appare colorata.

Il colore è l'insieme dei colori disponibili sulle stringhe dell'alfabeto.

Ogni piastrella può inoltre avere un'intensità differente (indicata con un *numero intero*).

Due piastrelle sono dette **circonvicine** se hanno in comune almeno un vertice.

### 1.1.2 Blocco

Il **blocco** è una regione massimale di *piastrelle accese*. Il blocco di appartenenza della *Piastrella(x, y)* è il blocco che la contiene. Definiamo inoltre un **blocco omogeneo** come un *blocco* nel quale tutte le piastrelle accese hanno il medesimo colore.

### 1.1.3 Propagazione del colore

L'intorno della piastrella  $p$  è l'insieme di tutte le piastrelle *circonvicine* ad  $p$  e differenti da  $p$ . Possiamo quindi definire *regole di propagazione* del tipo  $k_1\alpha_1 + k_2\alpha_2 + \dots + k_n\alpha_n \rightarrow \beta$  dove  $n$  è un intero,  $\alpha_1, \dots, \alpha_n$  sono stringhe sull'alfabeto  $a, b, \dots, z$  tutte differenti tra loro,  $\beta$  è una stringa sull'alfabeto  $a, b, \dots, z$ ,  $k_i$  sono interi positivi la cui somma non supera 8. Una regola è applicabile a una piastrella  $p$  se il suo intorno contiene almeno  $k_i$  piastrelle di colore  $\alpha_i, \forall i \in \{1, \dots, n\}$ .

L'applicazione della regola cambia in  $\beta$  il colore della piastrella  $p$ . Le regole di propagazione vengono memorizzate in ordine cronologico di inserimento (*FIFO*).

È possibile inoltre *propagare i colori* su un blocco: per ogni piastrella colorata  $p$  che appartiene al blocco, si applica la prima regola applicabile su  $p$  rispetto all'intorno di  $p$  come risulta prima dell'inizio della procedura di propagazione sul blocco.

Durante l'esecuzione definiamo *consumo* di una regola la quantità determinata dal numero totale di piastrelle a cui la regola è stata applicata dal momento della sua definizione.

## 2 Modellazione

### Generale e blocchi

Il problema è stato modellato come un *grafo* non orientato; indicando il grafo come  $G = (V, E)$  dove:

- $V$ : l'insieme dei vertici, ovvero una piastrella
- $E$ : insieme degli archi che in questo caso rappresentano il punto in comune tra le piastrelle

Gli *archi* non sono orientati, in quanto è possibile partire da qualsiasi vertice per definire un blocco, e non sono pesati.

I blocchi e blocchi omogenei comportano una visita nel *grafo*, per identificare le piastrelle circonvicine e quindi quelle contenute nel blocco, ma anche per verificarne l'intensità e il loro colore (blocco omogeneo).

Una possibile applicazione di questa visita, che poi è quella utilizzata nel progetto, è quella della **Depth First Search** (o ricerca in profondità) che fissa un vertice origine (ovvero una piastrella), segna come visitato tale vertice, esplora i vertici vicini non visitati e continua in modo ricorsivo fino a quando non trova un vertice che non ne ha altri non visitati.

### Propagazione

Relativamente alla *propagazione di colore* da una *piastrella*  $(x, y)$ , è necessario prima verificare la validità delle regole (individualmente), andando quindi a calcolare i valori delle piastrelle circonvicine a quella di partenza e compararli con la regola.

Una regola sarà valida e potrà essere applicata se l'intorno di piastrelle circonvicine è sufficiente per eseguirla.

Mentre per la *propagazione sul blocco*, è necessario iterare su ogni singola piastrella appartenente al blocco e applicare ad essa la prima *regola valida*.

### Pista

La modellazione scelta per la pista è quella di un *cammino in un grafo*, del quale si è a conoscenza del punto di origine (*piastrella* $(x_1, y_1)$ ) e di arrivo (*piastrella* $(x_2, y_2)$ ).

La *pista più breve* invece può essere rappresentata come il *cammino minimo nel grafo* tra due vertici (*piastrelle*).

Trattandosi di un grafo non pesato, la soluzione ideata è quella di applicare la **Breadth First Search** (o ricerca in ampiezza), che differentemente dalla *DFS* esplora i vertici vicini (adiacenti) prima di continuare l'esplorazione in quelli più lontani, in modo da garantire che il percorso risultante sia quello più breve.

### Lunghezza

La *lunghezza* di una *pista* richiesta dalla specifica, rappresenta la distanza tra i vertici e può essere definita (con la modellazione data) come il numero di archi della pista +1, quindi il valore del *cammino* +1.

## 3 Strutture Dati

Di seguito sono riportate le *strutture dati* utilizzate e le relative stime dei costi spaziali. Sono presenti inoltre, nel codice sorgente, ulteriori commenti utili a chiarire alcuni aspetti e scelte adottate.

## Piano

La struttura **Piano** (modellato come grafo) rappresenta un piano costituito da piastrelle e regole. Utilizzo una mappa (`map[punto]*Piastrrella`) per memorizzare le piastrelle e una slice (`*[]Regola`) per le regole. Memorizzo solo le piastrelle *accese*, ciò comporta che quelle *spente* risultano inesistenti e vengono aggiunte alla mappa esclusivamente se vengono accese.

- **piastrelle:**  $O(n)$ ,  $n$  numero di piastrelle
- **regole:**  $O(m)$ ,  $m$  numero di regole

**Spazio totale:**  $O(n) + O(m)$

L'implementazione scelta è dovuta ad alcune considerazioni in merito ad altre rappresentazioni possibili. Nello specifico:

- *matrice di incidenza/adiacenza:* essendo il piano potenzialmente infinito (da specifica), lo spazio occuperebbe  $O(n^2)$  con  $n$  numero di vertici
- *lista di incidenza/adiacenza:* l'operazione di inserimento (eseguita spesso) avrebbe comportato l'utilizzo di `append()` e quindi un tempo pari a  $O(1^*)$

I vantaggi quindi dell'implementazione utilizzata sono una *maggiore efficienza di spazio e tempo*, infatti attraverso le mappe utilizzate le operazioni di *inserimento*, *accesso* e *modifica* sono in tempo costante.

## Piastrrella

Rappresenta una singola **piastrella** con coordinate ( $(x \text{ int}, y \text{ int})$ ), colore (*string*) e intensità (*int*).

Le coordinate della piastrella indicano il *vertice di origine* (in basso a sinistra), il colore è semplicemente la stringa sull'alfabeto e l'intensità è un numero intero  $\geq 0$  (0 indica che la piastrella è *spenta*).

**Spazio totale:**  $O(1)$  per ogni piastrella ( $3 \text{ interi} + 1 \text{ stringa}$ )

## Punto

Rappresenta un punto con coordinate ( $(x \text{ int}, y \text{ int})$ ), ovvero un vertice del grafo.

**Spazio totale:**  $O(1)$  per ogni *punto* (tipo *intero*)

## Regola

Rappresenta una **regola** di propagazione con una stringa completa (*istruzioneCompleta string*), un colore (*string*), un consumo (*uint*) e una mappa dei valori di colore (*map[string]int*). Rispetto alla specifica ho scelto di aggiungere la mappa dei valori (*valColore*) che mi risulta utile nel verificare ad esempio la possibilità di applicare una regola.

**Spazio totale:**  $O(k)$ ,  $k$  numero di colori nella mappa **valColore**

## 4 Funzioni

### 4.1 colora

Colora una piastrella specifica identificata dalle coordinate  $(x, y)$  con un dato colore (*alpha*) e intensità (*i*).

**Tempo:**  $O(1)$  per aggiungere/aggiornare una piastrella nella mappa

**Spazio:**  $O(1)$ , aggiunge una singola piastrella alla mappa

## 4.2 spegni

Spegne una piastrella, impostando la sua intensità a zero.

**Tempo:**  $O(1)$  per aggiornare l'intensità della piastrella

**Spazio:**  $O(1)$ , non aggiunge nuove piastrelle

## 4.3 regola

Aggiunge una nuova regola ( $r$ ) di propagazione al piano. Questa operazione viene fatta ottenendo prima le istruzioni della regola, iterando su di esse e successivamente aggiungendo la regola alla slice.

**Tempo:**  $O(n)$ ,  $n$  lunghezza della stringa della regola  $r$

- $O(n)$ , *strings.Fields*( $r$ )
- $O(1)$ , creazione della mappa dei valori
- $O(t)$ , iterazione sulle istruzioni con  $t$  pari al numero di istruzioni
- $O(1^*)$ , append della regola alla slice di regole in tempo ammortizzato (se nel caso peggiore la slice debba essere ridimensionata)

**Spazio:**  $O(t)$ ,  $t$  numero di istruzioni della regola

## 4.4 stato

Stampa e restituisce il colore e l'intensità di una piastrella identificata dalle coordinate  $(x, y)$ .

Se la piastrella è *spenta*, non stampa nulla e restituisce la stringa vuota e l'intero 0.

**Tempo:**  $O(1)$ , accesso diretto alla mappa

**Spazio:**  $O(1)$ , non modifica/utilizza strutture dati aggiuntive

## 4.5 stampa

Stampa tutte le regole di propagazione, nell'ordine attuale.

Viene effettuata un iterazione sulle regole del *piano* ed effettuo poi una divisione della stringa *istruzioneCompleta* che rappresenta l'intera stringa della regola.

**Tempo:**  $O(n \cdot k_{max})$ ,  $n$  numero di regole e  $k_{max}$  lunghezza massima della stringa *istruzioneCompleta* tra tutte le regole

- $O(n)$ , iterazione su  $n$  numero di regole
- $O(k_i)$ , complessità di *strings.SplitN* con  $k_i$  lunghezza della stringa completa della regola  $i$ -esima
- $O(1)$ , per ogni regola passata a *fmt.Printf*

**Spazio:**  $O(1)$ , non alloco strutture dati aggiuntive (che crescerebbero con l'input)

## 4.6 blocco e bloccoOmog

Calcola la somma delle intensità delle piastrelle in un *blocco omogeneo* o non omogeneo, utilizzando la dfs ed gestisce sia il caso di *blocco* che di *blocco omogeneo*.

Viene preventivamente verificato che le coordinate passate corrispondano ad una *piastrella accesa* e in caso affermativo si inizializza l'intensità ad valore della piastrella attuale.

**Tempo:**  $O(n + m)$ , la *DFS*  $O(n + m)$  con  $n$  numero di piastrelle nel blocco e  $m$  numero di archi; accesso alla mappa e stampa in tempo costante  $O(1)$

**Spazio:**  $O(n)$  per la mappa delle *visite* nel caso peggiore di  $n$  piastrelle

## 4.7 propaga

Applica la prima regola valida a una piastrella specifica, se nessuna regola valida viene trovata (la funzione di utilità "verificaRegola" restituisce *nil*) non fa niente, altrimenti viene effettuata la funzione "colora" e viene incrementato il *consumo* di tale *regola*.

**Tempo:**  $O(R \cdot C)$  nel caso peggiore, dove  $R$  è il numero di regole e  $C$  è il numero di colori per regola **Spazio:**  $O(1)$ , non vengono utilizzate strutture dati aggiuntive, ma quelle già esistenti (quindi una quantità di memoria costante)

## 4.8 propagaBlocco

Propaga il colore su un blocco di piastrelle dove appartiene la piastrella di coordinate  $(x, y)$  passate come parametro.

Utilizzo la dfs ( $O(n + m)$ ) per ottenere il *blocco* di appartenenza della piastrella  $(x, y)$ , itero sulle piastrelle del blocco ( $O(v)$ ) e all'interno dell'iterazione, per ogni piastrella, verifico la validità della regola ( $O(r)$ )

**Tempo:**  $O(n) + O(m \cdot r)$ ,  $n$  è il numero di piastrelle nel blocco,  $m$  è il numero di regole, e  $r$  è il numero di regole; essendo un piano, si può semplificare  $n \approx m$ , perciò  $O(n + n \cdot r)$

**Spazio:**  $O(n)$  nel caso peggiore tutte le strutture dati di memorizzazione hanno  $O(n)$ , sia per la mappa *visite*, la *DFS*, che lo *stack di ricorsione*

## 4.9 ordina

Ordina le *regole di propagazione* (specificate nella sezione "Regola") per *consumo*.

Quando due regole hanno lo stesso *consumo* mantengo il loro ordine precedente rispetto all'applicazione della funzione *ordina*.

**Tempo:**  $O(n \log n)$ ,  $n$  numero di regole, dovuto all'algoritmo di ordinamento *sliceStable* (si tratta di una versione modificata di *mergeSort*)

**Spazio:**  $O(1)$ , poiché l'ordinamento avviene in loco, sulla slice di regole da applicare, senza utilizzare spazio aggiuntivo oltre alla memoria già allocata per le *regole* stesse

## 4.10 pista

Stampa la *pista* (un cammino) seguendo una *sequenza di direzioni* a partire da una piastrella  $(x, y)$ , solo se tale pista è definita, altrimenti non stampa niente.

Internamente chiama la funzione "calcolaPista" (descritta nell'apposita sezione) che segue la *sequenza di direzioni* e aggiorna la *stringa da stampare* se trova le piastrelle.

Se la stringa risultante è vuota, significa che nessuna pista è stata trovata e non viene stampato nulla, altrimenti viene stampata la sequenza delle piastrelle, con le relative informazioni come da specifica.

**Tempo:**  $O(k)$ ,  $k$  lunghezza della *sequenza di direzioni*, stesso costo per la funzione *strings.Split* di  $k$  numero di direzioni e la funzione *calcolaPista*

**Spazio:**  $O(k)$ ,  $k$  lunghezza della *sequenza di direzioni* (è richiesto lo spazio per la memorizzazione della stringa dipendente appunto da esse)

## 4.11 lung

Calcola la lunghezza della pista più breve tra due piastrelle usando la *BFS* (*calcolaPistaBreve*), in pratica, cerca il *cammino minimo* tra due vertici e ne stampa la *lunghezza* (se definito). La complessità spaziale e temporale quindi di questa funzione sono strettamente influenzate dalla funzione *calcolaPistaBreve* (e quindi la *BFS*).

**Tempo:**  $O(V + E)$ , con  $V$  numero di piastrelle del piano da esplorare,  $E$  numero di archi (vertici in comune delle piastrelle), pari quindi alla *BFS*

**Spazio:**  $O(n)$  per le strutture dati di *BFS*, con  $n$  numero di vertici

## 5 Funzioni di utilità

In questa sezione sono presenti le *funzioni* utili alle altre Funzioni principali che permettono di evitare la ridondanza di codice e di raggruppare comportamenti condivisi richiesti.

### 5.1 calcolaDeltaVertice

Calcola e restituisce un *punto* dati in input il *punto di origine* e lo spostamento sulla coordinata  $x$  (*deltaX int*) e  $y$  (*deltaY int*).

**Tempo:**  $O(1)$ , operazione aritmetica basilare in tempo costante

**Spazio:**  $O(1)$ , restituisce un *punto* che come descritto precedentemente è una struttura composta da due interi ( $x$ ,  $y$ )

### 5.2 dfs

La visita in profondità (*DFS*) viene usata per le funzioni *blocco* e *bloccoOmog* e *propagaBlocco* per ottenere le piastrelle di un blocco partendo da una piastrella indicata dal vertice (*punto* di coordinate  $x$ ,  $y$ ).

La ricerca procederà ricorsivamente se la *piastrella* (vertice) attuale non è stata ancora visitata ed è *accesa*.

Ho definito questa funzione utilizzando ulteriori parametri per renderla versatile per ulteriori calcoli.

Sono presenti infatti un *booleano omogeneo*, una *mappa blocco* ed un *puntatore ad intero sum*.

- **omogeneo bool:** *true* per calcolare appunto le piastrelle circonvicine nel caso di un *blocco omogeneo*, *false* altrimenti
- **blocco map[punto]\*Piastrella:** conterrà un puntatore alla mappa (le mappe in Go sono passate per riferimento) nel caso di *blocco omogeneo*, altrimenti sarà *nil* e non verrà modificato
- **sum \*int:** nel caso di *blocco/blocco omogeneo*, conterrà l'indirizzo di un intero inizializzato nella funzione chiamante (*blocco()*), questo per sommare il valore dell'intensità delle piastrelle in loco ed evitare di restituire valori; sarà *nil* se non è richiesta la somma come nella funzione *propagaBlocco*

**Tempo:**  $O(n + m)$ ,  $n$  numero di piastrelle nel blocco,  $m$  numero di archi

- $O(1)$ : aggiunta vertice alla mappa
- $O(1)$ : per iterazione sulle direzioni possibili (8 nel caso del progetto)
- $O(1)$ : *calcolaDeltaVertice* come riportato in precedenza
- $O(1)$ : verifica delle condizioni e ricorsione

**Spazio:**  $O(n)$  per la mappa delle visite,  $n$  numero di vertici visitati

- $O(n)$ : memorizzazione del booleano per ogni piastrella (vertice)
- $O(n)$ : caso peggiore, la mappa può richiedere  $O(n)$  piastrelle
- $O(n)$ : stack della chiamata ricorsiva, pari al numero di piastrelle  $n$
- $O(1)$ : spazio della variabile *sum*

### 5.3 piastrelleCirconvicine

Restituisce le piastrelle circonvicine (definite nella sezione "Piano e Piastrelle") come una mappa (*map[punto]\*Piastrella*), partendo da un vertice (*punto*) di origine, seguendo le 8 direzioni possibili, e se viene passato anche il 3° parametro (*colori map[string]int*), cioè diverso da *nil* aggiorna la mappa dei colori delle piastrelle circostanti.

**Tempo:**  $O(1)$ , *for-range* esegue 8 iterazioni (sono 8 le direzioni), mantenendo costante l'accesso alla mappa ed eventualmente aggiungendo un elemento a quest'ultima (in  $O(1)$ ), e la stessa cosa avviene per la mappa colori se deve essere aggiornata

**Spazio:**  $O(1)$ , la mappa *vicine* contiene al massimo 8 elementi, mentre quella *colori* non richiede ulteriore spazio in quanto passata per riferimento

### 5.4 verificaRegola

Restituisce la *prima regola* valida (applicabile) su una piastrella e ne aggiorna il *consumo*, in caso nessuna regola può essere applicata non vengono effettuate altre operazioni.

Nel dettaglio, verifico le piastrelle circonvicine e i loro colori e successivamente controllo se la regola che sto valutando può essere applicata, cioè se il valore dei colori dell'intorno è sufficiente al valore dei colori della regola.

**Tempo:**  $O(c)$ ,  $c$  pari al numero di colori distinti, che essendo limitato cresce in modo lineare rispetto al suo numero

- $O(1)$ : creazione mappa *valoriColore*
- $O(1)$ : funzione *piastrelleCirconvicine* come precedentemente indicato
- $O(c)$ : ciclo di iterazione sui colori indicati dalla regola, numero massimo di colori pari a  $c$

**Spazio:**  $O(1)$ , la mappa *valoriColore* contiene al massimo 8 colori (pari a tutte le piastrelle circonvicine di colore diverso) garantendo un tempo trascurabile e costante



## 5.5 calcolaPista

Calcola la *pista* (definita nella sezione "Pista"), seguendo la sequenza di direzioni passate come parametro ed aggiorna la *stringa da stampare* (come richiesto da specifica) se tale pista è definita. Se la pista non esiste, non restituisce nulla e la stringa da stampare sarà vuota. Inizializzo la *stringa* inserendo le coordinate (date dal *punto*) della piastrella di origine e seguo la sequenza di direzioni, se trovo una *piastrella spenta* (e quindi non esistente) mi fermo e resetto la stringa da stampare.

**Tempo:**  $O(n)$ ,  $n$  numero di direzioni specificate (sulle quali iterare)

- $O(n)$ : iterazione sulle  $n$  direzioni
- $O(1)$ : funzione *calcolaDeltaVertice* come precedentemente indicato per ogni nuova piastrella in tempo costante

**Spazio:**  $O(n)$ , dovuto alle variabili locali utilizzate per le piastrelle visitate e la stringa di output che verrà stampata; questo quindi è pari al numero  $n$  di piastrelle visitate durante il calcolo

## 5.6 calcolaPistaBreve

Calcola la *pista più breve* (definita nella sezione "Pista"), cioè il cammino minimo tra due vertici  $(x_1, y_1)$  e  $(x_2, y_2)$  dati e ne restituisce la *lunghezza*.

Viene utilizzata, come indicato nella sezione "Pista", una ricerca in ampiezza (*BFS*) trattando il problema come grafo non pesato.

La funzione *calcolaPistaBreve*, compresa la *BFS* è stata implementata seguendo questi passaggi:

- $O(1)$ : verifica che la piastrella di *origine* e *arrivo* siano *accese* altrimenti si ferma la computazione
- $O(1)$ : inizializzazione della *mappa visitate* e *coda per i vertici*
- $O(1)$ : segno nella mappa la piastrella di origine come visitata
- $O(V + E)$ : fin quando la *coda* non è *vuota*, algoritmo *BFS* ( $V$  vertici,  $E$  archi)
  - estraggo il primo vertice della coda
  - se il vertice estratto è quello di arrivo, fermo la computazione e restituisco la *lunghezza*
  - ottengo le piastrelle circoscrutte al vertice corrente
  - per ogni piastrella adiacente, non ancora visitata, la aggiungo alla coda e segno come visitata
  - incremento alla fine la *lunghezza*
  - se la coda è vuota e non ho trovato il vertice, non esiste una pista e la lunghezza è 0 come da specifica

**Tempo:**  $O(V + E)$ , con  $V$  numero di piastrelle del piano da esplorare,  $E$  numero di archi (vertici in comune delle piastrelle)

- $O(1)$ : Inizializzo le strutture dati e inserisco la *piastrella d'origine* nella coda
- $O(1)$ : verifico che la piastrella attuale sia quella di *arrivo* ed in quel caso

- $O(V + E)$ : ciclo della *BFS*, quindi per ogni *piastrella circconvicina*

La complessità temporale di questa funzione è influenzata quindi dalla **BFS**.

**Spazio:**  $O(V)$ , vertici  $V$  memorizzati (piastrelle) nelle mappe, proporzionalmente al loro numero; le variabili locali utilizzate hanno un costo di spazio pari a  $O(1)$

- $O(V)$ : la *coda* contiene al massimo  $V$  vertici
- $O(V)$ : la mappa *visitate* tiene traccia di  $V$  vertici visitati (piastrelle)

## 6 Requisiti e Testing

Per testare ed utilizzare il programma contenuto all'interno di questo archivio sono sufficienti una versione di **go** recente presente nel  $\$PATH$  ed una *shell* dove eseguire alcuni semplici comandi. Se questi requisiti sono soddisfatti, dopo essersi posizionati nella cartella principale (978578\_manca\_kevin) basta eseguire il comando *go build* che assembla il codice sorgente presente nel file *go* che contiene la funzione *main*, generando un eseguibile, e successivamente eseguire *go test* che eseguirà tutte le funzioni presenti nel file **\*\_test.go**.

Un altro possibile metodo è quello di utilizzare l'eseguibile generato dal comando *go build* e reindirizzarli dei file di input o usare il pipe per ottenere il risultato. Ecco alcuni esempi con i *placeholder* da sostituire

Listing 1: Esempio comandi shell

```
cd "978578_manca_kevin"
go build
# Utilizzando il file eseguibile (passando delle stringhe separate da '\n')
echo "STRINGA_COMANDO" | ./978578_manca_kevin
# Utilizzando il file eseguibile e reindirizzando l'input dei file da testare
./978578_manca_kevin < FILE_INPUT
# oppure
echo "STRINGA_COMANDO" | go run ./978578_manca_kevin.go
```

### 6.1 Test

Come indicato nella sezione "Introduzione", sono presenti all'interno della cartella **test/** ulteriori cartelle con i file di *input* e quelli di *output* che ci si aspetta (expected).

Tali file mostrano il comportamento del programma relativo a particolarità della specifica, e ciascuno ha il formato **BaseNomeFunzioni**.

Ecco alcuni esempi:

- Esempio input *blocco* e *blocco omogeneo* e il loro output
 

C 1 1 rosso 2	9
C 1 2 rosso 2	4
C 1 0 verde 5	4
b 1 1	
B 1 1	
B 1 0	
q	

Questo esempio mostra come il calcolo di *blocco* e *blocco omogeneo* applicato nella stessa piastrella(1, 1) restituisce nel primo caso l'intensità pari a quella totale delle piastrelle inizializzate è pari a  $2 + 2 + 5 = 9$ , mentre nel secondo caso  $2 + 2 = 4$  in quanto la piastrella (1, 0) non ha lo stesso colore della piastrella sulla quale viene

chiamato il blocco.

- Esempio input *propaga* e *propagaBlocco* e il loro output

C 2 2 g 3	(
C 2 3 g 2	v: 1 g 1 r
C 2 1 r 5	m: 2 g
r v 1 g 1 r	)
r m 2 g	
p 2 2	
p 2 3	
P 2 1	
s	

Con le piastrelle di coordinate (2, 2), (2, 3), (2, 0), e le regole definite questo esempio mostra come la propagazione del colore tramite *propaga* e *propagaBlocco*.

La propagazione sulla piastrella (2, 2) comporta che la prima regola venga applicata (*r v 1 g 1 r*) ricolorando la piastrella di v.

La propagazione sulla piastrella (2, 1) comporta che la seconda regola venga applicata (*r e 2 g*) ricolorando la piastrella di m.

Effettuando infine *propagaBlocco* sulla piastrella (2, 1) le piastrelle risultano:

- Piastrella(2, 2): v 3
- Piastrella(2, 3): g 2
- Piastrella(2, 1): r 5

- Esempio input *pista* e *lung* e il loro output

C 3 0 r 8	[
C 2 1 h 4	3 0 r 8
C 2 2 i 8	2 1 h 4
t 3 0 NO,NN	2 2 i 8
L 3 0 2 1	]
q	2

Con le piastrelle di coordinate (3, 0), (2, 1), (2, 2), viene mostrato nell'esempio la *pista* risultante e la sua *lunghezza*.

La *pista*, seguendo le *direzioni* *NO,NN* partendo dalla piastrella(3, 0) risulta appunto (3, 0) → (2, 1) → (2, 2).

Mentre la *lunghezza* partendo dalla piastrella(3, 0) e arrivando alla piastrella((2, 1) risulta pari a 2.