```
[elia@arch-elia ~/github/algo_exam/]$ ./tiles
[elia@arch-elia ~/github/algo_exam/]$ C 0 0 elia_cortesi_01911A 1
[elia@arch-elia ~/github/algo_exam/]$ C 0 1 tilesProject 1
[elia@arch-elia ~/github/algo_exam/]$ ? 0 1
```

tilesProject by Elia Cortesi 01911A

Indice:

1 – Introduzione

1.1 – Struttura del progetto

```
tiles/
├── tiles.go
├── unionFind.go
├── esegui.go
├── utils.go
├── colora.go
├── stato.go
├── spegni.go
├── regola.go
├── stampa.go
├── blocco.go
├── bloccoOmog.go
├── propaga.go
├── propagaBlocco.go
├── ordina.go
├── insertionSort.go
├── mergeSort.go
├── test/
│   ├── input/
│   │   ├── colora_stato.txt
│   │   ├── regola_stampa.txt
│   │   ├── blocco_intensity.txt
│   │   ├── propaga_regole.txt
│   │   └── spegni.txt
│   ├── output/
│   │   ├── colora_stato.txt
│   │   ├── regola_stampa.txt
│   │   ├── blocco_intensity.txt
│   │   ├── propaga_regole.txt
│   │   └── spegni.txt
```

I file all'interno della directory `tiles/test/input` sono dei file in formato `.txt` che servono a simulare una serie di comandi per il programma.
I file all'interno della directory `tiles/test/output` sono dei file in formato `.txt` che contengono l'output atteso dei corrispettivi file di input.
I file all'interno della directory `tiles/` sono tutti i file in formato `.go` che compongono il programma.
Per avviare il programma correttamente eseguire il seguente comando all'interno della directory principale del progetto:

```
[elia@arch-elia ~/github/algo_exam/]$ go run .
```

e poi inserire manualmente a piacimento i comandi relativi al programma elencati alla fine di questa sezione.
Se si volesse utilizzare un input presente nella cartella `tiles/test/input` eseguire i seguenti comandi all'interno della directory principale del progetto:

```
[elia@arch-elia ~/github/algo_exam/]$ go build .
[elia@arch-elia ~/github/algo_exam/]$ ./tiles < test/input/input_a_piacere.txt
```

Se invece si volesse usare un input personale spostare il file `.txt` nella cartella `tiles/test/input` ed eseguire i comandi sopracitati con il nome del vostro file.

Lista dei comandi:
`C x y` α `i`: colora la piastrella di coordinate (x,y) di colore a con intensità i
`S x y`: spegne la piastrella di coordinate (x,y)
`r` β k_1 α_1 `...` k_n α_n: inserisce la regola definita nel comando secondo il formato specificato nella traccia all'interno dell'elenco delle regole
`? x y`: stampa le informazioni della piastrella di coordinate (x,y) secondo il formato specificato nella traccia
`s`: stampa l'elenco delle regole secondo il formato specificato nella traccia
`b x y`: stampa la somma delle intensità delle piastrelle appartenenti al blocco in cui è presente la piastrella (x,y)
`B x y`: stampa la somma delle intensità delle piastrelle, appartenenti al blocco in cui è presente la piastrella (x,y), che sono dello stesso colore della piastrella (x,y)
`p x y`: cambia il colore della piastrella (x,y) in quello definito dalla prima regola applicabile dall'elenco
`P x y`: cambia il colore di tutte le piastrella appartenenti al blocco in cui è presente la piastrella (x,y) in quello definito dalla prima regola applicabile dall'elenco a ciascuna piastrella.
`o`: ordina l'elenco delle regole in base al numero di utilizzi di ciascuna regola in modo crescente
`q`: termina l'esecuzione del programma

1.2 – Descrizione del problema

Il problema proposto è una simulazione di una griglia di piastrelle quadrate digitali che si possono accendere, colorare e spegnere a proprio piacimento. Ogni piastrella è identificata da una coppia di numeri naturali, che corrispondono al vertice posto in basso a sinistra di essa. Quindi, i quattro vertici di una piastrella corrispondono alle coordinate (partendo dal basso in senso orario) (x,y), (x,y+1), (x+1,y+1) e (x+1,y). Due piastrelle possono essere tra loro circonvicine, ovvero se condividono almeno un vertice: ne consegue che ogni piastrella può avere 8 possibili piastrelle circonvicine. Ogni piastrella si può colorare con un colore e un'intensità di colore. Più piastrelle circonvicine formano un blocco di piastrelle, e se quelle piastrelle sono dello stesso colore allora il blocco si dice omogeneo. In questa simulazione è presente anche un elenco di regole: ogni regola è formata da un colore e da un elenco di requisiti. I requisiti sono nel formato `numero colore`, e se l'intorno di una piastrella (ovvero l'insieme delle piastrelle circonvicine a una piastrella) soddisfa i requisiti della regola, allora la piastrella in questione viene colorata del colore specificato nella regola. Ulteriori informazioni e specifiche tecniche possono essere trovate all'interno del file `tiles/traccia.pdf`.

2 – Modellazione e implementazione

2.1 – Modellazione del problema

Analizzando attentamente la traccia fornita e le varie istruzioni che il programma deve elaborare, emergono alcune considerazioni importanti. Innanzitutto, è necessaria una struttura dati efficiente per gestire sia i blocchi sia le relative piastrelle. Dato che i blocchi sono *insiemi* di piastrelle, viene subito in mente la struttura dati Union-Find, che mantiene la tracciabilità di un insieme di elementi suddivisi in una collezione di insiemi disgiunti. In parole semplici, possiamo considerare una piastrella come un insieme formato solo dalla piastrella stessa. Ogni volta che si aggiunge al sistema una piastrella in una posizione circonvicina a una piastrella già esistente, si uniscono i due insiemi per crearne uno unico formato da entrambe le piastrelle. In questo modo, si può tenere traccia delle piastrelle circonvicine in modo efficiente. Sebbene non abbia esplicitato questa struttura dati nel programma, ho insistito il più possibile sul concetto di piastrella piuttosto che su quello di blocco. Ulteriori dettagli sull'implementazione sono forniti nella sezione 2.2.2.

Inoltre, è presente anche il concetto di regola, descritto nella sezione 1.2. Parlando delle regole, mi collego a un'operazione che il programma deve essere in grado di eseguire: `ordina`. Questa operazione deve riordinare in modo crescente, in base al consumo delle regole (ovvero quante volte sono state applicate al sistema), l'elenco delle regole. L'idea di "riordinare" fa subito pensare a un algoritmo di sorting. E quale migliore struttura dati per applicare un algoritmo di sorting se non un array? Una particolarità di questo algoritmo di sorting deve essere la sua stabilità. Infatti, nella traccia è richiesto che, in caso di consumo uguale tra due regole, venga mantenuto l'ordine relativo tra di esse. Ulteriori dettagli sono forniti nella sezione 3.5.

2.2 – Implementazione delle strutture dati

2.2.1 – Implementazione di gioco

La struttura dati `piano` rappresenta il sistema nella sua interezza. L'ho implementata con una struct contenente due campi. Il primo campo, `tiles`, consiste in una mappa con come chiave una coppia di coordinate di numeri interi (definite dal tipo `piastrella`) e come valore un puntatore a una struct `properties`. Ho deciso di usare un puntatore perché, nonostante in Golang le slices, le mappe e i channels vengano passati per referenza, durante lo sviluppo ho riscontrato dei problemi relativi alla visibilità delle modifiche sulle strutture originali. La struct `properties` è una rappresentazione molto nascosta della struttura dati Union-Find (ulteriori dettagli nella sezione 2.2.2) e al suo interno ci sono vari campi corrispondenti alle caratteristiche di una singola piastrella: colore, intensità del colore, genitore, rango e intensità di colore del blocco. Come intuibile, le ultime tre caratteristiche sono relative solo alla *radice* del blocco, ovvero la prima piastrella inserita nel sistema che non ha piastrelle accese nel suo intorno al momento della creazione (ulteriori dettagli nella sezione 2.2.2). Il secondo campo di piano è `rules`, un puntatore a un array di regole, definite dalle struct `rule` e `ruleset`. In questo caso, ho dovuto usare un puntatore a un array, invece che un array, per questioni di riferimenti (ulteriori dettagli nella sezione 2.2.3).

2.2.2 – Implementazione di Union-Find

Come detto nella sezione 2.2.1 ogni blocco ha una *radice*, ma questo termine è usato principalmente in ambito di strutture dati ad albero. Effettivamente, le Union-Find sono rappresentate da alberi (implementati a loro volta in vari modi) e si può associare ogni insieme della Union-Find ad un albero, con una radice, dei figli/genitori e delle foglie. Solitamente, la struttura dati Union-Find ha due principali componenti: un array dei genitori e un array dei ranghi. Queste due componenti permettono alla Union-Find di essere molto efficiente. L'array dei genitori è un array in cui l'indice rappresenta un elemento e il valore rappresenta il genitore di tale elemento nell'albero. Se l'elemento è una radice, il valore è l'indice stesso. L'array dei ranghi è un array che traccia l'altezza degli alberi (o altre metriche).

La Union-Find supporta due operazioni fondamentali: `Find` e `Union`. La `Find` determina in quale insieme si trova un particolare elemento. Questa operazione non solo restituisce il rappresentante dell'insieme contenente l'elemento, ma esegue anche un'ottimizzazione chiamata *path compression* che appiattisce la struttura dell'albero per future operazioni di `Find`. La `Union` unisce due insiemi in un unico insieme. Questa operazione è ottimizzata utilizzando un'altra tecnica chiamata *union by rank* (o *union by size*), che unisce sempre l'albero più piccolo sotto la radice dell'albero più grande per mantenere la struttura più piatta possibile. Queste tecniche di ottimizzazione rendono la struttura Union-Find molto efficiente: la

complessità ammortizzata di queste funzioni eè molto vicina a O(1), mentre lo spazio occupato è O(n), dove n è il numero di elementi del sistema (nel nostro caso, il numero di piastrelle).

Nel programma non ho esplicitato né l'array dei genitori né l'array dei ranghi, ma ho inserito questi valori come caratteristiche di ogni singola piastrella. Ognuna di esse ha un campo `parent` che contiene le coordinate della radice (tranne la radice, che avrà le coordinate di sé stessa). Inoltre, le varie radici hanno il campo `rank` che rappresenta il numero di piastrelle che hanno quella radice come genitore. Infine, ogni radice ha un campo `blockIntensity` che tiene traccia della somma delle intensità delle piastrelle appartenenti a quel blocco (in modo da rendere più immediata l'esecuzione della funzione `blocco` e della `Union` stessa (ulteriori dettagli nella sezione 3.3).

L'utilizzo di questo tipo di implementazione implicita ha diversi vantaggi. Integrare le informazioni della Union-Find direttamente con le proprietà delle piastrelle consente di avere tutte le informazioni rilevanti in un'unica struttura, semplificando l'accesso e la gestione dei dati. Rendere esplicite le relazioni tra le piastrelle e le loro proprietà rende il codice più leggibile e le relazioni più facili da tracciare. Inoltre permette di manipolare facilmente le piastrelle, aggiungendo o rimuovendo elementi dalla mappa (utlie nel caso dello spegnimento e conseguente riorganizzazione).

2.2.3 – Implementazione delle regole

Come già accennato nella sezione 2.2.1 le regole sono memorizzate in un array di `rule`, la struct che ho definito per rappresentare una regola. Essa contiene quattro campi: `raw`, che contiene il comando integrale di inserimento della regola; `color`, che contiene il colore di cui si colorerà la piastrella che soddisferà la regola; `usage`, che tiene traccia del numero di volte che una regola viene applicata; `ruleset`, che contiene l'insieme dei requisiti che una piastrella deve soddisfare per poterci applicare la regola. Il compito di queste struct è di scomporre la regola in più parti rendendo i dati più facili da gestire e utilizzare.

Come citato in fondo alla sezione 2.2.1, ho utilizzato un puntatore ad un array di rule, invece che un normale array, per colpa di `append`. `append` non è una funzione magica ma fa in automatico quello che, in linguaggi come C, bisogna fare manualmente. Ovvero, se si deve aggiungere un record ad un array, ma quell'array è pieno, si crea un nuovo array con la dimensione aggiornata, si copia il contenuto di quello vecchio in quello nuovo, e si elimina quello vecchio. Quindi, è vero che una slice all'interno di una struct viene passata per referenza anche se la struct viene passata per valore, ma dopo che si fa `append` viene eliminata la slice originaria (nel caso in cui la capacità della slice sia piena). Quindi, se non si restituisce nulla, quando si prova ad accedere a quella slice, sarà sempre vuota.

3 – Spiegazione delle operazioni con analisi delle prestazioni

3.1 – Operazioni di Union-Find

Già descritte nella sezione 2.2.2, le operazioni di unionFind sono principalmente due: `Find` e `Union`. `Find` permette di trovare il rappresentante di un blocco a cui appartiene una determinata piastrella e per farlo sfrutta la tecnica di ottimizzazione chiamata *path compression* che permette, ad ogni chiamata di `Find`, di "appiattire" l'albero, ovvero che tutti i nodi sul percorso dalla radice all'elemento diventano direttamente collegati alla radice, rendendo sempre più veloci le ricerche future. `Union` permette di unire due insieme disgiunti in un solo insieme e per farlo sfrutta la tecnica di ottimizzazione chiamata *union by rank* che permette di mantenere l'albero piatto combinando sempre quello più piccolo sotto la radice di quello più grande (in termini di rango). Questo limita la profondità degli alberi e rende più veloci le operazioni successive.

A queste due si aggiungono `makeSet` e `Add`. `makeSet` permette di creare un nuovo sistema di piastrelle e regole: il piano è come se coincidesse con la struttura Union-Find stessa. `add` permette di creare una piastrella e di aggiungerla al sistema. Queste funzioni sono le fondamenta della struttura Union-Find, e vengono chiamate spesso all'interno di altre funzioni, quindi è necessario che siano il più efficienti possibili.

Come già scritto nella sezione 2.2.2, la complessità delle funzioni `Find` e `Union` è molto vicina a O(1). Per la precisione $\theta(\alpha(n))$, dove $\alpha(n)$ è la *funzione inversa di Ackermann,* una funzione che cresce così lentamente che per tutti i valori pratici di n (il numero di elementi nell'Union-Find), $\alpha(n)$ è minore di 5. Questo rende la complessità delle operazioni di `Find` e `Union` quasi costante nella pratica (https://en.wikipedia.org/wiki/Disjoint-set_data_structure). `Add,` nel caso in cui la piastrella

esiste già nel sistema, ha complessità θ(α(n)), mentre se la piastrella non esiste nel sistema ha complessità O(1). `makeSet` ha complessità O(1). La complessità spaziale di tutte le funzioni è O(1).

Come si può osservare queste funzioni sono incredibilmente veloci e permettono a questa struttura dati di performare molto bene in questa circostanza.

3.2 – Operazioni di supporto

In fase di sviluppo mi sono ritrovato più volte a dover effettuare delle procedure identiche tra loro, così ho deciso di incapsularle all'interno di funzioni da usare come supporto in altre funzioni. Queste funzioni sono `getAdiacenti`, `ruleOk`, `trovaBlocco` e `cambiaRadice`. `getAdiacenti` permette di trovare in automatico tutte le coordinate delle piastrelle che costituiscono l'intorno di una piastrella. `RuleOk` permette di verificare se una regola è applicabile ad una piastrella oppure no. `trovaBlocco` permette di trovare tutte le piastrelle facenti parte di un blocco. `CambiaRadice` permette di cambiare la radice di un blocco quando se ne verifica la necessità (ulteriori dettagli nella sezione 3.3).

Così come per le funzioni di Union-Find, anche queste funzioni devono essere veloci ed efficienti, essendo di supporto. `GetAdiacenti` e `ruleOk` hanno complessità temporale e spaziale pari a O(1) (in realtà `ruleOk` ha complessità temporale di O(k), dove k è il numero di elementi di `rule.ruleset`, però, siccome un'intorno di una piastrella può avere al massimo otto piastrelle, significa che la complessità temporale di `ruleOk` è O(8), quindi O(1)).

`trovaBlocco` è la funzione più interessante tra queste, perché sfrutta una DFS (*depth-first search*) per trovare tutte le piastrelle facenti parte di un blocco. La DFS è un algoritmo utilizzato per esplorare tutti i vertici e gli spigoli di un grafo, ma in questo caso è applicata all'albero che rappresenta il blocco (rappresentato dai valori delle proprietà delle piastrelle). L'idea principale è di andare il più a fondo possibile lungo un ramo dell'albero prima di tornare indietro e visitare i rami rimanenti. E' implementata con una pila ed è un potente strumento per esplorare completamente i percorsi e le connessioni. La complessità temporale è O(V+E), dove V è il numero di piastrelle nel blocco ed E è il numero di connessioni tra le piastrelle. Siccome nella mia implementazione non ho delle connessioni vere e proprie, posso considerare E come il numero di adiacenze o connessioni esaminate durante la connessione o come la rappresentazione indiretta della profondità dell'albero. Quindi, ricapitolando, la complessità temporale è O(V+E), dove V è il numero di piastrelle del blocco ed E è il rango dell'albero. La complessità spaziale invece è O(V), dove V è sempre il numero di piastrelle del piano.
Un altro aspetto interessante di questa funzione è l'utilizzo di una *funzione di livello superiore* (chiamata anche *funzione anonima* o *lambda function*) per influenzare la visita del blocco. Questa funzione anonima viene passata come parametro a `trovaBlocco` e in base a se dobbiamo trovare il blocco omogeneo partendo da una piastrella, o se dobbiamo trovare l'intero blocco, o se dobbiamo trovare tutte le piastrelle che soddisfano la condizione x, basta mettere quella condizione nella funzione anonima ed essa funzionerà come un filtro. Questa tecnica è incredibilmente potente e utile, e ci permette di semplificare il codice e renderlo modulare, per fare in modo che si adatti alle nostre esigenze.

`cambiaRadice` dipende da `trovaBlocco`, quindi la sua complessità viene dominata da quella di `trovaBlocco`. Ne consegue che entrambe le complessità siano uguali a quelle di `trovaBlocco`.

3.3 – Operazioni per le piastrelle

Gestire le piastrelle significa accenderle (e colorarle), spegnerle e verificarne lo stato. Per fare ciò ho implementato tre operazioni con delle funzioni che mi permettono di gestire questi compiti in modo efficiente, sfruttando sia le funzioni di Union-Find sia quelle di supporto.

Per implementare le operazioni di accensione e colorazione ho scritto la funzione `colora`. Essa utilizza le funzioni `Add`, `getAdiacenti` e `Union` per creare la piastrella nel sistema, attribuirne le proprietà ed eseguire eventuali `Union` nel caso in cui la piastrella da inserire non sia una radice. Di conseguenza la complessità temporale è θ(α(n)) (~O(1)) e quella spaziale è O(1).

Per implementare l'operazione di verifica dello stato di una piastrella non ho usato funzioni particolari, ma ho semplicemente sfruttato le proprietà di accesso e ricerca in tempo costante delle mappe. Quindi la complessità temporale e spaziale è O(1).

Per implementare l'operazione di spegnimento ho scritto la funzione `spegni`. Non lo nego, è la funzione che, con questa implementazione, mi ha creato più problemi. Infatti, quando spegniamo una piastrella bisogna fare molte valutazioni. Innanzitutto bisogna controllare che la piastrella che vogliamo spegnere non sia la radice, in quanto, se la spegnessimo, non avremmo più un riferimento per le altre piastrelle. Se capita di spegnere la radice allora tramite la funzione `cambiaRadice` si può cambiare il riferimento al blocco usando una piastrella circonvicina all'attuale radice. Questa operazione non crea nessun problema perché il nostro blocco è un insieme di piastrelle senza relazioni tra di esse (se non il fatto di essere adiacenti) quindi cambiare la radice è paragonabile a cambiare il nome di quel blocco, ma le proprietà dello stesso rimangono inalterate. Un'altra considerazione che bisogna fare è la seguente: quando si spegne una piastrella essa potrebbe spezzare un blocco in due o più blocchi, quindi bisogna *crearne* altri. Come già detto in precedenza non creiamo altri blocchi, ma modifichiamo le proprietà delle piastrelle (una per blocco) in modo da avere dei riferimenti per i nuovi insiemi di piastrelle appena formati. Con `trovaBlocco` troviamo i vari nuovi blocchi e poi modifichiamo le proprietà delle varie pistrelle per farle adattare alle nostre esigenze (C'è da notare che possiamo trovare al massimo 4 nuovi blocchi ad ogni applicazione di `spegni`, perché la configurazione di piastrelle circonvicine ottimale, per massimizzare il numero di blocchi distinti uniti da una singola piastrella, è quella dove le piastrelle dei blocchi condividono con la piastrella centrale solo uno spigolo). Le funzioni utilizzate sono `Find`, `cambiaRadice` e `trovaBlocco`, di conseguenza la complessità temporale è O(V+E), dove V è il numero di piastrelle del blocco ed E è il rango dell'albero, e la complessità spaziale è essere O(n), in quanto nel caso peggiore abbiamo un solo blocco nel sistema e spegnendone una piastrella si spezza in più blocchi, coinvolgendo tutte le piastrelle del piano.

3.4 – Operazioni per i blocchi

Le operazioni che riguardano i blocchi sono quelle implementate dalle funzioni `blocco` e `bloccoOmog`. Entrambe svolgono la stessa funzione: trovare la somma delle intensità delle piastrelle che compongono un blocco, con la differenza tra le due che `bloccoOmog` tiene conto solamente delle intensità delle piastrelle dello stesso colore della piastrella su cui viene chiamata la funzione.

L'operazione `blocco` è implementata con complessità, sia temporale sia spaziale, $\theta(\alpha(n))$ (~O(1)) perché usa una `Find` per trovare la radice del blocco, e una volta trovata basta solo restituire il campo `blockIntensity` di essa.

L'operazione `bloccoOmog`, invece, utilizza la funzione di support trovaBlocco insieme alla lambda function che filtra tutte le piastrelle di un determinato colore. Da ciò ne consegue che la complessità temporale di `bloccoOmog` è O(V+E), dove V è il numero di piastrelle del blocco omogeneo ed E è il rango dell'albero. La complessità spaziale è O(V), dove V è il numero di piastrelle del blocco.

EDIT: In fase di stesura di questa relazione mi è venuta in mente una diversa possibile implementazione di questa operazione. Dentro `properties` si può tenere traccia della somma delle intensità delle piastrelle dei vari colori con una mappa, con come chiave il colore e come valore la somma delle intensità. Quei valori andrebbero aggiornati al momento dell'inserimento delle piastrelle nel sistema e permetterebbero a `bloccoOmog` di avere una complessità temporale di O(k), dove k è il numero di colori diversi all'interno del blocco, e come complessità spaziale O(1). Dato la fase avanzata dello sviluppo del programma ho convenuto che fosse meglio lasciarlo nello stato attuale e parlarne in questo paragrafo.

3.5 – Operazioni per le regole

Le operazioni che il programma deve poter compiere, che riguardano le regole, sono diverse e variegate. Deve essere in grado di creare e memorizzare le regole in ordine di inserimento, deve essere in grado di stampare l'inisieme delle regole, deve essere in grado di riordinare in ordine crescente l'insieme delle regole e deve essere in grado di applicare una regola ad una piastrella o ad ogni piastrella di un blocco.

Le regole, come gia' detto nella sezione 2.2.3, vengono scomposte e memorizzate in delle strutture dati a noi convenienti. L'operazione di aggiunta di una regola è adibita alla funzione `regola`. In essa non utilizzo particolari funzioni, ma per lo più funzioni di libreria di Golang, come `strings.Split` e `append`. `strings.Split` divide la stringa r in un array di sottostringhe, quindi, se la lunghezza di r è n e ci sono k spazi, la complessità è O(n). Successivamente, ho un ciclo for che itera sulla lunghezza dell'array risultante da `strings.Split`, quindi la complessità è O(m), dove m è il numero di elementi dell'array. Ne consegue che la complessità totale di questa funzione è O(n+m), dove n è la lunghezza dell'input della regola, e m è la lunghezza risultante dell'array `rulesSplitted`. Per quanto riguarda la complessità spaziale

`strings.Split` crea un array di sottostringhe di lunghezza k, quindi O(k), e m/2 è il numero di `ruleset` aggiunti alla slice. La complessità spaziale finale è O(k+m/2), dove k è il numero di spazi ed m è il numero di parole nella stringa.

L'operazione di stampa dell'elenco delle regole è gestita dalla funzione `stampa`, che itera sull'elenco delle regole e sull'elenco di `ruleset`. Si può osservare, come già fatto nella sezione 1.2, che un intorno di una piastrella può avere al massimo 8 piastrelle, quindi anche `ruleset` può avere al massimo 8 elementi. La complessità temporale quindi si può definire come O(n*m), dove n è il numero di regole e m è il numero di elementi di `ruleset`, ma per l'osservazione precedente si può semplificare con O(8n), quindi ~O(n). La complessità temporale è O(1).

EDIT: Inizialmente avevo implementato `ruleset` con una mappa, con come chiave il colore (potendo comparire solo una volta nei requisiti è di fatto univoco) e con come valore il numero di piastrelle di quel colore che devono essere presenti nell'intorno. Tuttavia, al momento della stampa delle regole, accadeva una cosa curiosa. Quando si itera su una mappa, Golang lo fa automaticamente in ordine alfabetico, quindi, se aggiungo una regola `1 rosso 1 blu → giallo`, viene stampata `giallo: 1 blu 1 rosso`. Questa cosa va contro le specifiche di `tiles/traccia.pdf`, quindi ho sostituito la mappa con un array.

L'operazione di applicazione delle regole è implementata dalle due funzioni di propagazione: `propaga` e `propagaBlocco`.
`propaga` controlla per ogni regola se è applicabile alla piastrella presa in esame, di conseguenza la complessitò è O(r*m), dove r è il numero di regole e m è il numero di elementi di `ruleset`. Per le medesime osservazioni precedenti la complessità si riduce a O(r), dove r è il numero di regole dell'elenco.
`propagaBlocco` invece ha al suo interno due principali cicli for che ne determinano la complessità. Il primo è quello che itera sul blocco di piastrelle, quindi la complessità temporale è O(V), dove V è il numero di piastrelle del blocco. Il secondo è quello che itera sull'elenco delle regole, quindi ha complessità O(r). Quella totale diventa O(V*r). Poi c'è anche la funzione `trovaBlocco`, che fa aumentare la complessità totale a O(V*r + V + E). La complessità spaziale totale è O(V), dove V è il numero di elementi di `block`, `originalBlock` e `seen`, ovvero il numero di piastrelle nel blocco.

Infine c'è la funzione `ordina`, che deve implementare l'operazione di ordinamento dell'elenco delle regole. Siccome esistono tanti algoritmi di sorting bisogna trovare quello più adatto alle nostre esigenze. L'ordinamento deve avere una caratteristica principale, già descritta nella sezione 2.1, ovvero quella della stabilità. Un algoritmo di sorting stabile è un algoritmo che, a parità di "criterio di valutazione", mantiene inalterato l'ordine relativo tra due record. Con questa specifica si riduce il numero di scelte disponibili. Quello designato per questo compito era, inizialmente, il `mergeSort`, un algoritmo di ordinamento basato sul paradigma *divide-et-impera* che funziona dividendo l'array da ordinare in due metà, ordinando ciascuna metà in modo ricorsivo, e poi combinando (*merging*) le due metà ordinate per ottenere l'array completamente ordinato. L'array viene diviso log(n) volte, e ogni livello di divisione richiede O(n) per la fusione. La complessità temporale è quindi O(n*log(n)) e quella spaziale è O(n). Tuttavia, in fase di sviluppo, ho riscontrato dei problemi a riguardo, a cui purtroppo non sono stato in grado di trovare una soluzione. Ho deciso quindi di virare su un ben più modesto `insertionSort`, un algoritmo di ordinamento semplice e intuitivo che funziona costruendo l'array ordinato un elemento alla volta, inserendo ogni nuovo elemento nella posizione corretta rispetto agli elementi già ordinati. Ha una complessità temporale di $O(n^2)$ nel caso in cui l'array da ordinare sia ordinato in modo contrario a quello che vogliamo (nel nostro caso, descrescente), mentre una complessità di O(n) nel caso in cui l'array sia già ordinato. Quello che lo differenzia dal `mergeSort` è la complessità spaziale, ridotta soltanto a O(1). Infatti, si dice che l'`insertionSort` riordina l'array *in loco*, ovvero non utilizza strutture dati ausiliarie, come invece fa il `mergeSort`.

4- Conclusione

Il problema di questo applicativo è stato affrontato con un approccio mirato a garantire efficienza e flessibilità nell'implementazione delle regole e nella gestione dei blocchi di piastrelle. Attraverso una struttura dati Union-Find, ho ottimizzato il tracciamento delle piastrelle circonvicine, consentendo una gestione efficiente dei blocchi. Inoltre, l'uso di un algoritmo di sorting stabile per ordinare le regole assicurata un'applicazione coerente e sequenziale delle stesse.

```
[elia@arch-elia ~/github/algo_exam/]$ q
[elia@arch-elia ~/github/algo_exam/]$
```