```
[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 – Differenze tra consegna e revisione

Questa relazione è la versione aggiornata e revisionata rispetto a quella precedente. In particolare, gli aspetti per cui si differenzia sono i seguenti:
- struttura del codice modificata
- modellazione del problema migliorata con l'utilizzo di concetti dei grafi
- aggiunta descrizione migliore del funzionamento delle operazioni

1.2 – Modellazione del problema

L'uso di strutture dati efficienti è cruciale nella risoluzione di problemi complessi, specialmente quando si lavora con insiemi di oggetti connessi come, nel nostro caso, le piastrelle in un piano. Una delle strutture dati migliori per la gestione di insiemi disgiunti è la Union-Find, conosciuta anche come Disjoint-Set, fornendo operazioni efficienti per unire insiemi e trovare i rappresentanti dei singoli elementi. È comunemente usata in problemi che richiedono la gestione di connessioni dinamiche, come il calcolo delle componenti connesse in un grafo. I grafi sono strutture composte da nodi (o vertici) e archi che connettono coppie di nodi. I nodi sono gli oggetti che dobbiamo rappresentare, nel nostro caso le piastrelle, e gli archi sono le relazioni tra gli oggetti, nel nostro caso la condizione di adiacenza tra due piastrelle. La rappresentazione di un grafo può avvenire in diversi modi: liste di archi, matrici di adiacenza/incidenza, liste di adiacenza/incidenza, o altre strutture specifiche a seconda delle esigenze dell'applicazione. In alcuni scenari, pero', non è necessario rappresentare esplicitamente il grafo, come ho deciso di fare per questo progetto, in quanto, invece di memorizzare esplicitamente le connessioni, possiamo utilizzare una struttura union-find per mantenere traccia delle componenti connesse, rappresentando così il grafo in modo implicito. Un grafo implicito è una struttura in cui i nodi e gli archi non sono esplicitamente memorizzati come tali, ma possono essere dedotti attraverso regole o funzioni. Il dualismo tra grafi e union-find

emerge particolarmente quando consideriamo le operazioni di connettività nel sistema di piastrelle, in quanto i grafi forniscono un ottimo modo per rappresentare le connessioni tra le piastrelle, mentre la union-find offre un modo efficiente per gestire queste connessioni e per eseguire operazioni come l'unione di insiemi e la determinazione delle componenti connesse.

2 - Descrizione delle strutture dati

Ho definito diversi tipi complessi per rappresentare al meglio il grafo e l'intero sistema. Ho creato dei tipi per rappresentare le piastrelle (quindi i nodi del grafo) e dei tipi per rappresentare le adiacenze/blocchi (le connessioni tra i nodi). Come vedremo nella sezione 2.1 piastrelle, adiacenze e blocchi, nella mia rappresentazione implicita del grafo, sono strettamente correlate. Invece, per quanto riguarda le regole, ho usato una rappresentazione più tradizionale come l'array (sezione 2.2).

2.1 - Piastrelle, adiacenze e blocchi

La struct `piastrella` rappresenta una singola piastrella del sistema. È identificata univocamente con le coordinate x e y, che identificano la posizione della piastrella nel piano, ed è la rappresentazione di base di un nodo del grafo. Infatti, in un grafo un nodo può essere identificato da un'etichetta o da un identificatore univoco. Questa rappresentazione è oltremodo efficace poiché le operazioni di accesso e modifica delle piastrelle possono essere eseguite direttamente utilizzando le coordinate.

La struct `properties` rappresenta le proprietà di una piastrella ed è ciò che permette la rappresentazione implicita del grafo. Infatti nella struct vengono immagazzinate varie informazioni, elencate nel dettaglio in seguito, che permettono di tenere traccia dei vari blocchi di piastrelle, e di conseguenza delle componenti connesse del grafo. In particolare, la struct include informazioni come il colore della piastrella (`color`), l'intensità del colore della piastrella (`intensity`), il genitore di una piastrella, ovvero la prima piastrella del piano accesa con nessuna piastrella circonvicina accesa (`parent`) (vedi sezione 3.2), la dimensione del blocco, ovvero il numero di piastrelle che compongono un blocco (`size`) e l'intensità totale del blocco di piastrelle (`blockIntensity`). Questa implementazione permette di lavorare contemporaneamente sia con i grafi sia con la Union-Find, in modo complementare, facendo in modo di avere tutte le informazioni rilevanti in un'unica struttura, semplificando l'accesso e la gestione dei dati. Inoltre permette di manipolare facilmente le piastrelle, aggiungendo o rimuovendo elementi dal piano (vedi sezione 2.3).

2.2 - Regole

La struct `rule` rappresenta una singola regola all'interno del sistema. Una regola è un meccanismo che permette ad una piastrella di cambiare colore a secoda delle piastrelle circonvicine accese. Include varie informazioni come il colore che la regola applica alla piastrella (`color`), il numero di volte che una regola è stata applicata (`usage`) e una lista di ruleset, ovvero l'elenco di piastrelle circonvicine necessarie affinchè a regola possa essere applicata (`ruleset`).

La struct `ruleset` definisce un insieme di coppie numero-colore corrispondenti alle piastrelle circonvicine che devono essere accese affinchè la regola possa essere applicata. Include due informazioni: il colore della piastrella (`color`) e il numero di piastrelle di quel colore che devono essere accese (`count`).

La combinazione di questi due tipi di dati ci permette di scomporre la regola in componenti più comode da manipolare, analizzare ed applicare.

2.3 - Piano

La struct `piano` rappresenta il sistema nella sua interezza. Contiene due campi al suo interno: `tiles` e `rules`, rispettivamente l'iniseme delle piastrelle e l'insieme delle regole. Il primo è niente di meno che il grafo implicito. Infatti `tiles` è una mappa, con come chiave il tipo piastrella, una coppia di coordinate univoca, che rappresenta i nodi del grafo, e con come valore il tipo properties, che rappresenta le relazioni che il nodo (x,y) ha con le altre piastrelle. L'utilizzo di una mappa permette l'inserimento, la ricerca e l'eliminazione di un elemento molto conveniente. Il secondo è invece un array di elementi di tipo `ruleset`, molto comodo per applicarci un algoritmo di sorting. Per la precisione, `ruleset` è un puntatore ad un array. Ciò è stato necessario in quanto `append`, la funzione usata dentro `regola` (sezione 3.3) fa in automatico quello che, in linguaggi come C, bisogna fare manualmente (*memory management*). 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 – Operazioni

3.1 - Operazioni di Union-Find

La struttura dati Union-Find, conosciuta anche come Disjoint Set Union (DSU), è una struttura che gestisce un insieme di elementi partizionati in più insiemi disgiunti. Nel caso specifico di questo progetto il piano coincide con l'intera struttura Union-Find e viene inizializzato nel main tramite la funzione `makeSet`, che semplicemente inizializza `tiles` e `rules`.

Quando una piastrella viene accesa (vedi sezione 3.2) bisogna aggiungerla al sistema di piastrelle. Per fare ciò si utilizza la funzione `Add`. Essa permette di creare una piastrella, qual'ora non fosse già accesa, e di inserirla all'interno di `tiles`. In termini di grafo, `Add` permette di creare un nuovo nodo senza relazioni con altri nodi. Nella pratica `Add` aggiunge alla mappa `tiles` un nuovo elemento nel caso in cui quell'elemento non sia presente, mentre aggiorna i vari campi della struct `properties` associata nel caso in cui l'elemento sia già presente.

Per quanto riguarda le mere operazioni di Union-Find, sono principalmente due: `Find` e `Union`. `Find` permette, dato un elemento, di risalire a quale insieme appartiene. Applicata nel contesto delle piastrelle, questa funzione permette di risalire a quale blocco di piastrelle una piastrella appartiene. Infatti, come identificativo di un blocco di piastrelle, si intende la prima piastrella accesa che non ha piastrelle circonvicine accese (ulteriori dettagli nella sezione 3.2). Nella pratica quando si chiama `Find` su una piastrella la funzione controlla ricorsivamente il campo parent della struct `properties` associata alla piastrella, aggiornando per ogni piastrella esaminata il relativo campo `parent`. Questa operazione di aggiornamento viene definita *path compression*, una tecnica di ottimizzazione che permette di rendere le future chiamate di `Find` molto veloci. `Union` permette di unire due insieme disgiunti in un solo insieme. Applicata nel contesto delle piastrelle, questa funzione permette di creare una relazione di adiacenza tra due piastrelle, ovvero crea un arco tra due nodi del grafo. `Union` permette anche di unire due blocchi diversi, formati ciascuno da più piastrelle, in un unico grande blocco, e per decidere quale dei due identificativi dei due blocchi continuare ad usare, sfrutta la tecnica di ottimizzazione chiamata *union by size* che permette di mantenere come identificativo quello del blocco con il campo `size` maggiore. Nella pratica `Union` utilizza `Find` per trovare le due piastrella da cui si sono originati i due blocchi, e confronta il campo `size` di ciascuna. In base a quale dei due campi è maggiore si modificano le proprietà di una delle due

piastrelle di conseguenza, in modo tale che la piastrella "vincente" risulti come l'origine del nuovo blocco appena formato.

Le tecniche di ottimizzazione usate, *path compression* e *union by size*, rendono la struttura Union-Find estremamente efficiente per gestire le piastrelle e le loro manipolazioni, usando una mappa per tenere traccia delle piastrelle accese. Inoltre permette di tenere traccia in modo ordinato ed efficiente delle componenti connesse all'intero del grafo. Le mappe assicurano tempi di accesso, ricerca ed eliminazione costanti, quindi O(1), così come la complessità temporale delle funzioni `makeSet` e `Add` (per quanto riguarda `Add`, se la piastrella non è già presente nella mappa, la complessità è O(1), se invece è già presente la complessità è pari a quella di `Find` e `Union`). Le funzioni `Find` e `Union` hanno una complessità di ~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). Lo spazio occupato da queste funzioni, invece, è sempre costante, in quanto non usano strutture dati esterne per eseguire le loro operazioni, ma solo variabili locali temporanee.

Un'altra funzione importante dal punto di vista teorico è `getAdiacenti`. Essa restituisce un'array contenente 8 piastrelle, ovvero tutte le possibili piastrelle circonvicine che una piastrella può avere. In termini di grafo questa funzione permette di trovare tutti i possibili archi che possono essere instaurati tra il nodo corrente (la piastrella di cui si vogliono sapere le piastrelle circonvicine) e gli altri nodi (le piastrelle circonvicine). Il concetto di arco è racchiuso all'interno del concetto di adiacenza, infatti se due piastrelle non sono tra loro circonvicine esse non possono essere connesse tra loro (senza altre piastrelle intermedie), quindi non può esistere un arco diretto tra i due nodi. Anche per getAdiacenti la complessità è costante, quindi O(1), in quanto la grandezza dell'array restituito è sempre 8. La stessa cosa vale per lo spazio occupato, costante O(1).

3.2 - Operazioni per le piastrelle

Gestire le piastrelle significa accenderle (e colorarle), spegnerle e verificarne lo stato. In termini di grafo significa creare, aggiungere, collegare ed eliminare dei nodi.

Per implementare le operazioni di accensione e colorazione ho definito la funzione `colora`. Quando una piastrella deve essere colorata viene creata e aggiumta al sistema con la funzione `Add`. Successivamente bisogna controllare se essa ha piastrelle circonvicine accese attraverso la funzione `getAdiacenti`. Se non risultano piastrelle circonvicine accese allora la nuova piastrella creata sarà quella da cui avrà origine un blocco. Altrimenti la funzione controlla ogni piastrella circonvicina ed effettua una'operazione di `Union` tra la nuova piastrella e ogni piastrella già accesa. In realtà la vera e propria operazione di `Union` viene effettuata solo con la prima piastrella circonvicina accesa, perchè successivamente gli identificativi della nuova piastrella accesa e le piastrelle circonvicine coincidono, e non c'è bisogno di unire altri insiemi. Questo si traduce in: quando si crea un nuovo nodo del grafo si controlla se si può creare un arco tra esso e un'altro nodo già esistente. `colora` gestisce egregiamente anche il caso in cui una nuova piastrella, con la sua accensione, crea un collegamento tra due blocchi diversi. In quel caso l'operazione di `Union` viene eseguita almeno 2 volte: la prima per aggiungere la nuova piastrella ad un blocco, e la seconda (o più) per collegare i blocchi tra loro.

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 per ottenere le informazioni relative ad una piastrella (nodo).

Per implementare l'operazione di spegnimento ho definito la funzione `spegni`. Probabilmente è la funzione più complessa in termini di operazioni che bisogna eseguire. Infatti, quando spegniamo una piastrella bisogna fare molte valutazioni. Innanzitutto bisogna controllare che la piastrella che vogliamo spegnere non sia quella da cui

si è generato il blocco. Se dovessimo spegnerla, parlando in termini di programmazione, non avremmo più un identificativo per il blocco, e quindi non avremmo più la possibilità di identificare le componenti connesse. Se dovesse capitare di dover spegnere una piastrella di questo tipo (quindi un "nodo-radice" del grafo) allora tramite la funzione `cambiaRadice` si può cambiare il riferimento al blocco usando una piastrella circonvicina a quella attuale. Questa operazione non crea nessun problema perché il blocco è un insieme di piastrelle senza relazioni dirette tra di loro (se non il fatto di essere adiacenti) quindi cambiare la piastrella iniziale è paragonabile a cambiare il nome di quel blocco, ma le proprietà dello stesso rimangono inalterate.

Per descrivere `cambiaRadice` occorre introdurre un altro concetto fondamentale dei grafi: le visite. Siccome con l'attuale implementazione non ho modo di tenere traccia esplicitamente delle componenti connesse necessito di un sistema che mi permetta di ricavare all'occorrenza queste componenti (le piastrelle appartenenti ad un blocco). Per fare ciò si utilizza una tecnica di attraversamento dei grafi chiamata *visita in profondità*, o DFS (*depth-first search*). La DFS è un algoritmo utilizzato per esplorare tutti i vertici e gli spigoli di un grafo, ma in questo caso è applicata alla struttura Union-Find che rappresenta il grafo (rappresentato dai valori delle proprietà delle piastrelle). L'idea principale è di andare il più a fondo possibile lungo una direzione del grafo prima di tornare indietro e visitare le direzioni rimanenti. E' implementata con una pila: ad ogni iterazione prelevo dalla pila una piastrella, ne trovo gli adiacenti e li aggiungo a loro volta alla pila. Oltre alla pila si usa anche una mappa per tenere traccia delle piastrelle già visitate.
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 (vedi sezione 3.4), o se dobbiamo trovare l'intero blocco, o se dobbiamo trovare tutte le piastrelle che soddisfano la condizione x, basta specificare quella condizione nella chiamata della 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 a più esigenze (ulteriori esempi di utilizzo nelle sezioni 3.3 e 3.4).

La DFS è incapsulata nella funzione `trovaBlocco`, che viene a sua volta usata nella funzione `cambiaRadice`. Quando è necessario spegnere una piastrella da cui si è formato un blocco si copiano le informazioni da `properties` dell'attuale piastrella a quelle della prima piastrella circonvicina esistente ottenuta con la funzione `getAdiacenti`. Una volta fatto ciò si procede a recuperare, grazie a `trovaBlocco`, tutte le piastrelle appartenenti al blocco e ad aggiornarne `properties`.

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 fare in modo che ognuno dei nuovi blocchi creati abbia una piastrella di riferimento. 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, blocco per blocco, modifichiamo le proprietà delle varie pistrelle per fare in modo che ogni blocco appena formato abbia un riferimento univoco. (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).

Tra le operazioni appena descritte quelle meno dispendiose e complesse sono `colora` e `stato`, rispettivamente $\theta(\alpha(n))$ (~O(1)) e O(1). Anche lo spazio occupato è costante O(1). La complessità di `spegni` invece è maggiore, grazie all'utilizzo della funzione più complessa `trovaBlocco` che ne determina la compessità totale (`cambiaRadice` è dello stesso ordine di grandezza, usando al suo interno `trovaBlocco`). La complessità temporale di una DFS è O(V+E), dove V è il numero di nodi del grafo ed E è il numero di archi. Siccome nella mia implementazione non ho delle connessioni vere e proprie, posso considerare V come il numero di piastrelle ed E come il numero di adiacenze o connessioni esaminate durante la procedura. Quindi, ricapitolando, la

complessità temporale è O(V+E), dove V è il numero di piastrelle del blocco ed E è il numero di adiacenze. Lo spazio occupato invece è O(V), dove V è sempre il numero di piastrelle del piano.

3.3 - 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, 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` per scomporre una regola nelle sue componenti da aggiungere al sistema. `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 lo spazio occupato, `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 spazio occupato totale quindi 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 sezioni 3.1 e 3.2, 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). Lo spazio occupato da questa funzione è costante 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 della traccia fornita, 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 essa è applicabile alla piastrella presa in esame. Quando si chiama questa funzione, tramite `getAdiacenti` si controlla se l'intorno di piastrelle circonvicine accese ad una piastrella rispetta le condizioni delle regole disponibili. Quando le condizioni di una regola vengono soddisfatte allora essa è applicabile. Per controllare se l'intorno di una piastrella soddisfa le condizioni di una regola si usa la funzione `ruleOK`, che è composta da un semplice ciclo iterativo sul campo `ruleset` della regola. Siccome il capo `ruleset` di una regola può avere al massimo 8 elementi, allora la complessità temporale di essa è semplificabile a O(1). Lo spazio occupato è anch'esso costante, perchè non utilizza strutture aggiuntive. Successivamente, una volta appurata l'applicabilità di una regola si colora la piastrella selezionata. La complessità temporale 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 bisogno di più passaggi. Quando vogliamo applicare delle regole ad un intero blocco, dobbiamo sempre considerare come blocco di riferimento quello originale. Significa che se cambiamo il colore di una piastrella, il cambiamento non deve influire nelle successive propagazioni, e negli intorni delle altre piastrelle essa deve rimanere con il colore originale. Per fare ciò, la funzione, tramite `trovaBlocco`, recupera tutte le piastrelle del blocco (usando `true` come condizione della funzione anonima) e crea una copia locale temporanea del blocco appena trovato, in modo da usarlo come riferimento immutabile, nonostante si cambino i colori delle piastrelle del piano. Successivamente si verifica l'applicabilità di ogni regola e si cambia il colore. `propagaBlocco` ha al suo interno due principali cicli 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). Non bisogna dimenticare la funzione `trovaBlocco`, che fa aumentare la complessità totale a O(V*r + V + E). Si può semplificare in O(V(r+1) + E), e di conseguenza in O(V*r+E). Lo spazio occupato totale è O(V), dove V è il numero di elementi di `block`, `originalBlock` e `seen`, ovvero il numero di piastrelle nel blocco.

Infine, ma non meno importante, c'è la funzione `ordina`, che deve implementare l'operazione di ordinamento dell'elenco delle regole una volta ddefinito un campo chiave (in questo contesto `usage`). Siccome esistono tanti algoritmi di sorting bisogna trovare quello più adatto alle nostre esigenze. L'ordinamento deve avere una caratteristica principale: la *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 lo spazio occupato 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 (ipotizzo per colpa di problemi legati ai riferimenti dei vari sotto-array). Ho deciso quindi di virare su un approccio diverso. Ho deciso di usare un `bucketSort`. Si compone di due parti: la prima parte consiste nell'iterare sull'elenco di regole, in modo tale da smistarle in diversi *buckets*. I buckets sono implementati con un array, dove ogni cella contiene un rierimento ad una lista concatenata, in cui l'indice della cella dell'array corrisponde al valore di usege di una regola. Quinid, per esempio, se una regola ha usage uguale a 2 verrà inserita nella lista concatenata corrispondente alla cella di indice 2 nell'array. La seconda parte consiste nel ricostruire l'array originale prendendo di volta in volta l'elemento con usage minore dai bucket.

La complessità temporale di questo algoritmo è facilmente deducibile: bisogna iterare sull'elenco di regole, sull'array di bucket e sulle varie liste concatenate. Si può notare che la somma totale delle lunghezze delle liste concatenate è uguale al numero di regole presenti nel sistema. Di conseguenza la complessità temporale si semplifica in O(n+k), dove n è il numero di regole del sistema e k è il numero di bucket (ovvero il valore di `usage` della regola più utilizzata). Lo spazio occupato è anch'esso O(n+k) dove n è il numero di elementi delle liste concatenate (il numero di regole del sistema) e k è il numero di buckets.

Per quanto riguarda le liste concatenate, ho deciso di implementare da zero una struttura dati `linkedList` che mi permettesse di operare con facilità. Per la precisione `linkedList` rappresenta una lista concatenata singola, con due riferimenti, l'inizio (`head`) e la fine (`tail`) della lista. Ciò mi permette di avere una complessità temporale costante O(1) sia per la creazione di una lista sia per l'inserimento di un nodo. Lo spaazio occupato è invece lineare O(n), dove n è il numero di nodi di cui è composta la lista.

3.4 - Operazioni per i blocchi

I blocchi, come già detto in precedenza, corrispondono agli insiemi delle componenti connesse del grafo. Se due piastrelle appartengono allo stesso blocco significa che esiste una sequenza di piastrelle che collega due piastrelle

distanti tra loro. In termini di grafi significa che esiste un insieme di nodi e archi che collega due nodi: questo insieme è detto *cammino*. Nel sistema i blocchi non sono rappresentati in modo esplicito, ma tramite il campo `parent` della struct `properties` delle piastrelle. Se due o più piastrelle presentano lo stesso campo `parent`, significa che appartengono allo stesso blocco. Se una piastrella ha se stessa come campo `parent`, significa che è l'identificatrice del blocco.

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 in modo molto intelligente. Ogni piastrella ha dua informazioni che tornano utili in questo contesto: `intensity` e `blockIntensity`. La prima corrisponde all'intensità del colore di una piastrella, mentre la seconda corrisponde alla somma delle intensità del blocco a cui appartiene. Come intuibile, in tutte le piastrella queste due informazioni coincidono, tranne nella piastrella identificatrice del blocco. Di conseguenza per trovare l'intesità totale di un blocco è necessario solamente accedere al valore di `blockIntensity` della piastrella identificatrice del blocco. Per trovarla, si usa `Find`: da ciò deriva la complessità temporale di $\theta(\alpha(n))$ (~O(1)). Lo spazio occupato è invece costante O(1).

L'operazione `bloccoOmog`, invece, utilizza la funzione di supporto `trovaBlocco` insieme alla lambda function che filtra tutte le piastrelle di un determinato colore, in modo tale da tenere in considerazione solo le piastrelle necessarie. Da ciò ne consegue che la complessità temporale di `bloccoOmog` è dominata dalla complessità temporale di `trovaBlocco`, ovvero O(V+E), dove V è il numero di piastrelle del blocco omogeneo ed E è il numero di adiacenze. La spazio occupato è invece O(V), dove V è il numero di piastrelle del blocco omogeneo.

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 che non userebbe strutture dati aggiuntive. Data 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 piste

Come già descritto nella sezione 3.4, un blocco è un insieme di componenti connesse di un grafo, e se due piastrelle appartengono allo stesso blocco significa che esiste un cammino tra due nodi.

La funzione `pista` serve per, data una sequenza di direzioni (corrispondenti alle direzioni della rosa dei venti), verificare se esiste un cammino tra due piastrelle. Per fare ciò viene utilizzata la funzione `verificaPista`. Essa associa ad ogni possibile direzione una delle 8 piastrelle circonvicine alla piastrella di partenza, e procede iterativamente a controllare se la piastrella circonvicina corrispondente alla direzione è accesa o meno. Una volta terminato il controllo restituisce l'elenco delle piastrelle che permettono di arrivare alla fine della sequenza partendo da una piastrella iniziale. In termini di grafo `verificaPista` controlla se esiste un cammino tra due nodi, e se esiste restituisce l'insieme dei nodi che compongono il cammino. La complessità temporale di questa funzione di supporto è lineare, quindi O(n) dove n è il numero di direzioni (equivale al numero di piastrella che compongono il cammino). Lo spazio utilizzato è dello stesso ordine di grandezza della complessità temporale, quindi lineare.

Siccome, all'interno dell'insieme delle componenti connesse, ci possono essere più cammini possibili che collegano due nodi, è utile avere una funzione che permetta di ricavare il cammino minimo tra due nodi all'interno di un grafo. Ciò è adibito alla funzione `lung`. Essa, inizialmente effettua alcuni controlli, che non sono altro che condizioni necessarie affinchè possa esistere un cammino tra due nodi. Se i due nodi non appartengono allo stesso insieme delle componenti connesse significa che non esiste un cammino disponibile che li collega. Inoltre, se i due nodi coincidono, la lunghezza del cammino minimo risultante è unitaria.
Una volta eseguiti questi controlli la funzione ricava la lunghezza del cammino minimo tramite la funzione `camminoMinimo`. Questa funzione di supporto utilizza un'altra tecnica di visita di grafi, chiamata *visita in ampiezza*, o BFS (*breadth-first search*). La BFS è un algoritmo di ricerca che permette di trovare il cammino minimo all'interno di un grafo non pesato. L'idea principale è quella di visitare tutti i nodi del grafo, livello per livello. L'algoritmo si avvale di una coda in cui inserire i vari nodi da visitare e di una mappa per tenere traccia dei nodi già visitati e delle distanze parziali dal nodo iniziale. Una volta completata l'iterazione si ottiene la lunghezza del cammino minimo tra due nodi.

La complessità temporale e lo spazio occupato sono identici a quelli dell'algoritmo DFS descritto nella sezione 3.2, ovvero rispettivamente O(V+E), dove V è il numero di nodi (numero di piastrelle del blocco) ed E è il numero di archi (numemro di adiacenze) e O(V), dove V è il numero di nodi (numero di piastrelle).

4 - Esempi di esecuzione

I file all'interno della directory `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 `test/output` sono dei file in formato `.txt` che contengono l'output atteso dei corrispettivi file di input.

In particolare alcuni input sono utili a capire se il programma si comporta correttamente in situazioni particolari.
Per esempio il file `spegni_radice.txt` mette alla prova il programma provando a spegnere la piastrella identificativa del blocco. Come già descritto precedentemente nella sezione 3.2, grazie a `cambiaRadice`, questo problema viene superato egregiamente.
Un altro file interessante è `propaga_regole.txt`, che chiede al programma di effettuare molte operazioni, tra cui quella di riordinare l'elenco delle regole. Con questo file si può chiaramente vedere come l'algoritmo `bucketSort` funzioni in modo corretto mantenendo l'ordine relativo tra due regole con chiave uguale.

I restanti file di input sono:
- `blocco_intensity.txt` (`colora`, `blocco`, `bloccoOmog`)
- `colora_stato.txt` (`colora`, `stato`)
- `direzioni_cammino.txt` (`colora`, `pista`, `lung`)
- `regole_stampa.txt` (`regola`, `stampa`)
- `spegni.txt` (`colora`, `blocco`, `bloccoOmog`, `spegni`)

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