Relazione progetto di Agloritmi e Strutture Dati

Luca Dal Mas, 21118A

Gennaio 2024

Contents

1 Introduzione **3**

2 Struture Dati **3**
2.1 Sintesi . . . 3
2.2 Analisi . . . 5
2.2.1 Gioco . . . 5
2.2.2 Mattoncino . . . 5
2.2.3 Fila . . . 6

3 Algoritmi **6**
3.1 Inserisci mattoncino . . . 6
3.1.1 Sintesi . . . 6
3.1.2 Analisi . . . 6
3.2 Stampa Mattoncino . . . 6
3.2.1 Sintesi . . . 6
3.2.2 Analisi . . . 6
3.3 Disponi Fila . . . 7
3.3.1 Sintesi . . . 7
3.3.2 Analisi . . . 7
3.4 Stampa Fila . . . 7
3.4.1 Sintesi . . . 7
3.4.2 Analisi . . . 7
3.5 Elimina Fila . . . 7
3.5.1 Sintesi . . . 7
3.5.2 Analisi . . . 7
3.6 Disponi Fila Minima . . . 8
3.6.1 Sintesi . . . 8
3.6.2 Analisi . . . 8
3.7 Sotto Stringa Massima . . . 9
3.7.1 Sintesi . . . 9
3.7.2 Analisi . . . 9
3.8 Indice cacofonia . . . 9
3.8.1 Sintesi . . . 9

3.8.2 Analisi 9
3.9 Costo 10
3.9.1 Sintesi 10
3.9.2 Analisi 10

1 Introduzione

L'obiettivo di questa relazione è quello di esaminare approfonditamente le strutture dati e le funzioni coinvolte nell'implementazione del gioco dei mattoncini. Attraverso un'analisi dettagliata di tali componenti, si mira a comprendere appieno il funzionamento del sistema di gioco ai fini di progettare ed implementare un programma che massimizzi l'efficienza nella manipolazione dei mattoncini.

In questa relazione, verranno esaminate le seguenti componenti chiave del sistema di gioco:

- Le strutture dati fondamentali utilizzate per rappresentare i mattoncini e le loro interazioni sul tavolo di gioco.
- Le funzioni principali implementate per manipolare e gestire i mattoncini durante il gioco.
- Una valutazione critica delle complessità temporali e spaziali delle funzioni e delle operazioni coinvolte, al fine di comprendere meglio le prestazioni complessive del sistema.

2 Struture Dati

La selezione e la progettazione delle strutture dati svolge un ruolo fondamentale nella realizzazione di un sistema efficiente e funzionale. In questa sezione, analizzo le strutture dati scelte per rappresentare il gioco.

2.1 Sintesi

L'obiettivo principale di questa analisi è esaminare la natura del problema per identificare la struttura dati idonea. Questo comprende un'esplorazione delle caratteristiche delle strutture dati adottate e una valutazione delle implicazioni di prestazione legate alle operazioni richieste dal gioco.

Il gioco dei mattoncini prevede:

- I mattoncini come unità base del gioco, sono caratterizzati da un nome identificativo unico, noto come *sigma*, e da due lati con forme distinte, denominati rispettivamente forma *alpha* e forma *beta* del mattoncino.
- Il tavolo, che rappresenta l'ambiente su cui si sviluppa il gioco
- Le file sono sequenze di mattoncini con la stessa forma adiacente.
- Il sacchetto che contiene i mattoncini ancora disponibili da disporre in file

Una volta individuati i concetti fondamentali per la rappresentazione del problema possiamo procedere nella creazione di strutture dati che rappresentano la realtà.

```
type mattoncino struct {
    alpha, beta, sigma string
}

type fila = []mattoncino

type gioco struct {
    tavolo       map[sigma]mattoncino
    sacchetto    map[sigma]mattoncino
    file         []fila
}
```

La selezione delle variabili di tipo "map", sfruttando l'implementazione delle HashMap all'interno del linguaggio Go, è giustificata dalla necessità di identificare i mattoncini in base al loro nome. Le mappe, infatti, si distinguono per la capacità di individuare ogni elemento tramite una chiave, che in questo caso è di tipo stringa.

Tuttavia, le strutture dati identificate inizialmente, oggetto di un'analisi preliminare, possono beneficiare di un miglioramento mediante un'esame più approfondito. Si potrebbe considerare la fusione delle mappe `tavolo` e `sacchetto` in un'unica mappa denominata `mattoncini`, la quale includerebbe tutti i mattoncini in gioco, sia quelli posizionati sul tavolo che quelli presenti nel sacchetto. La differenziazione tra mattoncini sul tavolo e quelli nel sacchetto potrebbe essere gestita mediante un campo booleano all'interno di ciascun mattoncino, indicando la sua presenza o assenza nel sacchetto.

Un ulteriore ottimizzazione può essere implementata mediante l'utilizzo di puntatori alla struttura dati `mattoncino` all'interno della mappa `mattoncini`. Si evidenzi che se un mattoncino è posizionato sul tavolo, secondo le specifiche del gioco, è necessariamente associato a una fila. Pertanto, è possibile sostituire il booleano `sulTavolo` con un puntatore che fa riferimento a una fila nel caso in cui il mattoncino appartenga a una fila e, di conseguenza, si trovi sul tavolo. Questa modifica consente di eliminare l'elenco completo di tutte le file dalla struttura, poiché ora tali informazioni sono memorizzate tramite i puntatori nei singoli mattoncini.

Riflettendo sull'utilizzo di una fila, osserviamo che verrà sempre rappresentata come una sequenza di mattoncini, senza la necessità di effettuare accessi diretti in base alla posizione. Di conseguenza, possiamo considerare la sostituzione della struttura dati `slice` con una `linkeList` al fine di migliorare la gestione della memoria, specialmente in scenari con input di grandi dimensioni, senza compromettere le prestazioni nel nostro contesto d'uso. Considerando che un mattoncino, per costituire una fila, può essere girato, invertendo le forme dei lati `alpha` e `beta`, sarà necessario introdurre un booleano per memorizzare lo stato della direzione: se naturale o se il mattoncino è stato capovolto per appartenere a una fila. Infine, in considerazione delle future analisi dettagliate, che faremo per la funzione `disponiFilaMinima`, risulta necessario identificare il gioco come una struttura grafo. Pertanto, introduciamo una mappa aggiuntiva denominata `forme` al fine di archiviare tutti i mattoncini con una determinata forma.

```
type mattoncino struct {
  direzione          bool
  alpha, beta, sigma string
  fila               *fila
}

type fila = *list.List

type giocoStruct struct {
  mattoncini map[string]*mattoncino
  forme      map[string]*list.List
}

type gioco = *giocoStruct
```

2.2 Analisi

Analizziamo i costi delle strutture dati utilizzate e le considerazioni relative alla loro progettazione e implementazione.

2.2.1 Gioco

La struttura del gioco è implementata con due mappe: `mattoncini` e `forme`. La mappa dei mattoncini permette di identificare rapidamente ogni mattoncino attraverso il suo nome univoco `sigma`, facilitando operazioni come l'inserimento e la ricerca. Allo stesso modo, la mappa delle forme consente di organizzare i mattoncini in base alla loro forma, fornendo un meccanismo per individuare rapidamente tutti i mattoncini con una determinata forma durante le operazioni di gioco. Costo di tempo delle operazioni di inserimento, ricerca per chiave e eliminazione: $O(1)$. Lo spazio di memoria occupato è lineare al numero di mattoncini dati in input.

2.2.2 Mattoncino

La struttura del mattoncino è implementata come una struttura dati composta da diversi campi, tra cui le informazioni sui lati `alpha` e `beta` del mattoncino, il suo nome identificativo `sigma`, e un campo booleano per rappresentare lo stato della `direzione` (naturale o invertita). L'utilizzo di puntatori e campi booleani consente di gestire in modo efficiente informazioni aggiuntive, come lo stato del mattoncino sul tavolo di gioco o il suo appartenere a una fila. La scelta di inserire il campo `fila` come un puntatore ad un puntatore di una linkeList è ragionata: in questo modo tutti i puntatori `fila` di mattoncini appartenenti ad una stessa fila conteranno al loro interno l'indirizzo di un altro puntatore che punterà, questa volta alla fila vera e propria. Per eliminare una fila basterà quindi deferenziare quest'ultimo puntatore.

2.2.3 Fila

La struttura della fila è implementata come una lista collegata. Ogni nodo della lista rappresenta un singolo mattoncino, e i puntatori vengono utilizzati per collegare i mattoncini tra loro. Questa struttura permette una gestione ottimale della memoria nel caso di file di grandi dimensioni. Costo in termini di spazio in memoria lineare al numero di mattoncini dati in input.

3 Algoritmi

In questa sezione, esamineremo attentamente le funzioni implementate. Attraverso un'analisi dettagliata di ciascuna funzione, esploreremo la loro logica di implementazione e complessità computazionale. L'obiettivo è fornire una visione chiara e completa delle strategie algoritmiche utilizzate.

3.1 Inserisci mattoncino

3.1.1 Sintesi

Come da specifiche l'obiettivo della funzione `inserisciMattoncino` è quello di inserire correttamente un nuovo mattoncino all'interno del gioco.

3.1.2 Analisi

La complessità dell'algoritmo è legata alle operazione di inserimento del nuovo mattoncino nelle mappe dei `mattoncini` e `forme` Queste operazioni grazie alla struttura dati di tipo hashmap, hanno una complessità temporale costante $O(1)$. Anche l'inserimento nelle `linkedList` e il controllo della presenza di una chiave nelle mappe hanno complessità temporale costante $O(1)$. La complessità totale della funzione è $O(1)$.

3.2 Stampa Mattoncino

3.2.1 Sintesi

Come da specifiche l'obiettivo della funzione `stampaMattoncno` è di stampare nel formato indicato un mattoncino dato il suo nome identificativo `sigma`

3.2.2 Analisi

L'implementazione della funzione è diretta, con una complessità temporale costante $O(1)$, poiché accede direttamente alla mappa `mattoncini` usando `sigma` come chiave.

3.3 Disponi Fila

3.3.1 Sintesi

Come da specifiche, la funzione, presa in input una sequenza di mattoncini deve verificare l'esistenza di ogni mattoncino nel gioco e la coerenza delle relazioni tra di essi prima di disporli sulla fila. Se un mattoncino non è presente nel gioco o non può essere posizionato correttamente, la funzione non esegue alcuna operazione.

3.3.2 Analisi

La complessità temporale della funzione dipende principalmente dalla lunghezza dell'elenco dei nomi dei mattoncini da disporre sulla fila. In generale, la funzione richiede un tempo proporzionale al numero di mattoncini nell'elenco, quindi la sua complessità temporale può essere espressa come $O(n)$, dove n è il numero di mattoncini da disporre. La funzione, per ogni mattoncino dell'elenco controlla la sua esistenza, e la presenza di esso in una fila $O(1)$. Per la scelta implementativa delle strutture dati, l'appartenenza di un mattoncino in una fila è rappresentato dal suo campo `fila`, se è nil significa che non è mai appartenuto ad una fila, altrimenti punterà ad un puntatore fila.

3.4 Stampa Fila

3.4.1 Sintesi

Come da specifiche la funzione deve stampare una fila dato un certo mattoncino `sigma`.

3.4.2 Analisi

La complessità temporale della funzione dipende dalla ricerca per chiave all'interno della mappa `mattoncino` che come già visto per le proprieta delle hashmap è costante $O(1)$, e dalla stampa di ogni mattoncino all'interno della fila in caso positivo. Dato che la stampa si ripete per ogni mattoncino il costo complessivo di questa funzinoe è $O(n)$ dove n è il numero di mattoncini di cui è formata la fila.

3.5 Elimina Fila

3.5.1 Sintesi

Come da specifiche la funzione deve rimuovere una fila, riponenedo tutti i mattoncini nel sacchetto.

3.5.2 Analisi

La funzione deve cercare il mattoncino specificato nella mappa dei mattoncini del gioco e quindi rimuovere il puntatore alla fila associato a quel mattoncino.

Poiché l'accesso alla mappa dei mattoncini avviene in tempo costante $O(1)$, e la modifica di un punatatore avviene in tempo costante, il costo complessivo di questa fuznione è $O(1)$. La costanza in questa fuznione è garantita dal camo fila nel mattonicno. Impostando il puntatore ad una `linkedList`, puntato da tutti i campi `fila` dei mattoncini appartenneti alla stessa fila, a nil si modificherà in tempo costante l'informazione per tutti i mattoncini. Questa soluzione sfrutta la presenza del garbage collector all'interno di Go che si ocuperà poi di liberare la memoria occupata dalla linkedList che non è più puntata dal programma.

3.6 Disponi Fila Minima

3.6.1 Sintesi

Come da specifiche la funzione deve essere in grado di disporre una tra le file minime (ovvero con il minor numero di mattoncini) tra due forme `alpha` e `beta` passate in ingresso. Per risolvere in maniera ottima il problema conviene rappresentare il gioco come un grafo, dove i vertici sono le forme e i mattoncini risultano gli archi che collegano due forme. In questo modo per stabilire il percorso minimo tra nodi si può adottare una visita per ampiezza partendo dal nodo `alpha` per fermarsi appena si incontra il nodo `beta` La nostra struttura dati `gioco` si presta perfettamente per questa rappresentazione, in particolare la mappa `forme` è una rappresentazione per incidenza del grafo.

3.6.2 Analisi

La complessita temporale per una visita in ampiezza, con una rappresentazione attraverso lista di incidenza sappiamo essere $O(n + m)$ dove n è il numero dei nodi del grafo mentre m il numero di archi. In questo algoritmo si possono evidenziare due casi separati, nel caso in cui le due forme `alpha` e `beta` sono diverse eseguirà unicamente una visita in ampiezza fermandosi al vertice `beta` target. Nel caso peggiore, se volessimo disporre la fila minima che collega la stessa forma `alpha` abbiamo la necessità di eseguire una visita in ampiezza a partire da tutti i nodi adiacenti ad `alpha` per poi confrontare i cammini trovati e scegliere il minore. Nel caso peggiore, nel quale il nodo target è collegato con tutti gli altri nodi del grafo bisognerà ripetere la visita in ampiezza n volte. La complessità temporale nel nei casi peggiori è $O(n * (n + m))$

Per la riuscita del risultato la funzione si avvale di variabili di supporto, in particolare `visitedArch` che è una mappa nella quale vengono memorizzati gli archi già visitati, nel caso peggiore questa mappa avra dimensione $O(n)$. Anche la fuznione di supporto `BFSCamminoMinimo` utilizza come variabili di supporto una coda ed una mappa. Nel caso peggiore le loro dimensioni possono raggiungere entrambe $O(n)$. In conclusione la complessità di tempo di questa fuzione è $O(n * (n + m))$ mentre la complessità di spazio è $O(n) + O(n) + O(n) = O(n)$.

3.7 Sotto Stringa Massima

3.7.1 Sintesi

Come da specofoche, la funzionde deve restituire la sottostringa comune massima tr adue stringhe `sigma` e `tau` passate come input. L'algoritmo si basa su un approccio dinamico. Inizia crea una matrice bidimensionale per memorizzare i risultati intermedi dei sottoproblemi. Successivamente, riempie questa matrice utilizzando un approccio bottom-up, calcolando la lunghezza della sottosequenza comune massima per tutte le combinazioni di prefissi delle due sequenze. Infine, ricostruisce la sottosequenza comune massima utilizzando la matrice risultante.

3.7.2 Analisi

La complessità temporale della funzione dipende dalla lunghezza delle due sequenze di input. Poiché l'algoritmo utilizza un approccio dinamico bottom-up, la sua complessità temporale è $O(n * m)$, dove n è la lunghezza della prima sequenza e m è la lunghezza della seconda sequenza.

La funzione utilizza una matrice bidimensionale per memorizzare i risultati intermedi dei sottoproblemi. La dimensione di questa matrice è determinata dalle lunghezze delle due sequenze di input. Pertanto, la complessità di spazio dell'algoritmo è $O(n * m)$, dove n è la lunghezza della prima sequenza e m è la lunghezza della seconda sequenza.

3.8 Indice cacofonia

3.8.1 Sintesi

Come da specifiche, la funzione di calcolo dell'indice di cacofonia, esamina i mattoncini adiacenti nella fila e confronta i nomi. Utilizzando un approccio iterativo, calcola l'indice di cacofonia sommando la lunghezza della sottosequenza comune massima tra ogni coppia di mattoncini adiacenti nella fila.

3.8.2 Analisi

La complessità temporale della funzione dipende dalla lunghezza della fila. Poiché la funzione deve confrontare i nomi dei mattoncini adiacenti, la complessità temporale è influenzata dalla lunghezza dei nomi dei mattoncini. Per ogni coppia di nomi viene seguita la funzione di sottostringa massima che ha complessita $O(s * t)$, dove s e t sono i nomi dei due mattoncini. Si potrebbe quindi concludere che la complessità totale sia $O(n(s * t))$, dove n è il numero di mattoncini della fila. In generale, se il numero dei mattoncini è molto più grande rispetto alle lunghezze dei loro nomi la complessità temporale può essere approssimata a $O(n)$.

La funzione richiede una quantità di memoria aggiuntiva, la quantità di spazio richiesta dipende principalmente dalla lunghezza dei nomi dei mattoncini. Richiamando la funzione sottostringa massima verrà occupato uno spazio pari a $O(s*t)$ dove s e t sono i nomi dei due mattoncini, spazio che può essere usato

dalla coppia successiva di mattoncini. Si può quindi dire che lo spazio occupato è costante rispetto alla dimensione della fila ma dipende dalla lunghezza media dei nomi dei mattoncini.

3.9 Costo

3.9.1 Sintesi

Come da specifiche la funzione `costo` deve valultare il costo associato alla trasformazione di una fila di mattoncini iniziale ad una fila che rispecchi la sequenza di forme indicata. Leggendo attentamente le specifiche, si può notare che per trovare il costo minimo significhi fare il minor numero di operazioni elementari sulla fila originaria, conservando il maggior numero di mattoncini possibili. Individuando quindi la sotto sequenza massima M tra la fila iniziale F e la fila finale F', possiamo definire il costo della trasformazione come la somma tra il numero di eliminazioni che portano la fila F ad essere uguale alla sotto sequenza M ed il numero di inseriemtni neccessari per passare dalla sottosequenza M alla fila F'. Per risolvere quindi questo problema ci si può avvalere della funzione precedentemente scritta per individuare la sotto stringa massima, oppurtunamente modificata.

3.9.2 Analisi

Per un analisi completa di questa funzione dobbiamo analizza in dettaglio tutte i vari passaggi della funzione. come primo passaggio si va a mappare la fila desiderata all'interno di una mappa di supporto. Questo passaggio ha complessita $O(n)$, dove n è la dimensione della fila. Successivamente si va a controllare per ogni coppia di elementi all'interno della sequenza di forme passata in input si controlla se esiste almeno un mattoncino che le collega e si salvano tutti i possibili mattoncini. Questo passaggio ha complessita $O(m * a)$ dove m è la dimensione della lista di forme passata in input e a la media del numero dei mattocini per ogni forma. Una volta individuati tutti i possibili mattoncini per unire le varie forme, individua i mattoncini in comune con la fila precedente e li utilizza, conservando l'ordine, per creare la nuova fila di mattoncini. Questo passaggio ha una complessità di $O(m * a)$ dove m è la dimensione della lista di forme e a il numero medio di mattoncini per forma. Successivamente esegue la funzione di sottosequenza massima tra due slice di string per due volte. Come precedentemente analizzato questa funzione ha complessità pari a $O(n * m)$, che nel caso peggiore coincide a $O(n * 2)$, quando la sottoswquenza massima è pari alla lista di forme. La complessità temporale totale è quindi $O(n) + O(m * a) + O(m * a) + O(n^2)$. Notiamo quindi come la complessita di questa funzione dipenda sia dalla lunghezza delle file in input ma anche dal numero di mattoncini nel gioco.

La funzione richiede una quantità aggiuntiva di memoria per memorizzare i nomi dei mattoncini e i risultati intermedi dei confronti. Si avvale di due mappe ausiliare di dimensione $O(n)$, dove n è il numero di mattoncini presenti

nel gioco, e di una matrice di dimensioni $O(m * a)$ dove m è la dimensione della sequenza di forme presa come input e a è il numero medio di mattoncini per ogni forma. La complessità di spazio totale è $O(n) + O(m * a)$