Laboratorio di Algoritmi e Strutture Dati

Relazione del progetto “Automi e segnali”

Filippo Vannini 28984A

1 Introduzione

Questo documento contiene la relazione del progetto “Automi e segnali”. Al suo interno saranno presentati e discussi l’approccio utilizzato per risolvere il problema dato, le scelte di modellazione e di implementazione con particolare concentrazione sulle strutture dati utilizzate e all’implementazione degli algoritmi con tanto di studio di complessità. Il file zip che verrà consegnato sul sito https://upload.di.unimi.it/ contiene:

- il file in formato .go nominato “28984A_filippo_vannini.go” contenente il codice sorgente del progetto;
- la relazione in un file formato .pdf nominato “28984A_filippo_vannini.pdf”;
- una cartella contenente dei file in formato .txt che rappresentano dei file di test ed anche di esempio per il progetto (i dettagli saranno spiegati nella sezione dedicata ai test).

2 Analisi del problema e considerazioni generali

Il problema chiede di creare un **piano** (di dimensioni non definite) al cui interno è possibile trovare **automi** e **ostacoli**. Gli automi in particolare sono in grado, dato un segnale di richiamo, di spostarsi verso quest’ultimo se esiste un percorso libero di distanza ***D(P(automa), (x, y))*** minima dalla posizione dell’automa al punto di richiamo.

Detto questo si può intuire che, le entità principali da dover modellare sono il piano, gli automi e gli ostacoli.

Per capire la risoluzione del problema dell’esistenza del cammino da un automa a un punto, così come anche la ricerca della tortuosità minima, è importante affermare che:

- per ricercare un percorso libero di lunghezza minima è necessario muoversi solo su al massimo 2 direzioni delle 4 disponibili da ogni punto (solo quelle verso il punto di destinazione). Questo perchè se anche si fa solo un passo nella direzione opposta, si allunga il cammino, non ricadendo più nella condizione necessaria che il cammino sia di lunghezza minima;
- il cammino è “circoscritto” all’interno di un rettangolo che ha come vertici il punto di partenza (la posizione dell’automa) e il punto di arrivo (posizione del richiamo o del punto per il quale si vuole trovare la tortuosità minima). È importante rimanere all’interno del rettangolo in quanto questo grantisce, a meno di spsotamenti con direzione opposta rispetto alla destinazione, che il percorso abbia lunghezza minima;
- il punto di partenza così come quello di arrivo non appartengono sicuramente ad un ostacolo per richiesta esplicita del problema. Questo ci permette di dire che, per tutti i punti che analiziamo del cammino, per determinare se siamo o meno capitati in un ostacolo, ci basta controllare se il punto giace sul suo perimetro ed è quindi inutile fare il controllo per verifcare se il punto appartiene all’area dell’ostacolo. Questo perché, se un punto si trova sicuramente all’esterno di un ostacolo, i punti vicini possono al massimo risiedere sul perimetro dell’ostacolo.

3 Modellazione del problema e strutture dati usate

Per poter modellare il problema è necessario analizzare ciascuna delle entità che necessitano di essere modellate. Prima però è opportuno definire come le posizioni, le coordinate o i punti del piano sono stati rappresentati in quanto vengono utilizzati sia dagli automi che dagli ostacoli all'interno della loro rappresentazione.

Ho quindi deciso di rappresentarli utilizzando una struttura chiamata **Punto** contenente due campi di tipo intero chiamati ***x*** e ***y.***

```
type Punto struct {
       x int
       y int
}
```

3.1 Automi

Ciasucn automa è caratterizzato dalle seguenti caratteristiche:

- presenta un nome univoco che lo identifica rappresentato da una stringa di lunghezza finita sull'alfabeto {0, 1};
- possiede una posizione rappresentata da un punto la cui rappresentazione è stata presentata precedentemente;
- non potrà mai giacere all'interno di alcun ostacolo presente nel piano.

Per ogni posizione è possibile che siano presenti più automi distinti contemporaneamente. Viene quindi utile usare una mappa per rappresentare gli automi scegliendo come chiave il loro nome e come valore associato la loro posizione.

```
automi map[string]Punto
```

Grazie a questo è possibile accedere in tempo costante alla loro posizione dato il nome; invece per accedere al nome data la posizione richiede nel caso peggiore di dover scorrere tutti gli automi presenti nel piano. Per ovviare il problema si potrebbe creare una seconda mappa che, per ogni posizione, associa una lista di nomi (si ricorda che per ogni posizione possono essere presenti più automi). Tuttavia, con l'aggiunta di questa rappresentazione, potrebbe risultare complicato l'aggiornamento della posizione dell'automa nel caso in cui tutti condividessero la medesima posizione: in questo caso, prima di trovare il nome corretto dell'automa che si deve spostare, si rischia di scorrere una lista contentente tutti gli automi andando ad aggiungere un'ulteriore inutile complessità alla risoluzione.

Ho quindi optato per tenere solo una mappa che per ogni nome salva una posizione in quanto l'unica operazione che ne beneficerebbe sarebbe **stato.**

3.2 Ostacoli

Ciascun ostacolo è caratterizzato dalle seguenti caratteristiche:

- è definito come l'insieme dei punti appartenenti a un rettangolo che ha come coordinate dei vertici due coppie di interi **(x0, y0)** e **(x1, y1)** che sono rispettivamente il vertice in basso a sinistra e quello in alto a destra;
- è possibile che un ostacolo si sovrapponi con altri presenti nel piano;

- un ostacolo non può essere creato se al suo interno ricade un automa già presente nel piano;
- gli ostacoli non possono essere spostati.

Per rappresentare un ostacolo si può utilizzare una struttura contenente due campi per ciasuno dei vertici (rappresentati come punti). Non è inoltre necessario salvare tutti i punti appartenenti ad un ostacolo perché soltanto l'operazione di **stato** ne beneficerebbe in termini di tempo e dunque sarebbe solo uno spreco di spazio.

Può invece, come vedremo nella sezione piano, tornare comodo salvare i punti presenti sul perimetro di un ostacolo per una maggiore efficienza dell'operazione di ricerca di un percorso.

```
type Ostacolo struct {
	puntoSX Punto
	puntoDX Punto
}
```

Per invece rappresentare l'elenco di tutti gli ostacoli ho optato per una slice di riferiementi agli ostacoli.

```
ostacoli []*Ostacolo
```

Grazie alla slice è possibile avere un inserimento a costo ***O*(1) ammortizzato** (questo perchè l'inserimento avviene sempre alla fine ed è ammortizzato perché, una volta occpuato tutto lo spazio a disposizione, la slice viene riallocata con conseguente spostamento di tutti i valori salvati precedentemente. Questa operazione è comunque abbastanza rarefatta nel tempo per come vengono gestite in GO e, quindi, si può considerare nullo il suo effetto nella complessità generale). Tuttavia il problema principale risiede nella ricerca: quando devo infatti determinare se un punto appartiene o meno ad un ostacolo, devo, nel caso peggiore, scorrere tutta la slice; il costo, quindi, sarà lineare rispetto al numero di ostacoli presenti nel piano. Ho cercato di trovare un modo per poter effettuare una ricerca binaria sulla slice così da avere costo logaritmico rispetto al numero di ostacoli, ma non sono riuscito a trovare una chiave rispetto a cui fare un ordinamento che mi permettesse di effettuare una ricerca in tempi migliori.

3.3 Piano

Il piano non ha una dimensione definita e può espandersi durante l'esecuzione del programma. Detto questo si intuisce che rappresentare il piano tramite una matrice risulterbbe molto inefficiente in termini di spazio (memorizzerei punti in cui non è presente niente) e in termini di tempo. Questo perché se vengono aggiunti automi o ostacoli all'esterno della porzione di piano rappresentata dalla matrice, dovrei allocare uno spazio maggiore e spostare tutti i punti già presenti al suo interno.

Se considero invece il piano come se fosse formato solo da automi e ostacoli lo potrei rappresentare sfruttando le rappresentazioni definite precedentemente per l'entità automa e l'entita ostacolo. Inoltre, per una questione di efficienza per l'operazione di ricerca di un percorso viene utile salvarsi nella rappresentazione di piano una mappa che contiene come chiavi tutti i punti appartenenti al perimetro di ciasun ostacolo in modo che, per ogni punto che devo controllare durante la ricerca di un cammino (che so sicuro può appartenere al massimo al perimetro) posso determinare in tempo costante se appartiene o meno ad un ostacolo senza dover così ricorrere all'uso della ricerca lineare.

La struttura finale per la rappresentazione di piano sarà quindi:

```
type Piano struct {
       automi map[string]Punto
       ostacoli []*Ostacolo
       perimetroOstacoli map[Punto]bool
}
```

4 Operazioni e analisi della loro complessità

In questa sezione della relazione verranno presentate le implementazioni scelte per le operazioni richieste con anche un'analisi della complessità in termini di tempo e spazio.

Da qui in avanti si assume che:

- il numero generale di automi nel piano verrà segnato come ***n***;
- il numero generale di ostacoli nel piano verrà segnato come ***m***;

4.1 stato

L'operazione di stato è implementata nel codice sorgente dal metodo che ne condivide il nome. Questa operazione richiede, data una coppia di interi che rappresentano una posizione nel piano, di restituire **A** se in quel punto è presente almeno un automa, **O** se appartiene ad un ostacolo, **E** se quel punto non appartiene né ad un automa, né ad un ostacolo.

Nell'implementazione viene usato il metodo **controllaOstacoli** che, dato un punto in input, determina se appartiene o meno ad un osatcolo.

Prima di cimentarci nel calcolo della complessità occorre fare qualche precisazione:

- in **controllaOstacoli,** se il punto dovesse ricadere sul perimetro di un ostacolo, grazie alla ridondanza usata per rappresentare il piano, la ricerca avrebbe un costo costante;
- sempre in **controllaOstacoli,** se il punto dovesse ricadere all'interno di un ostacolo, dovrei, nel caso peggiore, scorrere tutta la lista degli ostacoli;
- se il punto invece dovesse appartenere a uno o più automi dovrei nel caso peggiore scorrere tutta la mappa degli automi;
- se il punto non dovesse appartenere né agli ostacoli e neppure agli automi, dovrei, nel caso peggiore scorrere tutti gli automi e tutti gli ostacoli.

Tempo: nel caso peggiore, quando il punto è vuoto, occorre scorrere tutta la mappa degli automi e tutta la slice degli ostacoli dunque, il costo in termini di tempo risulta ***O*(n + m).**

Spazio: vengono utilizzati una variabile locale e due paremtri in input, dunque, il costo in termini di spazio sarà ***O*(1).**

4.2 stampa

L'operazione di stampa viene implementata nel sorgente dal metodo con il medesimo nome. Questa operazione, una volta chiamata, stamperà a schermo prima l'elenco di tutti gli automi e successivamente l'elenco di tutti gli ostacoli. Il layout di stampa viene descritto nelle specifiche di implementazione.

Questo metodo utilizza altri due metodi di supporto:

- **stampaAutomi** che, data una stringa in input, stampa l'elenco di tutti gli automi il cui nome ha come prefisso la stringa in input. Se la stringa dovesse essere vuota allora stampa l'elenco di tutti

gli automi (questa decisione è stata presa per evitare di avere funzioni molto simili fra loro). Bisogna dunque iterare nel peggiore dei casi lungo tutti gli automi e intanto confrontare tutti i prefissi. Per controllare se il nome di un automa inizia con il prefisso, ho utilizzato il metodo **hasPrefix** del package **strings** che in tempo proporzionale alla lunghezza del prefisso, mi consente di determinare se è presente nel nome. Considerando quindi ***l*** come lunghezza del prefisso, per ogni automa avrò un costo in termini di tempo pari a ***l.*** Dunque, il costo in termini di tempo è ***O*****(l x n);**

- **stampaOstacoli** che stampa un elenco di tutti gli ostacoli presenti nel piano. È necessario quindi iterare lungo tutta la slice degli ostacoli, dunque, il costo in termini di tempo è ***O*****(m).**

Tempo: il costo in termini di tempo di questa operazione dipende dai due metodi di supporto citati sopra e quindi sarà ***O*****(l x n + m)** (considerando **fmt.Println()** con costo costante).

Spazio: non vengono utilizzate variabili ausiliarie, dunque il suo costo sarà ***O*****(1).**

4.3 automa

L'operazione automa viene implementata nel sorgente dal metodo **aggiungiAutoma**. Questa operazione, data una coppia di interi e una stringa su alfabeto binario, permette di creare l'automa con quel determinato nome nel punto indicato dalla coppia di interi. Se il punto appartiene ad un ostacolo, non viene effettuato l'inserimento; se invece è già presente l'automa con lo stesso nome, viene aggiornata la sua posizione con quella nuova (sempre se non appartiene ad alcun ostacolo).

Nell'implementazione viene utilizzato il metodo **controllaOstacoli.**

Dunque, prima di inserire l'automa o modificare la posizione di uno già esistente, **controllaOstacoli** determina se il punto ricade o meno all'interno di almeno uno di essi. In caso negativo si procede con l'inserimento/aggiornamento dell'automa nella mappa. Le operazioni che permettono di Inserire e aggiornare la mappa hanno entrambe un costo di tempo ***O*****(1) ammortizzato**.

Tempo: nel caso peggiore in **controllaOstacoli** devo comunque scorrere tutta la slice degli ostacoli, dunque, il costo in termini di tempo sarà ***O*****(m).**

Spazio: vengono utilizzati una variabile locale e 3 parametri in input, quindi, il costo in termini di spazio sarà ***O*****(1).**

4.4 ostacolo

L'operazione viene implementata nel sorgente dal metodo **aggiungiOstacolo.** Questa operazione, date due coppie di interi, richiede che venga creato un ostacolo avente, come vertice in basso a sinistra la prima coppia e come vertice in alto a destra la seconda coppia.

Nell'implementazione viene usato il metodo **controllaAuotmi** che, dati due punti in input, verifica se almeno un automa ricade fra di loro.

Per il calcolo della complessità di questo metodo occorre studiare ciascuna parte nel dettaglio:

- prima di inserire un ostacolo è necessario dover scorrere tutto l'elenco degli automi. Questo perché, se anche solo un automa dovesse ricadere nella sua area, non sarebbe possibile l'inserimento (questo avviene grazie a **controllaAutomi**);
- se viene determinato che non è presente alcun automa all'interno dell'area di definizione dell'ostacolo, si procede con il suo inserimento nella slice che avviene con costo ***O*****(1) ammortizzato**;
- successivamente, a seguito dell'inserimento di un ostacolo, occorre salvare anche tutti i punti del

suo perimetro nella mappa **perimetroOstacoli** del piano. Il costo di questa operazione è proporzionale alla dimensione dell'ostacolo (ritenendo le operazioni di inserimento in una mappa con costo ***O*(1) ammortizzato**). In particolare, considerando **(x0, y0)** e **(x1, y1)** le coordinate dei vertici dell'ostacolo, le dimensioni di quest'ultimo saranno: **w = |x1 - x0|** e **h = |y1 - y0|.**

Tempo: nel caso peggiore (l'inserimento avviene con successo) devo scorrere tutta la mappa degli automi e successivamente devo inserire tutti i punti appartenenti al perimetro dell'ostacolo. Dunque, il costo in termini di tempo sarà ***O*(n + w + h).**

Spazio: vengono utilizzate 2 variabili locali e 4 parametri in input, quindi, il costo in termini di spazio sarà ***O*(1).**

4.5 posizioni

L'operazione posizioni viene implementata nel sorgente dal metodo che ha lo stesso nome. Questa operazione, data una stringa in input, richiede che venga stampato un elenco di tutti gli automi il cui nome ha come prefisso la stringa data. Il layout di stampa dell'elenco è definito, come per l'operazione stampa, dalle specifiche di implementazione.

Per stampare l'elenco degli automi, questo metodo si avvale dell'utilizzo del metodo **stampaAutomi** già presentato in precedenza. Anche in questo caso il costo, nel caso peggiore, sarebbe come quello in cui venga richiesto di stampare un elenco di tutti gli automi a prescindere da un dato prefisso. Questo perché nel caso in cui tutti gli automi possiedano un nome che inizi con il prefisso dato, bisognerebbe scorrere tutta la mappa.

Tempo: il costo in termini di tempo di questo metodo dipende dal metodo **stampaAutomi** che come detto prima, nel caso peggiore, scorre tutta la mappa degli automi e dal metodo **hasPrefix**. Dunque, il costo in termini di tempo sarà ***O*(l x n).**

Spazio: viene passato solo un parametro in input al metodo e **stampaAutomi** abbiamo già detto avere costo costante. Dunque, il costo complessivo in termini di spazio sarà ***O*(1).**

4.6 visitaPercorsi

Questo metodo serve per ricercare un percorso libero ed eventualmente, all'occorrenza, anche con tortuosità minima. Viene utilizzata dalle implementazioni delle operazioni **richiamo** e **tortuosità.**

Questo metodo, dati due parametri di tipo **Punto** e un parametro booleano, determina:

- la sola esistenza di un cammino **libero**, ovvero privo di ostacoli, avente distanza minima, dalla posizione dell'automa (secondo punto in input), al punto di destinazione (primo punto in input). Questo viene effettuato solo se il parametro booleano contiene come valore **false.** In questo caso, il metodo restituisce **true** se esiste almeno un percorso libero avente distanza minima e -1 come output della tortuosità che verrà scartato dall'operazione **richiamo**. Altrimenti, se non esiste alcun percorso libero, restitusice **false** e la tortuosità minima trovata finora come valore per la tortuosità che verrà comunque scartato dall'operazione di **richiamo.**
- la tortuosità minima fra tutti i percorsi liberi se il parametro booleano contiene come valore **true.** In questo caso, il metodo restituirà sempre **false** per l'esistenza del cammino in quanto la pila deve svuotarsi per garantire di aver visitato tutti i percorsi. Questo valore verrà poi scartato dall'operazione **tortuosità.** Restituisce anche il valore della tortuosità minima calcolata visitando

tutti i percorsi liberi. Nel caso in cui non sia presente alcun percorso libero, assieme a **false**, verrà restituito **-1** come richiesto.

Questa operazione è la più importante e allo stesso tempo la più onerosa in termini di risorse utilizzate se presa in considerazione singolarmente.

Per implementerla ho pensato di utilizzare la **DFS** con un po' di ottimizzaizioni per migliorare la sua efficacia. Questo perché, la condizione determinante per l'efficienza di questo metodo è quella di ridurre il più possibile il numero di nodi da visitare che, grazie a qualche accorgimento, è possibile ridurre drasticamente. Di seguito è presente una breve descrizione del suo funzionamento e delle modifiche apportate.

I) Quando si vuole soltanto ricercare l'esistenza di un percorso libero, per evitare di prolungare inutilmente la ricerca, appena si arriva alla destinazione, il metodo termina. Se invece si vuole ottenere la tortuosità minima, una volta arrivati al punto di arrivo, se la tortuosità del percorso appena visitato risulta essere 1 sono sicuro che non può essere fatto di meglio. Questo è possibile perché se il punto di arrivo e quello dell'automa dovessero condividere anche solo uno dei due campi e il percorso dovesse essere libero, verrebbe trovata tortuosità pari a 0 immediatamente.

II) Il punto cruciale della miglioria risiede tuttavia in come i nuovi punti vengono aggiunti alla pila per poi venire visitati:

- viene determinata la direzione del passo (servirà per il calcolo della tortuosità);
- viene calcolata la distanza fra la destinazione e il nuovo punto appena "creato" e fra la destinazione e il punto in cui si trova la visita. Successivamente se la nuova posizione giace sul perimetro di un ostacolo oppure la nuova distanza è maggiore o uguale di quella corrente, evito di aggiungerlo alla pila e passo ad analizzare direttamente il punto successivo senza effettuare ulteriori passaggi;
- viene poi calcolata la nuova tortuosità con: la direzione del passo, la direzione per arrivare al punto in cui si trova la visita e la tortuosità ottenuta per raggiungerlo. Se la nuova tortuosità è già maggiore o uguale di quella minima trovata finora, so che non posso fare di meglio e continuo con l'analizzare il punto successivo senza effettuare ulteriori passaggi;
- viene poi prelevato, da una mappa, il valore minimo della tortuosità ottenuto finora per il nuovo punto (se già stato raggiunto) e, se devo trovare la tortuosità minima, viene controllato se è già stato visitato con una tortuosità minore di quella calcolata attualmente; in caso affermativo analizzo il punto successivo. Questo perché, se rivisito il punto con una tortuosità maggiore o uguale, sicuramente non potrò trovare un percorso con tortuosità migliore partendo da lui. Se invece non devo calcolare la tortuosità, controllo se il punto è già stato visitato; in caso affermativo analizzo un altro punto. Questo perché se ho già visitato quel punto e non è ancora terminata la ricerca, significa che non ho trovato da lì un percorso libero e quindi posso evitare di visitare nuovamente tutti i punti che avrei visitato partendo da lui;
- viene, infine, aggiornata la tortuosità del nuovo punto nella mappa dei punti visitati e viene inserito il punto nella pila se e solo se tutti i controlli precedenti vanno a buon fine.

Per calcolare la complessità di questo metodo conviene studiare ogni sua parte singolarmente:

I) Per la DFS è necessaria la presenza di uno stack. In questo stack vengono inserite delle informazioni per i passi che vengono fatti durante la visita che possono essere al massimo 2 per ogni punto: uno con direzione

verticale o uno orizzontale. Le informazioni utili per ciascun passo sono rappresentate dalla seguente struttura:

```
type InfoPasso struct {
      puntoCorrente Punto
      direzione rune
      tortuositaCorrente int
}
```

In particolare:

- **puntoCorrente** contiene il punto in cui si trova la visita;
- **direzione** contiene un carattere: ***v*** se il punto è stato raggiunto da un passo verticale dal suo vicino, ***o*** se il punto è stato raggiunto da un passo orizzontale oppure **carattere vuoto** se il punto è quello di partenza;
- **tortuositaCorrente** contiene il valore della tortuosità trovata finora per raggiungere il punto corrente. In particolare, la tortuosità viene calcolata confrontando la direzione del passo con cui il punto corrente è stato raggiunto con quella del passo per raggiungere un suo punto vicino. Nel caso in cui siano diverse aggiungo 1 alla tortuosita per il punto vicino che devo aggiungere, altrimenti la lascio invariata; questa operazione si può considerare avere costo***O*(1).**

Tutti i metodi, che implementano le tipiche operazioni per lo stack richiedono tempo ***O*(1)**, mentre l'operazione di **Push** richiede un tempo ***O*(1) ammortizzato** quando viene chiamata; questo perché lo stack è rappresentato da una slice (per la spiegazione, occorre fare riferimento alla sezione 3.2). Per determinare invece l'altezza massima dello stack nel caso peggiore, bisogna innanzitutto considerare che ogni punto dell'area di ricerca potrebbe essere presente al suo interno. Siccome per la tortuosità devo controllare tutti i percorsi possibili (che sono in questo caso $\mathbf{2^D}$), nel caso peggiore avrò che la pila dovrà contenere tutti i punti presenti nell'area di ricerca (alcuni verranno anche inseriti nuovamente al suo interno in base anche alla condizione di tortuosità). Viene quindi utile definire **(x0, y0)** come punto di partenza della ricerca, mentre **(x1, y1)** come punto di arrivo; allora possiamo calcolare che la dimensione dell'area di ricerca sarà **w x h** dove **w = |x1 - x0|** e **h = |y1 - y0|.** Dunque, nel caso peggiore, lo spazio occupato dello stack sarà nell'ordine di ***O*(w x h) (è un caso limite superiore e vengono considerati anche i casi in cui un punto può essere inserito piu volte).**

II) Per controllare se un percorso è libero, man mano che i suoi punti vengono visitati, è necessario controllare se i loro vicini appartengono o meno al perimetro di un ostacolo. Qui entra in gioco l'efficienza dovuta al salvataggio di tutti i punti appartenenti al perimetro di un ostacolo: grazie a questa implementazione, si può evitare, per ogni vicino che viene raggiunto, di dover scorrere la slice di tutti gli ostacoli per determinare se appartiene o meno ad un ostacolo. Grazie alla mappa dei punti del perimetro, infatti, è possibile fare ciò in tempo costante evitando così di peggiorare la complessità ulteriormente.

Tempo: per calcolare il costo in termini di tempo si può considerare come caso peggiore quello in cui calcolo la tortuosità (perché visito tutti i percorsi). Assumiamo ora di visitare tutti i punti presenti nell'area e di tenerli tutti nello stack (caso che non si può mai verificare in quanto non avrò mai tutti i punti contemporaneamente nello stack, ma che tengo come limite superiore per una maggiore facilità del calcolo della complessità). Considerando infine le operazioni dello stack, così come il calcolo della nuova tortuosità e

dei nuovi vicini con costo costante il costo in termini di tempo sarà ***O*(w x h).**

Spazio: vengono utilizzate 13 variabili locali, 3 parametri in input e una mappa che nel caso peggiore possiamo dire conterrà tutti i punti dell'area di ricerca, occupando spazio pari a ***O*(w x h).** Viene inoltre utilizzato lo stack (che consideriamo contenga nel caso peggiore tutti i punti dell'area contemporaneamente al suo interno anche più volte) che insieme alla mappa determina lo spazio occupato dal metodo ovvero ***O*(w x h + w x h) = *O*(w x h).**

Per risolvere questo problema della ricerca si potevano usare altri metodi, magari anche più efficienti (A*, Dijsktra adattato per la tortuosità, programmazione dinamica…), tuttavia ho preferito utilizzare la **DFS** in quanto mi permette di trovare in modo semplice tutti i cammini liberi da un punto a un altro. Questo metodo tuttavia pecca e rallenta molto quando l'area di ricerca è molto ampia e ricca di ostacoli. Infatti, un grande fattore in gioco che ne influenza le prestazioni è la densità degli ostacoli all'interno dell'area di ricerca:

- se non sono presenti ostacoli lungo il cammino, essendo una DFS, quest'ultimo verrà trovato al primo tentativo e sarà anche quello con tortuosità minima;
- se invece sono presenti abbastanza ostacoli, la DFS sarà costretta a interrompere più spesso la ricerca di un cammino e dunque richiede di visitare molti più punti prima di poter trovare un percorso libero.

È invece efficiente quando all'interno dell'area di ricerca, sono presenti pochi ostacoli sparsi che non limitano troppo la ricerca del percorso specialmente lungo i bordi dell'area.

Infine, ho deciso di non implementare la DFS tramite la ricorsione perché, nel caso in cui l'area di ricerca sia molto grande, i tempi di ricerca sarebbero potuti essere molto più alti. Ho quindi, per questo, deciso di utilizzare una versione iterativa con uso di uno stack esplicito.

4.7 richiamo

L'operazione **richiamo** viene implementata dal metodo che presenta lo stesso nome e nella sua implementazione utilizza il metodo **visitaPercorsi** descritto precedentemente.

Questa operazione, dati in input una coppia di valori interi e una stringa, permette di spostare di posizione tutti gli automi che hanno come prefisso nel nome la stringa data e che hanno anche un percorso libero di distanza minima dalla loro posizione corrente al punto rappresentato dalla coppia di interi passata in input. Se il punto del richiamo dovesse essere all'interno di un ostacolo, non verrebbe eseguita questa operazione. È quindi necessario dover scorrere tutta la slice degli ostacoli prima di effettuare le altre operazioni.

Successivamente vengono iterati tutti gli automi della mappa e per tutti quelli, il cui nome inizia per il prefisso dato, viene calcolata la loro distanza dal punto di richiamo. Questa operazione, considerando costante il calcolo della distanza e considerando come ***l*** la lunghezza del prefisso, mi viene a costare ***O*(l x n)** tramite anche l'uso del metodo **hasPrefix.**

Viene infine determinata la distanza minima fra tutte quelle presenti (nel caso in cui tutti gli automi hanno il nome che inizia con lo stesso prefisso, le devo scorrere tutte) e successivamente, scorrendo tutta la mappa degli automi, se la distanza dell'automa dal punto di richiamo è uguale a quella minima, cerco il percorso libero tramite **visitaPercorsi**. In caso affermativo aggiorno la posizione dell'automa.

Tempo: il tempo complessivo dipende dal numero di ostacoli, dal numero di automi e da **visitaPercorsi**. Nel caso peggiore in cui tutti gli automi hanno il nome che inizia con il prefisso dato e hanno tutti distanza

minima, devo cercare ***n*** percorsi liberi. Dunque, il costo totale in termini di tempo sarà ***O*****(m + n x l + n x w x h).** Considerando che **m** e **n x l** possono essere considerati trascurabili rispetto a **n x w x h,** il costo complessivo in termini di tempo sarà ***O*****(n x w x h).**

Spazio: vengono usate 3 variabili locali di cui 2 occupano spazio costante mentre una è una mappa che nel caso peggiore contiene la distanza per tutti gli automi e 3 parametri in input; inoltre viene utilizzato il metodo **visitaPercorsi** che usa spazio ***O*****(w x h).** Lo spazio totale utilizzato sarà ***O*****(n + w x h).**

4.8 esistePercorso

L'operazione **esistePercorso** viene implementata dal metodo che presenta lo stesso nome. Nella sua implementazione utilizza il metodo **visitaPercorsi** per determinare se esiste o meno un percorso libero.

Questa operazione, dati in input una coppia di valori interi e una stringa, determina se esiste un percorso libero di lunghezza ***D(P(automa), (x, y))*** dalla posizione dell'automa fino alla posizione rappresentata dalla coppia di interi in input. In particolare, se il punto di destinazione appartiene ad un ostacolo o se non esiste l'automa, viene restituto **false**. Se non ci sono ostacoli nel piano, viene invece restituito **true** perché, in questo caso, esisterà sempre un percorso libero. Se invece nessuna delle precedenti condizioni risulta essere verificata, viene chiamato il metodo **visitaPercorsi** che cercerà la presenza di un percorso libero. Successivamente in **esistePercorso** verrà ignorato il parametro intero restituito da **visitaPercorsi**.

Tempo: devo scorrere tutti gli ostacoli per determinare se il punto di destinazione appartiene a uno di essi e devo usare **visitaPercorsi** per determinare l'esistenza di un percorso libero. Il tempo quindi sarà ***O*****(m + w x h) = O(w x h).**

Spazio: vengono usati 3 parametri in input e 4 variabili locali che occupano spazio costante; viene inoltre utilizzato il metodo **visitaPercorsi** che ne determina il costo in termini di spazio che sarà ***O*****(w x h).**

4.9 tortuosità

L'operazione **tortuosità** viene implementata dal metodo **tortuosita.** Nella sua implementazione utilizza il metodo **visitaPercorsi** per determinare il percorso con tortuosità minima**.**

Questa operazione, dati in input una coppia di valori interi e una stringa, restituisce la tortuosità minima del cammino per andare dal punto in cui giace l'automa di cui viene dato il nome al punto che viene rappresentato dalla coppia di interi passata in input. In particolare, se l'automa non esiste o il punto di destinazione appartiene ad un ostacolo, viene restituito -1. Se invece non sono presenti ostacoli nel piano, viene restituito 1 se entrambi i punti hanno campi diversi, 0 invece se almeno uno dei due campi è uguale fra i due punti. Se nessuna delle condizioni precedenti dovesse avverarsi, viene chiamato il metodo **visitaPercorsi** che restituirà la tortuosità minima fra tutti i cammini liberi. In questo caso, viene ignorato il parametro booleano che restituisce **visitaPercorsi**.

Tempo: devo scorrere tutti gli ostacoli per determinare se il punto di destinazione appartiene a uno di essi e devo usare **visitaPercorsi** per determinare la tortuosità minima. Il tempo quindi sarà ***O*****(m + w x h) = O(w x h).**

Spazio: vengono usate 3 variabili locali e 3 parametri in input che occupano spazio costante; inoltre viene utilizzato il metodo **visitaPercorsi** che ne determina il costo in termini di spazio che sarà ***O*****(w x h).**

5 Test ed esempi

Come detto nell'introduzione e come richiesto da traccia, vengono forniti nel file .zip una serie di test che

fungono da esempi, i quali vanno a mirare sulle criticità del sorgente e delle sue impelementazioni e anche sulla correttezza dell'output dei metodi. In particolare, all'interno della cartella test, saranno presenti:

- n file di input chiamati "input_*i*.txt" dove *i* rappresenta un numero intero
- n file di output chiamati "output_*i*.txt" dove *i* rappresenta un numero intero

L'indice serve per capire a quale input corrisponde ciascun output; a parità di indice c'è una corrispondenza fra input e output.

Tutti i test sono stati creati a "mano" con l'aiuto di GeoGebra per poter ricostrutire visivamente il piano e osservare se il loro risultato fosse corretto.

Di seguito è presente un elenco con una breve descrizione per ogni test/esempio:

1. Questo test serve per verificare il corretto funzionamento di **esistePercorso** e di **tortuosita;** in particolare, viene calcolata l'esistenza di un percorso e la tortuosità minima aggiungendo sempre più ostacoli lungo il percorso dell'automa verficando così il loro comportamento per ogni situazione nuova che si viene a creare.
2. Questo test è un caso in cui l'implementazione di **visitaPercorsi** risulta essere più efficiente soprattutto nel caso in cui si voglia calcolare la tortuosità minima. Gli ostacoli in particolare non intralciano il perimetro delle aree di ricerca dei percorsi consentendo alla DFS di trovare subito il percorso ottimale.
3. Questo test è un caso limite per l'esistenza dei percorsi, per il richiamo e anche per il calcolo della tortuosità: l'area di ricerca è abbastanza grande e la densità degli ostacoli nel piano, ma soprattutto nell'area, è molto alta costringendo la DFS a dover visitare più percorsi prima di poter arrivare alla destinazine (se possibile).
4. Questo test mostra un caso meno immediato per la ricerca dei percorsi e per il calcolo della tortuosità minima. Gli ostacoli infatti si trovano lungo il perimetro dell'area di ricerca che forza la DFS a cambiare più volte il percorso senza poter andare direttamente verso l'obiettivo con un'unica visita.
5. Questo test è un altro caso limite per **esistePercorso, tortuosita** e **richiamo** in quanto il piano è molto vasto, così come sono molto vasti pure gli ostacoli; inoltre, quest'ultimi, sono pure molto addensati (specialmente attorno all'automa 100011), constringendo tutti gli automi a dover visitare più percorsi prima di poter arrivare alla destinazione. Dato che è un caso limite per la DFS, il tempo di esecuzione rispetto agli altri test sarà abbastanza più alto.
6. Questo test è un modo interessante e divertente per testare la corretta funzionalità del calcolo della tortuosità minima: l'automa, infatti, dovrà effettuare un percorso a zig-zag prima di poter arrivare alla destinazione.