Laboratorio di Algoritmi e Strutture Dati
Relazione progetto "Piastrelle Digitali"

Gabriele Sarti 975884

10 luglio 2024

1 Introduzione

In questo documento, vengono presentate le diverse strutture dati utilizzate per modellare il problema, le scelte implementative adottate nel codice sorgente e l'analisi dei costi associati alle varie operazioni. L'obiettivo è fornire una visione chiara e dettagliata delle decisioni tecniche e delle loro implicazioni in termini di efficienza e prestazioni.
Sul sito `https://www.upload.di.unimi.it` e' stato caricato un file zip, come da specifica, contenente:

- il file go: "975884_sarti_gabriele.go" contenente il programma
- la relazione (questo file): "975884_sarti_gabriele_relazione.pdf"
- esempi di input/output nelle rispettive cartelle "inFiles" e "outFiles", inoltre nella cartella "outFilesMiei" verranno inseriti gli output dopo esecuzione dello script bash
- uno script bash "test.sh" per eseguire i test con il comando: "bash test.sh"

2 Modellazione del problema e considerazioni generali

2.1 Piano e piastrelle

Il problema puo' essere modellato come un grafo non orientato, sia un grafo $G = (V, E)$ il piano, si ha:

- V: insieme dei vertici, un vertice $v \in V$ e' una $piastrella(a, b)$ di lato unitario
- E: insieme dei archi, $E \subseteq V \text{x} V$, un arco $e \in E$ e' il punto in comune tra le due piastrelle (o vertice non del grafo ma del piano bidimensionale), quindi dato un arco $(x, y) \in E$ il vertice x e' circonvicino al vertice y e viceversa

Esempio: data la piastrella (0,0), e la piastrella (1,1), sono i vertici del grafo, il punto (1,1) è arco del grafo, perché due piastrelle hanno il punto (1,1) in comune, quindi la piastrella (0,0) e' circonvicina alla piastrella (1,1).
Inoltre gli archi non sono pesati e nessun arco è orientato (infatti quando si definisce un blocco di una piastrella si può partire da qualsiasi vertice).

2.2 Blocco e blocco omogeneo

Per quanto concerne il blocco e blocco omogeneo per poter calcolare e la somma delle intensità delle piastrelle contenute nel blocco omogeneo e non di appartenenza di $piastrella(x, y)$ bisognera' fare una visita sul grafo ad esempio effettuare una ricerca in profondità o DFS (Depth-First Search) in dettaglio:

1. fissato un vertice/nodo sorgente ossia la piastrella(x,y) del blocco di appartenenza
2. marca il nodo iniziale come visitato

3. esploro i nodi adiacenti non ancora visitati, ossia le piastrelle circonvicine alla piastrella(x,y) non ancora visitati
4. continua ricorsivamente fino a quando si raggiunge un nodo che non ha nodi adiacenti non ancora visitati.

2.3 Propagazione colore su piastrella e blocco

Come da specifica, data una piastrella(x,y) per propagare il colore devo analizzare tutte le regole e trovare la prima regola tale che l'intorno della piastrella, ossia l'insieme di tutte le piastrelle circonvicine ad p, contenga almeno le occorrenze dei colori specificati nella regola che sto considerando.
Invece per propagare il colore su un blocco per ogni piastrella del blocco si applica la prima regola applicabile a p rispetto all'intorno di p, cosi' come si presenta all'inizio della procedura di propagazione sul blocco, ossia qualsiasi modifica eseguita su una piastrella dovuta alla propagazione del blocco non influisce sugli intorni delle piastrelle ad esse circonvicine.

2.4 Pista e lunghezza

Una pista puo' essere modellata come un cammino in un grafo da una piastrella x di partenza e una y di arrivo, inoltre la lunghezza di una pista puo' essere calcolata come la lunghezza di un cammino+1 (ossia il numero di archi+1). Per calcolare la pista minima, ossia la pista con lunghezza del cammino minima, si puo' modellare come problema di "trovare il cammino minimo tra due vertici", avendo un grafo non pesato sara' possibile quindi applicare la BFS.
La BFS (o Breadth-First Search) esplora i nodi vicini (adiacenti) al nodo di partenza prima di passare ai nodi più lontani, garantendo così che il primo percorso trovato verso un dato nodo sia il più corto in termini di numero di archi.

3 Strutture dati

3.1 Introduzione

Questa sezione contiene l'analisi delle strutture dati scelte e la stima dei costi di spazio della loro rappresentazione e alla stima dei tempi di esecuzione delle loro operazioni.

3.2 Posizione

Ho implementato una struttura **posizione** che rappresenta una coppia di coordinate (x, y), i due campi essendo implementati con un tipo primitivo **int**, lo spazio occupato dalla struttura sara' $O(1)$.

3.3 Piastrella

La parte fondamentale del **piano** e' la piastrella, essa viene rappresentata da una struttura **Piastrella** contentente:

- **x, y** due campi **int**, le coordinate del vertice in basso a sinistra
- **colore** campo **string**, rappresentante il colore della piastrella
- **intensita** campo **int**, rappresentante l'intensita' della piastrella (0 se spenta)

Lo spazio occupato dalla struttura piastrella sara' di $O(1)$, in quanto msemorizza solo 3 interi e una stringa.

3.4 Regola

3.4.1 Termine

Come spiegato nella sezione 1.1.3 una regola di propagazione e' del tipo: $k_1\alpha_1 + k_2\alpha_2 + ... + k_n\alpha_n \leftarrow \beta$, chiamiamo **termine**, un termine della regola ossia un generico $k_n\alpha_n$, allora creo una struttura dati **termine** costituito da:

- **k** un campo **int**, ossia il numero di piastrelle che devono avere il colore **alpha**
- **alpha** un campo **string**, il colore

Lo spazio occupato dalla struttura **termine** sara' di $O(1)$, in quanto msemorizza solo un intero e una stringa.

3.4.2 Regola

Come spiegato spiegato precedentemente una regola di propagazione e' del tipo: $k_1\alpha_1 + k_2\alpha_2 + ... + k_n\alpha_n \leftarrow \beta$, quindi la regola viene rappresentata da una struttura **regola** formata da:

- **parti**: un elenco di **termini** rappresentato come una **slice** di **termini**
- **nuovoColore**: di tipo **string** ossia β il nuovo colore per la piastrella in esame quando la regola e' applicata
- **consumo**: di tipo **int** che tiene traccia di tutte le volte che la regola è stata applicata.

Lo spazio occupato dalla struttura **regola** sara' di $O(t)$, dove t e' il numero di termini della regola (gli altri campi non influiscono sulla complessita'). Ho utilizzato una slice di termini della regola in quanto e' sia intuitivo che veloce il calcolo per poter applicare la regola ad una piastrella basata sul suo intorno; anche se la mappa era piu' efficiente non e' una implementazione aduguata infatti come vedremo in seguito la stampa della regola non dovra' essere randomica rispetto ai termini di essa.

3.5 Piano

La struttura dati **piano** (grafo) rappresenta l'intero sistema di piastrelle e regole all'interno del programma, e' costituito da:

- **piastrelle** di tipo **map[posizione]*Piastrella**: dove memorizzo tutte le piastrelle *accese* del piano, come chiave uso la **posizione** del piano (sezione 3.2), come valore salvo il *puntatore* ad una piastrella (tipo **Piastrella** sezione 3.3)
- **regole** di tipo ***[]Regola**: dove memorizzo tutte le regole di propagazione del piano in ordine grazie alla slice

Salvo nel piano soltanto le piastelle accese, quelle spente vengono considerate non esistenti nel piano, quanto una piastrella viene accesa viene inserita nella mappa.

Alcune considerazioni in merito alla rappresentazione del grafo/piano scelto: non ho scelto una rappresentazione "didattica" del grafo per le seguenti ragioni:

- **lista di adiacenza/incidenza**: anche se ha il vantaggio di restituire i vertici adiacenti in tempo costante come vedremo in seguito anche questa rappresentazione trova i vertici/pistrelle adiacenti in tempo costante, ed e' quindi uno **spreco in termini di spazio memorizzare una informazione inutile**. Inoltre mentre l'inserimento di un nuovo vertice con la rappresentazione scelta impiega $O(1)$ di contro implementando una lista di adiancenza ed essendo obbligati ad usare **append** il tempo sarebbe $O(1^*)$
- **matrice di adiacenza/incidenza**: anche se sembra essere una rappresentazione "valida" leggendo la specifica si evince che **il piano potrebbe essere infinito** dunque potrebbe essere molto sparso quindi anche in questo caso (in modo piu evidente) uno spreco di spazio ($O(n^2)$ con n numero di vertici)

Il vantaggio principale della rappresentazione scelta e' **l'efficienza in termini di spazio e tempo**: l'accesso, la modifica e l'inserimento di un vertice/piastrella del grafo/piano viene eseguito in tempo costante grazie alle mappe (l'accesso e' molto simile alle matrici di adiacenza/incidenza grazie alla chiave **posizione** sezione 3.2). Sia m il numero di regole ed n il numero di piastrelle del piano lo **spazio totale occupato dalla struttura piano** sara' di $O(n) + O(m)$.

4 Funzioni e costi

4.1 restituisciPiastrella

Restituisce la $piastrella(x, y)$ (puntatore) dato un x e y oppure nil se la piastrella non esiste (ossia spenta e senza colore), utilizzo l'accesso alle mappe.
Tempo: $O(1)$, vengono eseguiti un numero costante di accessi.
Spazio: $O(1)$, non uso strutture dati aggiuntive.

4.2 piastrelleCirconvicine

Restituisce le piastrelle circonvicine dato un x e y. L'algoritmo cicla sulle 8 direzioni possibili trovando le piastrelle circonvicine presenti e le aggiunge ad una mappa **vicini**.
Tempo: $O(1)$, il for-range esegue 8 iterazioni, con un numero costante di accessi a mappa, e al massimo una aggiunta a mappa $O(1)$.
Spazio: $O(1)$, la mappa **vicini** occupa massimo 8 piastrelle.

4.3 colora

Colora $piastrella(x, y)$ di colore $alpha$ e intensità i, qualunque sia lo stato di $piastrella(x, y)$ prima dell'operazione; utilizzo quindi **restituisciPiastrella** per verificare se la piastrellla e' gia presente nel piano (accesa o spenta con colore), in caso in cui non esista nel piano creo una nuova **Piastrella** con colore ed intensita' assegnata.
Tempo: $O(1)$, vengono eseguiti un numero costante di confronti, assegnamenti, inserimenti in mappa (max 1).
Spazio: $O(1)$, istanzio soltanto una **Piastrella**.

4.4 spegni

Spegne $piastrella(x, y)$. Se $piastrella(x, y)$ è già spenta, non fa nulla; controllo se la piastrella sia presente nel piano e controllo che l'intensita sia $\neq 0$ ossia non sia spenta, in caso affermativo la spengo (pongo intensita a 0), viceversa non faccio nulla.
Tempo: $O(1)$, eseguo un numero costante di confronti ed assegnamenti.
Spazio: $O(1)$, non uso strutture dati aggiuntive.

4.5 regola

Definisce la regola di propagazione $k_1\alpha_1 + k_2\alpha_2 + ... + k_n\alpha_n \leftarrow \beta$ e la inserisce in fondo all'elenco delle regole; parso la stringa rappresentante la regola (utilizzando la funzione **Fields e Atoi** di strings e strconv rispettivamente), formando i suoi termini ed aggiungendoli ad una slice di termini tramite append (sezione 3.4.1), ed infine aggiungendo la regola in fondo tramite append.
Tempo: $O(n)$, sia **n** la lunghezza della stringa r:

- 1 chiamata di **strings.Fields(r)** in tempo $O(n)$
- un for-ternario con t iterazioni dove t e' il numero di parti che esegue una append $O(t)$, dato che t è proporzionale alla lunghezza di r, allora $O(n)$
- un append finale per aggiungere la regola in tempo ammortizzato $O(1^*)$

Spazio: $O(t)$, sia **t** il numero di termini della regola allora lo spazio aggiuntivo occupato sara' dell'ordine $O(t)$

4.6 stato

Stampa e restituisce il colore e l'intensità di $piastrella(x, y)$. Se $piastrella(x, y)$ è spenta, non stampa nulla e restituisce la stringa vuota e l'intero 0, quindi viene effettuato un controllo per verificare se nel piano e' presente la piastrella accesa.
Tempo: $O(1)$, eseguo un numero costante di confronti e una chiamata a **restituisciPiastrella** (sezione 4.1).
Spazio: $O(1)$, non uso strutture dati aggiuntive.

4.7 stampa

Stampa l'elenco delle regole di propagazione, nell'ordine attuale. Quindi ciclo dapprima sulle regole (indico con n il numero di regole nel piano), e ciclo in secondo luogo sulle parti/termini della regola considerata (indico con t il numero di termini di una regola).
Tempo: $O(n * t)$, eseguo il ciclo esterno n volte e quello sui termini interno t volte con un numero costante di stampe.
Spazio: $O(1)$, non uso strutture dati aggiuntive.

4.8 dfs

Come spiegato nella sezione 2.2 useremo la visita in profondità DFS per **blocco** e **bloccoOmog** ma anche per **propagaBlocco**; per tutte le piastrelle adiacenti alla $piastrella(x, y)$ si verifica che non siano già state visitate in precedenza e che siano attualmente accese, in caso positivo si procederà ricorsivamente con la dfs.
Sono presenti alcuni parametri:

- **omogeneo**: flag per differenziare una visita del blocco basata sul colore
- **blocco**: variabile contenente le piastrelle del blocco considerato, se nil rimane nil
- **somma**: somma del blocco considerato, se nil rimane nil

Il comportamento di questa funzione varia a seconda dei parametri sopracitati:

- **blocco**: **omogeneo=false**, **blocco=nil**, per blocco la dfs calcola la somma senza guardare il colore
- **bloccoOmog**: **omogeneo=true**, **blocco=nil**, per bloccoOmog la dfs calcola la somma guardando il colore
- **propagaBlocco**: **omogeneo=false**, **somma=nil**, per propagaBlocco la dfs trova le piastrelle del blocco di appartenenza di $piastrella(x, y)$

Tempo: $O(n + m)$, l'algoritmo visita ogni piastrella una volta, per ogni piastrella esplora le piastrelle adiacenti ad essa; sia **n** il numero di vertici/piastrelle e **m** il numero di archi del grafo allora la complessita temporale sara' dell'ordine di $O(n + m)$.
Spazio: $O(n)$, la complessità spaziale è influenzata da:

- **spazio per i nodi visitati**: la mappa **visite** tiene traccia delle piastrelle visitate, nel peggiore dei casi, questa mappa contiene una *entry* per ogni piastrella quindi la sua complessità spaziale è $O(n)$, medesimo discorso vale per la mappa **blocco**.
- **spazio per lo stack di ricorsione**: la profondità della ricorsione della funzione dfs può arrivare fino al numero di piastrelle nel peggiore dei casi, quindi la memoria utilizzata è $O(n)$.

4.9 bloccoGenerale

Questo metodo risolve sia il problema dell'operazione **blocco** che **bloccoOmog** sezione 2.2, infatti al metodo viene passato un flag **omog** true se si vuole calcolare la somma delle intesita' del blocco omogeneo di appartenenza, false altrimenti. In seguito ai dovuti controlli legati alla piastrella di appartenenza (se spenta etc..), si chiama il metodo **dfs** spiegato prima sezione 4.8, con gli opportuni argomenti.
Tempo: $O(n+m)$, eseguo una chiamata a **restituisciPiastrella** $O(1)$, una a **dfs** $O(n+m)$ con n numero di vertici/piastrelle ed m numero di archi, ed infine una **println** eseguita in tempo costante essendo un numero (generalmenete piccolo) $O(1)$.
Spazio: $O(n)$, la mappa **visite** nel peggiore dei casi la sua complessità spaziale è $O(n)$

4.10 restituisciRegola

Metodo che **restituisce la prima regola da applicare** data una piastrella e **aggiorna il consumo**, nil altrimenti e aggiorna il consumo di essa, o non fa nulla se nessuna regola e' applicabile.
Dapprima calcolo l'intorno attraverso una mappa dove salvo le occorrenze dei colori dell'intorno concetto spiegato nella sezione 1.1.3; in secondo luogo scorrendo le regole del piano controllo se la regola e' applicabile a seconda dell'intorno precedentemente calcolato come spiegato nella sezione 2.3.
Tempo: $O(r*t)$, infatti:

- per **calcolare l'intorno** eseguo un numero di iterazioni pari al numero di piastrelle circonvicine usando il metodo **piastrelleCirconcivine** sezione 4.2 eseguito in tempo costante $O(1)$ eseguo quindi un massimo di 8 iterazioni quindi tempo costante $O(1)$
- per **trovare la regola da applicare** eseguo un numero di iterazioni sul for esterno pari al numero di regole, chiamiamo **r** il numero di regole del piano, ed eseguo un numero di iterazioni sul for interno pari al numero di termini della regola, chiamiamo **t** il numero di termini di una regola, sia nel for interno che nel for esterno eseguo un numero costante di operazioni come confronti ed assegnamenti quindi $O(r*t)$

Spazio: $O(1)$: dato dalla mappa **intorno** al massimo puo' possedere 8 colori come coppie chiave-valore si puo' considerare costante e trascurabile.

4.11 propaga

Metodo che applica alla piastrella(x, y) la prima regola di propagazione applicabile dell'elenco, ricolorando la piastrella, come spiegato nella sezione 2.3; questo metodo utilizza **restituisciRegola** citato prima, trovando cosi' la prima regola applicabile, in caso essa non sia **nil**, la applica alla piastrella ricolorandola nel caso sia gia "esistente", viceversa si applica la funzione **colora**.
Tempo: $O(r*t)$, si effettua una chiamata a **restituisciPiastrella** $O(1)$, una chiamata a **restituisciRegola** $O(r*t)$, con r numero di regole e t numero di termini, e infine un numero costante di confronti con al massimo una chiamata di **colora** $O(1)$.
Spazio: $O(1)$, non uso strutture dati aggiuntive.

4.12 propagaBlocco

Metodo che propaga il colore sul blocco di appartenenza di Piastrella(x, y), come spiegato nella sezione 2.3.
In un primo momento restituisco il blocco di appartenenza della piastrella chiamando il metodo **dfs** sezione 4.8, in seguito scorrendo il blocco appena ottenuto aggiungo tutte le regole da applicare in una mappa **aggiornamenti**, questo perche' come da specifica la ricolorazione delle piastrelle deve essere eseguita sul piano originario.
Ed infine applico tutte le regole usando appunto la mappa **aggiornammenti**.
Tempo: $O(n*r*t)$, si effettua:

- una chiamata al metodo **restituisciPiastrella** $O(1)$ sezione 4.1
- una chiamata al metodo **dfs** $O(n + m)$, con n numero di vertici e m numero di archi, sezione 4.8
- un for-range sulle piastrelle del blocco dove al massimo possono essere n iterazioni, con all'interno una chiamata del metodo **restituisciRegola** $O(r * t)$ sezione 4.10, con r numero di regole e t numero di termini, quindi $O(n * r * t)$
- un for-range che esegue un numero di iterazioni pari al numero di aggiornamenti massimo ossia n (aggiornamento su ogni piastrella), quindi $O(n)$

Quindi $O(1) + O(n + m) + O(n * r * t) + O(n) = O(n * r * t)$.
Spazio: $O(n)$, lo spazio per le mappe **visite, blocco, aggiornamenti, stack di ricorsione della dfs** e' al massimo $O(n)$.

4.13 ordina

Ordina l'elenco delle regole di propagazione in base al consumo delle regole stesse: la regola con consumo maggiore diventa l'ultima dell'elenco. Se due regole hanno consumo uguale mantengono il loro ordine relativo.
Si puo' evincere che bisogna usare un algoritmo di ordinamento stabile infatti gli elementi che hanno la stessa chiave (consumo) mantengono il loro ordine. Tra gli algoritmi stabili in go e' presente il metodo **sort.SliceStable**, del package **sort**, ho cosi' quindi definito un metodo **ordina** usando l'ordinamento basato sul consumo (si basa su una versione modificata del mergeSort).
Tempo: $O(n \log n)$, con n numero di elementi da ordinare.
Spazio: $O(n)$, con n numero di elementi da ordinare.

4.14 pista

Stampa la pista che parte da Piastrella(x, y) e segue la sequenza di direzioni s, se tale pista è definita. Altrimenti non stampa nulla.
Come spiegato nella sezione 2.4 si tratta di stampare un cammino se esiste, quindi dapprima controllo se esiste il vertice/piastrella di partenza usando **restituisciPiastrella**, in secondo luogo scorrendo i comandi controllo che applicando il comando rispetto alla piastrella in cui sono collocato, essa esista e sia e' accesa per **tutte** le direzioni in input, in caso affermativo stampo il cammino in caso contrario termino e non stampo nulla.
Tempo: $O(m)$, viene eseguito:

- una chiamata a **restituisciPiastrella** $O(1)$
- **strings.Split** impiega $O(m)$ con m numero di comandi/direzioni
- un for-range che esegue m iterazioni una per ogni comando/direzione con un numero costante di assegnamenti e confronti $O(m)$

Spazio: $O(m)$, con m numero di comandi della slice **comandi**.

4.15 lung

Determina la lunghezza della pista più breve che parte da Piastrella(x1 , y1) e arriva in Piastrella(x2 , y2). Altrimenti non stampa nulla.
Come spiegato nella sezione 2.4 per trovare la lunghezza della pista minima tra due piastrelle fissate si puo' applicare l'algoritmo **BFS** (perche grafo non pesato come spiegato nella sezione 2.4) l'algoritmo procede nel seguente ordine (con calcolo della stima del tempo):

1. controllo se esiste ed e' accesa la piastrella di partenza e arrivo, effettuando 2 chiamate a **restituisciPiastrella** $O(1)$
2. **inizializzo una coda** inserendo il vertice di partenza $O(1)$

3. finche la coda non e' vuota
   (a) **estraggo la prima piastrella** dalla coda ed aggiorno la coda
   (b) **per ogni piastrella circonvicina se non l'ho visitata la aggiungo alla coda** e aggiorno la distanza (ossia la lunghezza del cammino); ogni piastrella viene estratta e aggiunta alla coda esattamente una volta nel caso peggiore, inoltre per ogni piastrella esaminiamo tutti i suoi archi quindi $O(n+m)$, con n numero di vertici/piastrelle e m numero di archi
   (c) se la piastrella circonvicina i-esima e' il vertice di arrivo stampo la distanza e termino $O(1)$

Tempo: $O(n+m)$, con n numero di vertici/piastrelle e m numero di archi.
Spazio: $O(n)$, con n numero di piastrelle, infatti **visite, distance e la coda** occupano tutte $O(n)$.

5 Test di input e output

Da eseguire come spiegato nella sezione 1

- test su **colora e stato**: **BaseColoraStato** per testare la funzionalita' generali delle funzioni **colora** e **stato**
- test su **colora, stato e spegni**: **BaseSpegni, BaseSpegni2** per testare la funzionalita' generali delle funzioni **colora**, **stato** e anche **spegni** sia su piastrella accesa che su piastrella spenta
- test su **blocco e bloccoOmog**: per testare la funzionalita' generali delle funzioni **blocco** e **bloccoOmog** sia su piastrelle che formando un blocco che su piastrelle spente
- test su **regole e stampa**: **BaseRegole** per testare la funzionalita' generale di **regola** e **stampa** per verificare il corretto inserimento e stampa in ordine di inserimento
- test su **propaga e ordina**: **BasePropaga, BasePropaga2, BasePropaga3**, per verificare la propagazione su una piastrella accesa e spenta
- test su **propagaBlocco**: per verificare la propagazione su un blocco di appartenenza ad una piastrella sia accesa che spenta con **BasePropagaBlocco, BasePropagaBlocco2, AvanzatoPropagaBlocco**
- test su **pista e lung**: **BasePista, BasePista2, BaseLung** per verificare la correttezza di **pista** sia su una piastrella spenta che su accesa ma con una direzione su una piastrella spenta, ma anche di lunghezza con il minimo del cammino

6 Esempi

6.1 Esempio di blocco e bloccoOmog

Avendo le piastrelle $(1,1,"g",3)$, $(1,2,"g",2)$, $(1,0,"r",5)$, il blocco di appartenenza della piastrella $(1,1)$ ha somma delle intensita' pari a 10, invece il blocco omogeneo di appartenenza ha somma delle intensita' pari a 5.

6.2 Esempio di propaga e propagaBlocco

Avendo le piastrelle $(1,1,"g",3)$, $(1,2,"g",2)$, $(1,0,"r",5)$, e le regole in ordine di inserimento $(1g+1r \to v)$, $(2g \to e)$, se effettuo la propagazione nella piastrella $(1,1)$ ad essa verra' applicata la prima regola ricolorando la piastrella $(1,1)$ di v, invece se effettuassi propaga sulla piastrella $(1,0)$ verrebbe applicata la seconda regola ricolorando la piastrella $(1,0)$ di e.
Se invece effettuassi propagaBlocco sulla piastrella $(1,2)$ a fine esecuzione avrei: le piastrelle $(1,1,"v",3)$, $(1,2,"v",2)$, $(1,0,"e",5)$.

6.3 Esempio di pista e lung

Avendo le piastrelle $(2, 0, "t", 9)$, $(1, 1, "w", 3)$, $(1, 2, "m", 9)$, con direzioni NO, NN partendo dalla piastrella $(2, 0)$ la pista avrebbe le seguenti coordinate: $(2, 0) \to (1, 1) \to (1, 2)$.
Invece la lunghezza tra $(2, 0)$ e $(1, 2)$ sarebbe 3.