UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Algoritmi per la risoluzione di labirinti in linguaggio C

Relatore: Prof. Antonio Giunta

Laureando: Francesco Boscolo Meneguolo

ANNO ACCADEMICO 2022 – 2023
Data di laurea: 25/09/2023

SOMMARIO

1
FINALITÀ DI QUESTO LAVORO

1.1
I labirinti in informatica

1.1.1 Definizioni

Un labirinto è un percorso o un insieme di percorsi in genere da un punto di partenza ad un punto di arrivo. In termini matematici, un labirinto è un grafo non orientato, planare e possibilmente ciclico.

Una cella è un punto in un labirinto, ossia è un'unità di passaggio che è collegata ad altre celle per formare il labirinto. Dal punto di vista della teoria dei grafi ogni cella corrisponde ad un vertice. Una cella può avere dei collegamenti, che corrispondono agli archi di un grafo e conducono da essa ad altre celle, oppure può avere dei muri. Le celle possono avere geometria differente in base alla tassellazione del labirinto.

Una giunzione è qualsiasi cella con più di due celle adiacenti collegate.

Un incrocio è un punto di un labirinto nel quale si intersecano due percorsi.

Un vicolo cieco è qualsiasi cella con esattamente una cella vicina collegata.

Un passaggio è un insieme di celle tra due giunzioni o tra un vicolo cieco e una giunzione, inclusi entrambi i punti estremi.

Un anello è un percorso che si collega con se stesso formando un ciclo.

1.1.2 Classificazione

I labirinti possono essere classificati principalmente secondo 5 criteri:

- dimensione: equivale al numero di coordinate necessarie ad individuare una cella nel labirinto.
- iperdimensione: si riferisce alla dimensione dell'oggetto che si muove nel labirinto, ovvero al numero di coordinate necessarie per rappresentare tale oggetto.
- topologia: descrive la geometria dello spazio in cui il labirinto si trova (euclideo o non euclideo).
- tassellazione: divide i labirinti in base alla geometria delle singole celle ed ai collegamenti tra esse.
- percorso: classifica il labirinto in base alle caratteristiche dei passaggi, ovvero alla presenza o meno di: cicli, vicoli ciechi, giunzioni, celle irraggiungibili e cammini unici.

1.1.3 Rappresentazione attraverso caratteri ASCII

Rappresentare un labirinto attraverso caratteri del codice ASCII non è il modo più elegante, ma senza dubbio è il più semplice. In particolare tale metodo si presta bene a rappresentare labirinti a tassellazione ortogonale, ovvero quei labirinti in cui le celle formano una griglia rettangolare ed i collegamenti avvengono perpendicolarmente rispetto ai lati delle celle.

Si possono usare i seguenti caratteri:

- 3 spazi (“ “) per i corpi delle celle;
- singolo spazio (‘ ‘) per i collegamenti tra due celle adiacenti;
- ‘+’ per gli angoli delle celle;
- ‘|’ per i muri verticali;
- 3 trattini (“---“) per i muri orizzontali;
- S con uno spazio a destra ed uno a sinistra (“ S “) per il corpo della cella di partenza;
- E con uno spazio a destra ed uno a sinistra (“ E “) per il corpo della cella di arrivo.

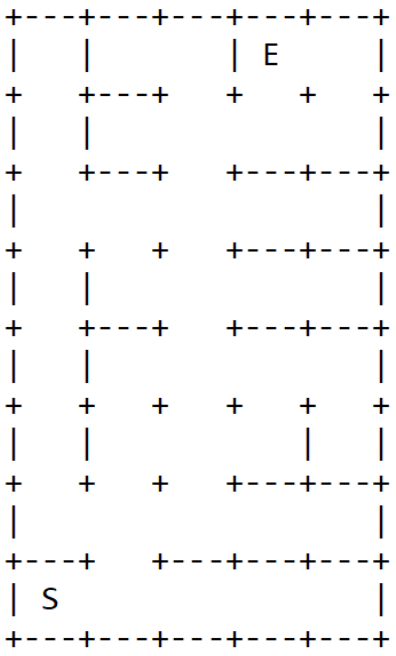

Figura 1.1 Esempio di labirinto di dimensioni 5×8 rappresentato con caratteri del codice ASCII

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)

Figura 1.2 Coordinate delle celle nella griglia rettangolare di dimensioni 5×8

1.1.4 Risoluzione

Il problema di trovare un percorso verso un obiettivo partendo da una posizione nota è ben conosciuto da molto tempo; ancora oggi risulta essere di interesse, dati l'invenzione di nuovi algoritmi ed i progressi nei campi della robotica e dell'intelligenza artificiale. Esistono diversi algoritmi di risoluzione di labirinti, ovvero metodi automatizzati per la loro risoluzione. Essi sono di due tipologie: la prima riguarda algoritmi progettati per essere utilizzati all'interno del labirinto da un viaggiatore senza alcuna conoscenza del labirinto, mentre la seconda riguarda algoritmi progettati per essere utilizzati da una persona o da un programma per computer in grado di vedere dall'esterno l'intero labirinto in una volta.

1.1.5 Progetto

Questa tesi si pone l'obiettivo di descrivere ed implementare in linguaggio C i principali algoritmi di risoluzione di labirinti, utilizzando le peculiarità di questo linguaggio di programmazione, che unisce i vantaggi della programmazione ad alto livello con quelli della programmazione a basso livello.

2
EVENTUALI LIMITI

Possono essere individuati i seguenti limiti:

- l'implementazione degli algoritmi di risoluzione è specifica per risolvere labirinti a tassellazione ortogonale;
- si suppone che esista sempre una soluzione al labirinto;
- non viene restituito nulla in output che faccia capire all'utente che l'algoritmo in questione non è in grado di risolvere il labirinto datogli in input (alcuni algoritmi risolvono solo certi tipi di labirinti); quindi l'utente deve prestare attenzione alle ipotesi di risoluzione di ciascun algoritmo;
- in una cella sono consentiti soltanto spostamenti verso l'alto, il basso, destra e sinistra;
- non avendo usato una libreria grafica, la rappresentazione del labirinto risulta più grezza, ma comunque efficace.
- non si eseguono controlli sulla correttezza dei dati forniti in input.

3
ALGORITI E STRUTTURE DI DATI PIÙ SIGNIFICATIVI

3.1
Algoritmi

3.1.1 Random mouse

Questo algoritmo è il più inefficiente. Simula un umano o un topo che si muove a caso all'interno di un labirinto senza nessuna memoria di dove è già stato. Si segue un passaggio finché si arriva ad una giunzione, dopo ciò si sceglie casualmente dove andare, evitando rotazioni di 180° (ammesse soltanto quando si giunge in un vicolo cieco e bisogna tornare indietro alla giunzione precedentemente visitata). Siccome l'algoritmo non ha memoria dei posti visitati può seguire lo stesso percorso più volte. Non è garantito che questo algoritmo raggiunga l'obiettivo perché potrebbe vagare all'infinito all'interno del labirinto.

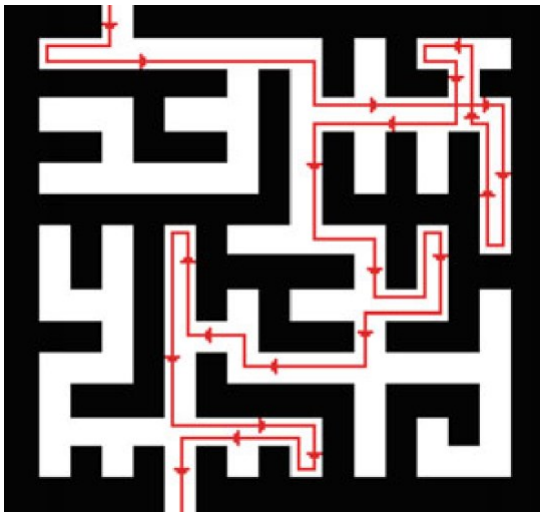

Figura 3.1 Random mouse si muove a caso all'interno del labirinto

3.1.2 Wall Follower

Si tratta di un semplice algoritmo per la risoluzione di labirinti. È sempre molto veloce e non utilizza memoria aggiuntiva. È come se un essere umano risolvesse un labirinto mettendo la mano destra (o sinistra sulla parete) e la lascia lì mentre cammina. Questo metodo non troverà necessariamente la soluzione più breve. Garantisce di trovare una soluzione solo nei labirinti perfetti, che sono quei labirinti in cui esiste un unico percorso tra due celle, ovvero non sono presenti anelli. Nei labirinti non perfetti la soluzione è garantita solo se la cella di partenza e la cella da raggiungere presentano un muro in uno dei loro 4 lati ed i muri sono tutti connessi tra loro. Se la cella di inizio e la cella da raggiungere sono all'interno del labirinto c'è ancora una possibilità di risolvere il labirinto, ma non è garantita la risoluzione. Partire all'interno potrebbe forzare l'algoritmo a seguire un muro che non è connesso al muro esterno del labirinto, creando un loop infinito. Se la cella da raggiungere si trova all'interno ci si potrebbe girare intorno all'infinito.

La logica seguita dal Right Hand Wall Follower è la seguente (regola della mano destra):

Destra	Di fronte	Azione
0	X	Ruota di 90° in senso orario e poi muoviti avanti
1	0	Muoviti avanti
1	1	Ruota di 90° in senso antiorario

0 = muro assente; 1 = muro presente; X = ignora muro

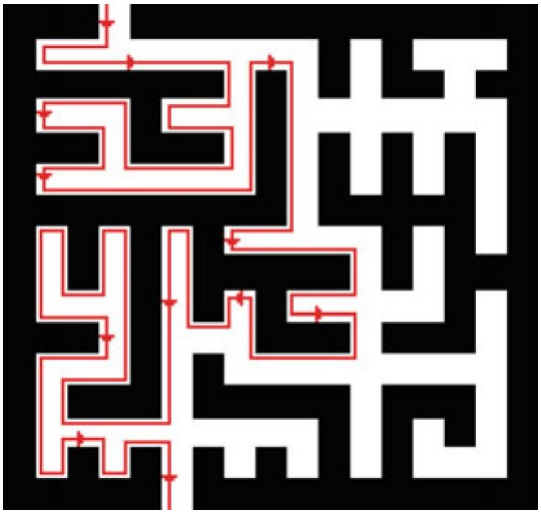

Figura 3.2 Soluzione al labirinto con il Right Hand Wall Follower

3.1.3 Pledge

Questo algoritmo è una versione modificata del Wall Follower e permette di trovare una soluzione nel caso in cui si voglia raggiungere una cella adiacente al muro esterno partendo da una cella all'interno del labirinto. All'inizio viene scelta una direzione principale che l'algoritmo seguirà. Prosegue verso quella direzione finché si trova di fronte ad un muro, successivamente inizia a seguire il muro usando una versione modificata del Wall Follower. Ogni volta che deve girare in senso orario incrementa di 1 un contatore, invece quando gira in senso antiorario lo decrementa di 1. L'algoritmo usa il wall follower finché tale contatore va a 0, quindi viene abbandonato l'ostacolo e viene ripercorsa la direzione principale. Tale procedura viene eseguita fino ad arrivare alla cella finale.

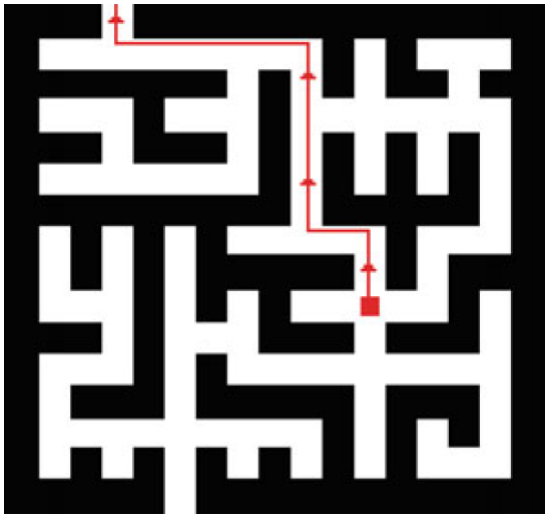

Figura 3.3 Soluzione trovata partendo dall'interno del labirinto usando l'algoritmo di Pledge

3.1.4 Trémaux

L’algoritmo di Trémaux è in grado di risolvere qualsiasi labirinto, ma non garantisce il percorso più breve. Può essere usato da un umano all’interno del labirinto. Mentre si percorre un passaggio, il risolutore traccia dietro di se una linea per segnare il percorso. L'algoritmo funziona secondo le seguenti regole:

- Segna ogni percorso una volta, quando lo segue.
- Non segue mai un percorso che ha già due segni su di esso.
- Se raggiunge un vicolo cieco ritorna indietro e lo contrassegna ulteriormente.
- Se arriva ad un incrocio che non ha segni, sceglie un percorso arbitrario non contrassegnato, lo segue e lo contrassegna.
- In caso contrario, sceglie arbitrariamente uno dei percorsi rimanenti con il minor numero di segni (zero se possibile, altrimenti uno), segue quel percorso e lo contrassegna.

Quando finalmente raggiungi la soluzione, i percorsi segnati esattamente una volta indicano una via di ritorno all'inizio, quindi sono una soluzione.

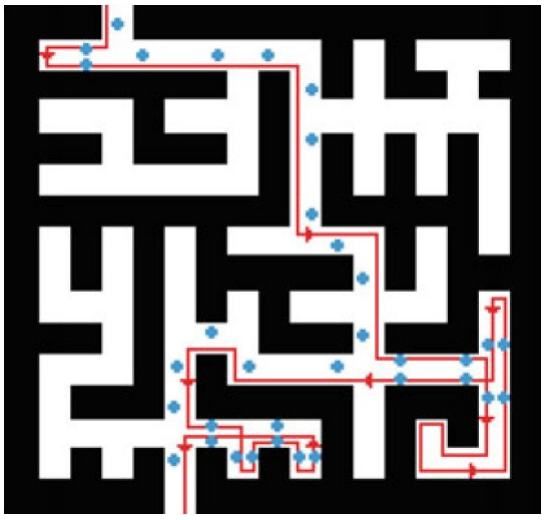

Figura 3.4 l’algoritmo di Trémaux raggiunge l’uscita contrassegnando i percorsi

3.1.5 Dead End Filler

È un algoritmo di trasformazione di labirinti che riempie tutti i vicoli ciechi che si vengono a formare. Può essere utilizzato per risolvere labirinti perfetti su carta o con un programma per computer. Non è utile per una persona all'interno di un labirinto sconosciuto poiché questo metodo esamina l'intero labirinto contemporaneamente. Il metodo consiste nel:

1) trovare tutti i vicoli ciechi nel labirinto;

2) "riempire" il percorso da ciascun vicolo cieco fino al raggiungimento di una giunzione;

3) alcune celle non diventeranno vicoli ciechi fino a quando non verranno rimossi altri vicoli ciechi.

Alla fine nei labirinti perfetti rimarrà soltanto la soluzione.

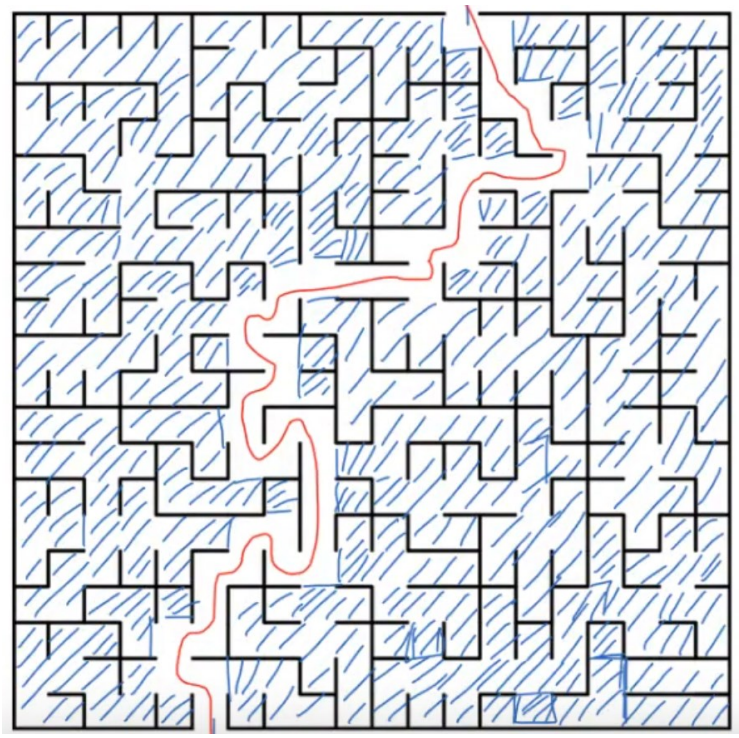

Figura 3.5 Esempio di labirinto perfetto trasformato dal Dead End Filler fino ad ottenere la soluzione

3.1.6 Recursive Backtracker

Questo algoritmo trova una soluzione, ma non necessariamente la più breve. È veloce per tutti i tipi di labirinto ed utilizza una pila di grandezza proporzionale alle dimensioni del labirinto. Utilizza la seguente logica: sceglie una direzione casuale e la segue, torna indietro se e solo se raggiunge un vicolo cieco. Quando giunge in una cella, la contrassegna come visitata, in modo da non visitarla di nuovo da una direzione diversa. In termini informatici, questo metodo è fondamentalmente una ricerca in profondità (DFS). Questo algoritmo viene comunemente chiamato "Recursive Backtracker", anche se la ricerca di profondità che utilizza ha bisogno solo di una pila e non deve utilizzare una vera e propria ricorsione.

Figura 3.6 Recursive Backtracker che ritorna alla giunzione precedente dopo essere arrivato ad un vicolo cieco

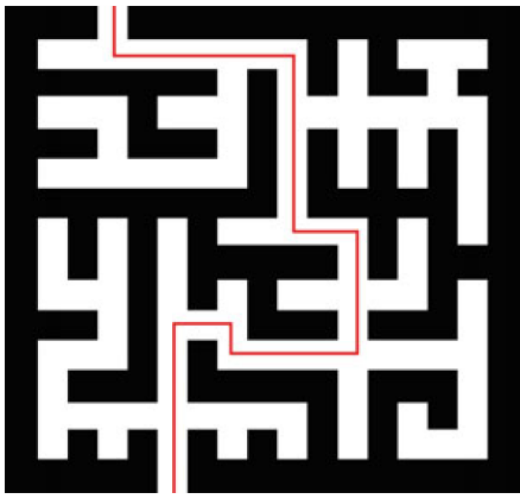

Figura 3.7 Soluzione trovata dal Recursive Backtracker

3.1.7 Chain

Il Chain algorithm risolve il labirinto trattandolo effettivamente come una serie di labirinti più piccoli, come gli anelli di una catena, e risolvendoli in sequenza. In alcuni casi ciò aiuta, in altri invece risulta essere un'idea terribile. È necessario specificare la posizione iniziale e quella finale desiderata e l'algoritmo troverà sempre un percorso dall'inizio alla fine. Si inizia tracciando una linea retta (o almeno una linea che non interseca se stessa) dall'inizio alla fine, lasciando che attraversi i muri se necessario. Poi segue la linea dall'inizio alla fine. Se il cammino d'uscita urta un muro, non può attraversarlo, e quindi deve girarci intorno. L'algoritmo manda dei "robot" in tutte le celle vicine, i quali devono raggiungere (utilizzando un algoritmo di risoluzione) il prossimo punto della linea guida; il percorso trovato più breve viene aggiunto alla soluzione. Continua a seguire la linea e ripete il processo fino alla fine.

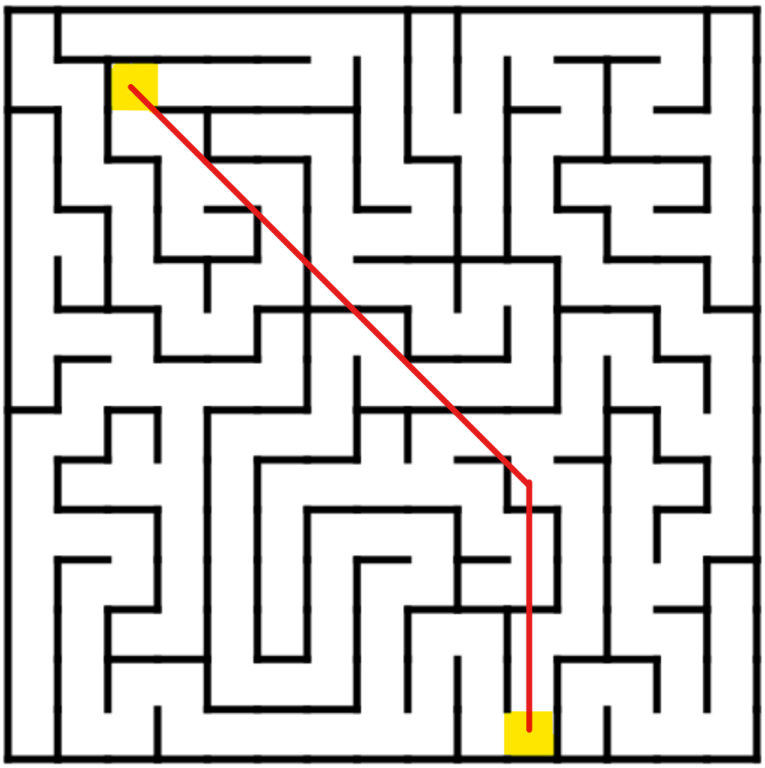

Figura 3.8 In rosso la linea guida, in giallo la celle iniziale e la cella finale

3.1.8 Collision Solver

È noto anche come "amoeba solver", dato che ricorda l'avanzamento di un'ameba (organismo unicellulare che presenta forma varia e non stabile, grazie alla sua capacità di cambiare conformazione, principalmente estendendo e ritraendo gli pseudopodi, estroflessioni temporanee utilizzate per il movimento ameboide e la fagocitosi). Questo algoritmo allaga il labirinto in modo tale che ogni cella alla stessa distanza dalla cella iniziale sia riempita allo stesso tempo; può essere visto come una Breadth-First Search. Quando due fronti d'onda si scontrano l'uno con l'altro, creano un muro in quella posizione. Se due onde arrivano nella posizione contemporaneamente e c'è almeno un passaggio non ancora riempito, il muro non si crea. Questo si spiega con il fatto che i due fronti d'onda hanno ancora un po' di strada da percorrere invece di di eliminarsi a vicenda, se l'algoritmo creasse un muro in quella posizione potremmo bloccare parte del labirinto dall'essere inondato. L'intero processo dev'essere ripetuto più volte finché non c'è più nessuna collisione. La soluzione finale consiste in almeno uno dei percorsi più brevi.

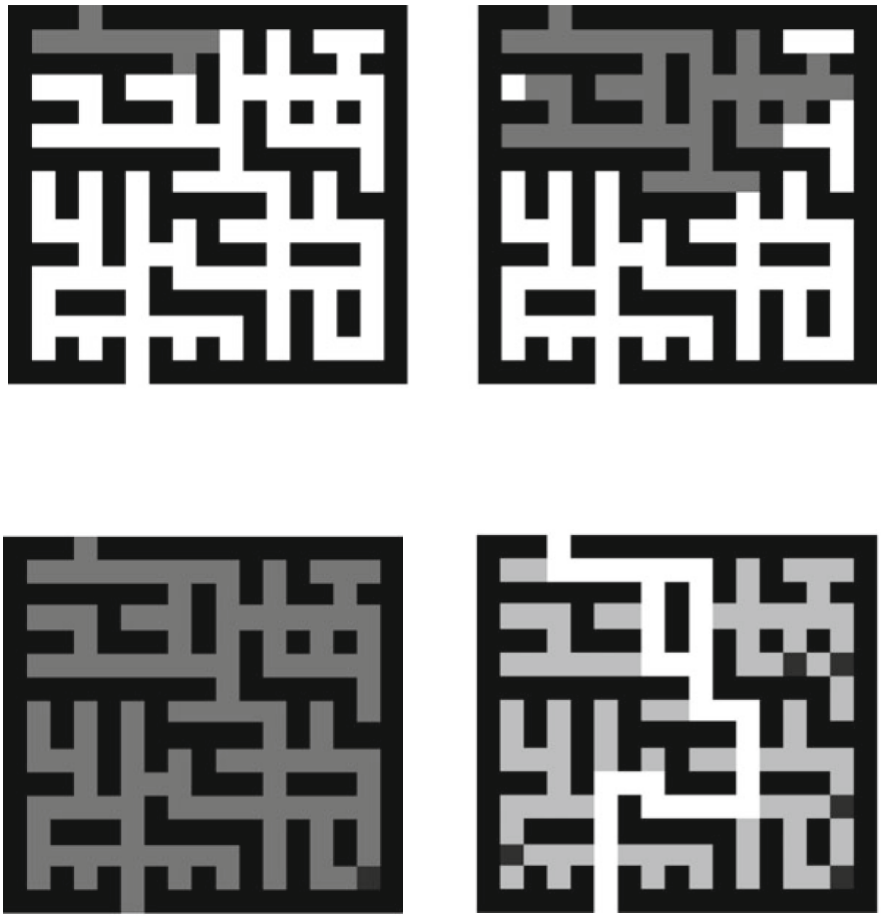

Figura 3.9 Riduzione del labirinto con l'algoritmo Collision Solver. Si possono notare: le celle inondate (grigio), le celle bloccate (grigio scuro), i percorsi che non fanno parte della soluzione (grigio chiaro) e le soluzioni (bianco)

3.1.9 Shortest Path Finder

Questo algoritmo restituisce una soluzione più breve. Può essere paragonato al Collision Solver e, a sua volta, alla Breadth-First Search, in quanto anch'esso allaga il labirinto, ma la soluzione viene ricavata in un altro modo. Ogni cella memorizza da quale cella è stata inondata, quindi l'algoritmo può creare un percorso dall'inizio alla fine. Nel momento in cui un fronte d'onda raggiunge l'obiettivo, l'algoritmo torna indietro. Poiché il percorso è a ritroso, esso sarà il più breve. Questo algoritmo è quasi una copia esatta della BFS, con l'unica differenza che termina dopo aver trovato la prima soluzione.

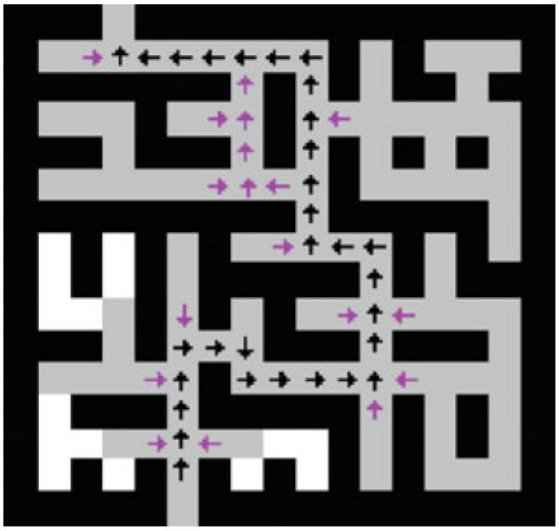

Figura 3.10 Shortest Path Finder in azione. Ogni cella sa da dove è stata visitata

3.1.10 Flood Fill

Questo algoritmo necessita di una visione globale del labirinto. Calcola la distanza di ogni cella dalla cella finale. Per costruire la soluzione parte dalla cella iniziale e sceglie mano a mano la cella vicina a minor distanza dalla cella finale; restituisce quindi il percorso più breve.

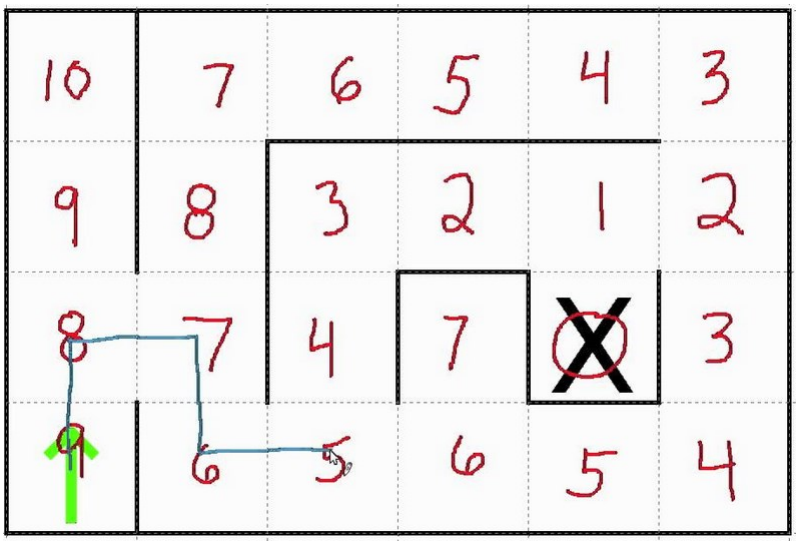

Figura 3.11 Flood Fill in azione in un labirinto

3.1.11 A*

A* è un algoritmo di ricerca informata best-first, che sfrutta la conoscenza di ulteriori dettagli sugli stati del problema da risolvere. La scelta del cammino si basa sia sul costo efettivo g(n) necessario per raggiungere la cella intermedia n dalla cella di partenza e sia sulla stima del costo h(n) necessario per raggiungere la cella finale dalla cella intermedia n. La funzione di ricerca f(n) è quindi composta da due funzioni di costo:

$$f(n) = g(n) + h(n)$$

Un algoritmo di ricerca che garantisce sempre di trovare il percorso più breve verso una meta è detto ammissibile. Se A* utilizza un'euristica, allora non bisogna mai sovrastimare la distanza (o in genere, il costo) verso la meta; si può così verificare che A* sarà ammissibile. Esempi di euristiche ammissibili sono la distanza euclidea o la distanza di Manhattan.

Le implementazioni tipiche di A* utilizzano una coda di priorità per effettuare la selezione ripetuta delle celle a costo minimo (stimato) da espandere. Questa coda di priorità è nota come insieme aperto, frangia o frontiera. Ad ogni passo dell'algoritmo, la cella con il valore f(n) più basso viene rimosso dalla coda; i valori f e g dei suoi vicini vengono aggiornati di conseguenza e questi vicini vengono aggiunti alla coda. L'algoritmo continua finché una cella rimossa (quindi la cella con il valore f più basso tra tutte le celle marginali) non è la cella finale. Il valore f di tale meta è anche il costo del percorso più breve, poiché h alla meta è zero in un'euristica ammissibile.

L'algoritmo descritto finora ci fornisce solo la lunghezza del percorso più breve. Per trovare l'effettiva sequenza di passi, l'algoritmo può essere facilmente rivisto in modo che ogni cella del percorso tenga traccia della precedente.

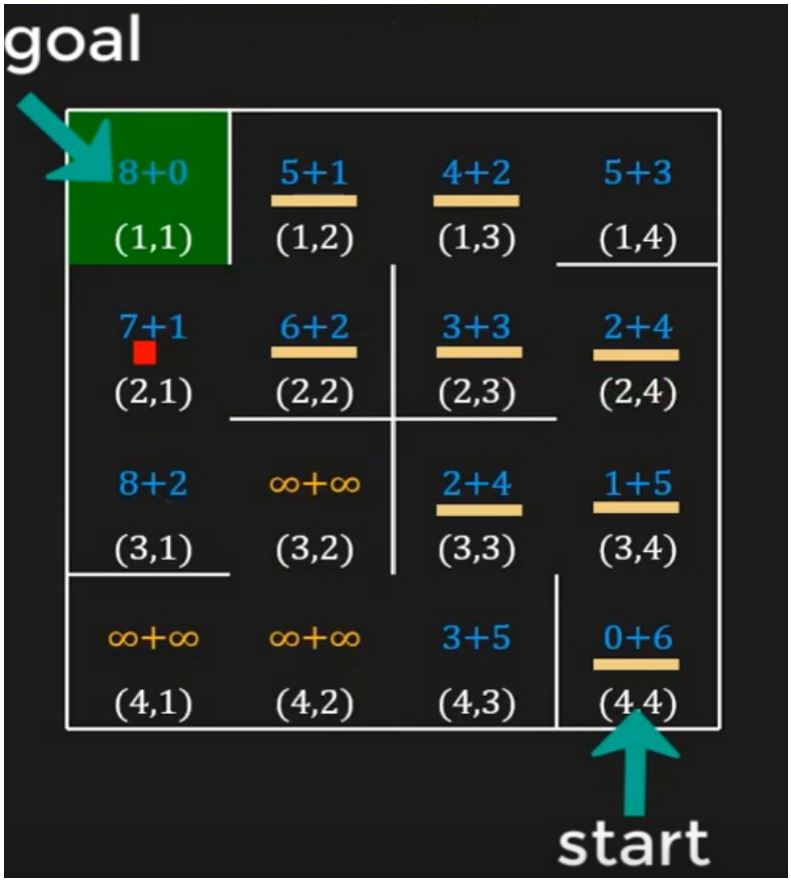

Figura 3.12 Esempio di applicazione di A* in un labirinto

3.1.12 Tabella riassuntiva

Algoritmo	Numero di soluzioni	Soluzione garantita?	Replicabile da una persona?	Memory Free?
Random mouse	1	No	Si, dall'interno / dall'esterno	Si
Wall Follower	1	No	Si, dall'interno / dall'esterno	Si
Pledge	1	No	Si, dall'interno / dall'esterno	Si
Trémaux	1	Si	Si, dall'interno / dall'esterno	No
Dead End Filler	Lascia tutte le soluzioni	No	Si, dall'esterno	Si
Recursive Backtracker	1	Si	No	No
Chain	1	Si	No	No
Collision Solver	Tutte le più brevi	Si	No	No
Shortest Path Finder	1 più breve	Si	No	No
Flood Fill	1 più breve	Si	Si, dall'esterno	No
A*	1 più breve	Si	No	No

3.2
Strutture di dati

3.2.1 Cell

È la struttura che rappresenta una cella, unità fondamentale di un labirinto, attraverso i seguenti campi:

- indice di riga;
- indice di colonna;
- puntatore alla cella in alto (se non è presente un muro tra le due celle);
- puntatore alla cella in basso (se non è presente un muro tra le due celle);
- puntatore alla cella a destra (se non è presente un muro tra le due celle);
- puntatore alla cella a sinistra (se non è presente un muro tra le due celle);
- array di 4 interi contenente zeri ed uni. Indica in ogni posizione la presenza di un muro o di un passaggio in ciascuno dei 4 lati della cella (rispettivamente in alto, in basso, a destra e a sinistra). Lo 0 indica la presenza di un passaggio, l'1 la presenza di un muro.

3.2.2 List

Per memorizzare le celle visitate è stata utilizzata una struttura $List, che ha come campi: un array di $Cell, la sua grandezza, e la sua capienza.

3.2.3 ListArray

Questa struttura è stata creata per memorizzare più $List (senza sapere il numero esatto a priori). Ha come campi: un'array di $List, la sua grandezza e la sua capienza.

3.2.4 Maze

Il labirinto è la più grande struttura del progetto. Rappresenta il labirinto attraverso una griglia rettangolare di celle. Possiede i seguenti campi: larghezza del labirinto, altezza del labirinto, array 2D di $Cell per rappresentare la griglia, puntatore alla cella di partenza, puntatore alla cella finale.

4
INPUT E OUTPUT DEL PROBLEMA

4.1
Input

Ciascuno degli 11 algoritmi di risoluzione è stato scritto in un file sorgente .c denominato con il suo nome e contente il main. Per ottenere gli 11 eseguibili .exe bisogna compilare i 15 file sorgente e successivamente linkare ciascuno degli 11 file oggetto del tipo NomeAlgoritmo.o con i seguenti 4 file: Cell.o, List.o, ListArray.o e Maze.o.

Per fare ciò ho utilizzato il compilatore GCC e mi sono servito dei file di batch forniti dal professor Giunta (PCA.bat per compilare e L9.bat per linkare).

In input a ciascun eseguibile si fornisce da riga di comando un file di testo contenente:

- nella prima riga la larghezza del labirinto in numero di celle, ovvero il numero di celle in una riga del labirinto;
- nella seconda riga l'altezza del labirinto in numero di celle, ovvero il numero di celle in una colonna del labirinto;
- nelle righe successive la rappresentazione del labirinto in caratteri del codice ASCII secondo le modalità spiegate al paragrafo 1.1.3.

Le celle di partenza e di destinazione possono essere scelte in modo arbitrario.

4.2
Output

Si ottengono i seguenti output sul terminale:

- soluzione/i o al labirinto rappresentata/e come lista/e di coordinate di ciascuna cella facente parte del/i percorso/i.
- la rappresentazione grafica del labirinto, con le celle appartenenti a quella soluzione contrassegnate con 'X', per ciascuna soluzione ottenuta.
- Il numero di passi eseguiti all'interno del labirinto solo per quegli algoritmi che simulano un risolutore all'interno del labirinto.

```
5
8
+---+---+---+---+---+
|   |       | E     |
+   +---+   +   +   +
|   |               |
+   +---+   +---+---+
|                   |
+   +   +   +---+---+
|   |               |
+   +---+   +---+---+
|   |               |
+   +   +   +   +   +
|   |           |   |
+   +   +   +---+---+
|                   |
+---+   +---+---+---+
| S                 |
+---+---+---+---+---+
```

Figura 4.1 Esempio di file di testo di input

```
SOLUZIONE: [(7, 0), (7, 1), (6, 1), (6, 2), (6, 3), (7, 3), (7, 4), (7, 5), (6, 5), (5, 5), (4, 5), (3, 5), (2, 5), (1,
5), (1, 4), (0, 4), (0, 5)]

+---+---+---+---+---+---+
|         |     X   E |
+   +---+   +   +   +---+
|   |           | X   X |
+---+---+   +---+---+   +
|           |       | X |
+---+   +---+---+   +   +
|                   | X |
+---+   +---+   +   +   +
|   |       |   |   | X |
+   +---+   +   +   +   +
|               |   | X |
+   +---+   +   +   +   +
|     X   X   X |   | X |
+---+   +---+   +   +   +
| S   X     | X   X   X |
+---+---+---+---+---+---+
```

Figura 4.2 Esempio di output su terminale

5
BIBLIOGRAFIA E SITOGRAFIA

- Jamis Buck. *Mazes for programmers*. Pragmatic Bookshelf, 2015.
- R Niemczyk, Stanisław Zawiślak. *Review of Maze Solving Algorithms for 2D Maze and Their Visualisation.* In: *Engineer of the XXI Century. Proceedings of the International Conference of Students, PhD Students and Young Scientists.* Springer, 2020, pp. 239-252.
- *Maze-solving algorithm.* URL: https://en.wikipedia.org/wiki/Maze-solving_algorithm.
- Walter D. Pullen. *Think Labyrinth!* URL: https://www.astrolog.org/labyrnth.htm.
- *A* search algorithm.* URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

6
CODICI SORGENTI

Il file Cell.h contiene la struttura dati per la rappresentazione di una cella del labirinto.

```
#include <stdbool.h>
#include <stdlib.h>

#define UP 0
#define RIGHT 1
#define DOWN 2
#define LEFT 3
#define NEIGHBOURS 4

/*
Cella.
row Riga.
col Colonna.
up Puntatore alla cella in alto, se collegata direttamente.
down Puntatore alla cella in basso, se collegata direttamente.
left Puntatore alla cella a sinistra, se collegata direttamente.
right Puntatore alla cella a destra, se collegata direttamente.
walls Array che indica la presenza di un muro rispettivamente in alto, a destra,
in basso e a sinistra.
      1: muro presente.
      0: muro assente.
*/
typedef struct Cell{
   int row;
   int col;
   struct Cell* up;
   struct Cell* down;
   struct Cell* left;
   struct Cell* right;
   int walls[NEIGHBOURS];
}Cell;

void swap(Cell**, Cell**);
bool isEqualCell(const Cell*, const Cell*);
```

Il file Cell.c contiene le funzioni di utilità per la struttura $Cell.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Cell.h"


/*Scambio tra loro due celle.
IOP a Prima cella.
IOP b Seconda cella.
*/
void swap(Cell** a, Cell** b){
   Cell* temp;
   temp = *a;
   *a = *b;
   *b = temp;
}/* swap */


/*Restituisce se due celle sono uguali tra loro.
IP p1 Prima cella.
IP p2 Seconda cella.
OR Esito del confronto.
*/
bool isEqualCell(const Cell* p1, const Cell* p2){
   return (p1->row == p2->row) && (p1->col == p2->col);
}/* isEqualCell */
```

Il file List.h contiene la struttura $List.

```
#include "Cell.h"

/*
   Lista di celle.
*/
typedef struct {
   Cell** list;/* lista contenente le celle */
   int length;/* lunghezza (grandezza) della lista */
   int capacity;/* capacità della lista */
   int first; /* Indice dell'elemento che e` nella lista da piu` tempo. */
} List;


void initList(List*);
bool isEmptyList(const List*);
void listFree(List*);
int listIndex(const List*, int);
void doubleIfFull(List*);
void listAdd(List*, Cell*);
void listAddFirst(List*, Cell*);
void listRemove(List*, int);
void listRemoveFirst(List*);
void listRemoveLast(List*);
void shuffle( List*);
Cell* chooseCell(const List*);
int searchCell(const Cell*, const List*, int);
void reduceList(List*, int, int);
bool isIn(const List*, const Cell*);
Cell* listGetLast(const List*);
Cell* listGetFirst(const List*);
void printList(const List*);
void concat(List*, List*);
void copyList(List*, const List*);
```

Il file List.c contiene le funzioni di utilità per la struttura $List.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "List.h"


/* Inizializza una lista.
IOP l Lista da inizializzare.
*/
void initList(List* l){
    l->list = malloc(sizeof(Cell*));
    assert(l->list != NULL);
    l->capacity = 1;
    l->length = 0;
    l->first = 0;
}/* initList */

/* Restituisce se una lista e' vuota.
IOP l Lista da esaminare.
OR Esito della verifica.
*/
bool isEmptyList(const List* l){
    return l->length == 0;
}/* isEmptyList */

/* Svuota la lista e libera la sua memoria allocata dinamicamente.
IOP l Lista da liberare.
*/
void listFree(List* l) {
    free(l->list);
    l->length = 0;
} /* listFree */

/* Ritorna l'indice che dista $i posizioni dall'inizio della lista.
IP l Lista da cui individuare l'indice dell'elemento $i-esimo.
IP i Indice dell'elemento da individuare.
OR L'indice contenente che dista $i posizioni dall'inizio della lista $l.
*/
int listIndex(const List* l, int i) {
    return (l->first + i) % l->capacity;
} /* listIndex */

/* Raddoppia la capienza della lista quando questa e` piena.
IOP l Lista di cui raddoppiare la capienza.
*/
```

```
void doubleIfFull(List* l) {
    int i;
    Cell** newList;
    if (l->length == l->capacity) {
        newList = malloc(2 * l->capacity * sizeof(Cell*));
        assert(newList != NULL);
        for(i = 0; i < l->length; i++)
            newList[i] = l->list[listIndex(l, i)];
        l->capacity *= 2;
        l->first = 0;
        free(l->list);
        l->list = newList;
    } /* if */
} /* doubleIfFull */


/* Aggiunge un elemento in fondo alla lista.
IOP l Lista a cui aggiungere un elemento.
IP elem Elemento da aggiungere alla lista.
*/
void listAdd(List* l, Cell* elem) {
    doubleIfFull(l);
    l->list[listIndex(l, l->length)] = elem;
    l->length++;
} /* listAdd */

/* Aggiunge un elemento all'inizio della lista.
IOP l Lista a cui aggiungere un elemento.
IP elem Elemento da aggiungere alla lista.
*/
void listAddFirst(List* l, Cell* elem) {
    int i;
    doubleIfFull(l);
    for(i = listIndex(l, l->length - 1); i >= l->first; i--)
        l->list[i+1] = l->list[i];
    l->list[l->first] = elem;
    l->length++;
} /* listAddFirst */

/* Rimuove l'i-esimo elemento della lista.
IOP l Lista da cui rimuovere un elemento.
IP index
*/
void listRemove(List* l, int index) {
    int i;
    if(index >= 0){
        for(i = listIndex(l, index); i < listIndex(l, l->length - 1); i++)
            l->list[listIndex(l, i)] = l->list[listIndex(l, i+1)];
        l->length--;
    }/* if */
    /*realloc(l->list, l->length * sizeof(Cell));*/
```

```
} /* listRemove */

/* Rimuove il primo elemento della lista.
IOP l Lista da cui rimuovere l'elemento.
*/
void listRemoveFirst(List* l) {
    if(!isEmptyList(l)){
        l->length --;
        l->first = listIndex(l, 1);
    }/* if */
} /* listRemoveFirst */

/* Rimuove l'ultimo elemento della lista.
IOP l Lista a cui rimuovere l'elemento.
*/
void listRemoveLast(List* l) {
    l->length --;
} /* listRemoveFirst */

/* Mescola casualmente gli elementi di una lista.
IOP Lista da mescolare.
*/
void shuffle(List* l) {
   int i, j;
   for(i = l->length - 1; i > 0; i--) {
      j = rand() % (i + 1);
      swap(l->list + i, l->list + j);
   }/* for */
}/* shuffle */

/* Sceglie un elemento a caso di una lista.
IP Lista dalla quale scegliere una cella.
OR Cella scelta.
*/
Cell* chooseCell(const List* l) {
    int randomIndex;
    /* Genera un indice casuale nell'intervallo [0, l->length - 1] */
    randomIndex = rand() % l->length;

    /* Restituisce l'elemento corrispondente all'indice casuale */
    return l->list[randomIndex];
}/* chooseCell */

/* Effettua una ricerca lineare su una lista.
IP key cella da cercare.
IP l Lista nella quale cercare $key.
IP start Indice da cui iniziare la ricerca.
OR Indice in cui è presente la cella, altrimenti -1.
*/
int searchCell(const Cell* key, const List* l, int start) {
    int i;
```

```
    if(l->length > 0){
        for(i = listIndex(l, l->length - 1); i >= listIndex(l, start); i--) {
            if(isEqualCell(key, l->list[i]))
                return i;
        } /* for */
    }/* if */
   return -1;
} /* searchCell */

/* Riduce una lista eliminando gli elementi da first + 1 a last.
IOP l Lista da ridurre.
IP first Indice dell'ultimo elemento da tenere.
IP last Indice dell'ultimo elemento da eliminare.
*/
void reduceList(List* l, int first, int last) {
   int i;
   for(i = 1; last + i <= l->length - 1; i++)
      l->list[listIndex(l, first + i)] = l->list[listIndex(l, last + i)];
   l->length = l->length - last + first;
}/* reduceList */

/* Restituisce se una cella è presente in una lista.
IP l Lista in cui cercare.
IP ce Cella da cercare.
OR Esito della ricerca.
*/
bool isIn(const List* l, const Cell* ce){
    return searchCell(ce, l, l->first) != -1;
}/* isIn */

/*Ritorna l'ultimo elemento aggiunto.
IP l Lista da cui individuare l'ultimo elemento.
OR Ultimo elemento.
*/
Cell* listGetLast(const List* l){
    return l->list[listIndex(l, l->length - 1)];
}/* listGetLast */

/*Ritorna il primo elemento aggiunto.
IP l Lista da cui individuare il primo elemento.
OR Primo elemento.
*/
Cell* listGetFirst(const List* l){
    return l->list[l->first];
}/* listGetLast */

/* Stampa a video una lista.
IP l Lista da stampare.
OV Contenuto della lista.
*/
void printList(const List* l){
```

```
    int i;
    printf("[");
    if(l->length > 0){
        printf("(%d, %d)", (l->list[l->first])->row, (l->list[l->first])->col);
        for(i = 1; i < l->length; i++)
        printf(",    (%d,    %d)",    (l->list[listIndex(l,    i)])->row,    (l-
>list[listIndex(l, i)])->col);
    }/* if */
     printf("]\n");
}/* printList */

/* Concatena la seconda lista alla prima.
IOP l1 Prima lista.
IP l2 Seconda lista.
*/
void concat(List* l1, List* l2){
    int i;
    for(i = 0; i < l2->length; i++)
        listAdd(l1, l2->list[listIndex(l2, i)]);
}/* concat */

/* Copia la seconda lista nella prima.
IOP l1 Prima lista.
IP l2 Seconda lista.
*/
void copyList(List* l1, const List* l2){
    int i;
    initList(l1);
    for(i = 0; i < l2->length; i++){
        listAdd(l1, l2->list[i]);
    }
}/* copyList */
```

Il file ListArray.h contiene la struttura $ListArray.

```
#include "List.h"

/*
   Array di liste.
*/
typedef struct {
   List* array;/* array contenente le liste */
   int size;/* grandezza dell'array di liste */
   int capacity;/* capacità dell'array di liste */
} ListArray;

void initListArray(ListArray*);
void listArrayFree(ListArray*);
bool isEmptyListArray(const ListArray*);
void doubleListArray(ListArray*);
void listArrayAdd(ListArray*, const List);
void listArrayRemove(ListArray*, int);
void swapList(List*, List*);
int partition(List[], int);
List* quickSelect(List[], int, int);
List* listMin(List[], int);
void copyListArray(ListArray*, const ListArray*);
void printListArray(const ListArray*);
```

Il file ListArray.c contiene le funzioni utili per la struttura $ListArray.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "ListArray.h"

/* Inizializza un array di liste.
IOP Array di liste.
*/
void initListArray(ListArray* l){
    l->array = malloc(sizeof(List));
    assert(l->array != NULL);
    l->capacity = 1;
    l->size = 0;
}/* initListArray */

/* Svuota l'array e libera la sua memoria allocata dinamicamente.
IOP l Array di liste da liberare.
*/
void listArrayFree(ListArray* l) {
      int i;
      for(i = 0; i < l->size; i++)
      listFree(&(l->array[i]));
    free(l->array);
} /* listOfListFree */

/* Verifico se un array di liste è vuoto.
IP l Array di liste.
OR esito della verifica.
*/
bool isEmptyListArray(const ListArray* l){
    return l->size == 0;
}/* isEmptyListArray */

/* Raddoppia la capienza dell'array di liste lista quando questo e` pieno.
IOP l Array di liste di cui raddoppiare la capienza.
*/
void doubleListArray(ListArray* l) {
    int i;
    List* newList;
    if (l->size == l->capacity) {
        newList = malloc(2 * l->capacity * sizeof(List));
        assert(newList != NULL);
        for(i = 0; i < l->size; i++){
            /*initList(newList + i);*/
            copyList(newList + i, l->array + i);
```

```
        }
        l->capacity *= 2;
        listArrayFree(l);
        l->array = newList;
    } /* if */
}/* doubleListArray */

/* Aggiunge un elemento all'array di liste.
IOP l Array di liste a cui aggiungere un elemento.
IP elem Elemento da aggiungere.
*/
void listArrayAdd(ListArray* l, const List elem) {
    doubleListArray(l);
    copyList(l->array + l->size, &elem);
    l->size++;
} /* listAdd */

/*
Elimina un elemento dall'array di liste.
IOP l Array di liste da cui eliminare un elemento.
IP index Indice dell'elemento da eliminare.
*/
void listArrayRemove(ListArray* l, int index){
    int i;
    for(i = index; i < l->size - 1; i++){
        listFree(l->array + i);
        copyList(l->array + i, l->array + i + 1);
    }
    l->size--;
}/* listArrayRemove */

/* Scambia due liste.
IOP l1 Primo elemento da scambiare.
IOP l2 Secondo elemento da scambiare.
*/
void swapList(List* l1, List* l2) {
    List temp;
    temp = *l1;
    *l1 = *l2;
    *l2 = temp;
} /* swapList */

/* Usa l'ultimo elemento come pivot, e poi:
~ Sposta gli elementi piu` piccoli a sinistra del pivot.
~ Sposta gli elementi piu` grandi a destra.
~ Sposta il pivot al confine tra i due insiemi suddetti.
IP l Array di liste da partizionare.
IP n Lunghezza dell'array di liste.
OR Indice del pivot.
*/
int partition(List l[], int n) {
```

```
    int i, indicePivot;
    List *pAI = l; /* Puntatore ad l[i], con i = 0, .., n - 1 */
    List *pASwap = l; /* Puntatore all'elemento di $l che sara` scambiato con
a[i] */
    List *pPivot = l + n - 1; /* Puntatore al pivot, che e` l[n - 1] */
    indicePivot = 0;
    for (i = 0; i < n; i++) {
        if (pAI->length < pPivot->length) {
            swapList(pAI, pASwap);
            pASwap ++;
            indicePivot++;
        } /* if */
        pAI++;
    } /* for */
    swapList(pPivot, pASwap);
    return indicePivot;
} /* partition */

/* Restituisce il puntatore alla k-esima lista piu' corta.
IP l Array di indirizzi da elaborare.
IP n Grandezza di a.
IP k Indice dell'indirizzo da ritornare. 0 <= k < n
OR Lista con la k-esima lunghezza piu' corta.
*/
List* quickSelect(List l[], int n, int k) {
    int indicePivot = partition(l, n);
    if (indicePivot == k)
        return l + indicePivot;
    else if (indicePivot < k) /* Cerca l'elemento in l[indicePivot + 1, ..., n -
1] */
        return quickSelect(l + indicePivot + 1, n - indicePivot - 1, k - indice-
Pivot - 1);
    else /* Cerca l'elemento in l[0, ..., indicePivot - 1] */
        return quickSelect(l, indicePivot, k);
} /* quickSelect */

/* Restituisce un puntatore alla lista di lunghezza minore.
IP l Array su cui cercare.
IP n Lunghezza dell'array.
OR Lista di lunghezza minore.
*/
List* listMin(List l[], int n){
    return quickSelect(l, n, 0);
}/* listMin */

/* Copia il secondo array di liste nel primo.
IOP l1 Primo array di liste.
IP l2 Secondo array di liste.
*/
void copyListArray(ListArray* l1, const ListArray* l2){
    int i;
```

```
    l1->size = 0;
    for(i = 0; i < l2->size; i++)
        listArrayAdd(l1, l2->array[i]);
}/* copyListArray */

/* Stampa su video un array di liste.
IP l Array di liste da stampare.
OV Contenuto dell'array di liste.
*/
void printListArray(const ListArray* l){
    int i;
    printf("[ ");
    printList(l->array);
    for(i = 1; i < l->size; i++){
        printf(", ");
        printList(l->array + i);
    }/* for */
    printf(" ]\n");
}/* printListArray */
```

Il file Maze.h contiene la struttura $Maze utile a rappresentare il labirinto.

```
#include "ListArray.h"

/*
Labirinto a pianta rettangolare.
*/
typedef struct {
   int w, h;/* larghezza (numero di celle per riga) ed altezza (numero di celle
per colonna) del labirinto */
   Cell** grid; /* griglia contenente la topologia del labirinto */
   Cell* start; /* puntatore alla cella d'inizio */
   Cell* end; /* puntatore alla cella da raggiungere */
} Maze;

char** saveMaze(FILE*, int*, int*);
bool isValid(int, int, int, int);
void createMazeGrid(char**, Maze*, int, int);
void findUnblockedNeighbours(const Cell*, List*);
void findUnvisitedNeighbours(const Cell*, bool**, List*);
bool withinOneMove(const Cell*, const Cell*);
bool areLinked(const Cell*, const Cell*);
bool isJunction(const Cell*, const Maze*);
bool isDeadEnd(const Cell*, const Maze*);
bool isBlocked(const Cell*);
void blockCell(Maze*, Cell*);
bool isWallRight(const Cell*, int);
bool isWallFront(const Cell*, int);
int turnClockwise(const Cell*, int);
int turnCounterClockwise(const Cell*, int);
Cell* moveForward(const Cell*, int);
void pruneSolution(List*);
void clean(char**, const Maze*);
void printSolution(char**, const Maze*, const List*);
void mazeFree(Maze*);
```

Il file Maze.c contiene le funzioni inerenti il labirinto.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>


#include "Maze.h"


/* Salva un labirinto rappresentato tramite una griglia di caratteri a partire
da un file di testo.
   Sono usati i seguenti caratteri del codice ASCII:
   x3 ' ': cella del labirinto,
   ' ': collegamento tra due celle adiacenti,
   '+': angoli delle celle,
   x3 '-': muro orizzontale,
   '|': muro verticale,
   'S': cella di partenza,
   'E': cella da raggiungere.
IF inF File di input.
OP w Larghezza.
OP h Altezza.
OR array 2D di caratteri contenente la topologia del labirinto.
*/
char** saveMaze(FILE* inF, int* w, int* h){
   char** a;
   int i, j;
   fscanf(inF, "%d", w);
   fscanf(inF, "%d", h);
   a = malloc(sizeof(char*) * (2 * (*h) + 1));
   for(i = 0; i < 2 * (*h) + 1; i++){
      a[i] = malloc((4 * (*w) + 2) * sizeof(char));
      for(j = 0; j < 4 * (*w) + 2; j++)
         fscanf(inF, "%c", &a[i][j]);
   }/* for */
   return a;
}/* saveMaze*/

/* Controllo se le coordinate della cella sono valide.
IP i Riga della cella.
IP j Colonna della cella.
IP w Larghezza del labirinto.
IP h Altezza del labirinto.
*/
bool isValid(int i, int j, int w, int h){
   return i >= 0 && j >= 0 && i < h && j < w;
}/* isValid */
```

```
/* Costruisco il labirinto a partire dall'array 2D di caratteri del codice ASCII
contenente la sua topologia.
IP a Array2D contenente la rappresentazione grafica tramite caratteri del codice
ASCII del labirinto.
OP maze Labirinto.
IP w Larghezza del labirinto.
IP h Altezza del labirinto.
*/
void createMazeGrid(char** a, Maze* maze, int w, int h){
   int i, j;
   maze->w = w;
   maze->h = h;
   maze->grid = malloc(sizeof(Cell*) * h);
   for(i = 0; i < h; i++)
      maze->grid[i] = malloc(sizeof(Cell) * w);

   for(i = 0; i < h; i++){
      for(j = 0; j < w; j++){

         /* Riga della cella */
         (maze->grid[i][j]).row = i;
         /* Colonna della cella */
         (maze->grid[i][j]).col = j;
         /* Cella di inizio */
         if(a[2*i+1][4*j+3] == 'S')
            maze->start = &(maze->grid[i][j]);
         /* Cella da raggiungere */
         else if(a[2*i+1][4*j+3] == 'E')
            maze->end = &(maze->grid[i][j]);
         /* Se la cella appartiene alla prima riga metto un muro in alto */
         if(i == 0){
            (maze->grid[i][j]).walls[UP] = 1;
            (maze->grid[i][j]).up = NULL;
         }/* if */
         /* Se la cella appartiene alla prima colonna metto un muro a sinistra
*/
         if(j == 0){
            (maze->grid[i][j]).walls[LEFT] = 1;
            (maze->grid[i][j]).left = NULL;
         }/* if */

         /* Ora basta inserire per ogni cella i muri in basso e a destra se pre-
senti */

         /* Se ci sono tre '-' vicini nell'array $a allora vuol dire che è pre-
sente un muro orizzontale in basso nella cella corrente */
         if((a[2*i+2][4*j+3]   ==   '-')   &&   (a[2*i+2][4*j+2]   ==   '-')   &&
(a[2*i+2][4*j+4] == '-')){
            (maze->grid[i][j]).walls[DOWN] = 1;
            (maze->grid[i][j]).down = NULL;
```

```
            /* controllo che la cella in basso esista */
            if(isValid(i+1, j, w, h)){
               /* aggiungo un muro in alto alla cella che si trova in basso ri-
spetto alla corrente */
               (maze->grid[i+1][j]).walls[UP] = 1;
               (maze->grid[i+1][j]).up = NULL;
            }/* if */
         }/* if */
         /* altrimenti la cella corrente comunica con quella in basso */
         else{
            (maze->grid[i][j]).walls[DOWN] = 0;
            (maze->grid[i][j]).down = &(maze->grid[i+1][j]);
            /* controllo che la cella in basso esista */
            if(isValid(i+1, j, w, h)){
               (maze->grid[i+1][j]).walls[UP] = 0;
               (maze->grid[i+1][j]).up = &(maze->grid[i][j]);
            }/* if */
         }/* else */
         /* Se c'e '|' nell'array $a allora vuol dire che è presente un muro ve-
ricale a destra nella cella corrente */
         if(a[2*i+1][4*j+5] == '|'){
            (maze->grid[i][j]).walls[RIGHT] = 1;
            (maze->grid[i][j]).right = NULL;
            /* controllo che la cella in basso esista */
            if(isValid(i, j+1, w, h)){
               /* aggiungo un muro a sinistra alla cella che si trova a destra
rispetto alla corrente */
               (maze->grid[i][j+1]).walls[LEFT] = 1;
               (maze->grid[i][j+1]).left = NULL;
            }/* if */
         }/* if */
         /* altrimenti la cella corrente comunica con quella a destra */
         else{
            (maze->grid[i][j]).walls[RIGHT] = 0;
            (maze->grid[i][j]).right = &(maze->grid[i][j+1]);
            /* controllo che la cella in basso esista */
            if(isValid(i, j+1, w, h)){
               (maze->grid[i][j+1]).walls[LEFT] = 0;
               (maze->grid[i][j+1]).left = &(maze->grid[i][j]);
            }/* if */
         }/* else */
      }/*for*/
   }/*for*/
}/* createMazeGrid */


/* Trova tutte le celle vicine alla cella corrente; visitate, o no.
IP ce Cella.
IOP ns Lista contente i puntatori alle celle vicine alla cella puntata da $ce.
*/
void findUnblockedNeighbours(const Cell* ce, List* ns) {
```

```
   initList(ns);
   /* controllo in alto */
   if (!ce->walls[UP]){
      listAdd(ns, ce->up);
   }
   /* controllo in basso */
   if (!ce->walls[DOWN]){
      listAdd(ns, ce->down);
   }
   /* controllo a sinistra*/
   if(!ce->walls[LEFT]){
      listAdd(ns, ce->left);
   }
   /*controllo a destra*/
   if (!ce->walls[RIGHT]){
      listAdd(ns, ce->right);
   }
   if(ns->length != 0)
      shuffle(ns);
}/* findUnblockedNeighbors */

/* Trova i vicini di una cella che non sono ancora stati visitati.
IP ce Cella.
IP visited Array 2D che tiene traccia se una cella e' stata visitata o no nella
posizione corrispondente alle sue coordinate.
IOP ns Lista contenente i vicini non visitati.
*/
void findUnvisitedNeighbours(const Cell* ce, bool** visited, List* ns) {
   initList(ns);
   /* controllo in alto */
   if (!ce->walls[UP] && !visited[ce->row-1][ce->col]){
      listAdd(ns, ce->up);
   }
   /* controllo in basso */
   if (!ce->walls[DOWN] && !visited[ce->row+1][ce->col]){
      listAdd(ns, ce->down);
   }
   /* controllo a sinistra*/
   if(!ce->walls[LEFT] && !visited[ce->row][ce->col-1]){
      listAdd(ns, ce->left);
   }
   /*controllo a destra*/
   if (!ce->walls[RIGHT] && !visited[ce->row][ce->col+1]){
      listAdd(ns, ce->right);
   }
   if( ns->length == NEIGHBOURS - 1){
      ns->list = realloc(ns->list, sizeof(Cell) * ns->length);
   }
   if(!isEmptyList(ns))
        shuffle(ns);
}/* findUnvisitedNeighbors */
```

```
/* Verifico se due celle distano una mossa.
IP ce1 Prima cella.
IP ce2 Seconda cella.
OR Ritorna true se due celle sono distano una mossa, altrimenti false.
*/
bool withinOneMove(const Cell* ce1, const Cell* ce2){
   return ( ((ce2->row - ce1->row == 0) && (fabs(ce2->col - ce1->col) == 1))
          || ((ce2->col - ce1->col == 0) && (fabs(ce2->row - ce1->row) == 1)) );

}/* withinOneMove */


/* Verifica se due celle sono collegate direttamente.
IP ce1 Prima cella.
IP ce2 Seconda cella.
OR Ritorna true se due celle sono collegate, altrimenti false.
*/
bool areLinked(const Cell* ce1, const Cell* ce2){
   if(isEqualCell(ce1, ce2))
      return true;
   if(withinOneMove(ce1, ce2)){
      if((ce2->row - ce1->row) == 1)
         return !ce1->walls[DOWN];
      else if((ce2->row - ce1->row) == -1)
         return !ce1->walls[UP];
      else if((ce2->col - ce1->col) == 1)
         return !ce1->walls[RIGHT];
      else
         return !ce1->walls[LEFT];
   }/* if */
   return false;
}/* areLinked */


/* Ritorna se una cella è una giunzione: è collegata con 3 o 4 celle.
IP ce Cella in questione.
OR Ritorna true se la cella è bloccata, false altrimenti.
*/
bool isJunction(const Cell* ce, const Maze* m){
   int sum, i;
   sum = 0;
   for(i = 0; i < NEIGHBOURS; i++)
      sum += ce->walls[i];
   return ((sum <= 1) && !isEqualCell(ce, m->end) && !isEqualCell(ce, m-
>start));
}/* isJunction */

/* Ritorna se una cella è un vicolo cieco: ha un muro in 3 dei suoi lati.
IP ce Cella in questione.
OR Ritorna true se la cella è bloccata, false altrimenti.
```

```
*/
bool isDeadEnd(const Cell* ce, const Maze* m){
   int sum, i;
   sum = 0;
   for(i = 0; i < NEIGHBOURS; i++)
      sum += ce->walls[i];
   /* una cella e' un vicolo cieco se ha 3 muri */
   return ((sum == NEIGHBOURS - 1) && !isEqualCell(ce, m->end) && !isEqual-
Cell(ce, m->start));

}/* isDeadEnd */

/* Ritorna se una cella è bloccata su tutti i suoi lati.
IP ce Cella in questione.
OR Ritorna true se la cella è bloccata, false altrimenti.
*/
bool isBlocked(const Cell* ce){
   int sum, i;
   sum = 0;
   for(i = 0; i < NEIGHBOURS; i++)
      sum += ce->walls[i];
   /* una cella e' un vicolo cieco se ha 4 muri */
   return sum == NEIGHBOURS;
}/* isBlocked */

/* Blocca una cella.
IOP m Labirinto.
IOP Cella da bloccare.
*/
void blockCell(Maze* m, Cell* ce){

    if(ce->walls[UP] == 0){
        ce->walls[UP] = 1;
        if(isValid(ce->row - 1, ce->col, m->w, m->h))
            (ce->up)->walls[DOWN] = 1;
    }/* if */

    if(ce->walls[DOWN] == 0){
        ce->walls[DOWN] = 1;
        if(isValid(ce->row + 1, ce->col, m->w, m->h))
            (ce->down)->walls[UP] = 1;
    }/* if */

    if(ce->walls[LEFT] == 0){
        ce->walls[LEFT] = 1;
        if(isValid(ce->row, ce->col - 1, m->w, m->h))
            (ce->left)->walls[RIGHT] = 1;
    }/* if */

    if(ce->walls[RIGHT] == 0){
        ce->walls[RIGHT] = 1;
```

```
        if(isValid(ce->row, ce->col + 1, m->w, m->h))
            (ce->right)->walls[LEFT] = 1;
    }/* if */
}/* blockCell */


/* Controlla se c'e' un muro a destra.
IP ce Cella.
IP dir Direzione attuale.
OR Esito della verifica.
*/
bool isWallRight(const Cell* ce, int dir){
   return ce->walls[(dir + 1) % NEIGHBOURS];
}/* is WallRight */

/* Controlla se c'e' un muro di fronte.
IP ce Cella.
IP dir Direzione attuale.
OR Esito della verifica
*/
bool isWallFront(const Cell* ce, int dir){
   return ce->walls[dir];
}/* is WallFront */

/* Gira in senso orario.
IP ce Cella.
IP dir Direzione attuale.
OR Nuova direzione.
*/
int turnClockwise(const Cell* ce, int dir){
   return (dir + 1) % NEIGHBOURS;
}/* turnClockwise */

/* Gira in senso antiorario.
IP ce Cella.
IP dir Direzione attuale.
OR Nuova direzione.
*/
int turnCounterClockwise(const Cell* ce, int dir){
   return (dir + NEIGHBOURS - 1) % NEIGHBOURS;
}/* turnCounterClockwise */

/* Muove avanti di una cella.
IP ce Cella.
IP dir Direzione attuale.
OR Nuova cella.
*/
Cell* moveForward(const Cell* ce, int dir){
   if(dir == UP)
      return ce->up;
   else if (dir == RIGHT)
```

```
      return ce->right;
   else if(dir == DOWN)
      return ce->down;
   else
      return ce->left;
}/* moveForward */

/* Elimina le celle duplicate dalla soluzione.
IOP solution Lista contenente le celle appartenenti alla soluzione.
*/
void pruneSolution(List* solution){
   int i, first_i, last_i, index;
   bool found = true;
   int attempt = 0;
   int maxAttempt = solution->length;

   while(found && solution->length > 1 && attempt < maxAttempt) {
      found = false;
      attempt++;

      for(i = 0; i < solution->length - 1; i++) {
         index = searchCell(solution->list[listIndex(solution, i)], solution,
listIndex(solution, i+1));
         if (index != -1) {
            first_i = listIndex(solution, i);
            last_i = listIndex(solution, index);
            found = true;
            break;
         }/* if */
      }/* for */
      if(found)
         reduceList(solution, first_i, last_i);
   }/* while */

   solution->list = realloc(solution->list, solution-> length * sizeof(Cell));
}/* pruneSolution */

/* Ripulisce la rappresentazione grafica del labirinto da celle contrassegnate
con 'X'.
IOP a Array2D contenente la rappresentazione grafica tramite caratteri del codi-
ce ASCII del labirinto.
IP m Labirinto.
*/
void clean(char** a, const Maze* m){
   int i, j;
   for(i = 0; i < m->h; i++){
      for(j = 0; j < m->w; j++){
         if(a[2*i+1][4*j+3] == 'X')
            a[2*i+1][4*j+3] = ' ';
      }/* for */
   }/* for */
```

```
}/* clean */

/* Stampa a video il labirinto con la soluzione, contrassegnando con 'X' le cel-
le appartenenti ad essa.
IP a Array2D contenente la rappresentazione grafica tramite caratteri del codice
ASCII del labirinto.
IP m Labirinto.
IP solution Lista contenente le celle appartenenti alla soluzione.
OV Labirinto con la soluzione.
*/
void printSolution(char** a, const Maze* m, const List* solution ){
   int i, j;
   for(i = 0; i < m->h; i++){
      for(j = 0; j < m->w; j++){
         if(isIn(solution,  &m->grid[i][j])  &&    a[2*i+1][4*j+3]  !=  'S'  &&
a[2*i+1][4*j+3] != 'E')
            a[2*i+1][4*j+3] = 'X';
      }/* for */
   }/* for */
   for(i = 0; i < 2 * m->h + 1; i++){
      for(j = 0; j < 4 * m->w + 2; j++){
         printf("%c", a[i][j]);
      }/* for */
   }/* for */
}/* printSolution */

/* Libera la memoria occupata dal labirinto.
IOP m Labirinto da liberare.
*/
void mazeFree(Maze* m){
   int i;
   for(i = 0; i < m->h; i++)
      free(m->grid[i]);
   free(m->grid);
   m->h = 0;
   m->w = 0;
   m->start = NULL;
   m->end = NULL;
}/* mazeFree */
```

Il file RandomMouse.c contiene l'algoritmo di risoluzione Random mouse ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/* Random Mouse: il risolutore procede seguendo il passaggio corrente fino al raggiungimento di un incrocio,
   quindi prendere una decisione casuale sulla direzione successiva da seguire, evitando di girarsi di 180° (tornare indietro) ove possibile.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
OP k Numero di passi.
*/
void RandomMouse(const Maze* m, List* solution, int* k){
      List ns;
      Cell* nxt;
    srand(time(NULL)); /* inizializzo un random seed */
    /* inizializzo la soluzione */
      initList(solution);
    /* inizializzo la lista che contiene i vicini di una cella */
      initList(&ns);
    /* aggiungo la cella iniziale alla soluzione */
      listAdd(solution, m->start);
    /* itero finche' non raggiungo la fine */
      while(!isEqualCell(solution->list[listIndex(solution, solution->length - 1)], m->end)){
        /* trovo i vicini della cella corrente */
        findUnblockedNeighbours(solution->list[listIndex(solution, solution->length - 1)], &ns);
        /* evita di tornare indietro */
        if(ns.length > 1 && solution->length > 1)
            listRemove(&ns, searchCell(solution->list[listIndex(solution, solution->length - 2)], &ns, ns.first));
        /* scelgo casualmente un vicino */
        nxt = chooseCell(&ns);
        /*libero la lista contenente le celle vicine alla corrente */
        listFree(&ns);
        /* aggiungo la prossima cella alla soluzione */
        listAdd(solution, nxt);
        /* incremento il numero di passi */
        (*k)++;
      }/* while */
    /* elimino le celle ripetute dalla soluzione */
    pruneSolution(solution);
}/* RandomMouse*/
```

```
int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h, k;
    char** a;
    Maze m;
    k = 0;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    RandomMouse(&m, &solution, &k);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    printf("\nPassi: %d\n", k);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file WallFollower.c contiene l'algoritmo di risoluzione Wall Follower ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/*Right Hand Wall Follower: algoritmo che risolve un labirinto che ha i muri connessi tra loro e le celle di partenza e di arrivo
  hanno un muro in almeno uno dei loro 4 lati.
  Utilizza la seguente logica rispetto alla direzione di percorrenza ("regola della mano destra"):
  1) se non c'e' un muro a destra ruota di 90° in senso orario e si muove in avanti;
  2) se c'e' un muro a destra:
           2.1) se non c'e' un muro di fronte si muove in avanti;
           2.2) se c'e' un muro di fronte ruota di 90° in senso antiorario.

IP m Labirinto.
IOP solution Lista contenente la soluzione.
OP k Numero di passi.
*/
void RightHand(const Maze* m, List* solution, int* k){
      Cell* curr;
      int dir;
      srand(time(NULL)); /* inizializzo un random seed */
      initList(solution);
      dir = rand() % NEIGHBOURS;
      curr = m->start;
      listAdd(solution, curr);
      while(!isEqualCell(curr, m->end)){
            if(isWallRight(curr, dir)){
                  if(isWallFront(curr, dir))
                        dir = turnCounterClockwise(curr, dir);
                  else{
                        curr = moveForward(curr, dir);
                        listAdd(solution, curr);
                        (*k)++;
                  }/* else */
            }/* if */
            else{
                  dir = turnClockwise(curr, dir);
                  curr = moveForward(curr, dir);
                  listAdd(solution, curr);
                  (*k)++;
            }/* else */
      }/* while */
      pruneSolution(solution);
```

```
}/* RightHand */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h, k;
    char** a;
    Maze m;
    k = 0;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    RightHand(&m, &solution, &k);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    printf("\nPassi: %d\n", k);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file Pledge.c contiene l'algoritmo di risoluzione di Pledge, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/*Right Hand Wall Follower: algoritmo che risolve un labirinto che ha le mura connesse tra loro, opportunamente modificato per Pledge.
  Utilizza la seguente logica rispetto alla direzione di percorrenza ("regola della mano destra"):
  1) se non c'e' un muro a destra ruota di 90° in senso orario e si muove in avanti;
  2) se c'e' un muro a destra:
           2.1) se non c'e' un muro di fronte si muove in avanti;
           2.2) se c'e' un muro di fronte ruota di 90° in senso antiorario.

IP m Labirinto.
IOP solution Lista contenente la soluzione.
IOP k Numero di passi.
IOP curr Cella corrente.
IOP count Contatore, viene incrementato di 1 per ogni rotazione in senso orario, decrementato di 1 per ogni rotazione in senso antiorario.
IP mainDir Direzione principale.
*/
void RightHand(const Maze* m, List* solution, int* k, Cell** curr, int* count, int mainDir){
      int dir;
      dir = turnCounterClockwise(*curr, mainDir);
      (*count)--;
      while(!isEqualCell(*curr, m->end) && *count != 0){
            if(isWallRight(*curr, dir)){
                  if(isWallFront(*curr, dir)){
                        dir = turnCounterClockwise(*curr, dir);
                        (*count)--;
                  }/* if */
                  else{
                        *curr = moveForward(*curr, dir);
                        listAdd(solution, *curr);
                        (*k)++;
                  }/* else */
            }/* if */
            else{
                  dir = turnClockwise(*curr, dir);
                  (*count)++;
                  *curr = moveForward(*curr, dir);
                  listAdd(solution, *curr);
                  (*k)++;
```

```
            }/* else */
      }/* while */
}/* RightHand */


/*Algoritmo di Pledge: progettato per aggirare gli ostacoli, richiede una direzione scelta arbitrariamente verso cui andare,
  che sarà preferenziale.
  Quando viene incontrato un ostacolo, una mano (la mano destra in questo implementazione) viene mantenuta lungo l'ostacolo,
  mentre vengono contati gli angoli ruotati (la rotazione in senso orario è positiva, la rotazione in senso antiorario è negativa).
  Quando il solutore si trova di nuovo nella direzione preferenziale originale, e la somma angolare delle virate eseguite è 0,
  il solutore lascia l'ostacolo e continua a muoversi nella sua direzione originale.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
OP k Numero di passi.
*/
void Pledge(const Maze* m, List* solution, int* k){
      Cell* curr;
      int dir, count, mainDir;
      srand(time(NULL)); /* inizializzo un random seed */
      initList(solution);
      mainDir = rand() % NEIGHBOURS;
      dir = mainDir;
      count = 0;
      curr = m->start;
      listAdd(solution, curr);
      while(!isEqualCell(curr, m->end)){
            if(!isWallFront(curr, mainDir)){
                  curr = moveForward(curr, dir);
                  listAdd(solution, curr);
                  (*k)++;
            }/* if */
            else{
                  RightHand(m, solution, k, &curr, &count, mainDir);
                  dir = mainDir;
            }/* else */
      }/* while */
      pruneSolution(solution);
}/* Pledge */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h, k;
    char** a;
    Maze m;
    k = 0;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
```

```
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    Pledge(&m, &solution, &k);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    printf("\nPassi: %d\n", k);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file Tremaux.c contiene l'algoritmo di risoluzione di Trémaux, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/* Restituisce la prossima cella da visitare, utilizzando la logica di Tremaux.
IOP visitedCells Array 2D contenete nelle posizioni corrispondenti quante volte sono state visitate le celle.
IP ns Lista contente i vicini alla cella corrente.
IOP solution Lista contenente la soluzione.
OR Prossima cella da visitare.
*/
Cell* whatNext(int** visitedCells, const List* ns, List* solution){
        List* visitCounts;
        int i;
        Cell* ce;
        /* gestisco lo scenario in cui vi sia una sola cella vicina alla corrente*/
        if(ns->length == 1)
            ce = ns->list[0];
        else{
            /* organizzo i vicini in base a quante volte sono stati visitati */
            visitCounts = malloc(3 * sizeof(List));
            /* inizializzo un array di 3 liste: nell'indice 0 c'e' la lista che conterra' le celle mai visitate, all'indice 1 quella che
            conterra' le celle visitate 1 volta, all'indice 2 quella che conterra' le celle visitate 2 volte */
            for(i = 0; i < 3; i++){
            initList(visitCounts + i);
            }/* for */
            for(i = 0; i < ns->length; i++)
                listAdd(visitCounts    +    visitedCells[(ns->list[i])->row][(ns->list[i])->col], ns->list[i]);

            /*
            rispetto la regola di Tremaux quando mi trovo in una cella: se ci sono celle che non sono mai state visitate ne scelgo una,
            altrimenti scelgo una cella che e' gia' stata visitata solo una volta, altrimenti ancora scelgo una cella gia' visitata piu' volte.
            Rimuovo dalla lista dei vicini la cella precedentemente visitata.
            */

            if(visitCounts->length != 0)
                ce = chooseCell(visitCounts);

            else if((visitCounts + 1)->length != 0){
                if((visitCounts + 1)->length > 1 &&
```

```
                    solution->length > 1 &&
                    isIn(visitCounts + 1, solution->list[solution->length - 2
]))
                    listRemove(visitCounts + 1, searchCell(solution-
>list[solution->length - 2], visitCounts + 1, (visitCounts + 1)->first));
                ce = chooseCell(visitCounts + 1);
            }/* else if */

            else{
                if((visitCounts + 2)->length > 1 &&
                    solution->length > 1 &&
                    isIn(visitCounts + 2, solution->list[solution->length - 2
]))
                    listRemove(visitCounts + 2, searchCell(solution-
>list[solution->length - 2], visitCounts + 2, (visitCounts + 2)->first));
                ce = chooseCell(visitCounts + 2);
            }/* else */

            for( i = 0; i < 3; i++)
                listFree(&visitCounts[i]);
            free(visitCounts);
        }/* else */
        return ce;
}/* whatNext */

/* Incremento le volte in cui e' stata visitata una cella di 1.
IP ce Cella visitata.
IOP visitedCells Array 2D contenete nelle posizioni corrispondenti quante volte
sono state visitate le celle.
*/
void visit(Cell* ce, int** visitedCells){
    visitedCells[ce->row][ce->col] += 1;
}/* visit */

/* Algoritmo di Tremaux.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
OP k Numero di passi eseguiti.
*/
void Tremaux(const Maze* m, List* solution, int* k){
    int i, j;
    List ns;
    Cell* next;
    int** visitedCells;
    srand(time(NULL)); /* Inizializzo un random seed */
    initList(solution);
    /* alloco ed inizializzo l'array 2D contenete nelle posizioni corrispondenti
quante volte sono state visitate le celle.*/
    visitedCells = malloc(m->h * sizeof(int*));
    for(i = 0; i < m->h; i++){
        visitedCells[i] = malloc(m->w * sizeof(int));
        for(j = 0; j < m->w; j++)
```

```
            visitedCells[i][j] = 0;
    }/* for */

    /* prima mossa */
    listAdd(solution, m->start);
    visit(m->start, visitedCells);

    /* prendo un vicino a caso usando la logica di Tremaux, ripeto finche' arri-
vo alla fine */
    while(!isEqualCell(solution->list[solution->length - 1], m->end)){
        /* trovo i vicini alla cella corrente */
        findUnblockedNeighbours(solution->list[solution->length - 1], &ns);

        /* scelgo e visito la prossima cella basandomi sulla logica di Tremaux*/
        next = whatNext(visitedCells, &ns, solution);
        listFree(&ns);
        /* incremento i passi */
        (*k)++;
        /* aggiungo la cella alla soluzione*/
        listAdd(solution, next);
        /* aggiorno il numero di volte in cui e' stata  visitata la cella solo
se la cella e' stata visitata meno di 2 volte */
        if(visitedCells[next->row][next->col] < 2)
            visit(next, visitedCells);
    }/* while */
    /* elimino dalla soluzione le celle presenti piu' volte */
    pruneSolution(solution);

    for(i = 0; i < m->h; i++)
        free(visitedCells[i]);
    free(visitedCells);
}/* Tremaux */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h, k;
    char** a;
    Maze m;
    k = 0;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    Tremaux(&m, &solution, &k);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    printf("\nPassi: %d\n", k);
    listFree(&solution);
```

```
    mazeFree(&m);
    return 0;
}/* main */
```

Il file DeadEndFiller.c contiene l'algoritmo di risoluzione Dead End Filler, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"


/* Ritorna se trova vicoli ciechi.
IP m Labirinto.
IOP deadEnd Vicolo cieco.
OR Esito della ricerca.
*/
bool findDeadEnd(const Maze* m, Cell** deadEnd){
      int i, j;
    for(i = 0; i < m->h; i++){
        for(j = 0; j < m->w; j++){
            if(isDeadEnd(&(m->grid[i][j]), m)){
                  *deadEnd = &(m->grid[i][j]);
                return true;
            }/* if */
        }/* for */
    }/* for */
    return false;
}/* findDeadEnd */

/* Riempie un passaggio.
IOP m Labirinto.
IP deadEnd Vicolo cieco.
*/
void fillPassage(Maze* m, Cell** deadEnd){
    List ns;
    /* dal vicolo cieco, mi muovo di una cella */
    findUnblockedNeighbours(*deadEnd, &ns);
    /* blocca il vicolo cieco */
    blockCell(m, *deadEnd);
    /* guardo alla prossima cella, se e' un vicolo cieco proseguo il ciclo */
    while(isDeadEnd(ns.list[0], m)){
        *deadEnd = ns.list[0];
        listFree(&ns);
        /* dal vicolo cieco, mi muovo di una cella */
        findUnblockedNeighbours(*deadEnd, &ns);
        /* blocca il vicolo cieco */
        blockCell(m, *deadEnd);
    }/* while */
}/* fillPassage */
```

```
/* Blocca tutti vicoli ciechi che si vengono a creare.
IOP m Labirinto.
OR Ritorna true se viene bloccata qualche cella, altrimenti false.
*/
void fillDeadEnds(Maze* m){
    Cell* deadEnd;
    deadEnd = NULL;
    while(findDeadEnd(m, &deadEnd)){
        fillPassage(m, &deadEnd);
    }
}/* fillDeadEnds */

/* DeadEndFiller: algoritmo di risoluzione per labirinti perfetti, riempie tutti
i vicoli ciechi fino ad ottenere la soluzione.
IOP m Labirinto.
IOP solution Lista contenente la soluzione.
*/
void DeadEndFiller(Maze* m, List* solution){
    int i, j;
    fillDeadEnds(m);
    initList(solution);
    for(i = 0; i < m->h; i++){
        for(j = 0; j < m->w; j++){
            if(!isBlocked(&(m->grid[i][j])))
                  listAdd(solution, &(m->grid[i][j]));
        }/* for */
    }/* for */
}/* DeadEndFiller */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h;
    char** a;
    Maze m;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    DeadEndFiller(&m, &solution);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file RecursiveBacktracker.c contiene l'algoritmo di risoluzione Ricursive Backtracker ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/* Recursive Backtracker: analogo ad una DFS in un grafo, che termina una volta raggiunta la cella finale.
IP m Labirinto.
IOP Lista contenente la soluzione.
OP Numero di passi.
*/
void RecursiveBacktracker(const Maze* m, List* solution, int* k){
  List ns;
  Cell* curr;
  int i, j;
  bool** visited;
  /* alloco ed inizializzo l'array che memorizza se una cella e' gia' stata visitata */
  visited = malloc(m->h * sizeof(bool*));
  for(i = 0; i < m->h; i++){
      visited[i] = malloc(m->w * sizeof(bool));
      for(j = 0; j < m->w; j++)
            visited[i][j] = false;
  }/* for */
  srand(time(NULL)); /* inizializzo un random seed */
  /* inizializzo la lista che conterra' la soluzione, che si comporta come una pila */
  initList(solution);
  initList(&ns);
  /* aggiungo la cella iniziale */
  listAdd(solution, m->start);
  /* cella corrente, ultima inserita */
  curr = listGetLast(solution);
  /* contrassegno la cella come visitata */
  visited[curr->row][curr->col] = true;
  *k = 0;
  while(!isEqualCell(curr, m->end)){
    /* trovo i vicini alla cella corrente */
    findUnvisitedNeighbours(curr, visited, &ns);
    /* aggiungo un vicino alla cella corrente casuale alla pila, se non ci sono rimuovo l'ultimo elemento inserito nella pila*/
    if(!isEmptyList(&ns))
        listAdd(solution, chooseCell(&ns));
    else
        listRemoveLast(solution);
    listFree(&ns);
```

```
    /* cella corrente, ultima inserita */
    curr = listGetLast(solution);
    /* contrassegno la cella come visitata */
    visited[curr->row][curr->col] = true;
    /* incremento i passi */
    (*k)++;
  }/* while */
  for(i = 0; i < m->h; i++)
    free(visited[i]);
  free(visited);
}/* RecursiveBacktracker */

int main(int argc, char **argv){
    FILE* inF;
    Maze m;
    int w, h, k;
    List solution;
    char** a;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    k = 0;
    RecursiveBacktracker(&m, &solution, &k);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    printf("\nPassi: %d\n", k);
    listFree(&solution);
    return 0;
}/* main */
```

Il file Chain.c contiene l'agoritmo di risoluzione Chain, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"


/*Salva la linea guida da seguire per raggiungere la cella finale.
IP m Labirinto.
IOP path Lista contenente la linea guida.
*/
void drawGuidingLine(const Maze* m, List* path){
    Cell* curr;
    int rdiff, cdiff;
    curr = m->start;
    listAdd(path, curr);
    while(!isEqualCell(curr, m->end)){
        rdiff = 0;
        cdiff = 0;
        if (fabs(curr->row - (m->end)->row) > 0){
            if(curr->row < (m->end)->row)
                  rdiff = 1;
            else
                  rdiff = -1;
        }/* if */
        if (fabs(curr->col - (m->end)->col) > 0){
            if(curr->col < (m->end)->col)
                  cdiff = 1;
            else
                  cdiff = -1;
        }/* if */
        curr = &m->grid[curr->row + rdiff][curr->col + cdiff];
        listAdd(path, curr);
    }/* while */
}/* drawGuidingLine */

/* Tenta la mossa diretta se due celle consecutive della linea guida sono colle-
gate tra loro.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
IP curr Cella corrente.
IP Cella della linea guida successiva da raggiungere.
OR Spostamento avvenuto o no.
*/
bool tryDirectMove(const Maze* m, List* solution,   const Cell* curr, Cell*
next){
    int rdiff, cdiff;
```

```
    rdiff = next->row - curr->row;
    cdiff = next->col - curr->col;
    /* calculate the cell next door and see if it's open */
    if (rdiff == 0 || cdiff == 0){
        if (areLinked(curr, next)){
            listAdd(solution, next);
            return true;
        }/* if */
    }/* if */
    else{
        if(areLinked(curr, &(m->grid[curr->row + rdiff][curr->col])) && areL-
inked(next, &(m->grid[curr->row + rdiff][curr->col]))){
            listAdd(solution, &m->grid[curr->row + rdiff][curr->col]);
            listAdd(solution, next);
            return true;
        }/* if */
        else if(areLinked(curr, &(m->grid[curr->row][curr->col + cdiff])) && ar-
eLinked(next, &(m->grid[curr->row][curr->col + cdiff]))){
            listAdd(solution, &m->grid[curr->row][curr->col + cdiff]);
            listAdd(solution, next);
            return true;
        }/* else if */
        else
            return false;
    }
      return false;
}/* tryDirectMove */

/* BacktrackingSolver: analogo ad una DFS in un grafo, che termina una volta
raggiunta la cella finale.
IOP solution Lista contenente la soluzione.
IP goal Cella da raggiungere.
IP m Labirinto.
*/
void BacktrackingSolver(List* solution, const Cell* goal, const Maze* m){
  List ns;
  Cell* curr;
  int i, j;
  bool** visited;
  /* alloco ed inizializzo l'array che memorizza se una cella e' gia' stata vi-
sitata */
  visited = malloc(m->h * sizeof(bool*));
  for(i = 0; i < m->h; i++){
    visited[i] = malloc(m->w * sizeof(bool));
    for(j = 0; j < m->w; j++)
        visited[i][j] = false;
  }/* for */
  initList(&ns);
  /* cella corrente, ultima inserita */
  curr = listGetLast(solution);
  /* contrassegno la cella come visitata */
```

```
  visited[curr->row][curr->col] = true;
  while(!isEqualCell(curr, goal)){
    /* trovo i vicini alla cella corrente */
    findUnvisitedNeighbours(curr, visited, &ns);
    /* aggiungo un vicino alla cella corrente casuale alla pila, se non ci sono
rimuovo l'ultimo elemento inserito nella pila*/
    if(!isEmptyList(&ns))
        listAdd(solution, chooseCell(&ns));
    else
        listRemoveLast(solution);
    listFree(&ns);
    /* cella corrente, ultima inserita */
    curr = listGetLast(solution);
    /* contrassegno la cella come visitata */
    visited[curr->row][curr->col] = true;
  }/* while */
  for(i = 0; i < m->h; i++)
    free(visited[i]);
  free(visited);
}/* BacktrackingSolver */

/* Invio un robot per ogni vicino alla cella corrente, ognuno dei quali ha l'o-
biettivo di raggiungere la prossima cella della linea guida.
IOP solution Lista contenente la soluzione.
IP curr Cella corrente.
IP Cella della linea guida successiva da raggiungere.
IP m Labirinto.
*/
 void sendOutRobots(List* solution, const Cell* curr, Cell* next, const Maze*
m){
    List ns, robotPath;
    List* robotPaths;
    int i;
    initList(&ns);
    findUnblockedNeighbours(curr, &ns);
    /* crea un robot per ogni vicino */
    robotPaths = malloc(ns.length * sizeof(List));
    assert(robotPaths != NULL);
    for (i = 0; i < ns.length; i++){
        initList(&robotPath);
        listAdd(&robotPath, ns.list[i]);
        robotPaths[i] = robotPath;
    }/* for */

    /* ogni robot deve raggiungere la prossima cella della linea guida */
    for (i = 0; i < ns.length; i++)
        BacktrackingSolver(&robotPaths[i], next, m);
    /* aggiunge il cammino piu' corto alla soluzione */
    if(ns.length > 1)
        concat(solution, listMin(robotPaths, ns.length));
    else
```

```
        concat(solution, robotPaths);
    for (i = 0; i < ns.length; i++)
        listFree(&robotPaths[i]);
    listFree(&ns);
}/* sendOutRobots */

/* Chain Algorithm: traccia una linea guida (piu' dritta possibile) da seguire,
che va dall'inizio alla fine. Quando si incontra un ostacolo
   invia dei robot che lo aggirano per arrivare alla prossima cella della linea
guida.
IP m Labirinto.
IOP solution Soluzione.
*/
void Chain(const Maze* m, List* solution){
    List guidingLine;
    int curr = 0;
    initList(&guidingLine);
    initList(solution);
    drawGuidingLine(m, &guidingLine);
    printf("GUIDING LINE: ");
    printList(&guidingLine);
    listAdd(solution, guidingLine.list[0]);
    while(curr < guidingLine.length - 1){
        if(!tryDirectMove(m,   solution,      guidingLine.list[curr],   guiding-
Line.list[curr + 1]))
            sendOutRobots(solution,       guidingLine.list[curr],       guiding-
Line.list[curr + 1], m);
        curr++;
    }/* while */
    listFree(&guidingLine);
    pruneSolution(solution);
}/* Chain */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h;
    char** a;
    Maze m;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    Chain(&m, &solution);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    listFree(&solution);
    return 0;
}/* main */
```

Il file CollisionSolver.c contiene l'algoritmo di risoluzione Collision Solver, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/* Ritorna se due cammini distinti hanno un'intersezione.
IP l1 Lista contenente il primo cammino.
IP l2 Lista contenente il secondo cammino.
IP intersections Lista contenete le intersezioni tra due o piu' percorsi.
OR Esito della verifica.
*/
bool foundIntersection(const List* l1, const List* l2, const List* intersec-
tions){
    int j, i1, i2;
    for(j = 0; j < intersections->length; j++){
        i1 = searchCell(intersections->list[j], l1, 0);
        i2 = searchCell(intersections->list[j], l2, 0);
        if(i1 == i2 && i1 != -1)
            return true;
    }/* for */
    return false;
}/* foundIntersection */

/* Ritorna se due cammini sono uguali.
IP l1 Lista contenente il primo cammino.
IP l2 Lista contenente il secondo cammino.
OR Esito della verifica.
*/
bool isEqualPath(const List* p1, const List* p2){
    int i;
    if(p1->length == p2->length){
        for(i = 0; i < p1->length; i++){
            if(!isEqualCell(p1->list[listIndex(p1,  i)],  p2->list[listIndex(p2,
i)]))
                return false;
        }/* for */
        return true;
    }/* if */
    return false;
}/* isEqualPath */

/* Ritorna se ci sono differenze tra due insiemi di percorsi.
IP paths1 Primo array di liste.
IP paths2 Secondo array di liste.
OR Esito del confronto.
```

```
*/
bool difference(const ListArray* paths1, const ListArray* paths2){
    int i, j;
    bool flag;
    if(paths1->size == paths2->size){
        for(i = 0; i < paths1->size; i++){
            flag = false;
            for(j = 0; j < paths2->size && !flag; j++){
                if(isEqualPath(&paths1->array[i], &paths2->array[j])){
                    flag = true;
                }
            }/* for */

            if(!flag)
                return true;

        }/* for */
        return false;
    }
    return true;
}/* difference */

/* Sistema le collisioni tra due percorsi bloccando la cella in cui si incontrano se questa NON e' una giunzione, la quale viene eliminata
   dai percorsi.
IOP m Labirinto.
IOP paths Array di liste contenete i percorsi.
IOP intersections Puntatore alla lista contenete le intersezioni tra due o piu' percorsi.
IOP prev Array di liste che contiene le celle dei percorsi da cui si e' giunti alla cella di intersezione nella posizione
   corrispondente.
*/
void fixCollisions(Maze* m, ListArray* paths, List* intersections, ListArray* prev){
    int i, j, index;
    Cell* ceI;
    Cell* prevI;
    Cell* ceJ;
    Cell*prevJ;
    List p;
    for(i = 0; i < paths->size - 1; i++){
        ceI = (paths->array + i)->list[listIndex(paths->array + i, (paths->array + i)->length - 1)];
        prevI = (paths->array + i)->list[listIndex(paths->array + i, (paths->array + i)->length - 2)];
        if(isDeadEnd(ceI, m))
            blockCell(m, ceI);
        else if(!isDeadEnd(ceI, m) && !isEqualCell(ceI, m->end)){
            for(j = i+1; j < paths->size; j++){
                ceJ = (paths->array + j)->list[listIndex(paths->array + j, (paths->array + j)->length - 1)];
```

```
                prevJ = (paths->array + j)->list[listIndex(paths->array + j,
(paths->array + j)->length - 2)];
                if(!isDeadEnd(ceJ, m) && !isEqualCell(ceJ, m->end)){
                    if(isEqualCell(ceI, ceJ)){
                        if(isJunction(ceI, m)){
                            if(!isIn(intersections, ceI) && !isEqualCell(prevI,
prevJ)){
                                listAdd(intersections, ceI);
                                initList(&p);
                                listAdd(&p, prevI);
                                listAdd(&p, prevJ);
                                listArrayAdd(prev, p);
                                listFree(&p);
                            }/* if */
                            if(isIn(intersections, ceI)){
                                index = searchCell(ceI, intersections, 0);
                                if(!isIn((prev->array + index), prevI))
                                    listAdd((prev->array + index), prevI);
                                if(!isIn((prev->array + index), prevJ))
                                    listAdd((prev->array + index), prevJ);
                            }/* if */
                        }/* if */
                        else{
                            if(!isEqualCell(prevI, prevJ)){
                                blockCell(m, ceI);
                                listRemoveLast(paths->array + i);
                                listRemoveLast(paths->array + j);
                            }/* if */
                        }/* else */

                    }/* if */
                }/* if */
            }/* for */
        }/* else if */
    }/* for */
}/* fixCollisions */

/* Fa il primo passo, inondando le celle vicine alla cella di inizio, se possi-
bile.
IOP m Labirinto.
IOP paths Array di liste contenete i percorsi.
IOP intersections Lista contenete le intersezioni tra due o piu' percorsi.
IOP prev Array di liste che contiene le celle dei percorsi da cui si e' giunti
alla cella di intersezione nella posizione
   corrispondente.
OR Ritorna true se almeno una nuova cella e' stata inondata, altrimenti false.
*/
bool oneTimeStepFirst(Maze* m, ListArray* paths, List* intersections, ListArray*
prev){
    int j;
    Cell* ce;
    List ns;
```

```
    ListArray tempPaths;
    bool stepMade = false;
    initListArray(&tempPaths);
        /* ultima cella */
        ce = (paths->array)->list[listIndex(paths->array, (paths->array)->length
- 1)];

            findUnblockedNeighbours(ce, &ns);
            if(ns.length > 1 && (paths->array)->length > 1)
                listRemove(&ns, searchCell((paths->array)->list[listIndex(paths-
>array, (paths->array)->length - 2)], &ns, ns.first));

            if(ns.length != 0){
                stepMade = true;
                for(j = 0; j < ns.length; j++){
                    listArrayAdd(&tempPaths, *(paths->array));
                    listAdd(tempPaths.array + tempPaths.size - 1, ns.list[j]);
                }/* for */
            }/* else */
    listFree(&ns);
    if (!stepMade)
        return false;

    /* gestisce le collisioni */
    fixCollisions(m, &tempPaths, intersections, prev);

    listArrayFree(paths);
    initListArray(paths);
    copyListArray(paths, &tempPaths);
    listArrayFree(&tempPaths);
    return true;
}/* oneTimeStepFirst */

/* Se possibile inonda le celle non ancora visitate vicine all'ultima cella di
ogni percorso.
IOP m Puntatore al labirinto.
IOP paths Array di liste contenete i percorsi.
IOP intersections Lista contenete le intersezioni tra due o piu' percorsi.
IOP prev Array di liste che contiene le celle dei percorsi da cui si e' giunti
alla cella di intersezione nella posizione
   corrispondente.
OR Ritorna true se almeno una nuova cella e' stata inondata, altrimenti false.
*/
bool oneTimeStep(Maze* m, ListArray* paths, List* intersections, ListArray*
prev){
    int i, j, index;
    Cell* ce;
    List ns;
    ListArray tempPaths;
    bool stepMade = false;
    initListArray(&tempPaths);
```

```
    for(i = 0; i < paths->size; i++){
        /* ultima cella */
        ce = (paths->array + i)->list[listIndex(paths->array + i, (paths->array
+ i)->length - 1)];

        if(isEqualCell(ce, m->end))
            listArrayAdd(&tempPaths, *(paths->array + i));
        else if((paths->array + i)->length != 0 && !isDeadEnd(ce, m) &&
!isBlocked(ce)){
            findUnblockedNeighbours(ce, &ns);
            if(ns.length > 1 && (paths->array[i]).length > 1){
                if(!isIn(intersections, ce))
                    listRemove(&ns,                                searchCell(paths-
>array[i].list[listIndex(&paths->array[i],  paths->array[i].length  -  2)],  &ns,
ns.first));
                else{
                    index = searchCell(ce, intersections, 0);
                    for(j = 0; j < (prev->array[index]).length; j++)
                        listRemove(&ns, searchCell((prev->array[index]).list[j],
&ns, ns.first));
                }/* else */
            }/* if */
            if(ns.length != 0){
                stepMade = true;
                for(j = 0; j < ns.length; j++){
                    listArrayAdd(&tempPaths, paths->array[i]);
                    listAdd(tempPaths.array + tempPaths.size - 1, ns.list[j]);
                }/* for */
            }/* if*/
            listFree(&ns);
        }/*else if */
    }/* for */
    if (!stepMade)
        return false;

    /* gestisce le collisioni */
    fixCollisions(m, &tempPaths, intersections, prev);
    listArrayFree(paths);
    initListArray(paths);
    copyListArray(paths, &tempPaths);
    listArrayFree(&tempPaths);
    return true;
}/* oneTimeStep */

/* Inonda il labirinto step-by-step finche' e' possibile, cioe' finche' tutti i
percorsi hannno raggiunto la cella finale oppure
   un vicolo cieco.
IOP m Labirinto.
IOP paths Array di liste contenete i percorsi.
*/
void floodMaze(Maze* m, ListArray* paths){
    List path, intersections;
```

```
    ListArray prev;
    bool stepMade;
    initListArray(&prev);
    initList(&path);
    initList(&intersections);
    listAdd(&path, m->start);
    /*printList(&path);*/
    listArrayAdd(paths, path);
    stepMade = oneTimeStepFirst(m, paths, &intersections, &prev);
    while(stepMade){
        stepMade = oneTimeStep(m, paths, &intersections, &prev);
    }/* while */
    listFree(&intersections);
    listArrayFree(&prev);
}/* floodMaze */

/* Sistema la lista di soluzioni eliminando i percorsi che terminano con un vi-
colo cieco, che ovviamente non sono soluzioni.
IOP m Labirinto.
IOP solutions Array di liste contenete le soluzioni.
*/
void fixSolutions(Maze* m, ListArray* solutions){
    int i;
    Cell* ce;
    i = 0;
    while(i < solutions->size){
        ce = (solutions->array + i)->list[((solutions->array + i)->length) - 1];
        if(!isEqualCell(ce, m->end))
            listArrayRemove(solutions, i);
        else
            i++;
    }/* while */
}/* fixSolutions */

/* Collision Solver: inonda il labirinto piu' volte finche' non vi sono piu'
differenze di percorsi tra un'inondazione e la precedente.
   Vengono trovate piu' soluzioni tra cui tutte le soluzioni piu' brevi.
IOP m Labirinto.
IOP solutions Array di liste contenete le soluzioni.
*/
void CollisionSolver(Maze* m, ListArray* solutions){
    bool diff;
    ListArray tempPaths;
    initListArray(&tempPaths);
    /* Inonda il labirinto 2 volte e compara i risultati */
    floodMaze(m, solutions);

    floodMaze(m, &tempPaths);

    /* controllo se ci sono differenze tra le due liste di percorsi */
    diff = difference(solutions, &tempPaths);
```

```
    /* inonda il labirinto finche' non ci sono differenze */
    while(diff){

        listArrayFree(solutions);
        initListArray(solutions);
        copyListArray(solutions, &tempPaths);
        listArrayFree(&tempPaths);
        initListArray(&tempPaths);
        floodMaze(m, &tempPaths);

        diff = difference(solutions, &tempPaths);
    }/* while */
    listArrayFree(&tempPaths);
    fixSolutions(m, solutions);
}/* CollisionSolver */

int main(int argc, char **argv){
    FILE* inF;
    ListArray solutions;
    int w, h, i;
    char** a;
    Maze m;
    srand(time(NULL)); /* inizializzo un random seed */
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    initListArray(&solutions);
    CollisionSolver(&m, &solutions);
    printf("SOLUZIONI: ");
    printListArray(&solutions);
    for(i = 0; i < solutions.size; i++){
        printf("\n");
        printf("\n");
        printf("%d\n", i+1);
        printSolution(a, &m, &solutions.array[i]);
        printf("\n");
        clean(a, &m);
    }/* for */
    listArrayFree(&solutions);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file ShortestPath.c contiene l'algoritmo di risoluzione Shortest Path Finder ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

/* Shortest Path: equivalente ad una BFS, che si interrompe quando raggiunge la
fine.
   Trova una delle soluzioni piu' brevi.
IP m Lista.
IOP solution Lista contenente la soluzione.
*/
void ShortestPath(const Maze* m, List* solution){
    int i, j;
    bool** visited;
    Cell*** prev;
    Cell* curr;
    List q, ns;
    srand(time(NULL)); /* inizializzo un random seed */
    /* alloco ed inizializzo l'array che tiene traccia se una cella e' gia' sta-
ta visitata */
    visited = malloc(m->h * sizeof(bool*));
    for(i = 0; i < m->h; i++){
        visited[i] = malloc(m->w * sizeof(bool));
        for(j = 0; j < m->w; j++)
            visited[i][j] = false;
   }/* for */
    /* alloco ed inizializzo l'array che memorizza la cella visitata precedente-
mente nella posizione corrispondente a quella appena visitata */
    prev = malloc(m->h * sizeof(Cell**));
    for(i = 0; i < m->h; i++)
        prev[i] = malloc(m->w * sizeof(Cell*));
    /* inizializzo una lista che si comporta come una coda */
    initList(&q);
    /* aggiungo la cella di inizio */
    listAdd(&q, m->start);
    /* itero finche' non raggiungo la fine */
    while(!isEqualCell(q.list[q.first], m->end)){
        /*cella corrente, cella da piu' tempo nella lista */
        curr = q.list[q.first];
        /* contrassegno la cella come visitata */
        visited[curr->row][curr->col] = true;
        /* trovo i vicini */
        findUnvisitedNeighbours(curr, visited, &ns);
        for(i = 0; i < ns.length; i++){
            /* salvo la cella corrente come precedente alle celle vicine */
            prev[ns.list[i]->row][ns.list[i]->col] = curr;
```

```
            /* aggiungo ogni vicino alla lista */
            listAdd(&q, ns.list[i]);
        }/* for */
        /* rimuovo la cella corrente dalla lista */
        listRemoveFirst(&q);
        listFree(&ns);
    }/* while */
    listFree(&q);
    /* inizializzo la soluzione */
    initList(solution);
    /* aggiungo la cella finale */
    listAddFirst(solution, m->end);
    curr = m->end;
    /* itero finche' non aggiungo la xella iniziale */
    while(!isEqualCell(curr, m->start)){
        /* aggiungo in prima posizione la cella che era stata visitata prima
della cella corrente */
        listAddFirst(solution, prev[curr->row][curr->col]);
        /* la cella precedente diventa la corrente */
        curr = prev[curr->row][curr->col];
    }/* while */
    for(i = 0; i < m->h; i++)
        free(visited[i]);
    free(visited);
    for(i = 0; i < m->h; i++)
        free(prev[i]);
    free(prev);
}/* ShortestPath */

int main(int argc, char **argv){
    FILE* inF;
    Maze m;
    int w, h;
    List solution;
    char** a;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    ShortestPath(&m, &solution);
    printf("SOLUZIONE: ");
    printList(&solution);
    printSolution(a, &m, &solution);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file FloodFill.c contiene l'algoritmo di risoluzione Flood Fill, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include "Maze.h"

#define UNINITIALISED -1

/* Restituisce la cella vicina alla corrente a minore distanza dalla cella fina-
le.
IP ce Cella corrente.
IP dist Array 2D contenente le distanze di ogni cella dalla cella finale.
OR Cella vicina alla corrente a minore distanza dalla cella finale.
*/
Cell* minDistNeigh(const Cell* ce, int** dist){
      List ns;
      Cell* min;
      int i;
      findUnblockedNeighbours(ce, &ns);
      min = ns.list[0];
      for(i = 1; i < ns.length; i++){
            if(dist[(ns.list[i])->row][(ns.list[i])->col]  <  dist[min->row][min-
>col])
                  min = ns.list[i];
      }/* for */
      return min;
}/* minDistNeigh */

/* Flood Fill Algorithm: calcola la distanza di ogni cella dalla cella finale e
poi costruisce la soluzione piu' breve passo a passo
   a partire dalla cella di partenza.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
*/
void floodFill(const Maze* m, List* solution){
      int** dist;
      int i, j;
      List ns, l;
      srand(time(NULL)); /* inizializzo un random seed */
      /* alloco ed inizializzo l'array contenente le distanze di ogni cella dal-
la cella finale */
      dist = malloc(m->h * sizeof(int*));
      for(i = 0; i < m->h; i++){
            dist[i] = malloc(m->w * sizeof(int));
            for(j = 0; j < m->w; j++)
                  dist[i][j] = UNINITIALISED;
      }/* for */
```

```
      /* inizializzo la lista */
      initList(&l);
      /* aggiungo la cella finale nella lista */
      listAdd(&l, m->end);
      /* imposto la sua distanza a 0 */
      dist[(l.list[l.first])->row][(l.list[l.first])->col] = 0;
      /* itero finche' la lista non e' vuota */
      while(!isEmptyList(&l)){
            /* trovo i vicini alla cella corrente */
            findUnblockedNeighbours(l.list[l.first], &ns);
            /* per ogni vicino calcolo la sua distanza dalla cella finale e lo
inserisco nella lista */
            for(i = 0; i < ns.length; i++){
                  if(dist[(ns.list[i])->row][(ns.list[i])->col]                ==
UNINITIALISED){
                        listAdd(&l, ns.list[i]);
                        dist[(ns.list[i])->row][(ns.list[i])->col]              =
dist[(l.list[l.first])->row][(l.list[l.first])->col] + 1;
                  }/* if */
            }/* for */
          listFree(&ns);
          /* rimuovo la cella corrente, la quale e' presente da piu' tempo nella
lista */
            listRemoveFirst(&l);
      }/* while */
      listFree(&l);
      /* inizializzo la lista contenente la soluzione */
      initList(solution);
      /* ci aggiungo la cella di inizio */
      listAdd(solution, m->start);
      /* itero aggiungendo ogni volta  la cella a minore distanza dalla cella
finale,
         parto dalla cella iniziale fino ad arrivare alla cella finale */
      while(!isEqualCell(solution->list[solution->length - 1], m->end))
            listAdd(solution, minDistNeigh(solution->list[solution->length - 1],
dist));
}/* floodFill */


int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h;
    char** a;
    Maze m;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    floodFill(&m, &solution);
    printf("SOLUZIONE: ");
```

```
    printList(&solution);
    printSolution(a, &m, &solution);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```

Il file aStar.c contiene l'algoritmo di risoluzione A*, le funzioni ausiliarie ed il main.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>
#include <limits.h>

#include "Maze.h"

/* Calcola la distanza di Manhattan tra due celle.
IP ce1 Prima cella.
IP ce2 Seconda cella.
OR Distanza di Manhattan.
*/
int ManhattanDist(const Cell* ce1, const Cell* ce2){
      return fabs(ce2->row - ce1->row) + fabs(ce2->col - ce1->col);
}/* ManhattanDist */

/* Inserisce nella posizione corretta un puntatore ad una cella in una lista,
che si comporta come una coda di priorità
    implementata attraverso un min-heap, sfruttando la corrispondenza biunivoca
tra heap ed array.
    - All'indice 0 si trova la radice dello heap.
    - I figli dell'elemento all'indice i si trovano agli indici 2i+1 e 2i+2.
    - Il padre dell'elemento all'indice i si trova all'indice (i-1)/2.
IOP l Lista in cui aggiungere l'elemento.
IP ce Elemento da aggiungere alla lista.
IP fScore Array 2D contenente il valore della funzione di costo di ogni cella.
*/
void listInsert(List* l, Cell* ce, int** fScore){
      int i;
      listAdd(l, ce);
      i = l->length - 1;
      while(i > 0 && (fScore[(l->list[(i-1)/2])->row][(l->list[(i-1)/2])->col] >
fScore[(l->list[i])->row][(l->list[i])->col])){
            swap(l->list + i, l->list + (i-1)/2);
            i = (i - 1) / 2;
      }/* while */
}/* listInsert */

/* Rimuove l'elemento con priorita' massima(minima funzione di costo) dalla li-
sta, che si comporta come una coda di priorità
   implementata attraverso un min-heap, sfruttando la corrispondenza biunivoca
tra heap ed array.
   Mantiene invariata la proprietà dello heap.
   - All'indice 0 si trova la radice dello heap.
   - I figli dell'elemento all'indice i si trovano agli indici 2i+1 e 2i+2.
   - Il padre dell'elemento all'indice i si trova all' indice (i-1)/2.
IOP l Lista in cui aggiungere l'elemento.
```

```
IP fScore Array 2D contenente il valore della funzione di costo di ogni cella.
OR Cella con massima priorita'.
*/
Cell* listRemoveMin(List* l, int** fScore){
      Cell* minCe;
      int i, j;
      minCe = l->list[0];
      l->list[0] = l->list[l->length - 1];
      l->length--;
      i = 0;
      if(2 * i + 1 <= l->length - 1){
            if(fScore[(l->list[2*i+1])->row][(l->list[2*i+1])->col] < fScore[(l-
>list[2*i+2])->row][(l->list[2*i+2])->col])
                  j = 2*i+1;
            else
                  j = 2*i+2;
      }/* if */
      while(2 * i + 1 <= l->length - 1 && fScore[(l->list[i])->row][(l-
>list[i])->col] > fScore[(l->list[j])->row][(l->list[j])->col]){
            swap(l->list + i, l->list + j);
            i = j;
            if(2 * i + 1 <= l->length - 1){
                  if(fScore[(l->list[2*i+1])->row][(l->list[2*i+1])->col]      <
fScore[(l->list[2*i+2])->row][(l->list[2*i+2])->col])
                        j = 2*i+1;
                  else
                        j = 2*i+2;
            }/* if */
      }/* while */
      return minCe;
}/* listRemoveMin */

/* Algoritmo A*.
   Sia c una cella del labirinto, introduco le seguenti funzioni:
   - g(c): distanza tra la cella c e la cella di partenza.
   - h(c): funzione euristica, distanza di Manhattan tra la cella c e la cella
da raggiungere.
   - f(c) = g(c) + h(c): funzione di costo.
IP m Labirinto.
IOP solution Lista contenente la soluzione.
*/
void aStar(const Maze* m, List* solution){
      List p, ns;
      Cell* curr;
      Cell*** prev;
      int** gScore;
      int** fScore;
      int i, j, tempGscore, tempFscore;
      srand(time(NULL)); /* Inizializzo un random seed */
      initList(&p);/* Inizializzo la lista che si comportera' come coda di prio-
rita' */
      initList(solution);/* Inizializzo la lista che conterra' la soluzione */
```

```
      /* alloco l'array 2D che conterra' in ogni posizione un puntatore alla
cella precedentemente visitata, quindi il puntatore alla cella
         dalla quale si e' arrivati a quella corrispondente */
      prev = malloc(m->h * sizeof(Cell**));
    for(i = 0; i < m->h; i++){
        prev[i] = malloc(m->w * sizeof(Cell*));
    }/* for */
    /* alloco ed inizializzo l'array 2D che conterra' in ogni posizione il valo-
re di g(c)
      (distanza tra la cella corrispondente e la cella di partenza) */
      gScore = malloc(m->h * sizeof(int*));
      for(i = 0; i < m->h; i++){
            gScore[i] = malloc(m->w * sizeof(int));
            for(j = 0; j < m->w; j++)
                  gScore[i][j] = INT_MAX;
      }/* for */
    /* alloco ed inizializzo l'array 2D che conterra' in ogni posizione la fun-
zione di costo f(c) */
      fScore = malloc(m->h * sizeof(int*));
      for(i = 0; i < m->h; i++){
            fScore[i] = malloc(m->w * sizeof(int));
            for(j = 0; j < m->w; j++)
                  fScore[i][j] = INT_MAX;
      }/* for */
      /*g(start)*/
      gScore[(m->start)->row][(m->start)->col] = 0;
      /*f(start)*/
      fScore[(m->start)->row][(m->start)->col]   =   ManhattanDist(m->end,   m-
>start);
      /* inserisco la cella di partenza nella coda di priorita' */
      listInsert(&p, m->start, fScore);
      /* inizializzo la cella corrente con la cella di partenza */
      curr = m->start;
      /* trovo i vicini della cella corrente */
      findUnblockedNeighbours(curr, &ns);
      /* per ogni vicino alla cella corrente calcolo g(c) e f(c) */
      for(i = 0; i < ns.length; i++){
            /* calcolo un valore temporaneo di g(c)*/
            tempGscore = gScore[curr->row][curr->col] + 1;
            /* calcolo un valore temporaneo di f(c) */
            tempFscore = tempGscore + ManhattanDist(m->end, ns.list[i]);
            /* aggiorno g(c) e f(c) se il valore calcolato e' minore di quello
precedente */
            if(tempFscore < fScore[(ns.list[i])->row][(ns.list[i])->col]){
                  gScore[(ns.list[i])->row][(ns.list[i])->col] = tempGscore;
                  fScore[(ns.list[i])->row][(ns.list[i])->col] = tempFscore;
                  /* inserisco la cella vicina nella coda di priorita' */
                  listInsert(&p, ns.list[i], fScore);
                  /* assegno la cella precedente nella posizione corrispondente
alla cella vicina in questione*/
                  prev[(ns.list[i])->row][(ns.list[i])->col] = curr;
```

```
            }/* if */
      }/* for */
      listFree(&ns);
      /* itero finche' raggiungo la fine */
      while(!isEqualCell(curr, m->end)){
            /* rimuovo l'elemnto con priorita' massima */
            curr = listRemoveMin(&p, fScore);
            /* trovo i vicini della cella corrente */
            findUnblockedNeighbours(curr, &ns);
            /* per ogni vicino alla cella corrente calcolo g(c) e f(c) */
            for(i = 0; i < ns.length; i++){
                  /* calcolo un valore temporaneo di g(c)*/
                  tempGscore = gScore[curr->row][curr->col] + 1;
                  /* calcolo un valore temporaneo di f(c)*/
                  tempFscore = tempGscore + ManhattanDist(m->end, ns.list[i]);
                  /* aggiorno g(c) e f(c) se il valore calcolato e' minore di
quello precedente */
                  if(tempFscore < fScore[(ns.list[i])->row][(ns.list[i])->col]){
                        gScore[(ns.list[i])->row][(ns.list[i])->col] = tempGsco-
re;
                        fScore[(ns.list[i])->row][(ns.list[i])->col] = tempFsco-
re;
                        /* inserisco il puntatore alla cella vicina nella coda
di priorita' */
                        listInsert(&p, ns.list[i], fScore);
                        /* assegno la cella precedente nella posizione corri-
spondente alla cella vicina in questione*/
                        prev[(ns.list[i])->row][(ns.list[i])->col] = curr;
                  }/* if */
            }/* for */
            listFree(&ns);
      }/* while */
      /* inserisco le celle appartenti alla soluzione in $solution */
      while(!isEqualCell(curr, m->start)){
            listAddFirst(solution, curr);
            curr = prev[curr->row][curr->col];
      }/* while */
}/* aStar */

int main(int argc, char **argv){
    FILE* inF;
    List solution;
    int w, h;
    char** a;
    Maze m;
    inF = fopen(argv[1], "r");
    if (inF == NULL)
        return -1;
    a = saveMaze(inF, &w, &h);
    createMazeGrid(a, &m, w, h);
    aStar(&m, &solution);
    printf("SOLUZIONE: ");
```

```
    printList(&solution);
    printSolution(a, &m, &solution);
    listFree(&solution);
    mazeFree(&m);
    return 0;
}/* main */
```