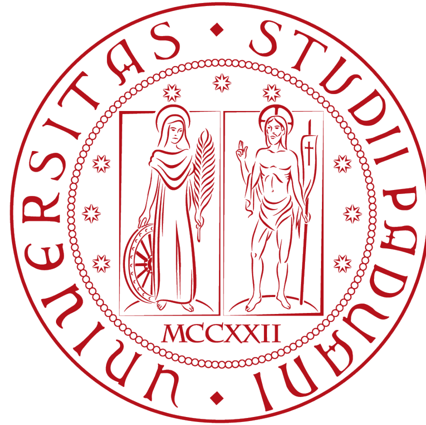


Università degli Studi di Padova

Dipartimento di ingegneria dell'informazione

Corso di laurea in Ingegneria Informatica

Algoritmi per labirinti

Maze Algorithms

Laureando:
Christian Marchiori

Relatore:
Carlo Fantozzi

Anno accademico 21-22

Data di laurea: 22/07/2022

Indice

1 Introduzione ai labirinti **3**
1.1 Definizioni 4

2 Categorizzazione dei labirinti **5**
2.1 Dimensione 5
2.2 Iper-dimensione 6
2.3 Topologia 6
2.4 Tassellazione 7
2.5 Percorso 8
2.6 Texture 8

3 Introduzione alla teoria dei grafi **10**

4 Implementazione della griglia **13**
4.1 Implementazione della classe *Cell* 13
4.2 Implementazione della classe *Grid* 17

5 Visualizzazione dei labirinti **21**
5.1 Visualizzare un labirinto nel terminale 21
5.2 Visualizzare un labirinto mediante libreria grafica 24

6 Algoritmo Binary Tree **29**
6.1 Introduzione 29
6.2 Descrizione dell'algoritmo 29
6.3 Implementazione 31

7 Algoritmo Aldous-Broder **33**
7.1 Introduzione 33
7.2 Descrizione dell'algoritmo 33
7.3 Implementazione 35

8 Algoritmo di Wilson **37**
8.1 Introduzione 37
8.2 Descrizione dell'algoritmo 38
8.3 Implementazione 40

9 Algoritmo Recursive Backtracker **44**
9.1 Introduzione 44
9.2 Descrizione dell'algoritmo 45
9.3 Implementazione 47

10 Algoritmo di Kruskal **49**
10.1 Introduzione 49
10.2 Descrizione dell'algoritmo 50
10.3 Implementazione 53

11 Confronto tra gli algoritmi di generazione **57**

12 Conclusioni **59**

Riferimenti bibliografici **60**

1 Introduzione ai labirinti

La parola labirinto è molto suggestiva, perché rappresenta un punto di contatto fra storia e mito. L'etimologia del termine deriva dal greco labýrinthos. Probabilmente questa era la parola egea che si riferiva al palazzo di Cnosso, a Creta, dove ne venne fatto costruire uno dal re Minosse per rinchiudervi il mostruoso Minotauro.

La prima idea che ricorre alla mente quando si parla di labirinto è quella di un gioco, dato che i labirinti sono strumenti di divertimento ed intrattenimento molto popolari.

La costruzione di un labirinto è caratterizzata da una rete di passaggi interconnessi complicati e tortuosi per rendere difficile l'orientamento e quindi l'uscita; lo scopo è di stimolare la pazienza, l'abilità manuale, la concentrazione e la perseveranza di raggiungere un traguardo, sia nell'adulto che nel bambino, ognuno declinato alla propria sfera di conoscenza e capacità. Strutturalmente un labirinto è formato da corridoi e pareti.

Attualmente sono considerati interessanti dal punto di vista matematico-logico; il labirinto non è solo usato come puzzle, ma è anche impiegato per altre funzioni.

- Nel campo dei giochi per computer, può essere utilizzato come struttura di base per un livello di gioco.
- Nel campo della robotica, può essere utilizzato come piattaforma per dimostrare la capacità di apprendimento di un robot.
- Nel campo dell'architettura, il suo motivo può essere utilizzato per decorare un edificio.

Poiché un labirinto può essere utilizzato per diverse finalità, un utente generico ha la possibilità di creare un labirinto che assuma delle proprietà specifiche in base allo scopo. Esistono molteplici tipologie di labirinti con caratteristiche diverse e contestualmente esistono diversi generatori di labirinti in base alle proprietà desiderate.[5]

In questo elaborato, inizialmente verranno illustrati: alcune definizioni di termini associati ai labirinti utili a comprendere meglio la tesi, una categorizzazione dei labirinti e un'introduzione ai concetti rilevanti della teoria dei grafi, al fine di rendere chiara e osservabile la struttura degli algoritmi.
Successivamente verranno introdotti alcuni algoritmi di generazione di labirinti basati su grafi includendo un'implementazione in linguaggio C++.

1.1 Definizioni

Un **labirinto** è una griglia rettangolare di celle in cui ogni cella è collegata alla cella iniziale da almeno un percorso. In termini matematici, un labirinto è un grafo non orientato, planare e possibilmente ciclico la cui chiusura transitiva è un grafo completamente connesso.

Una **cella** è un punto in un labirinto, o più tecnicamente le unità di passaggio che sono collegate in una griglia per formare il labirinto. Dal punto di vista dei grafi ogni cella corrisponde ad un vertice. Una cella può avere dei confini di passaggio, che corrispondono agli archi di un grafo e conducono da essa ad altre celle.

Una **giunzione** è qualsiasi cella con più di due celle adiacenti collegate, ad eccezione della cella iniziale, che è una giunzione anche se sono presenti solo due celle adiacenti collegate.

Un **incrocio** è un punto di un labirinto nel quale si incontrano quattro passaggi o si intersecano due percorsi . Può anche indicare un punto generale in cui si incontrano quattro o più passaggi.

Un **vicolo cieco** è qualsiasi cella con esattamente una cella vicina collegata, ma può anche riferirsi all'intero insieme che collega tale cella alla giunzione più vicina, nel qual caso la dimensione del vicolo cieco indica il numero di celle dall'estremità alla più vicina giunzione, esclusa la giunzione.

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 cerchio.

Un **passo di sovrapposizione** è un qualsiasi spostamento effettuato su una cella precedentemente già visitata. I passi di sovrapposizione possono essere utilizzati, durante la risoluzione di un labirinto, per misurarne la difficoltà. Se durante la risoluzione di un labirinto si effettuano numerosi passi di sovrapposizione significa che il risolutore, non riuscendo a trovare l'uscita, ripercorre spesso dei passaggi già attraversati.

2 Categorizzazione dei labirinti

I labirinti in generale possono essere categorizzati secondo sette diversi criteri [9]:

- dimensione,
- iperdimensione,
- topologia,
- tassellazione,
- percorso,
- texture,
- focus.

L'appartenenza di un labirinto, ad una o più categorie, solitamente influenza le caratteristiche del labirinto come difficoltà e tempo di risoluzione, relative alla fase di risoluzione.

2.1 Dimensione

La dimensione di un labirinto equivale al numero di dimensioni nello spazio che il labirinto copre, dove la dimensione di un oggetto nello spazio corrisponde al numero di coordinate necessarie per individuare un punto sull'oggetto (nel caso di un labirinto: una cella). I labirinti più comuni sono quelli bi-dimensionali; gli algoritmi di generazione di labirinti analizzati in seguito saranno di questo tipo.

Altri labirinti ampiamente utilizzati sono quelli a tre dimensioni. I labirinti 3D sono sostanzialmente molteplici labirinti 2D sovrapposti tra loro con una modalità di passaggio che permette a chi li risolve di attraversare i vari livelli. Questi labirinti sono spesso utilizzati dai progettisti di spazi virtuali (ad esempio videogiochi) per creare ambienti complessi randomizzati. Per rappresentare un labirinto 3D si utilizza un array contenente vari livelli 2D.

È possibile che un labirinto copra uno spazio dimensionale superiore rispetto allo spazio 3D, ad esempio, un labirinto a 4 dimensioni. Tale labirinto può essere difficile da immaginare ma è possibile rappresentarlo come un labirinto 3D, con la quarta dimensione accessibile attraverso dei portali.

Un altro tipo di labirinto con una dimensione particolare è il labirinto a trama (Figura 2.1). I labirinti a trama consentono la sovrapposizione o la stratificazione di percorsi all'interno del labirinto. Ovviamente durante la risoluzione di uno di questi labirinti non è possibile spostarsi tra gli strati quando in una cella sono presenti due passaggi sovrapposti. È possibile considerare questi labirinti con una dimensione intermedia tra la seconda e la terza.

2.2 Iper-dimensione

Questa categoria si basa sulla dimensione dell'oggetto che attraversa il labirinto durante la sua risoluzione. Supponendo che l'oggetto che deve attraversare il labirinto abbia una dimensione spaziale N, generalmente la soluzione è rappresentata da figure di dimensione N+1. Queste figure vengono create spostando l'oggetto lungo il percorso corretto che parte dal punto di ingresso e arriva al punto di uscita. L'ambiente che forma il labirinto stesso ha, invece, dimensione N+2 o superiore.

Ad esempio, un labirinto 2D standard è formato da:

- un oggetto di 0 dimensioni, che può essere rappresentato da un punto;
- un percorso (la soluzione) formato da elementi di 1 dimensione, che può essere rappresentato da delle linee;
- la struttura del labirinto di 2+ dimensioni, che può essere rappresentata da un piano o un solido.

Perché un labirinto venga considerato **iper-labirinto** l'oggetto attraversante deve avere almeno una dimensione. Supponendo che l'oggetto abbia dimensione N (con N>0), l'iper-labirinto verrà chiamato iper-labirinto di ordine N. L'esempio più comune di un iper-labirinto è quello di una linea che deve attraversare una struttura di almeno tre dimensioni. La soluzione viene rappresentata con dei piani.

2.3 Topologia

La topologia di un labirinto descrive la geometria dello spazio in cui si trova.

- **normale**: labirinto esistente in uno spazio euclideo standard. Tale spazio deve soddisfare tutti i postulati di Euclide.

- **aereo**: labirinto che non si adatta ad uno spazio euclideo regolare e quindi, non soddisfa tutti i postulati di Euclide. Un esempio è un insieme di labirinti piani interconnessi posti su una superficie di un cubo.

2.4 Tassellazione

La tassellazione divide i labirinti in base alla geometria delle singole celle (ovvero le unità base) che compongono il labirinto.

- **Ortogonale**: le celle formano una griglia rettangolare e i collegamenti avvengono perpendicolarmente rispetto alla direzione dei muri (ovvero ai lati delle celle).
- **Omega**: il termine "omega" si riferisce alla maggior parte dei labirinti con una tassellatura non ortogonale.
  - **Delta**: labirinti composti da triangoli intrecciati, in cui ogni cella può avere fino a tre collegamenti.
  - **Sigma**: labirinti composti da esagoni ad incastro, dove ogni cella può avere fino a sei collegamenti.
  - **Upsilon**: labirinti composti da ottagoni e quadrati ad incastro, dove ogni cella può avere quattro o otto possibili collegamenti.
  - **Theta**: labirinti composti da passaggi a forma di cerchi concentrici, dove la partenza è posizionata al centro del labirinto e l'arrivo sul bordo esterno, o viceversa. Tutti i passaggi sono allineati lungo cerchi con lo stesso centro ma con raggio diverso.
- **Zeta**: labirinti formati da una griglia rettangolare, ma oltre ai collegamenti tra le celle orizzontali e verticali, sono consentiti anche quelli diagonali a 45 gradi.
- **Deformato**: labirinti senza alcun sistema di tassellatura coerente. Le pareti e i collegamenti hanno angoli casuali.
- **Frattale**: labirinti composti da labirinti più piccoli. Le celle di questi labirinti sono nidificate, infatti, esse contengono altri labirinti tassellati al loro interno; il processo di nidificazione può essere anche ricorsivo.

2.5 Percorso

Il percorso classifica i labirinti in base alle proprietà del loro sistema di passaggio. Questa categoria è probabilmente la più interessante per l'analisi degli algoritmi di generazione.

- **Perfetto**: un labirinto con percorso perfetto deve soddisfare tre vincoli:

  1. non ci devono essere anelli
  2. non ci devono essere celle isolate (le cosiddette isole), ovvero aree inaccessibili
  3. ci deve essere esattamente un percorso tra qualsiasi coppia di celle

  Questo labirinto ha esattamente una soluzione. In termini informatici, tale labirinto può essere descritto come uno "spanning tree".

- **Treccia**: un labirinto "a treccia" corrisponde ad un labirinto senza vicoli ciechi. Tale labirinto utilizza passaggi che si avvolgono e si rincorrono l'uno nell'altro (da cui il termine "treccia").

- **Unicursale**: un labirinto unicursale è un labirinto senza giunzioni. Un labirinto unicursale ha solo un unico lungo passaggio simile a un serpente che si avvolge per tutta l'estensione del labirinto. Spesso il termine "labirinto" è usato erroneamente per riferirsi a costrutti di questo tipo, invece si riferisce ad un puzzle in cui sono coinvolte delle scelte.

- **Sparso**: un labirinto che può presentare aree inaccessibili. Durante la generazione di questa tipologia non vengono necessariamente "scavati" passaggi attraverso tutte le celle. Tale labirinto è l'opposto di un labirinto a treccia.

2.6 Texture

La texture di un labirinto descrive lo stile dei passaggi in qualsiasi percorso e in qualsiasi geometria.

- **Bias**: una texture di tipo bias è caratterizzata dalla tendenza dei percorsi del labirinto ad essere diretti orizzontalmente o verticalmente. Ad esempio, un labirinto con bias orizzontale avrà lunghi passaggi orizzontali e brevi passaggi verticali. Lo stesso, ma viceversa, vale per i labirinti con bias verticale.

- **Corsa**: il fattore "corsa" di un labirinto indica approssimativamente per quanto tempo vengono percorsi i rettilinei prima che si presentino svolte forzate. Un la-

birinto con una corsa corta ha passaggi rettilinei con durata di tre o quattro celle massimo e appare molto casuale. Un labirinto con una corsa lunga ha passaggi lunghi che attraversano una buona percentuale del labirinto.

- **Elite**: il fattore "elitarismo" di un labirinto indica la lunghezza della soluzione rispetto alle dimensioni del labirinto. Un labirinto elitario ha generalmente una soluzione diretta breve, mentre un labirinto non elitario ha la soluzione che vaga per una buona parte dell'area del labirinto.
- **Simmetria**: un labirinto è simmetrico se una porzione del labirinto è riflessa in un'altra area. Questo effetto può essere ottenuto applicando la simmetria rotazionale (rispetto al centro) o la riflessione utilizzando l'asse di riflessione (asse orizzontale). Un labirinto può essere parzialmente o totalmente simmetrico.
- **Uniformità**: un labirinto può essere descritto come avente una texture uniforme, se sembra generato da un algoritmo uniforme. Un algoritmo uniforme genera tutti i possibili labirinti con la stessa probabilità.
- **A fiume**: il fattore "fiume" indica la quantità e la dimensione dei vicoli ciechi presenti nel labirinto. Un labirinto "a fiume" presenta pochi vicoli ciechi di grandi dimensioni, mentre un labirinto con fattore "fiume" basso presenta molti vicoli ciechi con dimensioni esigue.[6][9]

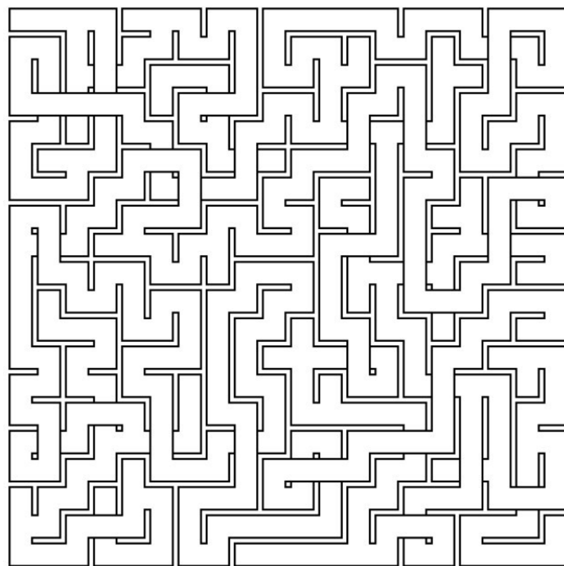


Figura 2.1: Esempio di labirinto a trama.[2]

3 Introduzione alla teoria dei grafi

Cos'è un grafo

Un grafo è un tipo di dato astratto che rappresenta relazioni esistenti tra coppie di oggetti. Un grafo, cioè, è un insieme di oggetti, chiamati vertici o nodi, e una raccolta di accoppiamenti tra loro, chiamati archi. I grafi hanno applicazioni in molti ambiti, tra i quali possiamo citare i trasporti pubblici, le reti di calcolatori, le reti di distribuzione dell'energia elettrica e le mappe di navigazione stradale. In astratto, un grafo G è semplicemente un insieme V di vertici e una raccolta E di coppie di vertici appartenenti a V, dette archi. Quindi, un grafo è un sistema per rappresentare connessioni o relazioni tra coppie di oggetti appartenenti all'insieme V. Solitamente i grafi vengono visualizzati disegnandone i vertici come cerchi, mentre gli archi sono rappresentati da segmenti che collegano coppie di vertici.

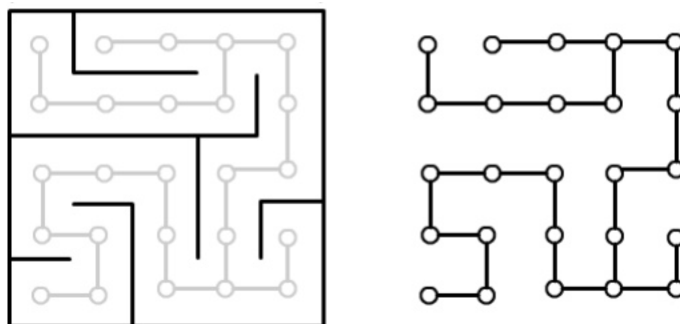


Figura 3.1:
Labirinto rappresentato da un grafo.[4]

Grafi orientati e non orientati

Gli archi di un grafo possono essere orientati oppure non orientati. Un arco (u, v) è detto orientato da u a v se la coppia (u, v) è una coppia orientata, con u che precede v. Un arco (u, v) si dice, invece, non orientato se la coppia (u, v) è una coppia non orientata. Nel caso in cui la coppia non sia orientata, (u, v) è equivalente a (v,u).

- Se tutti gli archi di un grafo sono orientati, il grafo è orientato (digrafo).
- Se tutti gli archi di un grafo sono non orientati, il grafo è non orientato. I grafi non orientati rappresentano, in generale, relazioni simmetriche tra i vertici.
- Se nessuna della condizioni precedenti è vera, il grafo è "misto".

Archi e vertici

Due vertici collegati da un arco sono i suoi vertici terminali o estremi. Se un arco è orientato, il suo primo vertice è la sua origine, mentre l'altro è la sua destinazione. Due vertici u e v si dicono adiacenti se esiste un arco i cui vertici terminali siano u e v. Un arco si dice incidente in un vertice se questo è uno dei suoi due vertici terminali.

- Gli archi uscenti da un vertice sono gli archi orientati la cui origine è quel vertice.
- Gli archi entranti in un vertice sono gli archi orientati la cui destinazione è quel vertice.

Il grado di un vertice v, indicato con deg(v), è il numero di archi incidenti in v. Il grado entrante e il grado uscente di un vertice v sono il numero di archi entranti in v e uscenti da v; si indicano, rispettivamente, con indeg(v) e outdeg(v).

Percorsi e cicli in un grafo

In un grafo, un percorso è una sequenza di vertici e archi alternati tra loro in modo che:

- la sequenza inizi con un vertice e termini con un vertice;
- ciascun arco sia incidente nel vertice che lo precede e nel vertice che lo segue nella sequenza;
- nessun arco compaia in due posizioni consecutive.

Un ciclo (loop), invece, è un percorso (avente almeno due archi) i cui vertici iniziale e finale coincidono. Sia un percorso che un ciclo possono essere:

- **Semplici**: se tutti i loro vertici sono distinti (ad eccezione del primo e l'ultimo nel caso del ciclo).
- **Orientati**: se tutti i loro archi sono orientati e vengono attraversati lungo la loro direzione.

Sottografi

Un sottografo di un grafo G è un grafo i cui insiemi di vertici e archi sono sottoinsiemi, rispettivamente, degli insiemi di vertici e archi di G.

Un sottografo ricoprente (*spanning subgraph*) è un sottografo di un grafo di partenza G contenente tutti i vertici e un sottoinsieme di archi di G

Grafi connessi e non connessi

Un grafo G è connesso se esiste un percorso (non necessariamente orientato) tra due qualsiasi vertici. Se G non è connesso, i suoi sottografi connessi di dimensioni massime si dicono componenti connesse di G.

Alberi e foreste

Un grafo connesso privo di cicli è un **albero**, o meglio è un grafo ad albero, perché in questo caso non è definita una radice.

Una proprietà fondamentale dei grafi ad albero è che contengono il numero minimo di archi che rendono connesso un grafo avente lo stesso insieme di vertici.

Una foresta, invece, è un grafo privo di cicli (indipendentemente dal fatto che sia connesso oppure no).

Un albero ricoprente (*spanning tree*) è un grafo ricoprente connesso e privo di cicli, quindi un albero. Ovviamente per un grafo G possono esistere molteplici spanning tree.[8]

4 Implementazione della griglia

Prima di iniziare l'implementazione vera e propria degli algoritmi di generazione, inizieremo a predisporre gli strumenti di contesto che ci aiuteranno a mettere in pratica gli algoritmi proposti. Gli strumenti necessari per la creazione di un labirinto sono essenzialmente due, ovvero la griglia e le celle. Essendo queste delle entità indipendenti dagli algoritmi è opportuno rappresentarle come delle classi di oggetti. La classe Grid fungerà da contenitore per tutte le celle singole, quindi in base alla grandezza desiderata del labirinto, una volta istanziata la griglia sarà essa stessa a occuparsi di istanziare un numero di celle pari al prodotto tra numero di righe e numero di colonne e organizzarle in maniera corretta.
L'implementazione degli algoritmi analizzati nei capitoli successivi si baserà su questa struttura. Prima di avviare un algoritmo, chiamandone la funzione corrispondente, è necessario creare un'istanza della griglia; dopodiché l'algoritmo si occupa di "scavare" i passaggi tra le celle, rimuovendone i muri di separazione, creando i percorsi tortuosi che caratterizzano un labirinto.

Per l'implementazione è stato utilizzato il linguaggio C++ assieme alle librerie: *stdio.h*, *string*, *ctime*. Per la rappresentazione di array, mappe, multi-mappe sono state utilizzate rispettivamente le librerie *vector*, *unordered_map* e *map*.

4.1 Implementazione della classe *Cell*

Essendo la cella il cuore del labirinto iniziamo da essa, quindi creiamo la classe cell:

```
class cell
{
public:
    int row;
    int column;
    cell *north;
    cell *east;
    cell *south;
    cell *west;
    vector<cell*> link_vec;

    cell(int r, int c)
    {
        row = r;
        column = c;
```

```
    }
    ...
}
```

Ogni cella ha una posizione all'interno della griglia, quindi la nostra classe registra le sue coordinate, identificate da riga e colonna. Ogni cella memorizza, inoltre, le celle adiacenti a nord, sud, est e ovest. Infine, viene memorizzato un vector chiamato link_vec, che verrà utilizzato per tenere traccia delle celle adiacenti unite a quella corrente tramite un collegamento. Se tra due celle qualsiasi dell'intera griglia non è presente un collegamento, allora significa che esse saranno separate da un muro, in modo che il risolutore non possa spostarsi direttamente da una all'altra.

Il costruttore della classe accetta due parametri, che sono la riga e la colonna in cui si trova la cella nella griglia, e li utilizza per assegnare le variabili d'istanza corrispondenti.

I due metodi successivi hanno la funzione di modificare link_vec aggiungendo o togliendo collegamenti con le celle adiacenti.

```
    void link(cell *c, bool bidi=true)
    {
        link_vec.push_back(c);
        if(bidi){
            (*c).link(this, false);
        }
    }

    void unlink(cell *c, bool bidi=true)
    {
        vector<cell*>:: iterator itr;
        for(itr = link_vec.begin(); itr != link_vec.end(); ++itr)
        {
            if((*(*itr)).isEqual(c))
            {
                link_vec.erase(itr);
            }
        }
        if(bidi)
        {
            (*c).unlink(this, false);
        }
    }
```

Il metodo link() collega la cella corrente con quella passata come parametro. Invece, il metodo unlink() agisce al contrario e scollega le due celle. In entrambi i casi, tuttavia, è necessario assicurarsi che l'operazione avvenga in modo bidirezionale, affinché la connessione venga registrata in entrambe le celle. Il parametro facoltativo "bidi" passato ad entrambi i metodi garantisce che ciò accada. Il collegamento effettuato in direzione opposta viene sempre eseguito impostando bidi a false, evitando che i due metodi si richiamino all'infinito.

```
void links()
{
    vector<cell*>:: iterator itr;
    for (itr = link_vec.begin(); itr != link_vec.end(); ++itr)
    {
        cout << (*(*itr)).row << " " << (*(*itr)).column<< endl;
    }
}
```

Il metodo links() stampa a schermo tutti i collegamenti della cella corrente. I collegamenti vengono identificati tramite le coordinate di riga e colonna.

```
bool linked(cell *c)
{
    vector<cell*>:: iterator itr;
    for (itr = link_vec.begin(); itr != link_vec.end(); ++itr)
    {
        if((*(*itr)).isEqual(c))
        {
            return true;
        }
    }
    return false;
}
```

Il metodo linked() verifica se la cella corrente è collegata alla cella passata come parametro, ritornando true se la verifica è andata a buon fine e false altrimenti.

```
vector<cell*> neighbors()
{
    vector<cell*> n;
    if((*north).row > 0 && (*north).column > 0)
    {
        n.push_back(north);
    }
    if((*east).row > 0 && (*east).column > 0)
    {
        n.push_back(east);
    }
    if((*south).row > 0 && (*south).column > 0)
    {
        n.push_back(south);
    }
    if((*west).row > 0 && (*west).column > 0)
    {
        n.push_back(west);
    }
    return n;
}

cell* rand_neighbor()
{
    vector<cell*> n = neighbors();
    return n[rand()%n.size()];
}
```

Il metodo neighbors() resistuisce un vector contenente i vicini della cella corrente, invece il secondo metodo rand-neighbor() restituisce un vicino casuale tra quelli della cella corrente.

```
bool isEqual(cell *c)
{
    return row == (*c).row && column == (*c).column;
}
```

Infine, l'ultimo metodo isEqual() viene definito per verificare se due celle sono uguali confrontando le loro coordinate.[2]

4.2 Implementazione della classe *Grid*

La classe Grid essenzialmente rappresenta un involucro contenente tutte le celle necessarie. Le celle vengono organizzare in una matrice bidimensionale di vector(array).
Di seguito l'implementazione della classe Grid e del suo costruttore.

```
class grid
{
public:
    int rows;
    int columns;
    vector<vector<cell>> g;
    cell* out_bound;

    grid(int r, int c)
    {
        rows = r;
        columns = c;
        cell invalid(0,0);
        out_bound = &invalid;        //cella non valida

        //riempimento iniziale griglia
        vector<cell> empty_vec;
        g.push_back(empty_vec);        //aggiungo riga iniziale non valida
        vector<cell> horizontal;
        for(int i=1;i<=rows;i++)
        {
            horizontal.clear();
            horizontal.push_back(*out_bound);    //aggiungo colonna iniziale
                non valida
            for(int j=1; j<=columns; j++)
            {
                cell ce(i, j);
                horizontal.push_back(ce);
            }
            g.push_back(horizontal);
        }

        //assegnazione vicini
        for(int i=1;i<=rows;i++)
        {
            for(int j=1; j<=columns; j++)
            {
```

```
                g[i][j].north = verify_bound(i-1, j);
                g[i][j].east = verify_bound(i, j+1);
                g[i][j].south = verify_bound(i+1, j);
                g[i][j].west = verify_bound(i, j-1);
            }
        }
    }
    ...
}
```

Il costruttore prende le dimensioni desiderate della griglia come parametri e le imposta come attributi. Successivamente, l'inizializzazione della griglia avviene in due fasi: il riempimento della griglia con il numero di celle necessarie e l'assegnazione dei vicini a ciascuna di esse.

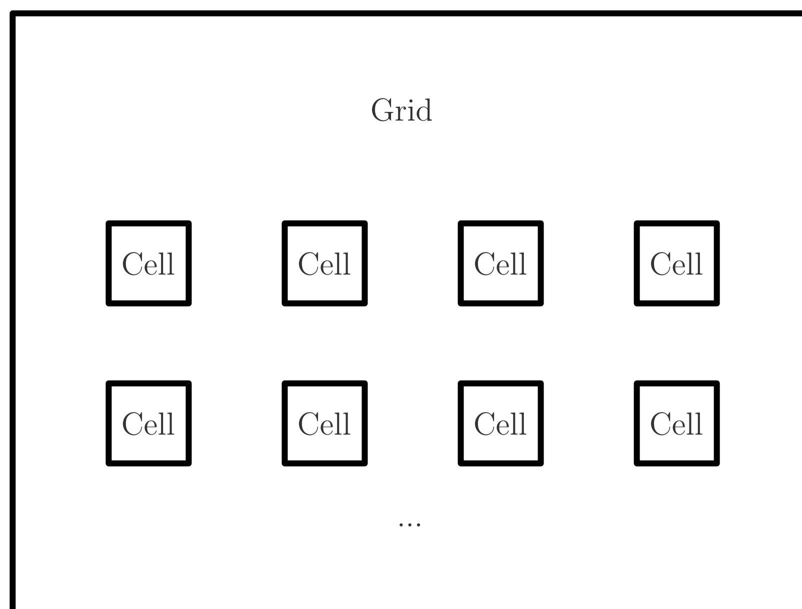


Figura 4.1:
Struttura astratta della griglia.

Per riempire la griglia è necessario creare una semplice matrice bidimensionale che contenga le varie istanze di cell. Per creare una matrice è necessario prima creare un vector di vector chiamato g, poi inserire in g un numero pari alle righe di vector di cell. I vector orizzontali sono riempiti con delle istanze di celle aventi righe e colonne corrispondenti alle coordinate della griglia. Per far corrispondere gli indici dei vector della matrice alle coordinate delle celle sono state aggiunte una riga e una colonna iniziali contenenti celle non valide (con coordinate [0,0]).

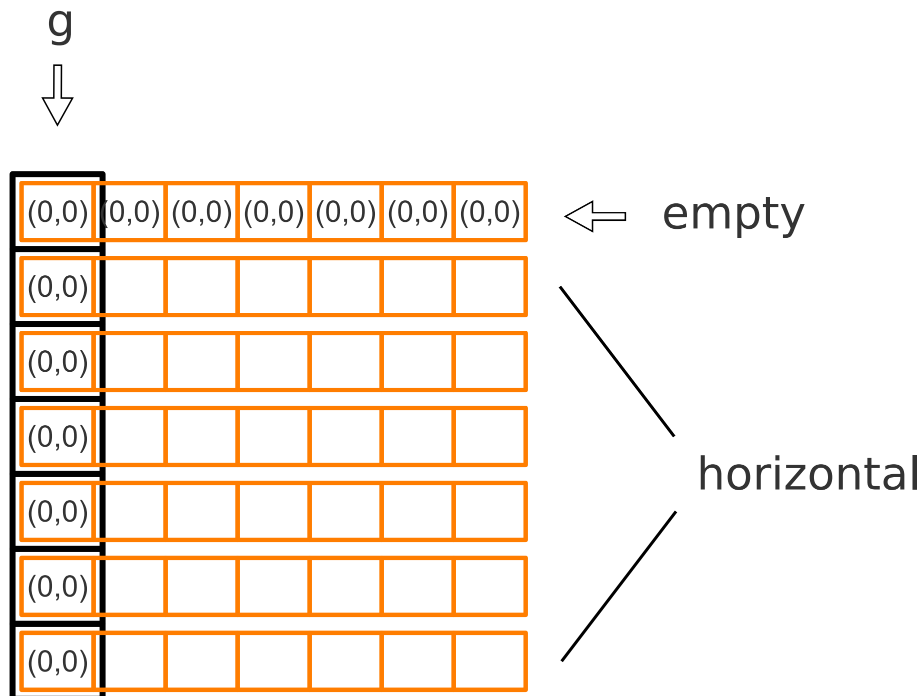


Figura 4.2:
Struttura della matrice contenente le celle.

Nella fase di assegnazione dei vicini le variabili d'istanza north, east, south, west di ciascuna cella vengono riempite con le celle rispettivamente a nord, est, sud, ovest. Si noti che i casi limite, come il confine settentrionale, nel quale le celle non hanno un vicino a nord, vengono gestiti dalla funzione verify_bound(). L'obiettivo principale di tale funzione è di controllare se le coordinate passate come parametro sono esterne ai limiti della griglia: se ciò si verifica viene restituita una cella non valida, avente coordinate [0,0].

```
cell* verify_bound(int i, int j)
{
    if(i<1 || i>rows || j<1 || j>columns)
    {
        return out_bound;
    }
    else
    {
        return &g[i][j];
    }
}
```

Oltre alla funzione verify_bound appena descritta è presente un'ulteriore funzione di supporto che verifica se una cella si trova all'interno della griglia, passando come parametri le sue coordinate.

```
bool verify_bound2(int i, int j)
{
    return !(i<1 || i>rows || j<1 || j>columns);
}
```

La funzione restituisce true se le coordinate appartengono alla griglia e false altrimenti. Ulteriori funzioni che saranno utili alla gestione degli algoritmi di generazione sono le seguenti.

```
cell* random_cell()
{
    int row = (rand()%rows)+1;
    int column = (rand()%columns)+1;
    return &g[row][column];
}

int size()
{
    return rows*columns;
}
```

La prima funzione restituisce l'indirizzo di una cella casuale all'interno della griglia. Per fare ciò vengono generate delle coordinate random tramite la funzione rand() della libreria *stdlib.h*.

La seconda funzione, invece, ritorna la dimensione della griglia, ovvero il numero di celle contenute al suo interno.[2]

5 Visualizzazione dei labirinti

5.1 Visualizzare un labirinto nel terminale

Disegnare utilizzando caratteri ASCII non è necessariamente il modo più fantasioso, né il più bello, ma è spesso il modo più conveniente per visualizzare i labirinti generati. Disegnando con il terminale non avremo bisogno di importare librerie esterne o API specifiche. Esaminiamo un possibile metodo per avvicinarci al disegno dei labirinti utilizzando solamente quattro caratteri diversi: spazio (" ") per celle e passaggi, barra verticale ("|") per le pareti verticali, il trattino ("-") per le pareti orizzontali ed il simbolo più ("+") per disegnare gli angoli. Ecco un esempio di un piccolo labirinto disegnato usando tali caratteri:

```
+---+---+---+---+---+
|   |               |
+   +   +   +---+---+
|       |           |
+   +---+---+---+   +
|       |           |
+---+   +   +---+---+
|       |           |
+   +---+---+---+   +
|   |               |
+---+---+---+---+---+
```

Figura 5.1:
Esempio di semplice labirinto visualizzato da terminale.

È evidente che ogni cella condivida i muri con i suoi vicini, quindi la parete orientale di una cella è la medesima parete occidentale del suo vicino a est. Ciò semplifica l'implementazione dello script, poiché quando disegniamo il labirinto, è sufficiente tracciare solo i confini orientali e meridionali di ogni cella. Non dobbiamo preoccuparci, invece, dei confini settentrionali e occidentali, quando la cella vicina a nord o ovest viene elaborata, automaticamente viene delineato anche il confine settentrionale e occidentale della cella corrente [2].
Tuttavia, i confini settentrionali e occidentali del labirinto stesso devono essere disegnati manualmente, dal momento che non esistono celle al di fuori della griglia a nord e ad ovest.

Essendo la funzione di visualizzazione frequentemente utilizzata, conviene aggiungerla come metodo della classe Grid. Tale metodo verrà chiamato ogni volta che un algoritmo termina la sua esecuzione, in modo da visualizzare a schermo il labirinto completato.

Aggiungiamo il seguente metodo alla classe Grid:

```
void display_grid()
{
    cout << "+";
    for(int j=1; j<=columns; j++)
    {
        cout << "---+";
    }
    cout << "\n";

    for(int i=1; i<=rows; i++)
    {
        string top = "|";
        string bottom = "+";
        for(int j=1; j<=columns; j++)
        {
            top = top + "   ";
            if(g[i][j].linked(g[i][j].east))
            {
                top = top + " ";
            }
            else
            {
                top = top + "|";
            }

            if(g[i][j].linked(g[i][j].south))
            {
                bottom = bottom + "   ";
            }
            else
            {
                bottom = bottom + "---";
            }
            bottom = bottom + "+";
        }
        cout << top << endl;
        cout << bottom << endl;
    }
}
```

Il labirinto viene rappresentato processando una riga alla volta, scorrendo le colonne da ovest ad est. Inizialmente, come accennato in precedenza, è necessario tracciare il confine settentrionale del labirinto. Il confine è rappresentato da un carattere "+" in ogni angolo superiore delle celle e da tre trattini per le pareti settentrionali delle celle appartenenti alla prima riga (linee 3-8).
Le righe successive vengono elaborate due per volta finché non si raggiunge il confine meridionale, completando la rappresentazione. Le coppie di righe elaborate contemporaneamente sono inizialmente costruite in due stringhe (top e bottom) e successivamente visualizzate nel terminale. Esse contengono:

- **top**: le pareti verticali e i corpi centrali delle celle.
- **bottom**: le pareti orizzontali e gli angoli delle celle.

Realizzazione della riga top

Innanzitutto viene inserito un carattere "|" per definire manualmente la parete occidentale del labirinto (linea 12), poi per ogni colonna vengono aggiunti:

- il corpo della cella, rappresentato da 3 spazi (linea 16)
- un passaggio, rappresentato da un singolo spazio, se la cella corrente è collegata con quella ad est (linea 19) o una parete verticale, rappresentata dal carattere "|", se la cella corrente non è collegata con quella ad est (linea 23).

Realizzazione della riga bottom

Inizialmente viene inserito un carattere "+" per definire manualmente la parete occidentale del labirinto (linea 13), successivamente per ogni colonna si aggiunge:

- un passaggio, rappresentato da 3 spazi, se la cella corrente è collegata con quella ad sud (linea 28) o una parete orizzontale, rappresentata da 3 trattini, se la cella corrente non è collegata con quella ad sud (linea 32)

Infine viene inserito un "+" per definire l'angolo inferiore-destro dell'ultima cella (linea 34).

Una volta che entrambe le stringhe sono state costruite possono essere stampate a terminale (linee 36-37).[2]

5.2 Visualizzare un labirinto mediante libreria grafica

Le illustrazioni con caratteri ASCII sono innegabilmente funzionali, ma non necessariamente attraenti. Fortunatamente, online sono disponibili librerie per eseguire il rendering grafico su aree di disegno incapsulate in finestre indipendenti del sistema operativo. Esploreremo questa tecnica con una libreria grafica scritta in C++ chiamata SFML, che ci permetterà di disegnare i labirinti in maniera più precisa ed elegante.

Esattamente come abbiamo proceduto con il metodo display_grid() per visualizzare la rappresentazione testuale del labirinto, introduciamo nella classe Grid un metodo display_grid_sfml() per sviluppare la rappresentazione grafica vettoriale.

```
1  void display_grid_sfml(int width, int height, int cell_size)
2  {
3      sf::RenderWindow window(sf::VideoMode(width*cell_size, height*
           cell_size), "Maze Generator");
4      while (window.isOpen())
5      {
6          sf::Event event;
7          while (window.pollEvent(event))
8          {
9              if (event.type == sf::Event::Closed)
10                 window.close();
11         }
12
13         window.clear();
14         grid_creation(window, cell_size);
15         digging(window, cell_size);
16         window.display();
17     }
18 }
```

Innanzitutto alla linea 3 viene creata una finestra chiamata "Maze Generator" con dimensioni width*cell_size e height*cell_size.
Width, heigth e cell_size vengono passati come parametri e corrispondono rispettivamente a: numero di colonne del labirinto, numero di righe del labirinto e dimensione della cella in pixel (si consiglia una dimensione compresa tra 20 e 40 px).
Dalla linea 4 alla 11 si configura la finestra in modo che termini automaticamente se riceve un Event di tipo Closed.

Il riempimento della finestra avviene in due fasi:

- creazione della struttura completa della griglia,
- eliminazione delle pareti delle celle a seguito della presenza di collegamenti.

Le due fasi vengono implementate in due metodi separati: grid_creation() e digging(). Infine la finestra viene visualizzata tramite chiamata window.display().

Implementazione della prima fase:

```
1   void grid_creation(sf::RenderWindow &window, int cell_size){
2       for(int i=1;i<=rows;i++)
3       {
4           for(int j=1; j<=columns; j++)
5           {
6               int gx = (j * cell_size)-cell_size;
7               int gy = (i * cell_size)-cell_size;
8
9               sf::RectangleShape rectangle_north;
10              rectangle_north.setFillColor(sf::Color::White);
11              rectangle_north.setSize(sf::Vector2f(cell_size, 2));
12              rectangle_north.setPosition(gx, gy);
13              window.draw(rectangle_north);
14
15              sf::RectangleShape rectangle_south;
16              rectangle_south.setFillColor(sf::Color::White);
17              rectangle_south.setSize(sf::Vector2f(cell_size, 2));
18              rectangle_south.setPosition(gx, gy+(cell_size-2));
19              window.draw(rectangle_south);
20
21              sf::RectangleShape rectangle_west;
22              rectangle_west.setFillColor(sf::Color::White);
23              rectangle_west.setSize(sf::Vector2f(2, cell_size));
24              rectangle_west.setPosition(gx, gy);
25              window.draw(rectangle_west);
26
27              sf::RectangleShape rectangle_east;
28              rectangle_east.setFillColor(sf::Color::White);
29              rectangle_east.setSize(sf::Vector2f(2, cell_size));
30              rectangle_east.setPosition(gx+(cell_size-2), gy);
31              window.draw(rectangle_east);
32          }
33      }
34  }
```

In questa funzione membro vengono passate la finestra (come reference) e la dimensione delle celle in pixel. I due cicli *for* alle linee 2 e 4 permettono la creazione dei bordi di ogni cella del labirinto.

Le due variabili gx e gy hanno la funzione di rappresentare la posizione di partenza di ogni cella. Tale posizione corrisponde all'angolo superiore sinistro della cella.
Successivamente vengono costruiti i vari bordi (di dimensione 2*cell_size) a nord, sud, ovest ed est. Un bordo viene generato tramite la creazione di un semplice rettangolo della classe *RectangleShape*. Per ogni rettangolo viene impostato il colore, la dimensione e la posizione. Quindi, i rettangoli vengono disegnati all'interno della finestra. Il risultato della creazione di una griglia 20x20 è mostrato in Figura 5.2.

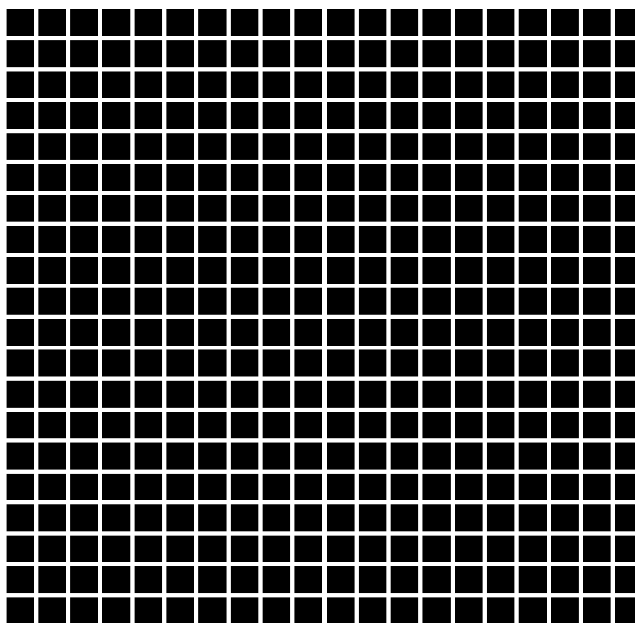


Figura 5.2:
Esempio di griglia 20x20 generata dalla prima fase di display_grid_sfml().

Implementazione della seconda fase:

```
void digging(sf::RenderWindow &window, int cell_size)
{
    for(int i=1;i<=rows;i++)
    {
        for(int j=1; j<=columns; j++)
        {
            int gx = (j * cell_size)-cell_size;
            int gy = (i * cell_size)-cell_size;
            vector<cell*> link_v = (g[i][j]).link_vec;
            vector<cell*>:: iterator itr;
            for(itr = link_v.begin(); itr != link_v.end(); ++itr)
            {
                if((*(*itr)).isEqual((g[i][j]).south))
                {
                    sf::RectangleShape passage;
                    passage.setFillColor(sf::Color::Black);
                    passage.setSize(sf::Vector2f(cell_size-4, 4));
                    passage.setPosition(gx+2, gy+(cell_size-2));
                    window.draw(passage);
                }
                if((*(*itr)).isEqual((g[i][j]).east))
                {
                    sf::RectangleShape passage2;
                    passage2.setFillColor(sf::Color::Black);
                    passage2.setSize(sf::Vector2f(4, cell_size-4));
                    passage2.setPosition(gx+(cell_size-2), gy+2);
                    window.draw(passage2);
                }
            }
        }
    }
}
```

In questa fase tutte le pareti posizionate tra coppie di celle collegate vengono rimosse. La rimozione viene effettuata semplicemente sovrapponendo dei rettangoli di colore nero (identico allo sfondo) alle pareti, in modo che non siano più visibili. Per ogni cella, vengono recuperate gx e gy ovvero le posizioni di partenza in pixel. Successivamente generati dei vector di puntatori a celle contenenti tutti i collegamenti delle varie celle (linea 9).

Come accennato precedentemente nella sezione 5.1, ogni cella del labirinto condivide i muri con i suoi vicini, quindi la parete orientale di una cella è la stessa della parete occidentale del suo vicino a est.
Di conseguenza non è necessario "scavare" le pareti in ogni direzione, ma è sufficiente farlo solo nelle direzioni sud ed est.
Se ad esempio è presente un collegamento verticale tra due celle adiacenti, è sufficiente rimuovere la parete sud della cella superiore evitando di eliminare la parete nord della cella inferiore. Per identificare i collegamenti di una cella viene utilizzato un iteratore che scorre il vettore link_v (linea 11). Se il collegamento identificato è associato alla cella a sud viene generato un rettangolo ricoprente il bordo inferiore della cella (linea 13); invece se il collegamento è associato alla cella ad est viene generato un rettangolo ricoprente il bordo ad est della cella (linea 21).
I rettangoli sovrapposti hanno dimensione 4*cell_size in modo da ricoprire due bordi contemporaneamente.
Anche in questo caso per ogni rettangolo viene impostato il colore, la dimensione e la posizione.
I rettangoli vengono disegnati all'interno della finestra. Il risultato della creazione di una griglia 20x20 comprensiva di collegamenti è mostrato in Figura 5.3.[10]

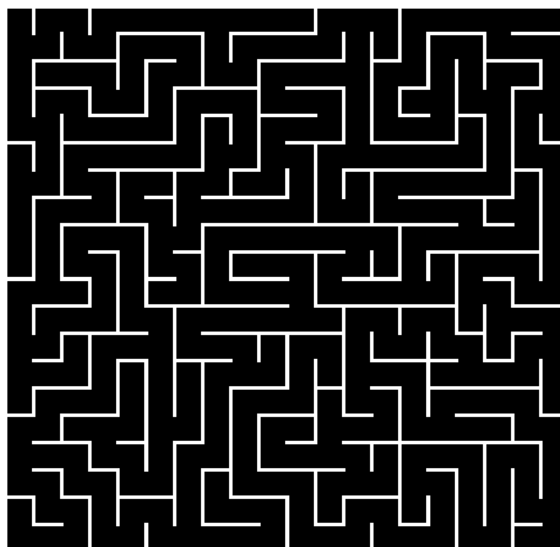


Figura 5.3:
Esempio di griglia 20x20 comprensiva di collegamenti generata da display_grid_sfml().

6 Algoritmo Binary Tree

6.1 Introduzione

L'algoritmo Binary Tree è, molto probabilmente, l'algoritmo più semplice in circolazione per generare un labirinto. Come suggerisce il nome, richiede semplicemente di scegliere tra due possibili opzioni in ogni step. Per ogni cella della griglia, è necessario decidere se scolpire un passaggio a nord o ad est. Dopo aver deciso la direzione del collegamento per tutte le celle, il labirinto è pronto per essere visualizzato a schermo.

I labirinti generati da tale algoritmo corrispondono a degli alberi binari. Gli alberi binari sono strutture dati i cui valori sono disposti gerarchicamente; ogni valore può avere al massimo due valori figlio propri.
Nel nostro caso, la radice di un albero binario generato da Binary Tree coincide con la cella posizionata nell'angolo nord-est del labirinto. Allontanandoci da tale angolo, percorrendo i vicini a sud o ad ovest di ogni cella, è possibile osservare che ogni cella successiva è essa stessa un vicolo cieco (cioè un nodo foglia dell'albero) o un genitore di massimo altri due figli.

6.2 Descrizione dell'algoritmo

Innanzitutto è necessario scegliere una cella di partenza. Questo algoritmo non dipende dall'ordine con cui procediamo; è possibile, quindi, scegliere una cella random dalla griglia (Figura 6.1). Possiamo scavare un passaggio verso la cella a nord o quella ad est scegliendo casualmente. Scaviamo verso nord (Figura 6.2).

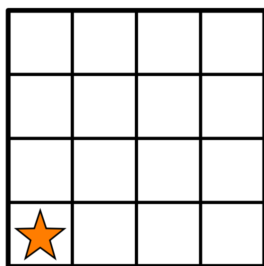


Figura 6.1

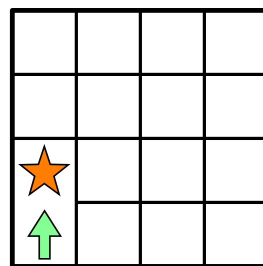


Figura 6.2

Successivamente ci spostiamo su un'altra cella e seguiamo lo stesso iter procedurale. Questa volta decidiamo di scavare ad est. Scaviamo ad est finché non raggiungiamo l'angolo inferiore destro (Figura 6.3), a questo punto l'opzione di scavare ad est non è più plausibile

in quanto usciamo dal confine orientale del labirinto e siamo costretti a scegliere il nord (Figura 6.4).

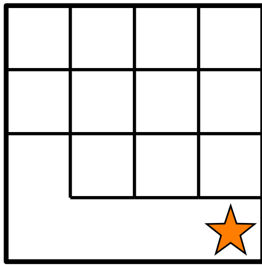


Figura 6.3

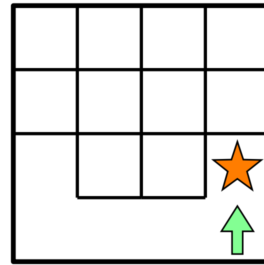


Figura 6.4

Un ulteriore caso particolare da considerare è quello riguardante la visita della cella posta nell'angolo in alto a destra. In questo caso non esiste alcuna cella né a nord né ad est: di conseguenza, siamo costretti a non scavare e passare alla cella successiva (Figura 6.5). L'algoritmo termina quando tutte le celle sono state visitate.

Analizzando l'algoritmo è possibile notare che tutte le celle poste nel confine settentrionale ed orientale del labirinto sono vincolate nel scegliere una sola direzione per creare collegamenti. Questa caratteristica dell'algoritmo si traduce necessariamente in un labirinto con un lungo passaggio che percorre il bordo nord-orientale (Figura 6.6).

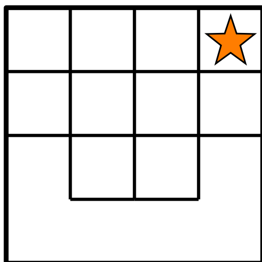


Figura 6.5

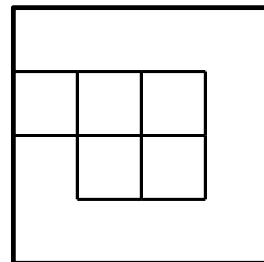


Figura 6.6

6.3 Implementazione

```
void binary_tree(grid &test)
{
    int rando;
    vector<cell*> n;
    for(int i=1;i<=test.rows;i++)
    {
        for(int j=1; j<=test.columns; j++)
        {
            n.clear();
            if(test.verify_bound2(i-1, j))  //north
            {
                n.push_back(&((test.g)[i-1][j]));
            }
            if(test.verify_bound2(i, j+1))  //east
            {
                n.push_back(&((test.g)[i][j+1]));
            }
            if(n.size() == 0)
            {
                continue;
            }

            rando = rand()%(n.size());
            ((test.g)[i][j]).link(n[rando]);
        }
    }
}
```

Il metodo binary_tree() accetta una reference ad una griglia e vi applica l'algoritmo Binary Tree, iterando su ciascuna delle sue celle.

Per ogni cella, vengono raccolte le celle adiacenti a nord e ad est. Prima di memorizzare le celle adiacenti, però, si verifica che esse facciano parte del labirinto. Se la verifica non va a buon fine semplicemente non vengono inserite nel vector di memorizzazione.

Successivamente si controlla che sia stata inserita almeno una scelta nel vector (linea 18). Se questo non accade significa che si è manifestato il caso limite della cella dell'angolo in alto a destra. In tal caso non si deve effettuare nessuna operazione ed è necessario visitare la cella successiva, tralasciando le elaborazioni seguenti (linea 20). Quindi, scegliamo

una cella in modo casuale dall'elenco rappresentato dal vector e la colleghiamo tramite il metodo link() alla cella corrente.

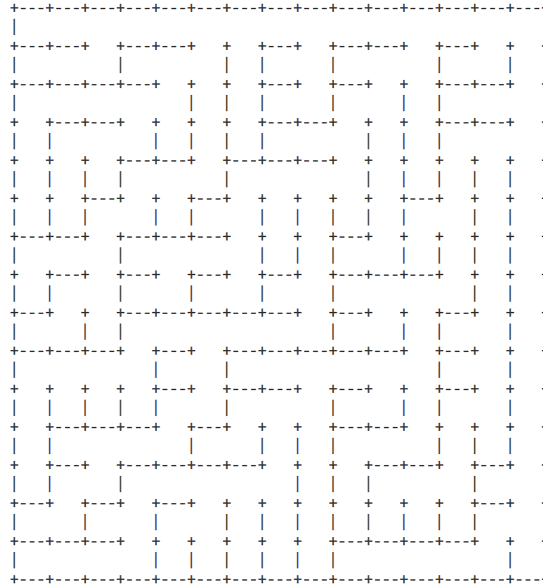


Figura 6.7:
Esempio di labirinto 15x15 generato da Binary Tree visualizzato con terminale.

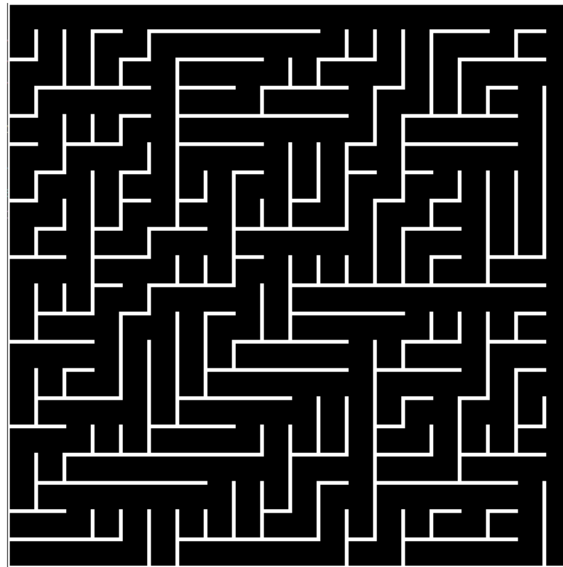


Figura 6.8:
Esempio di labirinto 20x20 generato da Binary Tree visualizzato con SFML.

Come è possibile notare dalle figure 6.7 e 6.8, i labirinti generati da Binary Tree hanno un forte *bias* diagonale, tendente verso l'angolo nord-est della griglia. I corridoi corrono per tutta la lunghezza della riga settentrionale e della colonna orientale.

7 Algoritmo Aldous-Broder

7.1 Introduzione

L'algoritmo Aldous-Broder è stato sviluppato indipendentemente sia da David Aldous, professore alla UC Berkeley, sia da Andrei Broder, attualmente Distinguished Scientist di Google. È un algoritmo semplice da implementare, quasi quanto Binary Tree.
L'idea è la seguente: iniziare da qualsiasi cella nella griglia e scegliere un vicino casuale. Spostarsi nella cella vicina scelta e, se non è già stata visitata in precedenza, collegarla alla cella precedente. Ripetere l'operazione fino a quando ogni cella non è stata ispezionata.

In realtà, l'obiettivo iniziale di questo algoritmo era quello di cercare degli Spanning-tree uniformi in un grafo. Come accennato nell'introduzione alla teoria dei grafi uno spanning tree, o albero ricoprente, è un albero contenente gli stessi identici vertici e un sottoinsieme di archi di un grafo. Con il termine uniformi si intendono spanning tree scelti in modo totalmente casuale (e con uguale probabilità) tra tutti i possibili spanning tree di un dato grafo.[1]
Nell'ambito dei labirinti le celle corrispondono ai vertici di un grafo, invece i collegamenti tra le celle corrispondono agli archi del grafo. [2]

7.2 Descrizione dell'algoritmo

Inizieremo scegliendo una cella a caso. Coloreremo di grigio le celle non ancora visitate e rappresenteremo la cella corrente con una stella.

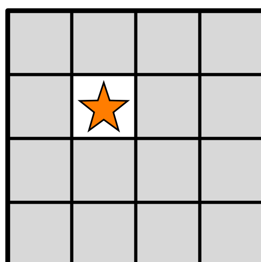


Figura 7.1

Individuata una cella random è necessario scegliere un vicino casuale, quindi scegliamo est. Il vicino tra i quali abbiamo optato non è sicuramente stato ancora visitato, quindi colleghiamo le due celle insieme, e poi reiteriamo il processo partendo dalla nuova cella. La figura seguente mostra tre passaggi consecutivi, ognuno dei quali ci porta a una nuova cella non visitata.

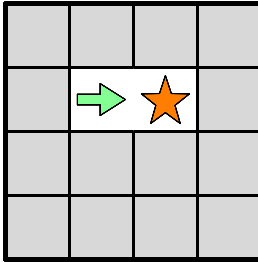


Figura 7.2

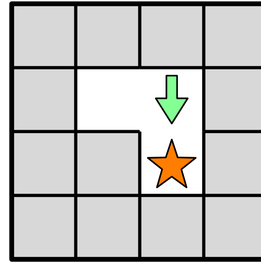


Figura 7.3

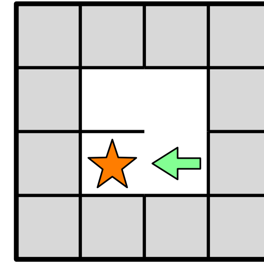


Figura 7.4

Scegliamo nuovamente un vicino casuale e optiamo per spostarci a nord. È possibile notare che è appena stato effettuato un passo di sovrapposizione, in quanto, la cella è già stata visitata precedentemente, quindi questa volta, non colleghiamo le due celle. Facciamo semplicemente del vicino la cella attuale e proseguiamo (Figura 7.5).

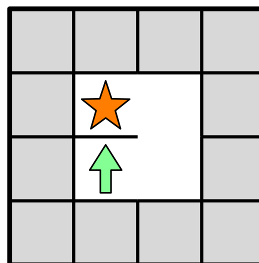


Figura 7.5

Il processo continua fino a quando ogni cella è stata perlustrata, il che, per i grandi labirinti, può richiedere un po' di tempo. Proseguendo è possibile osservare che il percorso casuale tenderà a serpeggiare, visitando e rivisitando alcune celle numerose volte. Nel momento in cui rimangono solo una o due celle non visitate, può essere decisamente esasperante osservare l'algoritmo ripetere frequentemente le stesse identiche operazioni.

7.3 Implementazione

```
void aldous_broder(grid &test)
{
    int rando;
    vector<cell*> n;
    cell *here = test.random_cell();
    cell *neighbor;
    int unvisited = test.size()-1;
    while(unvisited>0)
    {
        n.clear();
        n = (*here).neighbors();
        rando = rand()%(n.size());
        neighbor = n[rando];
        if(((*neighbor).link_vec).empty()){
            (*here).link(neighbor);
            unvisited -= 1;
        }
        here = neighbor;
    }
}
```

Il metodo aldous_broder() accetta una reference ad una griglia e vi applica l'algoritmo Aldous-Broder.
Dalla linea 5 l'algoritmo inizia scegliendo una cella della griglia a caso tramite il metodo random_cell() di Grid. Il percorso casuale inizia da questa cella. Per assicurarsi che l'algoritmo sia a conoscenza di quando tutto è stato visitato, alla linea 7 viene inizializzata una variabile "unvisited" che conta le celle non ancora visitate. In questo caso il left-value dell'inizializzazione corrispoinde alla dimensione della griglia-1, in quanto abbiamo già visitato una prima cella. Ogni volta che viene visitata una nuova cella, tale valore verrà decrementato.
Successivamente inizia un ciclo che continua fino a quando unvisited non diventa 0, ovvero quando tutte le celle sono state esplorate.

Ad ogni iterazione del ciclo, scegliamo un vicino della cella corrente in modo causale (linea 11) e lo rendiamo la nuova cella corrente (linea 19). Se la nuova cella non è ancora stata collegata ad altre celle (il che implica che non è stata visitata precedentemente), la colleghiamo alla cella corrente (linea 14).
Successivamente decrementiamo la variabile univisited e procediamo con l'iterazione.

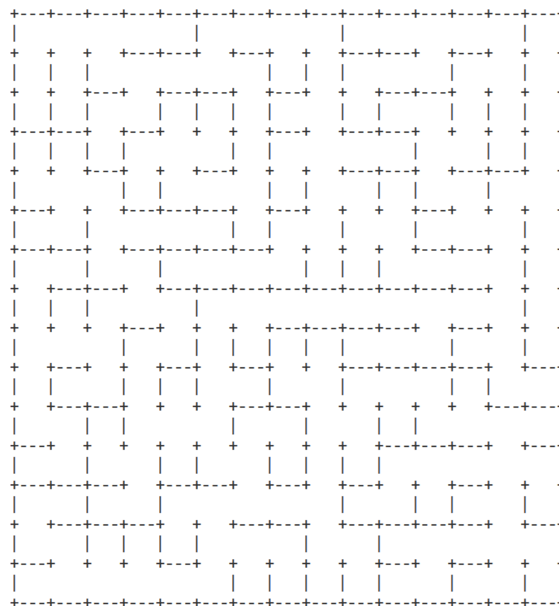


Figura 7.6:
Esempio di labirinto 15x15 generato da Aldous-Broder visualizzato con terminale.

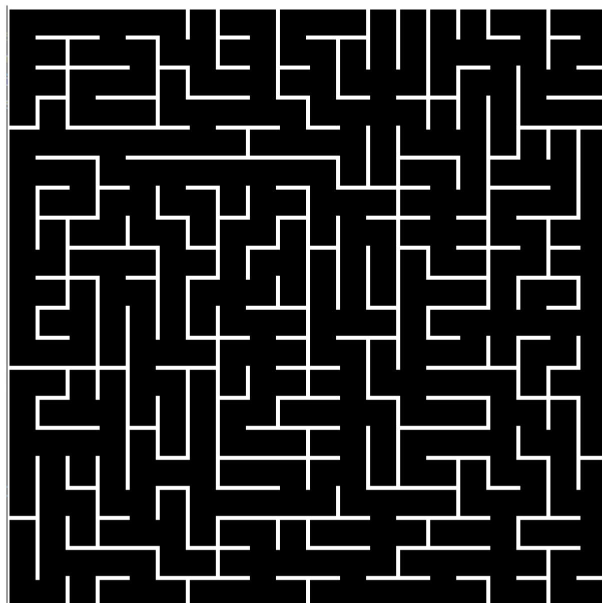


Figura 7.7:
Esempio di labirinto 20x20 generato da Aldous-Broder visualizzato con SFML.

8 Algoritmo di Wilson

8.1 Introduzione

L'algoritmo di Wilson è stato sviluppato da David Bruce Wilson, uno dei principali ricercatori di Microsoft e professore associato affiliato di matematica presso l'Università di Washington. Come Aldous-Broder, questo algoritmo dipende dall'idea di un percorso casuale, ma con una particolarità. Esegue quella che viene chiamata una *loop-erased random walk*, il che significa che mentre si sposta tra le celle, se il percorso che sta formando si interseca con se stesso e forma un loop, il loop viene eliminato prima di procedere.

L'algoritmo inizia scegliendo una qualsiasi cella nella griglia, e contrassegnandola come visitata. Si prosegue individuando un'altra qualsiasi cella non visitata nella griglia ed eseguendo una delle passeggiate casuali priva di loop fino a quando non ci si imbatte in una cella visitata. Dopo tale situazione l'algoritmo abbatte tutti i muri del labirinto incontrati durante la passeggiata, segnando come visitata ciascuna delle celle lungo il percorso. Tale processo si ripete finché tutte le celle della griglia non sono state visitate.

Inizialmente, l'algoritmo può essere frustrantemente lento da osservare, poiché è improbabile che i primi percorsi casuali si ricolleghino alle celle già perlustrate del labirinto esistente. Tuttavia, man mano che il labirinto cresce, è più probabile che le passeggiate casuali entrino in collisione con le celle del labirinto già visitate, e quindi l'algoritmo accelera notevolmente.

Come l'algoritmo di Aldous-Broder, anche questo può essere utilizzato per cercare spanning tree uniformi in un grafo. Ricordiamo che uno spanning tree è un albero che collega tutti i vertici di un grafo. Uno spanning tree uniforme è uno qualsiasi dei possibili spanning tree di un grafo, selezionato casualmente e con uguale probabilità. L'algoritmo di Wilson e quello di Aldous-Broder sono quindi molto simili e hanno alcune caratteristiche comuni:

- sono entrambi algoritmi che effettuano delle *"random walk"*;
- possono trovare spanning tree uniformi in un grafo e, contestualmente, labirinti perfetti in una griglia;
- sono entrambi *"unbiased"*, cioè garantiscono la generazione di labirinti in modo perfettamente casuale, senza preferenze per alcuna trama o caratteristica particolare.

Tuttavia l'algoritmo di Wilson converge alla soluzione molto più rapidamente, pertanto risulta notevolmente più efficiente.[3]

8.2 Descrizione dell'algoritmo

Scegliamo una cella casuale dalla quale iniziare e contrassegniamola come visitata (Figura 8.1).
Successivamente, scegliamo un'altra cella random. Indichiamo la nostra scelta con il simbolo della stella. La chiameremo "cella corrente", e sarà la cella dalla quale inizieremo la *loop-erased random walk*. Da notare che non consideriamo questa cella come visitata.

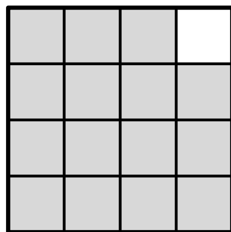


Figura 8.1

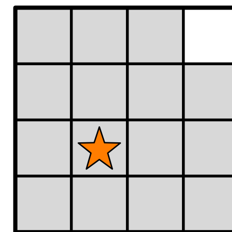


Figura 8.2

Partendo dalla cella corrente, scegliamo un vicino casuale e rendiamo quel vicino la nuova cella corrente, iteriamo questo processo. La figura 8.3 mostra il nostro percorso dopo quattro iterazioni successive. È fondamentale ribadire che in questa fase non stiamo intagliando passaggi, o modificando la griglia, stiamo solamente memorizzando il percorso.

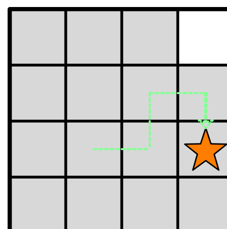


Figura 8.3

L'iterazione si interrompe solamente in due casi:

- il percorso casuale si interseca con sé stesso, creando un loop
- il percorso casuale si collega ad una cella già visitata

Simuliamo il primo caso: scegliamo come prossima cella quella posizionata ad ovest. creando un anello (Figura 8.4). Prima di procedere siamo obbligati ad eliminare dal-

la passeggiata tutte le celle appartenenti all'anello (Figura 8.5). Fatto ciò è possibile riprendere il percorso.

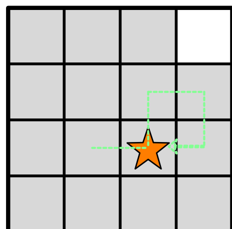


Figura 8.4

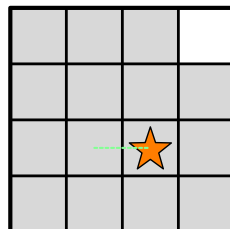


Figura 8.5

In questo esempio, l'unico caso per il quale la passeggiata termina definitivamente, è quando quest'ultima si collega alla sola cella vistata del nostro labirinto. Può essere necessario del tempo perché il percorso si riconduca alla cella in posizione [1,4]; è sicuro che presto o tardi tale situazione si verifichi. Il nostro percorso potrebbe essere simile a quello in figura 8.6.
Una volta raggiunta la cella vistata, scolpiamo il percorso nel labirinto collegando tutte le celle e contrassegnandole come visitate (Figura 8.7).

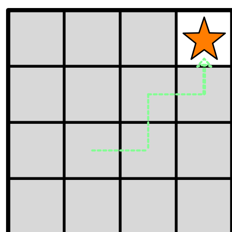


Figura 8.6

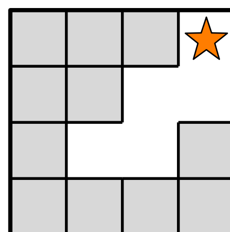


Figura 8.7

Quindi ripetiamo tutto: scegliamo una cella casuale e non visitata dalla griglia, eseguiamo una loop-erased random walk fino a quando non raggiungiamo una cella visitata e scolpiamo il percorso risultante nel labirinto (Figure 8.8,8.9,8.10).

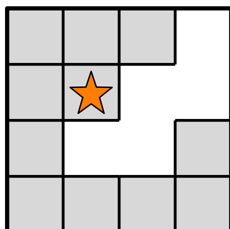


Figura 8.8

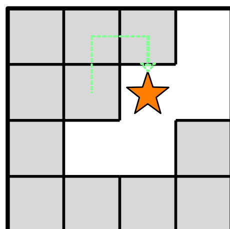


Figura 8.9

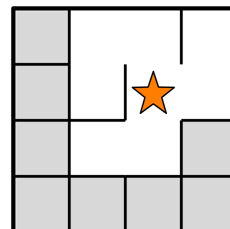


Figura 8.10

Ciò si ripete fino a quando non ci sono più celle non visitate nella griglia.

8.3 Implementazione

Prima di procedere con l'implementazione vera e propria dell'algoritmo includiamo delle *helper functions* che saranno utili in seguito:

```
bool find(unordered_map<int, cell*> &unvisited, cell* c){
    unordered_map<int, cell*>::iterator itr;
    for(itr = unvisited.begin(); itr != unvisited.end(); ++itr){
        if((*(itr->second)).isEqual(c)){
            return true;
        }
    }
    return false;
}

int find_index(unordered_map<int, cell*> &unvisited, cell* c){
    unordered_map<int, cell*>::iterator itr;
    for(itr = unvisited.begin(); itr != unvisited.end(); ++itr){
        if((*(itr->second)).isEqual(c)){
            return itr->first;
        }
    }
}

int contains(vector<cell*> path, cell *c){
    for(int i=0; i<path.size(); i++){
        if((*(path[i])).isEqual(c)){
            return i;
        }
    }
    return -1;
}
```

Le funzioni find() e find_index() consentono di cercare un elemento in una *unordered_map* partendo dal valore piuttosto che dalla chiave. Nella prima funzione viene restituito *true* se l'elemento è presente nella mappa o false altrimenti. Nella seconda funzione, invece, viene restituita la chiave corrispondente al valore passato.

La funzione contains() ha lo scopo di cercare una cella specifica in un vector restituendone l'indice corrispondente. Se la cella non è presente viene restituito il valore -1.

```
1  void wilsons(grid &test)
2  {
3      int rando;
4      vector<cell*> path;
5      unordered_map<int, cell*> unvisited;
6      //inizializzazione mappa unvisited
7      int a = 1;
8      for(int i=1;i<=test.rows;i++)
9      {
10         for(int j=1; j<=test.columns; j++)
11         {
12             unvisited.insert({a, &((test.g)[i][j])});
13             a++;
14         }
15     }
16     //visitazione prima cella
17     rando = (rand()%(unvisited.size()))+1;
18     cell *here = unvisited[rando];
19     unvisited.erase(rando);
20     int n;
21     while(!unvisited.empty())
22     {
23         //scelgo una cella casuale dalle unvisited
24         unordered_map<int, cell*>::iterator it_rand = unvisited.begin();
25         advance(it_rand, rand()%unvisited.size());
26         here = it_rand -> second;
27
28         path.clear();
29         path.push_back(here);
30         //loop-erased random walk
31         while(find(unvisited, here))
32         {
33             here = (*here).rand_neighbor();
34             if((n = contains(path, here))>=0)
35             {
36                 path.resize(n+1);
37                 path.shrink_to_fit();
38             }
39             else
40             {
41                 path.push_back(here);
```

```
42              }
43          }
44          //creazione collegamenti
45          for(int i=0; i < path.size()-1; i++)
46          {
47              (*path[i]).link(path[i+1]);
48              unvisited.erase(find_index(unvisited, path[i]));
49          }
50      }
51  }
```

L'implementazione sopra rappresentata consiste, in realtà, di tre parti diverse:

1. l'inizializzazione (linee 7-15)
2. la visita della prima cella (linee 17-19)
3. le loop-erased random walk (linee 21-49)

Nella fase di inizializzazione, viene definito un vector *path*, per memorizzare i percorsi delle loop-erased random walks. Viene poi definita una mappa non ordinata vuota (linea 5) per contrassegnare tutte le celle non visitate. L'inizializzazione della mappa unvisited avviene nelle linee 7-15. Ad ogni cella viene associata una chiave, rappresentata da un numero intero. I numeri vengono assegnati in modo crescente scorrendo le celle prima per colonne e poi per righe.

Successivamente nelle linee 17-19 viene scelta una cella casuale dalle unvisited, contrassegnata come cella corrente e rimossa dalla mappa contenente le celle non visitate. Essa è la nostra prima cella obiettivo, che la camminata casuale cercherà di raggiungere.

Le camminate casuali iniziano alla linea 21 e continuano finché non sono più presenti celle non visitate.
Una passeggiata casuale inizia scegliendo una cella non visitata a caso (linee 24-25) e aggiungendola al nostro percorso (linea 29). Questo percorso non termina fino a che non si raggiunge una delle celle visitate nella griglia (linea 31); continuiamo a camminare spostandoci tra i vicini di ogni cella (linea 33), controlliamo costantemente se si è creato o meno un loop (linea 34). Nel caso in cui si sia originato un anello, provvediamo a rimuoverlo troncando il percorso all'ultima cella aggiunta prima che il loop iniziasse a generarsi (linee 36-37). In caso contrario, se la cella non avesse dato origine ad un ciclo, la aggiungiamo al percorso e proseguiamo (linea 41).

Una volta raggiunta una cella visitata, la camminata termina. Il percorso memorizzato viene quindi "scavato" (linea 47), collegando ogni cella in sequenza tramite il metodo link(). Infine, le celle appartenenti al percorso vengono contrassegnate come visitate e quindi rimosse dalla mappa unvisited.

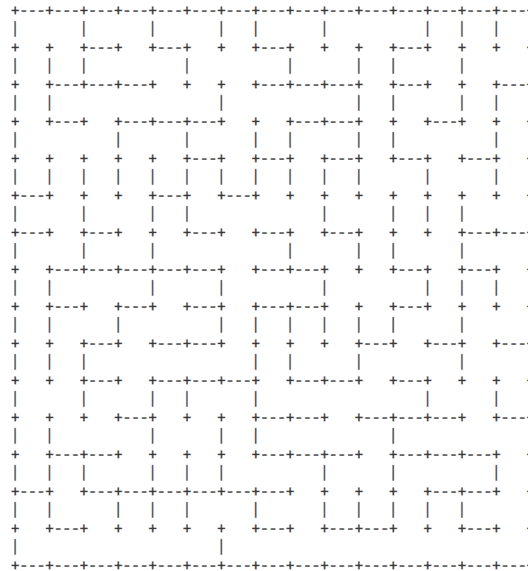


Figura 8.11:
Esempio di labirinto 15x15 generato da Wilson's visualizzato con terminale.

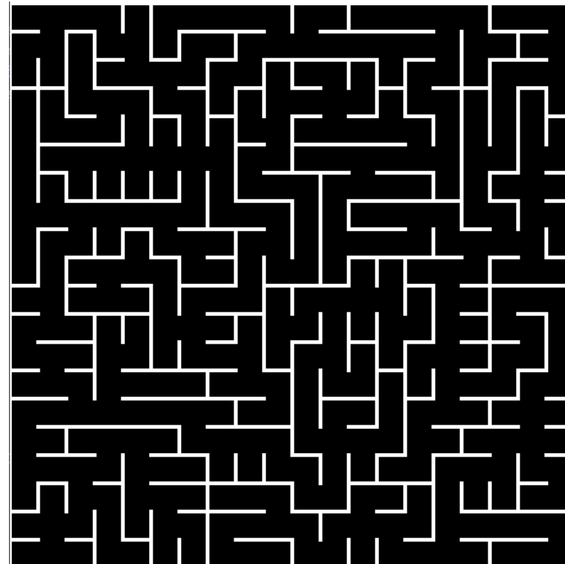


Figura 8.12:
Esempio di labirinto 20x20 generato da Wilson's visualizzato con SFML.

9 Algoritmo Recursive Backtracker

9.1 Introduzione

L'algoritmo Recursive Backtracker è una versione randomizzata dell'algoritmo di ricerca in profondità per grafi.
La ricerca in profondità (DFS) è un algoritmo utilizzato per l'attraversamento di strutture dati come alberi o grafi. Esso inizia dal nodo radice (selezionando un nodo arbitrario come nodo radice nel caso di un grafo) ed esplora ogni ramo il più lontano possibile prima di tornare indietro (backtracking).

Recursive Backtracker è veloce, intuitivo e semplice da implementare. Tuttavia, oltre alla memoria per archiviare l'intero labirinto, richiede la memorizzazione di uno stack[1] proporzionale alle dimensioni del labirinto, quindi per labirinti eccezionalmente grandi può essere abbastanza inefficiente. Ma per la maggior parte dei labirinti funziona egregiamente.

A partire da una cella qualunque, l'algoritmo seleziona una cella adiacente casuale che non è stata ancora visitata. rimuove il muro tra le due celle e contrassegna la nuova cella come visitata, aggiungendola allo stack per facilitare il backtracking. L'algoritmo continua questo processo finchè non incontra una cella priva di vicini non visitati, considerata un vicolo cieco. Quando si trova in un vicolo cieco, torna indietro lungo il percorso fino a raggiungere una cella con un vicino non visitato. In seguito prosegue con la generazione del percorso visitando la nuova cella non visitata. Tale processo continua fino a quando ogni cella non è stata esplorata, quindi retrocede fino alla cella iniziale.

Il backtracking utilizzato dall'algoritmo di ricerca in profondità può essere implementato in modo iterativo o anche tramite una routine ricorsiva. Uno svantaggio dell'ultimo approccio è una grande profondità di ricorsione: nel peggiore dei casi, la routine potrebbe dover ricorrere a ogni cella dell'area in elaborazione, superando in molti ambienti la profondità massima di ricorsione. In questa analisi ho utilizzato il backtracking iterativo, dichiarando esplicitamente lo stack.

I labirinti generati con una ricerca in profondità hanno un basso fattore di ramificazione e contengono corridoi molti lunghi, in quanto l'algoritmo esplora il più possibile lungo ogni arco prima di retrocedere.[7]

[1] Lo stack è un'area di memoria contigua gestita in modalità Last In First Out (LIFO), cioè l'ultimo oggetto inserito è il primo ad essere rimosso. Le due operazioni principali sono push (aggiunge un elemento in cima allo stack) e pop (rimuove un elemento dalla cima dello stack).

9.2 Descrizione dell'algoritmo

Iniziamo scegliendo una cella casuale dalla griglia e inseriamola nello stack (Figura 9.1). Rappresentiamo lo stack con una colonna arancione; ogni cella potrà essere inserita o rimossa solamente dalla cima. Qualsiasi cella si trovi nella parte superiore della pila sarà sempre considerata la cella corrente.

Osserviamo i vicini non visitati della nostra cella attuale, ne scegliamo uno a caso (ci spostiamo in A3) e scolpiamo un percorso, inserendolo contemporaneamente sul nostro stack. Questo ha l'effetto di rendere A3 la nostra nuova cella attuale.
Il processo continua, camminando casualmente attraverso la griglia, come dimostra la figura 9.2. Lo stack includerà ogni cella visitata finora.

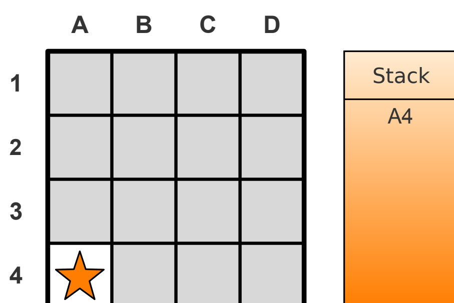


Figura 9.1

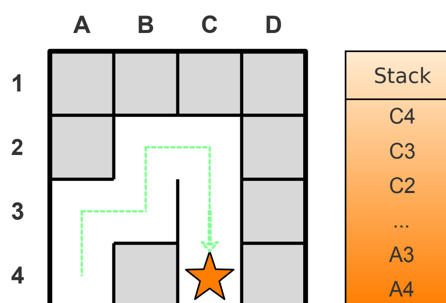


Figura 9.2

Scegliamo nuovamente un vicino casuale e optiamo per spostarci ad ovest (Figura 9.3). Purtroppo, a questo punto, la nuova cella corrente è priva di vicini non visitati: ci siamo imbattuti in un vicolo cieco. Siamo costretti ad eliminare la cella B4 dallo stack, il che ha l'effetto di rendere la cella precedente, C4, nuovamente la nostra cella corrente (Figura 9.4).

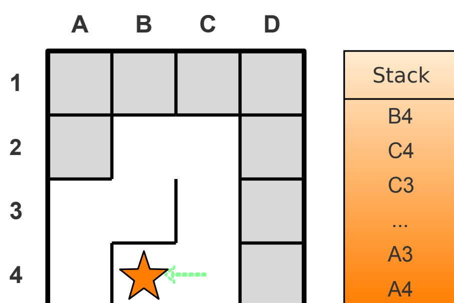


Figura 9.3

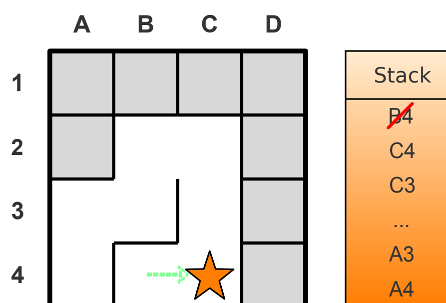


Figura 9.4

Il processo continua in questo modo, retrocedendo ad ogni vicolo cieco, fino a che ogni cellula è stata visitata (Figura 9.5).

L'ultima cella da visitare è sempre un vicolo cieco, di conseguenza è necessario tornare all'origine. Ripercorriamo i nostri passi, rimuovendo le celle dalla pila, cercandone una con un vicino non visitato. In questo caso, però, non sono rimaste celle non visitate, quindi rimuoviamo le celle dallo stack fino a quando non torniamo al punto di partenza, in A4 (Figura 9.6). Poiché tale cella è priva di vicini non visitati, estraiamo anch'essa dalla pila, il che lascia la pila vuota. Una volta che la pila è priva di elementi si deduce che il labirinto è stato completato e l'algoritmo termina.

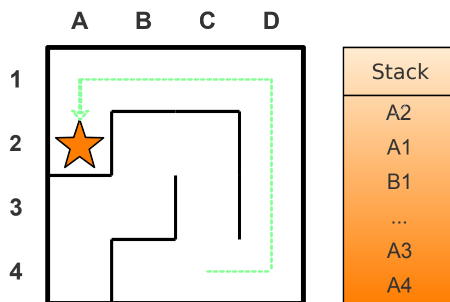


Figura 9.5

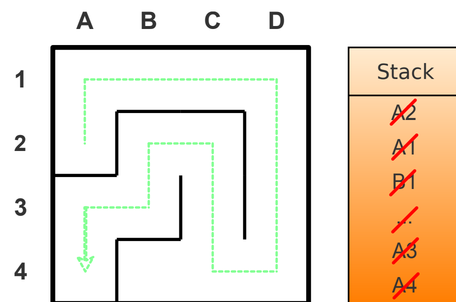


Figura 9.6

Il labirinto creato è rappresentato dalla figura seguente.

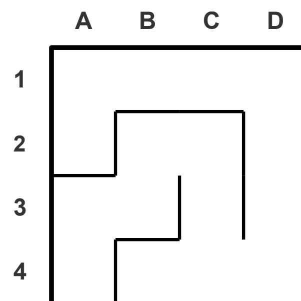


Figura 9.7

9.3 Implementazione

```
void recursive_backtraker(grid &test)
{
    vector<cell*> stack;
    vector<cell*> neighbors;
    vector<cell*> unvisited_neighbors;
    cell *neighbor;
    int initial_size;
    cell *here = test.random_cell();
    stack.push_back(here);
    while(!stack.empty())
    {
        neighbors.clear();
        unvisited_neighbors.clear();
        here = stack.back();
        neighbors = here->neighbors();
        initial_size = neighbors.size();
        for(int i=0; i<initial_size; i++)
        {
            if((neighbors[i]->link_vec).empty())
            {
                unvisited_neighbors.push_back(neighbors[i]);
            }
        }
        if(unvisited_neighbors.empty())
        {
            stack.pop_back();
        }
        else
        {
            neighbor = unvisited_neighbors[rand()%unvisited_neighbors.size
                ()];
            here->link(neighbor);
            stack.push_back(neighbor);
        }
    }
}
```

Innanzitutto vengono dichiarate alcune strutture dati utili in seguito.

- **stack**: vector di celle che rapparesenta lo stack.
- **neighbors**: vector contenente i vicini di una cella.
- **unvisited_neighbors**: vector contenente i vicini non ancora visitati di una cella, quindi un sottoinsieme di neighbors.

Successivamente, nelle linee 8-9 viene scelta una cella casuale dalla griglia, contrassegnata come cella corrente e aggiunta allo stack. In seguito, inizia un ciclo che continua fino a quando lo stack risulta vuoto. Viene inizializzato il vector neighbors con i vicini della cella corrente. Partendo da tali vicini si selezionano solamente quelli che non sono ancora stati visitati. Quest'ultimi vengono inseriti nel vector unvisited_neighbors.

Se non sono presenti vicini non ancora visitati, si rimuove la cella corrente dallo stack e si procede (linee 24-27), altrimenti si seleziona un vicino non visitato in modo casuale e lo si collega alla cella corrente. Infine il vicino viene aggiunto allo stack e si procede (linee 28-33).

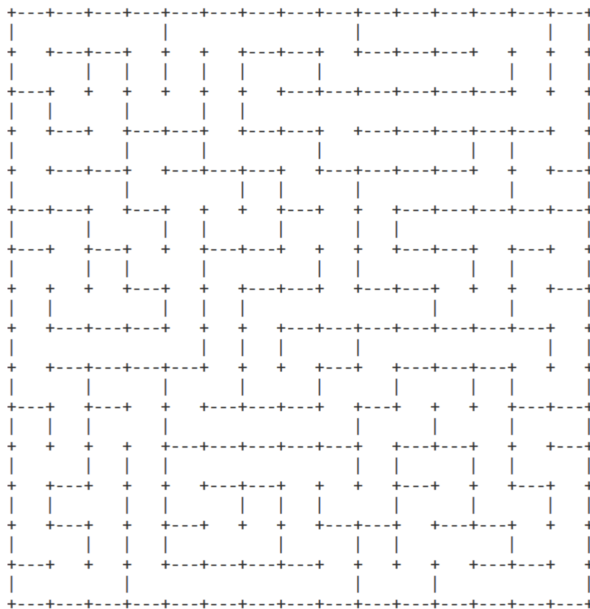


Figura 9.8:
Esempio di labirinto 15x15 generato da Kruskal's visualizzato con terminale.

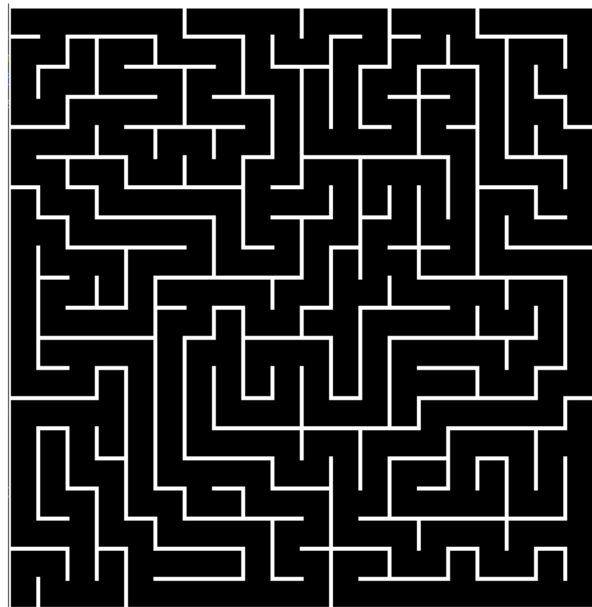


Figura 9.9:
Esempio di labirinto 20x20 generato da Kruskal's visualizzato con SFML.

10 Algoritmo di Kruskal

10.1 Introduzione

L'algoritmo di Kruskal è stato sviluppato dal matematico e informatico Joseph Kruskal nel 1956 per costruire quelli che vengono chiamati *minimum spanning tree*. In un grafo pesato, un minimum spanning tree è uno spanning tree con peso totale degli archi minimo.

L'algoritmo di Kruskal originale esordisce elaborando i vertici associati agli archi di peso minimo, iniziando a creare uno spanning tree minimo, che diventerà la soluzione complessiva. Se sono presenti archi con peso uguale, l'algoritmo ne sceglie uno casuale. Successivamente seleziona un altro arco con peso minimo non ancora elaborato e lo aggiunge allo spanning tree. Tale processo continua fino a quando tutti i nodi del grafo originale sono stati aggiunti. Una volta che tutti i nodi nel grafo appartengono allo spanning tree minimo, l'algoritmo si ferma.
Durante la creazione dello spanning tree minimo, è fondamentale che l'algoritmo non crei alcun ciclo, in modo da generare un labirinto perfetto.

Potremmo implementare l'algoritmo esattamente come descritto, completo di archi ponderati che collegano le celle, ma analizzato il nostro scopo, possiamo semplificarlo leggermente.
Al contrario di alcune applicazioni nelle quali i pesi degli archi sono importanti, solitamente non sono così rilevanti quando si generano labirinti. Per trasformare una griglia in un labirinto usando l'algoritmo di Kruskal, dovremmo assegnare pesi casuali a tutti i passaggi possibili, prima di poterli elaborare in ordine di peso. Questo approccio è un po' ridondante. In realtà è molto più facile inserire tutti i passaggi in un grande elenco e selezionarli casualmente. Tale piccolo cambiamento converte l'algoritmo di Kruskal nell'algoritmo di Kruskal randomizzato.

L'algoritmo di Kruskal, oltre ad essere ideale per generare labirinti perfetti, è ottimo anche per la generazioni di labirinti a trama (descritti nella sotto-sezione 2.1).
Essendo le applicazioni per questo tipo di labirinti svariate, le esigenze di generazione sono differenti. Kruskal è in grado di creare labirinti con un numero coerente e prevedibile di "trame"; inoltre, ha la peculiarità di poterle posizionare precisamente nella griglia, a piacimento dell'utente. In questo modo Kruskal permette di adattare i passaggi intrecciati a modelli e disegni specifici.[2]

10.2 Descrizione dell'algoritmo

Supponiamo di iniziare con un grafo, o griglia, in cui ogni possibile passaggio che potrebbe collegare celle vicine ha un costo/peso. Potrebbe assomigliare alla figura seguente.

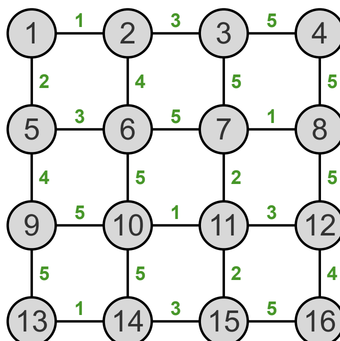


Figura 10.1

Associamo ad ogni vertice un insieme. Inizialmente ogni vertice appartiene ad un insieme diverso. I vari insiemi sono indicati da numeri crescenti, in questo caso da 1 a 16, scritti all'interno di ciascun vertice.
Anche i pesi degli archi sono rappresentati da un numero: più alto è il numero, maggiore è il peso. Nella figura i pesi sono posti vicino agli archi e colorati di verde.

Come accennato precedentemente, cominciamo ad elaborare gli archi con peso minore, quindi partiamo da quelli con peso 1. Prendiamo come esempio l'arco che collega le celle [1,1] e [1,2], notiamo che appartengono a due insiemi differenti, rispettivamente insiemi 1 e 2. Di conseguenza, possiamo collegare i due nodi aggiungendoli entrambi alla soluzione. Inoltre, è necessario che ambedue i vertici appartengano allo stesso insieme, quindi scegliamo casualmente uno degli insiemi di partenza e assegnamolo ad entrambi. Procediamo con la medesima modalità per gli altri archi di peso 1.

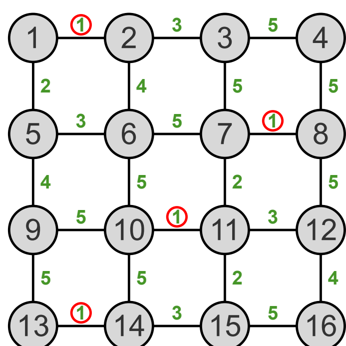


Figura 10.2

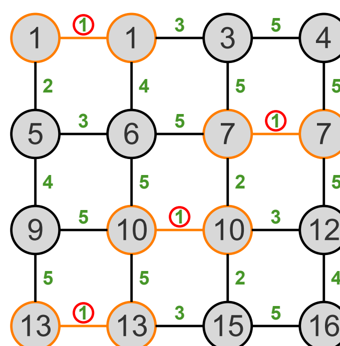


Figura 10.3

Come mostra la figura successiva, i passaggi più "economici" a questo punto hanno tutti un peso 2.
Ripetiamo, quindi, per ogni passaggio di peso 2, la stessa procedura seguita precedentemente. Gli insiemi di ogni coppia di celle vengono confrontati e risultano diversi, quindi le celle vengono collegate e gli insiemi fusi tra loro.

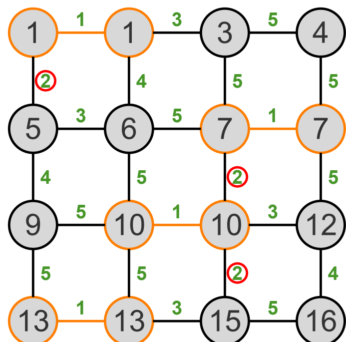


Figura 10.4

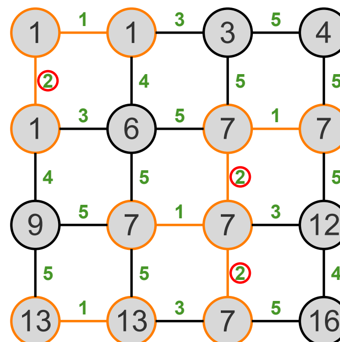


Figura 10.5

Nelle iterazioni successive, tutti i passaggi con peso 3 saranno elaborati, continuando a far crescere le porzioni che formeranno lo spanning tree finale.

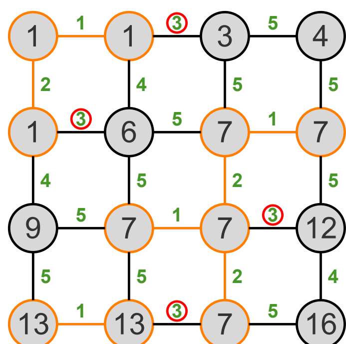


Figura 10.6

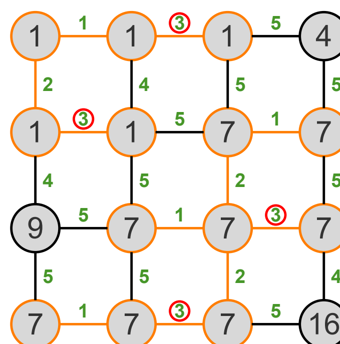


Figura 10.7

Passando allo step seguente, i passaggi più economici sono ora tutti ponderati 4; ci imbattiamo nel primo piccolo intoppo. Si consideri il passaggio che collega le celle [1,2] e [2,2], evidenziato a sinistra, in figura 10.8. Se dovessimo selezionare tale passaggio e unire le due celle, finiremmo per collegare due celle che appartengono entrambe allo stesso insieme. Poiché ogni insieme rappresenta un piccolo labirinto autonomo, ciò significherebbe aggiungere un ciclo al labirinto, cosa che non dobbiamo consentire.
Evitiamo ciò, tralasciando completamente l'arco e proseguendo.

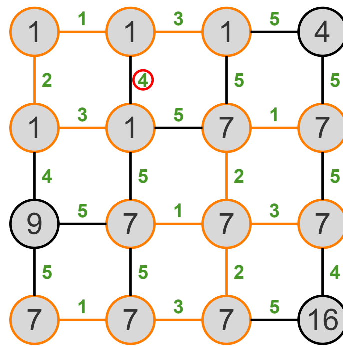


Figura 10.8

Continuiamo il processo finché tutti i vertici appartengono allo stesso insieme. Un possibile risultato è rappresentato dalla figura 10.9, alla quale corrisponde il labirinto di figura 10.10.

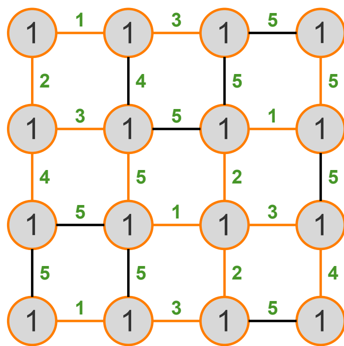


Figura 10.9

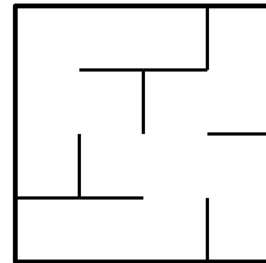


Figura 10.10

10.3 Implementazione

```
void kruskal(grid &test)
{
    vector<pair<cell*,cell*>> couples;
    pair<cell*, cell*> tmp;
    unordered_map<cell*, int> set_for_cell;
    multimap<int, cell*> cells_in_set;
    vector<cell*> losers;
    //INIZIALIZZAZIONE GRID
    int a = 1;
    for(int i=1;i<=test.rows;i++)
    {
        for(int j=1; j<=test.columns; j++)
        {
            set_for_cell.insert({&((test.g)[i][j]), a});
            cells_in_set.insert({a, &((test.g)[i][j])});
            a++;
        }
    }
    //RIEMPIMENTO COUPLES
    for(int i=1;i<=test.rows;i++)
    {
        for(int j=1; j<=test.columns; j++)
        {
            if(test.g[i][j].south->row > 0 && test.g[i][j].south->column > 0)
            {
                tmp = make_pair(&(test.g[i][j]), test.g[i][j].south);
                couples.push_back(tmp);
            }
            if(test.g[i][j].east->row > 0 && test.g[i][j].east->column > 0)
            {
                tmp = make_pair(&(test.g[i][j]), test.g[i][j].east);
                couples.push_back(tmp);
            }
        }
    }
    //INIZIO ALGORITMO
    //ora tmp viene utilizzato per estrarre le coppie
    int winner;
    int loser;
    while(!couples.empty())
```

```
41      {
42          vector<pair<cell*,cell*>>::iterator it_rand = couples.begin();
43          advance(it_rand, rand()%couples.size());
44          tmp = *it_rand;
45          couples.erase(it_rand);
46          if(set_for_cell[tmp.first] != set_for_cell[tmp.second])
47          {
48              tmp.first->link(tmp.second);
49              winner = set_for_cell[tmp.first];
50              loser = set_for_cell[tmp.second];
51
52              //RIEMPIMENTO LOSERS
53              losers.clear();
54              multimap<int,cell*>::iterator itr;
55              pair<multimap<int, cell*>::iterator, multimap<int, cell*>::
                    iterator> ret;
56              ret = cells_in_set.equal_range(loser);
57              for (itr=ret.first; itr!=ret.second; ++itr)
58              {
59                  losers.push_back(itr->second);
60              }
61
62              //AGGIORNAMENTO DELLE DUE MAPPE
63              for(int i=0; i<losers.size(); i++)
64              {
65                  //esempio se winner=1 e loser=2 e losers=a,b,c
66                  cells_in_set.insert({winner, losers[i]});        //inserisco
                        a,b,c in cells_in_set con chiave 1
67                  set_for_cell.erase(losers[i]);                   //elimino a
                        ,b,c da set_for_cell
68                  set_for_cell.insert({losers[i], winner});        //
                        reinserisco a,b,c in set_for_cell con valore 1 al posto
                        di 2
69              }
70              cells_in_set.erase(loser);                           //elimino
                    gli elementi con chiave 2 da cells_in_set
71          }
72      }
73  }
```

Innanzitutto vengono dichiarate alcune strutture dati utili in seguito.

- **couples**: vector di pair contenente gli archi da elaborare.
- **tmp**: pair temporaneo per inserire/estrarre gli archi in couples, contiene una coppia di celle.
- **set_for_cell**: mappa che associa ad ogni cella un preciso insieme.
- **cells_in_set**: multi-mappa che associata ad un insieme le celle che ne appartengono.
- **losers**: vector temporaneo contenente le celle che devono cambiare insieme.

Successivamente, alle linee 9-18 vengono inizializzate la mappa set_for_cell e la multimappa cells_in_set. Ad ogni cella viene associato in modo univoco un numero crescente, rappresentante l'insieme, scorrendo prima le colonne e poi le righe.

Dalla linea 20 alla linea 35, viene inizializzato il vector couples. Come accennato nella sezione 5.1, ogni muro interno del labirinto viene condiviso tra due celle, quindi è sufficiente aggiungere i passaggi con direzione sud ed est di ogni cella. Ogni passaggio è rappresentato dalla struttura *pair*, contenente una coppia di celle.

L'algoritmo vero e proprio inizia con un ciclo *while* che termina solamente nel caso in cui tutti i passaggi sono stati elaborati. Ad ogni iterazione si sceglie in modo casuale un passaggio dal labirinto e lo si elimina dall'array *couples* (linee 42-45). Si verifica poi che le celle congiunte dal passaggio siano appartenenti a due insiemi differenti. Se ciò non si verifica l'algoritmo continua con il passaggio successivo, senza effettuare nessuna operazione.

Alla linea 48, le celle del passaggio vengono semplicemente collegate. Nelle linee seguenti vengono memorizzati gli indici rappresentati gli insiemi di appartenenza delle due celle collegate. In modo casuale si associa alla variabile winner l'insieme di una cella e alla variabile loser l'insieme dell'altra. Tutte le celle appartenenti all'insieme loser dovranno spostarsi nell'insieme identificato da winner.

Per fare ciò in maniera più chiara, viene riempito il vector losers con tutte le celle appartenenti all'insieme loser (linee 53-60). Per ogni cella di losers vengono effettuate le seguenti operazioni:

- la cella viene inserita in cells_in_set con la nuova chiave winner (linea 66),

- la cella viene eliminata dalla mappa set_for_cell (linea 67),
- la cella viene reinserita nella mappa set_for_cell con il nuovo valore winner (linea 68).

Infine viene rimosso l'insieme loser da cells_in_set (linea 70).

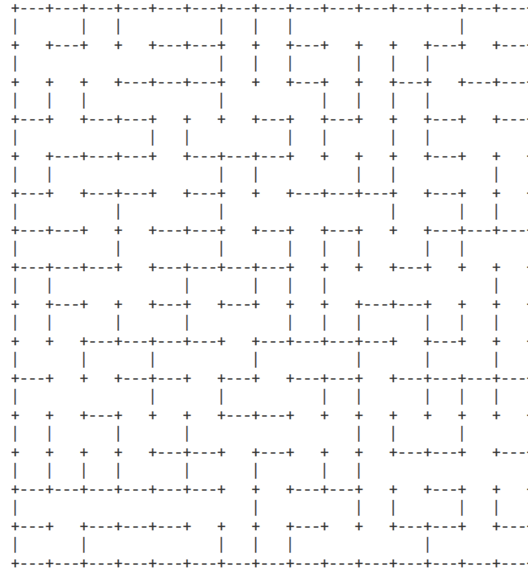


Figura 10.11:
Esempio di labirinto 15x15 generato da Kruskal's visualizzato con terminale.

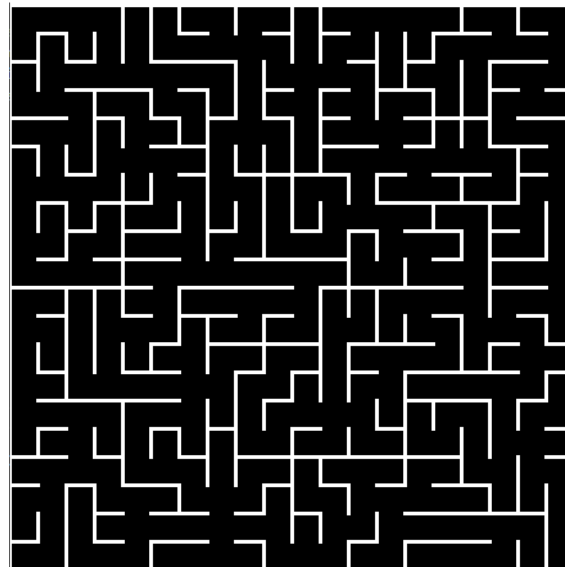


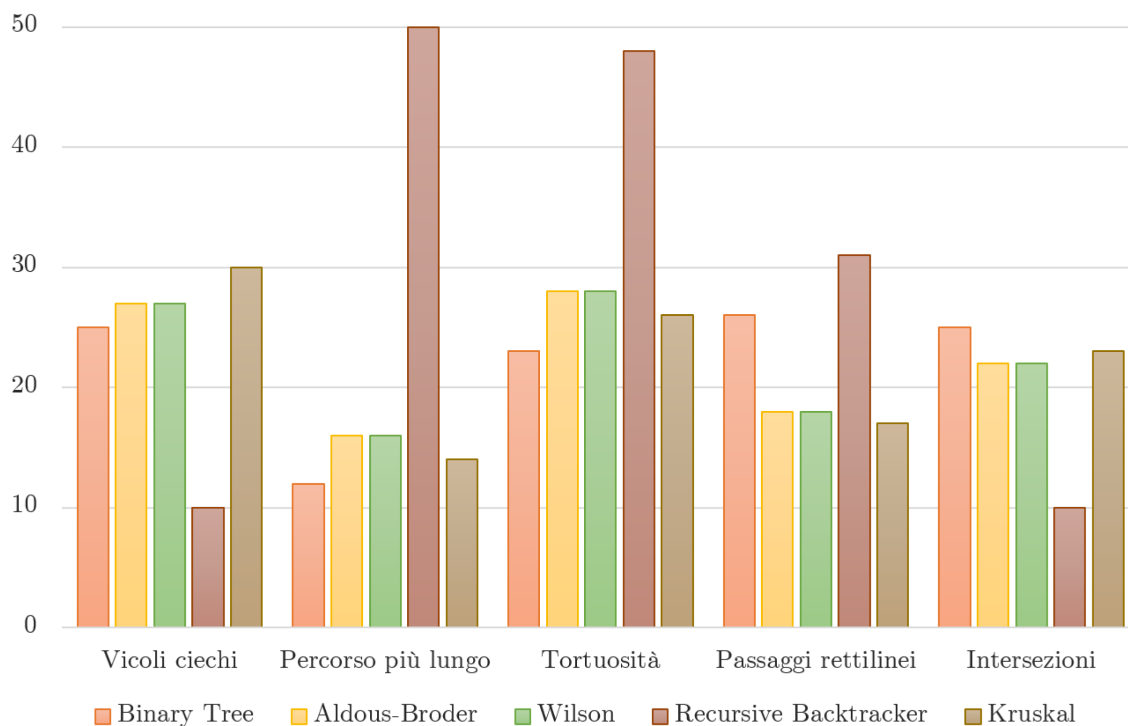
Figura 10.12:
Esempio di labirinto 20x20 generato da Kruskal's visualizzato con SFML.

11 Confronto tra gli algoritmi di generazione

Non è significativo confrontare gli algoritmi semplicemente osservando i labirinti generati: per una analisi più precisa è utile approfondire varie caratteristiche separatamente. Per questo motivo sono state eseguite alcune statistiche considerando sei proprietà fondamentali.

- **Vicoli ciechi**: Un vicolo cieco è definito come una cella collegata ad un solo vicino. Pochi vicoli ciechi e lunghi percorsi potrebbero suggerire lunghi tragitti, che possono portare il risolutore lontano.
- **Percorso più lungo**: Il percorso più lungo è stato calcolato utilizzando l'algoritmo di Dijkstra. È rappresentato come la percentuale di celle che si trovano sul percorso più lungo del labirinto.
- **Tortuosità**: la tortuosità è una misura di quanto spesso un passaggio cambi direzione. È rappresentata come la percentuale di celle con un passaggio che svolta improvvisamente verso sinistra o verso destra.
- **Passaggi rettilinei**: è l'opposto della tortuosità. È una misura di quante celle in una griglia formano percorsi in linea retta, orizzontalmente o verticalmente.
- **Intersezioni**: le intersezioni danno un'indicazione di quanto spesso qualcuno che attraversa il labirinto dovrà prendere una decisione. È stato calcolato il numero di intersezioni a tre direzioni.

Per generare i dati necessari è stato eseguito ogni algoritmo un determinato numero di volte, su una griglia di dimensioni fisse. Il grafico seguente è stato creato eseguendo ogni algoritmo mille volte, su una griglia 32×32.[2]



Osservando le statistiche è possibile, nella maggior parte dei casi, confermare le peculiarità di ogni algoritmo introdotte nei capitoli precedenti.

Sebbene l'algoritmo Binary Tree non sia *unbiased*, presenta una percentuale media in tutti i settori.

Essendo gli algoritmi di Aldous-Broder e Wilson entrambi *unbiased* ed entrambi utilizzati per la generazione di spanning tree uniformi, i loro output sono statisticamente identici.

L'algoritmo Recursive Backtracker ha un numero notevolmente alto di passaggi lunghi e tortuosi e un numero esiguo di vicoli ciechi e intersezioni. Ciò significa che esso genera labirinti che portano il risolutore ad eseguire estenuanti camminate, attraversando frequentemente l'intera area. Recurive Backtracker è un algoritmo potenzialmente veloce, garantisce di visitare ogni cella solamente due volte e ha bisogno di un'elevata memoria per tenere traccia delle celle visitate in precedenza.

I labirinti generati da Kruskal, invece, sono molto simili a quelli generati da Aldous-Broder e Wilson. Kruskal produce labirinti molto regolari e uniformi. Eccelle nella produzione di labirinti che sono l'unione di insiemi disgiunti. I labirinti di Kruskal presentano un buon numero di intersezioni e vicoli ciechi.

12 Conclusioni

Lo sviluppo di questo elaborato mi ha permesso di comprendere quanto l'argomento trattato non sia banale. La generazione di labirinti e, in genere, i labirinti rappresentano un fenomeno molto stimolante che merita uno studio approfondito.
In questa tesi sono stati analizzati solamente cinque algoritmi per generare labirinti, ma ne esistono numerosi, come ad esempio: Eller's, Growing Tree, Hunt-and-kill, Prim's, Recursive Division, Sidewinder. Ciò permette di intuire quanto l'argomento sia ampio e complesso. L'esistenza di un numero così elevato di algoritmi garantisce la possibilità di generare labirinti con caratteristiche precise desiderate dall'utente.
Durante la stesura della tesi, ho appreso quanto il concetto di labirinto sia simile a quello di grafo. Questo permette di utilizzare la maggior parte degli algoritmi di generazione anche per elaborazioni sui grafi, struttura dati ampiamente utilizzata nell'ambito dell'informatica.
Uno sviluppo futuro di questo elaborato potrebbe riguardare la creazione di labirinti più complessi come quelli a treccia o a trama. Un'ulteriore espansione consiste nel modificare la struttura di base composta da griglia e celle, descritta nel capitolo 4, in modo da poterla adattare a forme particolari, creando labirinti con tassellazione differente da quella ortogonale come: Theta, Sigma e Delta.
Oltre agli algoritmi per generare labirinti sono presenti anche algoritmi per la risoluzione. Uno di questi è il famoso algoritmo di Dijkstra, usato comunemente per trovare uno *shortest path* (cammino minimo) in un grafo.

Riferimenti bibliografici

[1] David J. Aldous. "The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees". In: *SIAM Journal on Discrete Mathematics* 3.4 (1990), pp. 450–465. DOI: 10.1137/0403039. eprint: https://doi.org/10.1137/0403039. URL: https://doi.org/10.1137/0403039.

[2] Jamis Buck. *Mazes for programmers*. Pragmatic Bookshelf, 2015.

[3] Jamis Buck. *The buckblog*. URL: https://weblog.jamisbuck.org/.

[4] Fawzi Emad. *Object-Oriented Programming II*. URL: http://www.cs.umd.edu/class/spring2019/cmsc132-020X-040X/Project8/proj8.html.

[5] Martin Foltin. "Automated Maze Generation and Human Interaction". 2011. URL: https://theses.cz/id/k1d3n5/.

[6] Petros L. Ioannidis. "Procedural Maze Generation". 2016. URL: https://pergamos.lib.uoa.gr/uoa/dl/frontend/file/lib/default/data/1324569/theFile.

[7] *Maze generation algorithm*. URL: https://en.wikipedia.org/wiki/Maze_generation_algorithm.

[8] Roberto Tamassia e Michael H. Goldwasser Micheal T. Goodrich. *Data structures and algorithm analysis in Java*. John Wiley Sons Inc, 2014.

[9] Walter D. Pullen. *Think Labyrinth!* URL: http://www.astrolog.org/labyrnth.htm.

[10] *SFML Graphics module Documentation*. URL: https://www.sfml-dev.org/documentation/2.5.1/group__graphics.php.