Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea in Informatica

Analisi di algoritmi generativi di labirinti

Relatore: *Prof. Alberto Ottavio Leporati*

Relazione della prova finale di:
Luca Cortinovis
Matricola 869598

Anno Accademico 2024-2025

Ai miei genitori per avermi sostenuto per tutti questi anni.

Ad Alexandra, con cui ho un rapporto unico,
impossibile da esprimere a parole.

A Gabriele, con cui coltivo una lunga amicizia,
nonostante le nostre differenze.

Ai miei amici Matteo e Sara.

Infine, ringrazio il Professore Alberto Ottavio Leporati,
per essere stato un ottimo insegnante, e per la sua pazienza
e disponibilità durante lo sviluppo di questa tesi.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

La storia dei labirinti è millenaria, la sua origine si perde nel tempo, e nel corso della storia ha subito delle evoluzioni nel suo ruolo e significato. Nell'antica Grecia, per esempio, non erano pensati come rompicapi da risolvere, ma come generici luoghi o edifici intricati, dalla geometria complicata o oscura, e quasi sempre di tipo **unicursale**, ovvero con una sola entrata e un vicolo cieco alla fine del percorso. Il labirinto era spesso a spirale, o raramente di forma quadrata.

Dal punto di vista strutturale, invece, i labirinti moderni sono detti **multicursali**, ovvero con incroci, deviazioni e molti vicoli ciechi. Infatti oggi è uno dei simboli maggiormente riconosciuti di rompicampo, considerati quindi come giochi o sfide da risolvere.

In questa tesi studieremo come generare questi tipi di labirinti, poichè dal punto di vista dell'Informatica, in particolare di Algoritmi e di Teoria dei Grafi, sono formalmente definibili e sicuramente più interessanti da analizzare.

Nel primo capitolo verranno definiti alcuni concetti di Teoria dei Grafi e come possono essere usati per definire un labirinto.

Nel secondo capitolo verrà descritto un algoritmo di risoluzione dei labirinti con relativo pseudocodice, complessità computazionale e spaziale, e vedremo come la velocità di esecuzione cambia in base al tipo di algoritmo generativo usato.

Nel terzo capitolo verranno descritti una serie algoritmi di generazione con relativi pseudocodici, complessità computazionali e spaziali, e ne verranno discusse le peculiarità che li contraddistinguono gli uni dagli altri.

Nel quarto capitolo verranno presentate delle statistiche sugli algoritmi, per capire quali sono ideali in determinate situazioni o rispetto ad eventuali restrizioni.

Infine, verranno tratte le conclusioni sul lavoro svolto, discutendo anche possibili sviluppi e miglioramenti.

Tutte le implementazioni degli algoritmi sono state sviluppate tramite il linguaggio di programmazione Python versione 3.13 [10] tramite l'IDE PyCharm 2025.1 (Community Edition) [11]. Per poter visualizzare in formato grafico i labirinti è stata utilizzata la libreria Pillow [12].

Il progetto Python è liberamente consultabile alla seguente pagina:
https://github.com/LucerysLightbringer/ariadnes-thread.

Capitolo 1

Definizioni e terminologia

1.1 Teoria dei Grafi

Di seguito vengono riportate alcune definizioni di Teoria dei Grafi utili per poter definire in modo formale un labirinto.

Grafo
Un grafo è una struttura dotata di vertici collegati tra loro da archi. Un grafo G è quindi descrivibile come $G = (V, E)$, dove V è l'insieme dei vertici, ed E è l'insieme degli archi che collegano i vertici.

Grafo orientato e non orientato
Gli archi di un grafo possono essere orientati, rappresentati come una freccia, ovvero percorribili solamente in quel verso, oppure non orientati, rappresentati come una semplice linea, ovvero percorribili arbitrariamente lungo ogni verso della linea stessa.

Grafo connesso e non connesso
Un grafo connesso è un grafo per cui, dati due qualsiasi vertici, esiste sempre almeno un percorso tra loro. Un grafo non connesso è semplicemente un grafo composto da più sottografi non connessi tra loro. I sottografi prendono il nome di componenti connesse.

Percorso o cammino
Un percorso (o cammino) è una sequenza di archi che connettono dei vertici. In questa tesi i percorsi sono implicitamente definiti senza cicli, ovvero in cui ogni arco del percorso compare una sola volta.

Ciclo
Un ciclo è un percorso in cui il vertice iniziale e finale coincidono.

Albero
Un albero è un grafo connesso aciclico. Ogni nodo ha un solo padre e un numero variabile di figli. Il primo nodo dell'albero è chiamato radice, e non ha nodi padri, ma tutti gli altri nodi discendono da esso. I nodi senza figli, ovvero quelli terminali, sono chiamati foglie.

Albero n-ario
Un albero in cui ogni nodo non foglia ha al massimo n nodi figli.

1.2 Termini utili

Le seguenti sono definizioni utili per comprendere meglio la tesi.

Cella
La componente elementare di un labirinto. Secondo la rappresentazione dei labirinti proposta, equivale ad un vertice.

Corridoio o passaggio
Una serie di celle collegate. Secondo la rappresentazione dei labirinti proposta, equivale ad un percorso.

Griglia
Il labirinto stesso, implementato come una griglia, ovvero una matrice di celle RxC dove R è il numero di righe e C il numero di colonne della griglia. Secondo la rappresentazione dei labirinti proposta, equivale al grafo stesso.

In particolare, un labirinto perfetto può essere interpretato come un albero, ovvero un grafo connesso aciclico. Nel nostro caso, un labirinto a griglia è considerabile come un albero ternario.

Vicolo cieco
Una cella connessa solamente ad una sola altra cella. Secondo la rappresentazione dei labirinti proposta, equivale ad una foglia dell'albero.

Cella root o ingresso
L'ingresso del labirinto. Secondo la rappresentazione dei labirinti proposta, equi-

vale alla radice dell'albero.

Cella obiettivo o uscita
L'uscita del labirinto. Secondo la rappresentazione dei labirinti proposta, equivale ad un qualsiasi nodo non radice dell'albero.

1.3 Tipologie di labirinti

I labirinti possono essere suddivisi secondo la seguente serie di criteri:

- Dimensione
- Topologia
- Tassellazione
- Percorso
- Texture
- Focus

Un labirinto può quindi essere costruito utilizzando una serie di combinazioni dei criteri appena detti.

1.3.1 Dimensione

La dimensione [5] del labirinto indica in quante dimensioni geometriche il labirinto è definito. Ovvero, per ogni sua componente elementare quanta informazione è richiesta per descriverlo.

I labirinti più semplici possibili sono 2D, per cui basta salvare le coordinate 2D di ogni cella.

Ovviamente esistono labirinti 3D, che sono una serie di labirinti 2D sovrapposti in livelli uno sopra l'altro, e viene quindi aggiunto il concetto di pavimento/soffitto per ogni cella. Il passaggio tra i livelli viene effettuato quando ad esempio si completa il precedente. Potrebbero essere facilmente implementati come un array di labirinti 2D, con ogni labirinto avente un indicatore per il passaggio tra livelli superiori o inferiori. Si possono anche pensare labirinti di dimensioni maggiori, dove i livelli sono a loro volta labirinti di dimensione $n - 1$, e il passaggio a dimensioni

superiori viene effettuato ancora una volta con delle flag.

Esistono infine dei particolari tipi di labirinti, detti weave, traducibili come intrecciati o a treccia. Sono labirinti 2.5D, in cui i vari corridoi del labirinto possono intrecciarsi tra di loro. Un esempio potrebbe essere un labirinto con corridoi-ponti che connettono parti più alte del labirinto, che se visti dall'alto come una mappa 2D, apparirebbero come corridoi che si sovrappongono ad altri.

1.3.2 Topologia

La topologia [5] descrive la geometria dello spazio in cui il labirinto esiste. Può essere di tipo:

- Normale: un labirinto definito in un classico spazio euclideo.
- Planair: un labirinto che non appartiene ad uno spazio euclideo. Ad esempio labirinti che si sviluppano sulla superficie di una sfera, di un toro o su una striscia di Möbius, come visto in Figura 1.1.

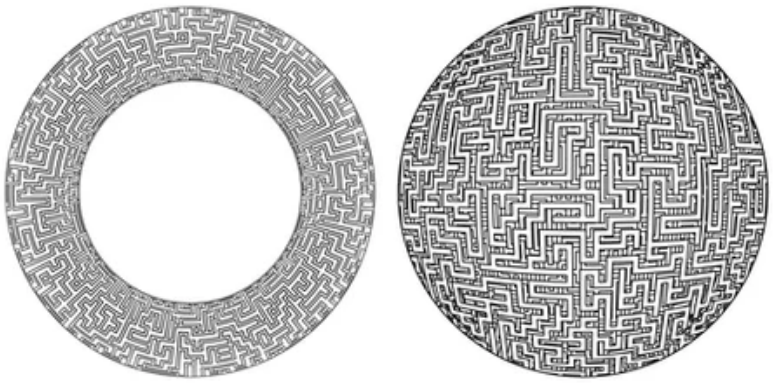

Figura 1.1: Esempi di labirinti sviluppati rispettivamente su un toro e su una sfera. Immagine ottenuta dal sito [13].

1.3.3 Tassellazione

La tassellazione [5] è la geometria delle singole celle. Può essere di tipo:

- Ortogonale: una cella quadrata. Il labirinto che ne deriva è rettangolare, come una griglia, dove ogni cella si connette a 2 (per i vertici), 3 (per le celle lungo i lati) oppure 4 altre celle.

- Non-ortogonale: una cella triangolare, esagonale o in generale non rettangolare. Il labirinto che ne deriva è quindi formato da celle che possono avere fino ad altre N celle adiacenti, dove N è il numero di lati del poligono.

- Crack: un labirinto in cui le celle non hanno una tassellazione specifica, ma ha corridoi che si sviluppano su angoli qualsiasi, come visto in Figura 1.2.

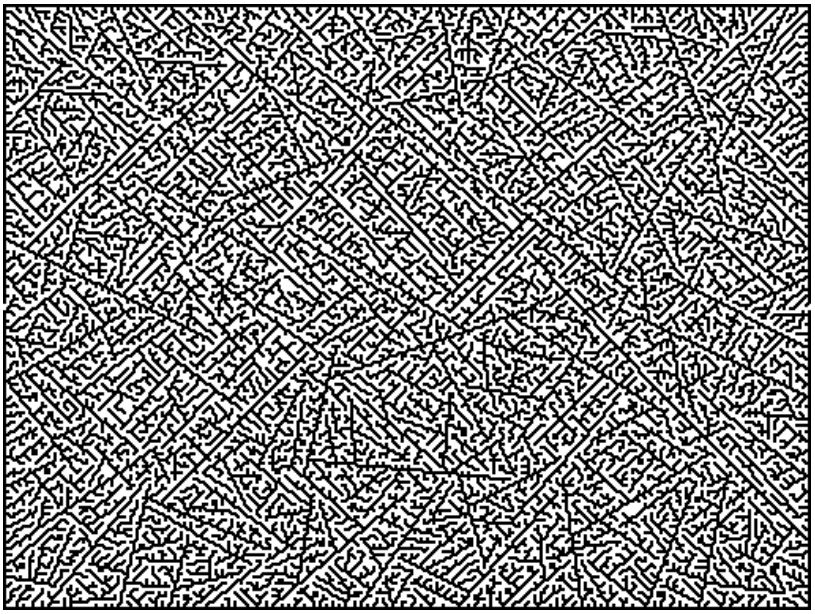

Figura 1.2: Esempio di labirinto a tassellazione crack. Immagine ottenuta dal sito [14].

- Frattale: un labirinto composto da labirinti più piccoli. Si può cioè suddividere in celle di grandezza e tassellazione arbitraria, in cui ogni cella è a sua volta un labirinto. Si può vedere un esempio nella Figura 1.3.

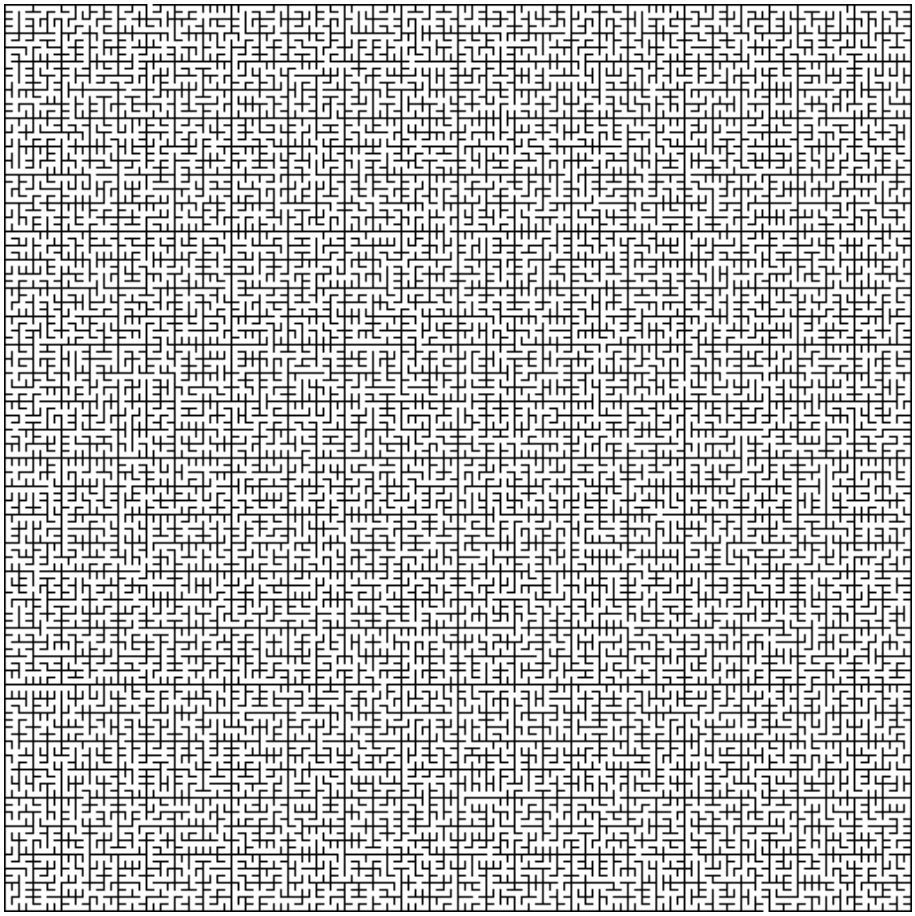

Figura 1.3: Esempio di labirinto a tassellazione frattale. Immagine ottenuta dal sito [15].

1.3.4 Percorso

Il percorso [5] è l'insieme delle caratteristiche che definiscono i passaggi del labirinto. Può essere di tipo:

- Perfetto: un labirinto in cui non ci sono cicli e non esistono celle inaccessibili. Cioè da e per ogni cella esiste un unico percorso che le collega, ovvero il labirinto ha solamente una soluzione.
- Treccia o intrecciato: un labirinto in cui compaiono cicli e i passaggi possono intrecciarsi tra loro, come si vede in Figura 1.4.

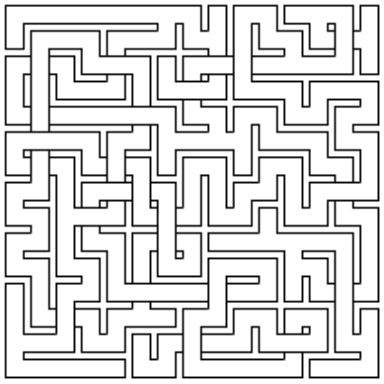

Figura 1.4: Esempio di labirinto intrecciato. Immagine ottenuta dal sito [16].

- Unicursale: un labirinto senza incroci, ovvero esiste un unico percorso, che attraversa tutto il labirinto. Un esempio è dato dalla Figura 1.5.

Figura 1.5: Esempio di labirinto unicursale. Immagine ottenuta dal sito [17].

- Multicursale: un labirinto con incroci e vicoli ciechi, per cui possono esistere anche più percorsi che risolvono il labirinto.
- Sparso: un labirinto in cui alcune celle rimangono isolate, ovvero senza passaggi che le connettono ad altre celle. Quindi esistono aree inaccessibili del labirinto.

1.3.5 Texture

La texture [5] si riferisce allo stile del labirinto. Non è una caratteristica facilmente definibile, e infatti definisce molteplici fattori che definiscono un labirinto, alcuni esempi possono essere:

- Bias: la texture bias definisce un labirinto che tende ad avere un bias verso passaggi orizzontali piuttosto che verticali, o viceversa.
- Run: la texture run indica quanto sono lunghi i passaggi descritti dal bias. Si parla di texture run lunga o corta a seconda del fatto che i passaggi siano più lunghi o corti di n celle, per un valore specifico di n.
- Elite: la texture elite definisce la lunghezza della soluzione rispetto alla grandezza del labirinto. Cioè un labirinto elitario ha una soluzione relativamente corta, mentre un labirinto non elitario (o poco elitario) ha una soluzione che ricopre un'area del labirinto significativa.
- Uniformità: la texture uniformità indica se un labirinto sembra essere stato generato da un algoritmo uniforme. Certi algoritmi infatti, non possono affatto generare alcuni tipi di labirinti, mentre altri possono generare qualsiasi tipo di labirinto in modo statisticamente uniforme.

1.3.6 Focus

Il focus [5] non è tecnicamente una caratteristica del labirinto, quanto una caratteristica che definisce il modo in cui l'algoritmo che lo genera funziona. Gli algoritmi di generazione possono essere suddivisi in due tipi:

- Intagliatori: 'scavano' il labirinto creando i passaggi.
- Costruttori: innalzano i muri attorno alle celle del labirinto.

1.4 Labirinti perfetti

La tesi tratta solamente di labirinti 2D ortogonali perfetti. Ovvero labirinti 2D, formati da celle quadrate, che rispettano le seguenti proprietà [2]:

- il labirinto non ha cicli;
- il labirinto non ha celle isolate;
- tra una qualsiasi coppia di celle del labirinto esiste esattamente un percorso che le collega.

È stato scelto di limitarsi solamente a questo tipo di labirinti perché lo scopo della tesi è concentrarsi sugli algoritmi generativi piuttosto che sugli algoritmi risolutivi, ed anche per semplicità implementativa.

Capitolo 2

Algoritmo risolutivo A*

L'algoritmo risolutivo A* è in grado di trovare il percorso minimo tra due vertici all'interno di un grafo. È un algoritmo informato, cioè utilizza altri dati oltre al solo labirinto per risolverlo, infatti richiede che il labirinto sia definibile come un grafo pesato [6]. È considerabile come un ibrido tra l'algoritmo di Dijkstra e un algoritmo Best-First (infatti utilizza delle euristiche sui pesi) [7]. Il peso degli archi in un labirinto ortogonale perfetto è sempre 1.

L'algoritmo si basa sull'utilizzo di tre funzioni di valutazione della distanza:

- $\boldsymbol{F(n) = g(n) + h(n)}$: funzione che assegna ad ogni cella n del labirinto un valore, selezionando la prossima cella da esplorare come quella con il valore minore $F(n)$;
- $\boldsymbol{g(n)}$: funzione sempre nota che assegna il costo esatto dalla cella iniziale alla cella attuale n, viene calcolata mano a mano che il labirinto viene esplorato;
- $\boldsymbol{h(n)}$: euristica del costo (stimato) dalla cella attuale n alla cella obiettivo. Le euristiche più utilizzate per l'algoritmo sono la distanza euclidea (usata se si ammettono distanze anche diagonali) e la distanza di Manhattan (usata se si ammettono solamente distanze verticali ed orizzontali). Proprio per questo motivo noi utilizzeremo come euristica la distanza di Manhattan, definita come $\boldsymbol{|x_1 - x_2| + |y_1 - y_2|}$.

L'algoritmo garantisce di trovare il percorso più corto solamente se l'euristica è **ammissibile**, ovvero non sovrastima mai le distanze effettive ed è **consistente**, ovvero se la stima del percorso dal vertice iniziale al vertice obiettivo è sempre minore o uguale alla distanza stimata da qualsiasi vertice adiacente all'obiettivo, più il costo per raggiungere tale adiacente.

Considerando che il labirinto ortogonale perfetto è un grafo in una geometria euclidea definita in solamente due direzioni e non ammette distanze diagonali, e in cui le distanze sono unitarie, allora la distanza di Manhattan sarà sempre ammissibile, perché non può mai sovrastimare il costo effettivo del percorso.

Ogni volta che il costo di una cella viene calcolato, viene memorizzata la cella precedente usata per il calcolo di quella attuale. Per ricostruire il percorso, è sufficiente fare il backtracking di tutte le celle predecessori salvate fino ad arrivare alla cella root [8].

I passi dell'algoritmo sono i seguenti:

1. Inizializza una coda di priorità, che ordina le celle da esplorare. L'ordine è definito dalla n-upla $(f\_score, tie\_breaker, cella)$, e garantisce che la cella con f_score più basso abbia sempre la priorità più alta.
2. Inizializza un contatore $tie\_breaker$ per evitare conflitti durante l'estrazione dalla coda di priorità. Cioè a parità di f_score, verrà estratta la cella inserita prima nella coda, ovvero con valore di $tie\_breaker$ minore.
3. Inizializza tre dizionari, g_score per memorizzare il costo dalla cella iniziale ad una cella n, f_score per calcolare il costo totale (stimato) e infine $temp\_path$ per ricostruire il percorso, memorizzando per ogni cella la sua cella genitore.
4. Per la cella iniziale imposta $g(n) = 0$, e calcola $h(n)$ e $F(n)$. Per tutte le altre celle, imposta $g(n) = \infty$, $F(n) = \infty$.
5. Ripeti finché la coda delle celle da esplorare non è vuota:
   (a) Ottieni dalla coda delle celle da esplorare la cella con il valore di $F(n)$ minore. Questa cella sarà la cella attuale e in questo momento viene considerata esplorata.
   (b) Se la cella attuale è la cella obiettivo, il percorso è stato trovato e viene ricostruito facendo il backtracking dalla cella tramite il dizionario $temp\_path$.
   (c) Se la cella attuale non è la cella obiettivo, allora per ogni cella adiacente alla cella attuale:
      i. Calcola il valore $g(n)$ della cella, ovvero il costo per raggiungere la cella adiacente considerata passando dalla cella attuale. Siccome i costi degli spostamenti nei nostri tipi di labirinti sono sempre unitari, ci si riduce a calcolare $g\_score[cella\_attuale] + 1$.

ii. Confronta il costo appena calcolato con il costo precedente, ovvero quello di *g_score*[*cella_adiacente*]. Se il nuovo percorso è migliore, cioè ha valore *g_score* minore, allora abbiamo trovato un percorso più efficiente per raggiungere la cella adiacente considerata.

iii. Quindi aggiorniamo i seguenti valori della cella adiacente: la sua cella genitore, impostandola come la cella attuale, e i costi *g_score* e *f_score*.

iv. Infine aggiungiamo alla coda delle celle da esplorare la cella adiacente con i nuovi valori *f_score* e *tie_breaker* ed aumentiamo di uno il valore di *tie_breaker* per la prossima cella.

Pseudocodice

```
# Inizializza una coda di priorità (min-heap)
# La tupla inserita è (f_score, tie_breaker, cell)
visitate = queue_min_heap[]

# Inizializza un contatore tie breaker
# per risolvere parità dei valori F(n)
tie_breaker = 0

# Inizializza dei dizionari per salvare i valori g(n), F(n)
# e per fare backtracking del percorso
temp_path = dict[]
g_score = dict[]
f_score = dict[]

# Inizializza i costi iniziali per tutte le celle ad infinito
for each cell in maze
    g_score[cell] = inf
    f_score[cell] = inf

# Inizializza i costi cella root
g_score[cell_root] = 0
f_score[cell_root] = manhattan_dist(cell_root, cell_goal)


# Aggiungi la cella root alla coda di priorità
# La cella root avrà valore tie_breaker = 0
visitate.append(f_score[cell_root], tie_breaker, cell_root)
```

```
# Visita ogni cella del labirinto
while visitate is not empty

  # Estrai dalla coda di priorità la cella con
  # valore F(n) minore, rendendola la cella attuale
  current_f, _, cella_attuale = visitate.pop(cell_root)

  # Se la cella corrente è la cella goal
  # ricostruisco il percorso
  if (cella_attuale == cella_goal)
    solution_path = []
    temp = cell_goal

    while temp in temp_path
      solution_path.append(temp)
      temp = temp_path[temp]

    solution_path.append(cell_root)
    return solution_path.reverse() # inverti percorso


# Osserva tutte le celle adiacenti alla cella attuale
for cella_adiacente in cella_attuale.get_adiacenti()

  # Calcola il costo provvisorio per raggiungere
  # la cella adiacente
  current_g = g_score[cella_attuale] + 1

  # Se il percorso trovato è minore di
  # quello precedente, aggiorna i valori
  if (current_g < g_score[cella_adiacente])

    temp_path[cella_adiacente] = cella_attuale
    g_score[cella_adiacente] = current_g
    f_score[cella_adiacente] =
      g_score[cella_adiacente] +
      manhattan_dist(cella_adiacente, cella_goal)
```

```
        # Aggiungi la cella adiacente alla coda di priorità
        visitate.append(
          f_score[cella_adiacente],
          tie_breaker,
          cella_adiacente)

        # Incrementa per evitare conflitti
        # nella coda di priorità
        tie_breaker += 1

# Se non troviamo la cella goal, il percorso è vuoto
return []
```

Complessità computazionale: L'algoritmo A* nel caso peggiore si comporta come l'algoritmo di Dijkstra, cioè ha una complessità in tempo pari a $O(|E| + V * log|V|)$, dove V rappresenta l'insieme dei nodi del grafo e E l'insieme degli archi. In realtà abbiamo osservato che l'algoritmo A* applicato ai labirinti presi in considerazione utilizza un euristica ammissibile e consistente, e viene implementato con una coda di priorità, utile per trovare la cella con valore $F(n)$ minore, quindi la complessità computazionale è in realtà $O(E_{visitati} + V_{visitati} * log(V_{visitati}))$, dove $E_{visitati}$ indica l'insieme degli archi effettivamente visitati e $V_{visitati}$ indica l'insieme dei vertici effettivamente visitati.

Efficienza spaziale: L'algoritmo richiede tre dizionari di dimensione $O(K)$ e una coda di priorità di dimensione $O(K)$, dove K nel caso peggiore, rappresenta un numero che si avvicina a N (il massimo di celle del labirinto). Pertanto lo spazio richiesto tende a $O(3N + N) = O(4N) = O(N)$.

Capitolo 3

Algoritmi generativi

3.1 Algoritmo Binary Tree

È il più semplice tra gli algoritmi per la generazione di labirinti perfetti. Richiede ad ogni passo di decidere casualmente tra due opzioni, basate sul bias principale dell'algoritmo. Infatti tecnicamente, esistono quattro versioni di questo algoritmo, in cui il bias consiste in quale direzione cancellare i muri:

- nord-est;
- nord-ovest;
- sud-est;
- sud-ovest.

In questa tesi abbiamo scelto in modo arbitrario il bias nord-est. La scelta del bias in questo algoritmo è particolarmente importante, perché crea sempre un labirinto con una texture particolare, ovvero con i corridoi in direzione del bias generati senza interruzioni (in questo caso i corridoi nord ed est).

I passi dell'algoritmo sono i seguenti [1]: Per ogni cella del labirinto, decidi casualmente se:

- continuare verso la cella adiacente verso est (se esiste);
- continuare verso la cella adiacente verso nord (se esiste).

Partendo da una cella qualsiasi, l'algoritmo viene applicato per ogni cella del labirinto (applicando ad esempio l'algoritmo ad ogni cella in ordine, da [0][0] a

$[max-1][max-1]$), e se mi trovo in una cella lungo il bordo del labirinto, ovvero non ho celle adiacenti verso est oppure verso nord, mi sposto verso l'unica cella disponibile.

Un esempio di esecuzione dell'algoritmo (con bias nord-est) applicato riga per riga può essere visto nella Figura 3.1.

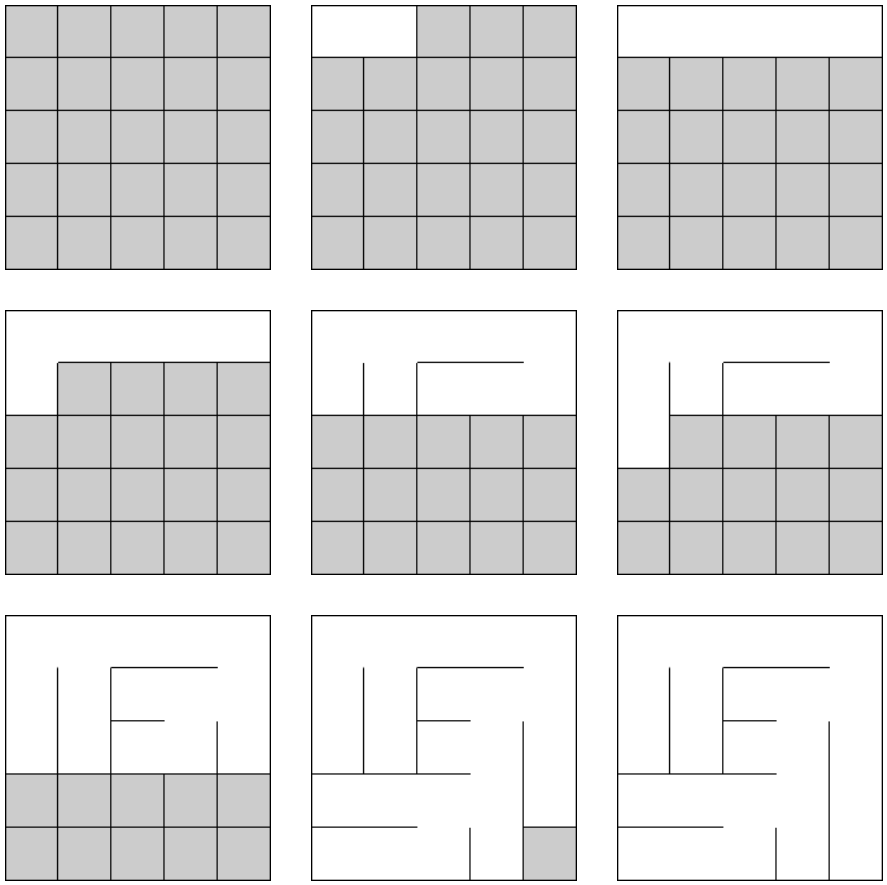

Figura 3.1: Esempio di esecuzione dell'algoritmo Binary Tree con bias nord-est. Immagine ottenuta dal sito [3].

Caratteristiche

- Algoritmo di tipo intagliatore;
- non possono esistere vicoli ciechi nelle direzioni del bias;

- si possono creare facilmente labirinti di grandi dimensioni senza preoccuparsi del tempo di esecuzione;
- non può creare alcuni tipi di labirinti. Ad esempio, limitandosi ai labirinti 2x2, non potrà mai creare la metà di essi (nel nostro caso, i labirinti C,D, mostrati in Figura 3.2).

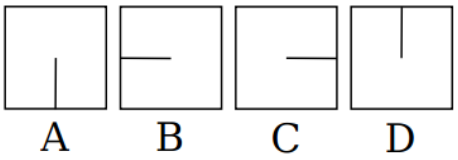

Figura 3.2: Esempio di labirinti 2x2. Immagine ottenuta dal libro [1].

Pseudocodice

```
for each row in maze
  for each cell in row

    direzioni = []

    # Scelta casuale tra nord oppure est
    if (cell.nord exists)
      direzioni.append(cell.nord)

    if (cell.est exists)
      direzioni.append(cell.est)

    # Collega la cella attuale alla cella scelta
      if (direzioni is not empty)
        temp = random(direzioni)
        cell.link(temp)
```

Complessità computazionale: $O(N)$, dove N rappresenta il numero totale di celle del labirinto. L'algoritmo itera su ogni cella una sola volta ed effettua operazioni in tempo costante.

Efficienza spaziale: $O(1)$. Ha bisogno solamente di memorizzare una cella in ogni passo dell'algoritmo, perciò non richiede ulteriori strutture di supporto oltre al labirinto stesso.

3.2 Algoritmo Sidewinder

Un algoritmo anch'esso semplice, ed evoluzione dell'algoritmo Binary Tree con una leggera miglioria, che permette di rimuovere il bias su uno dei corridoi.

I passi dell'algoritmo sono i seguenti [1]: Per ogni cella del labirinto,

1. Memorizzo un gruppo di celle, inizialmente inizializzato vuoto.
2. Aggiungo la cella attuale al gruppo di celle.
3. Dalla cella attuale, decidi casualmente se tagliare verso la cella adiacente nord oppure verso la cella adiacente est.
4. Se dopo il taglio è stato creato un passaggio, rendi la cella appena collegata la nuova cella attuale e ripeti i passi 2-4.
5. Se dopo il taglio, non è stato creato un nuovo passaggio, scegli una cella casuale tra quelle del gruppo e taglia verso nord. Svuota il gruppo, scegli come prossima cella la cella successiva della riga attuale e ripeti i passi 2-5.
6. Se ci troviamo lungo i bordi, semplicemente tagliamo verso nord e poi ripartiamo dalla riga successiva.

Rispetto all'algoritmo Binary Tree, l'algoritmo Sidewinder rimuove il bias su uno dei due corridoi. Come per l'algoritmo Binary Tree, è stato scelto il bias nord-est e in questo caso l'algoritmo Sidewinder genera solamente il corridoio nord totalmente ininterrotto, come visto in Figura 3.4.

Caratteristiche

- Algoritmo di tipo intagliatore;
- non possono esistere vicoli ciechi in direzione nord;
- si possono creare facilmente labirinti di grandi dimensioni senza preoccuparsi del tempo di esecuzione;
- non può creare alcuni tipi di labirinti. Ad esempio limitandosi ai labirinti 2x2, potrà crearne solo il 75% (il labirinto D mostrato in Figura 3.3, infatti, non può essere generato).

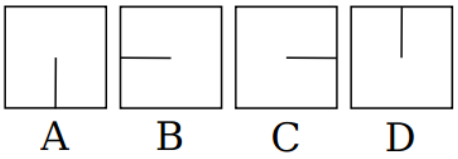

Figura 3.3: Esempio di labirinti 2x2. Immagine ottenuta da [1].

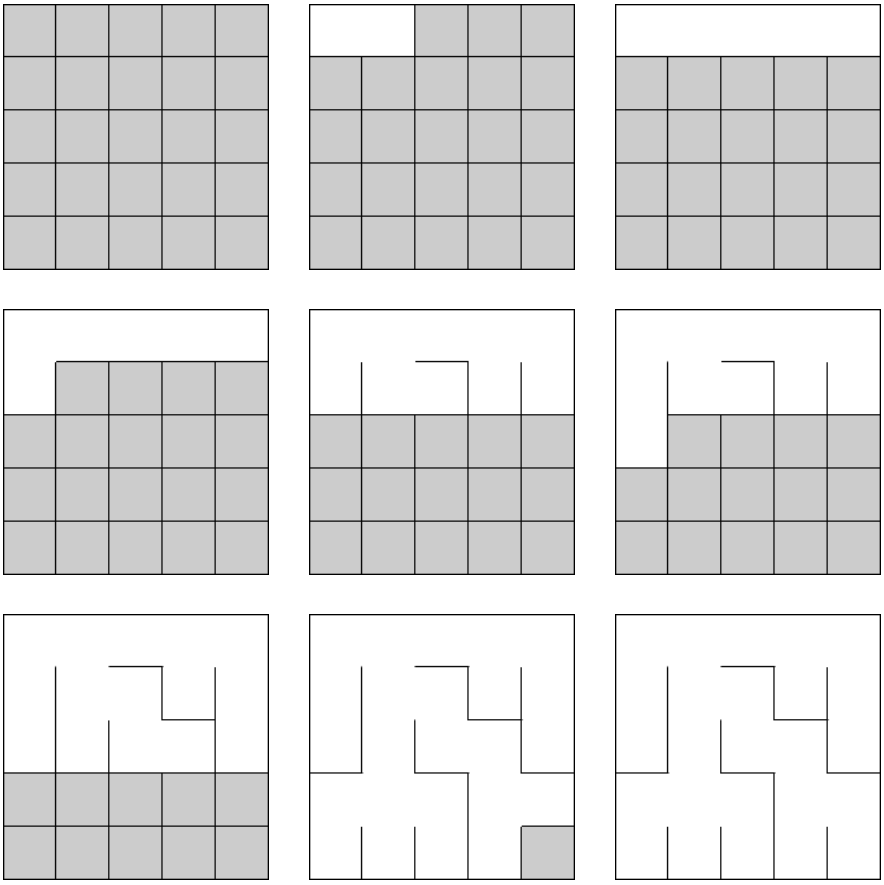

Figura 3.4: Esempio di esecuzione dell'algoritmo Sidewinder con bias nord-est. Immagine ottenuta dal sito [3].

Pseudocodice

```
for each row in maze

  gruppo_celle = []

  for each cell in row

    # Aggiungi la cella attuale al gruppo
    gruppo_celle.append(cell)

    # Verifica di essere sui bordi nord o est
    bord_nord = cell.nord is None
    bordo_est = cell.est is None

    # Se ti trovi sul bordo est, chiudi il gruppo, oppure
    # se non ti trovi sul bordo nord, e decidi casualmente
    # di chiudere il gruppo (random(0,1) == 0)
    gruppo_chiuso = bordo_est OR
                    (NOT bordo_nord AND
                    random.randint(0,1) == 0)


    # Se chiudi il gruppo, seleziona la nuova cella attuale
    # e taglia verso nord
    if (gruppo_chiuso == TRUE)

      nuova_cella = random.choice(gruppo_celle)

      if (nuova_cella.nord exists)
        nuova_cella.link(nuova_cella.nord)

      gruppo_celle.clear()

    # Altrimenti taglia ad est
    else
      cell.link(cell.est)
```

Complessità computazionale: $O(N)$, dove N rappresenta il numero totale di celle del labirinto. L'algoritmo itera su ogni cella una sola volta, ed effettua operazioni in tempo costante. La grandezza del set *gruppo_celle* viene resettata ad ogni

nuova visita, quindi il costo complessivo rimane lineare rispetto alla grandezza del labirinto.

Efficienza spaziale: $O(K)$, dove K rappresenta la grandezza dell'array *gruppo_celle*, che può al massimo contenere ogni cella della riga, ovvero il numero di colonne della griglia.

3.3 Algoritmo di Aldous-Broder

Si tratta di un algoritmo che, a differenza di Binary Tree e Sidewinder, è di tipo **unbiased**, ovvero non ha favoritismi di scelta strutturale durante la generazione. Questo perché l'idea alla base è la **random walk**, ovvero muoversi in modo casuale finché non si trova una cella non visitata. Ovviamente questa caratteristica permette di generare labirinti senza alcun tipo di bias, ma aumenta di molto i tempi di esecuzione.

I passi dell'algoritmo sono i seguenti:

1. Scegli una cella casuale del labirinto come cella iniziale.
2. Muoviti in una direzione casuale, rendendo la cella appena visitata la nuova cella attuale.
3. Se la nuova cella attuale non è già stata visitata, la collego alla cella precedente, creando un passaggio.
4. Ripeti gli step 1-3 finché tutte le celle non sono visitate.

Caratteristiche

- Algoritmo di tipo intagliatore;
- l'algoritmo è in grado di generare qualsiasi tipo di labirinto perfetto. Nel senso che ogni labirinto ha la stessa probabilità di essere generato. Si dice quindi che l'algoritmo di Aldous-Broder è un algoritmo generativo uniforme [4] [1];
- l'algoritmo è fortemente sconsigliato per la creazione di labirinti di grandi dimensioni, in quanto potrebbe anche non terminare mai proprio perché sceglie casualmente la direzione in cui muoversi.

Come si vede nella Figura 3.5, nessuna logica viene seguita, ma appunto, vengono fatte scelte casuali. Le celle di colore grigio sono le celle non ancora esplorate, le celle di colore verde sono l'attuale cella in esplorazione, mentre le celle di colore bianco sono le celle già esplorate.

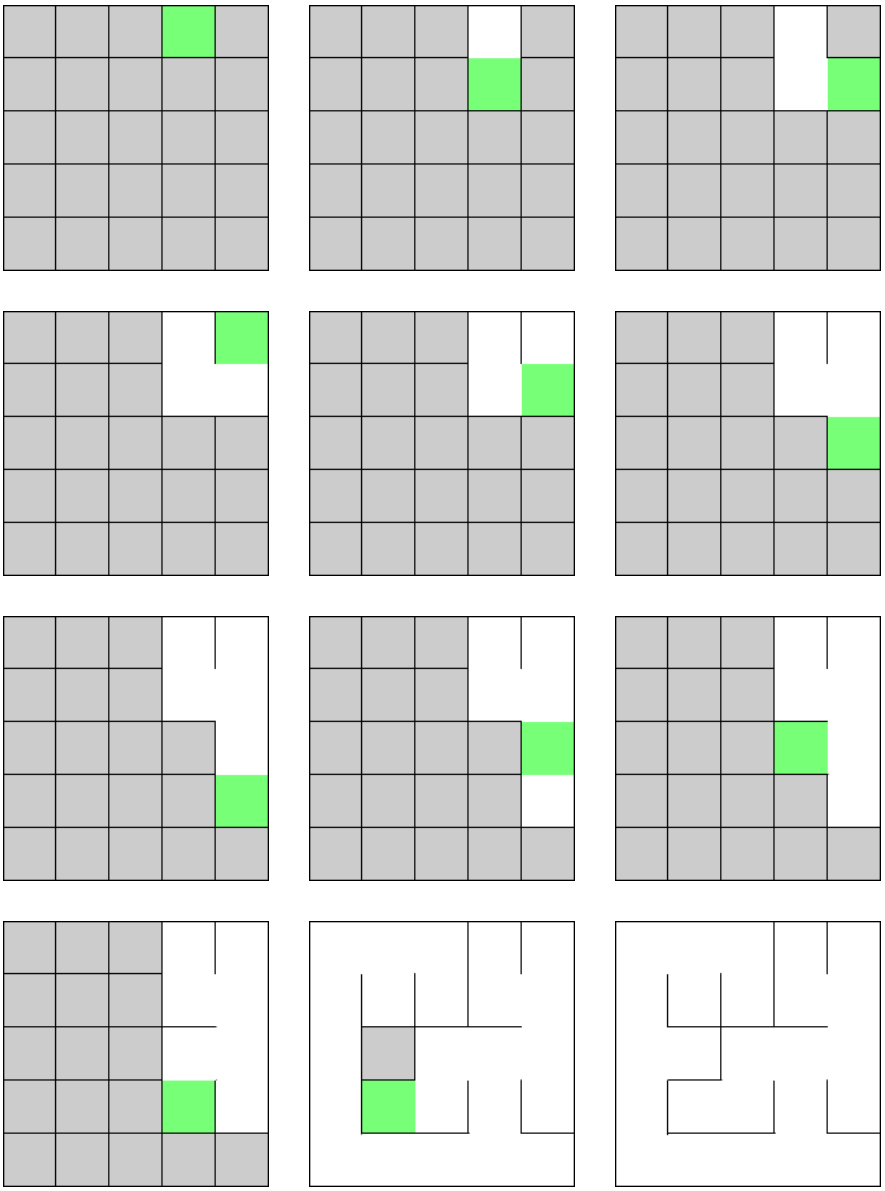

Figura 3.5: Esempio di esecuzione dell'algoritmo di Aldous-Broder. Immagine ottenuta dal sito [3].

Pseudocodice

```
# Seleziona una cella casuale
cella_attuale = maze.random_cell()

# Inizializza una lista con tutte le celle della griglia
non_visitate = [maze.each_cell]

# La cella iniziale viene già considerata visitata
non_visitate.remove(cella_attuale)

# Itera per ogni cella del labirinto
while non_visitate is not empty

  # Scegli casualmente una cella adiacente a quella attuale
  cella_adiacente = random(cella_attuale.get_adiacenti())

  # Se la cella adiacente non è già stata visitata
  # viene collegata a quella precedente
  if (cella_adiacente in non_visitate)
    cella_attuale.link(cella_adiacente)
    non_visitate.remove(cella_adiacente)

  # La cella adiacente diventa la nuova cella attuale
  cella_attuale = cella_adiacente
```

Complessità computazionale: non è facilmente quantificabile. Nel caso peggiore, l'algoritmo visita spesso celle già visitate, infatti potrebbe anche non terminare mai, a causa della natura casuale delle random walk. Si può dire che in media è significativamente più lenta di $O(N)$, dove N è il numero di celle del labirinto.

Efficienza spaziale: $O(K)$, dove K rappresenta la grandezza del set *non_visitate*, che all'inizio contiene tutte le celle del labirinto, ma man mano che l'algoritmo procede, la dimensione diminuisce. Lo spazio richiesto è comunque proporzionale alla grandezza del labirinto.

3.4 Algoritmo Recursive Backtracker

È un algoritmo di generazione anch'esso **unbiased**, infatti come l'algoritmo di Aldous-Broder, anch'esso si basa sull'idea di **random walk**, ma con una restrizione, ovvero se si incontra un vicolo cieco si fa il backtracking, finché non si trova una cella adiacente non visitata. È in grado di generare ogni possibile labirinto perfetto. Per questo motivo i labirinti generati dall'algoritmo sembrano più "convincenti" a noi esseri umani rispetto ai labirinti generati da Binary Tree o da Sidewinder.

I passi dell'algoritmo sono i seguenti:

1. Inizializza uno stack per tenere traccia delle celle visitate, chiamato *visitate*.
2. Si sceglie casualmente una cella iniziale e la si inserisce nello stack *visitate*.
3. Iterando sullo stack *visitate*, ripeti i seguenti passi finché ci sono celle nel percorso per cui fare backtracking:
   (a) Esplora le celle adiacenti alla cella attuale non ancora visitate, inserendole in un altro stack, chiamato *adiacenti*.
   (b) Iterando sullo stack *adiacenti*, se ci sono celle adiacenti non ancora visitate, allora:
      i. scegli casualmente una di queste celle;
      ii. crea un passaggio tra la cella attuale e la nuova cella scelta;
      iii. inserisci la cella collegata nello stack *visitate*.
   (c) Se non ci sono celle adiacenti non ancora visitate, rimuovi la cella attuale dalla cima dello stack *visitate*. Questo passo è la parte di backtracking dell'algoritmo.

Caratteristiche

- Algoritmo di tipo intagliatore;
- crea pochi vicoli ciechi, risultando in labirinti con percorsi molto lunghi;
- si possono creare facilmente labirinti di grandi dimensioni senza preoccuparsi del tempo di esecuzione, ma richiede più memoria a causa dello stack;
- tende a generare labirinti con corridoi tortuosi, questo perché l'algoritmo esplora il più possibile prima di fare il backtracking.

La Figura 3.6 permette di vedere in azione l'algoritmo, e in particolare come inizia il backtracking se si incontra un vicolo cieco. Le celle di colore grigio sono celle non ancora esplorate, le celle di colore rosso sono le celle esplorate, mentre le celle di colore bianco sono le celle esplorate per cui è già stato svolto il backtracking.

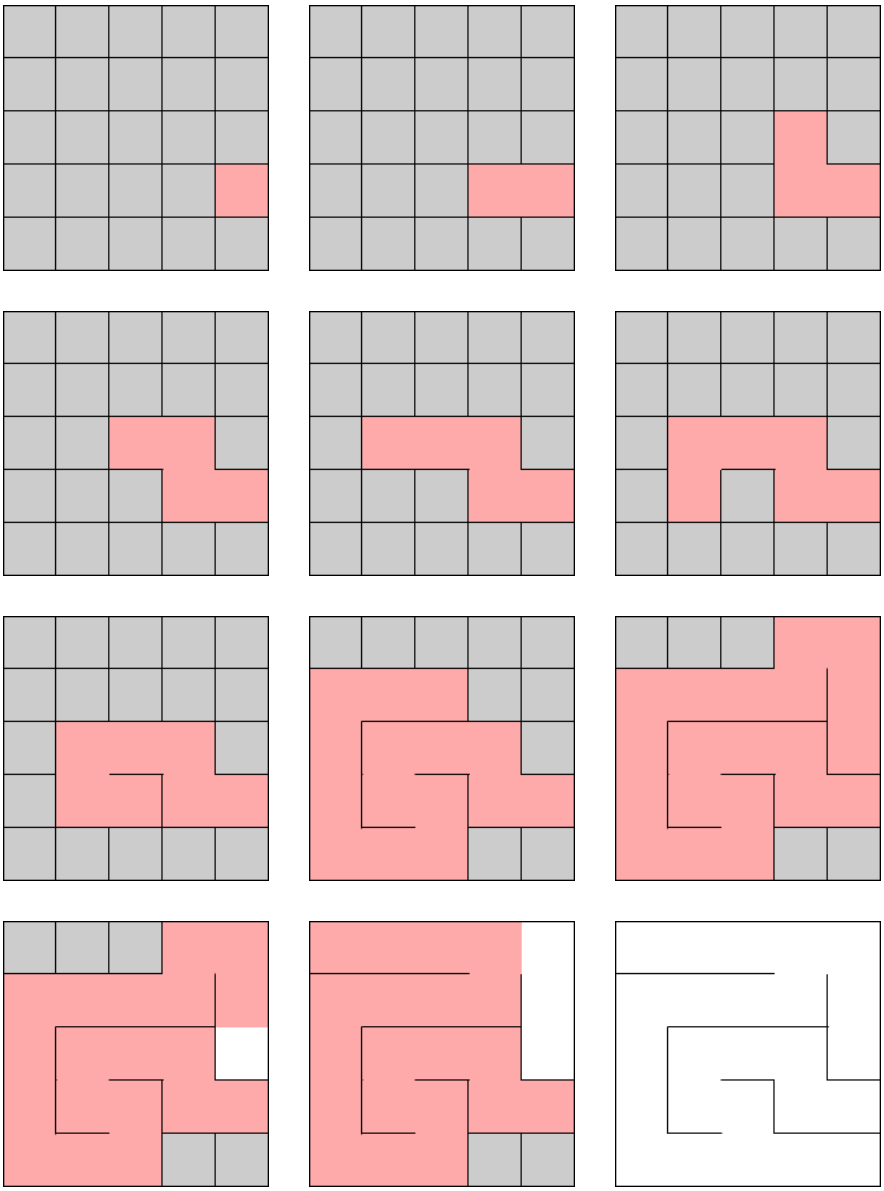

Figura 3.6: Esempio di esecuzione dell'algoritmo Recursive Backtracker. Immagine ottenuta dal sito [3].

Pseudocodice

```
# Inizializza lo stack con una cella iniziale casuale
visitate = []
visitate.push(maze.random_cell())

# Itera per ogni cella del labirinto
while visitate is not empty

    # Ottieni la cella attuale (in cima allo stack)
    cella_attuale = visitate.top()

    # Ottieni le celle adiacenti non collegate
    # alla cella attuale
    non_visitate = []
    adiacenti [] = cella_attuale.get_adiacenti()
    for each temp in adiacenti
        if (temp.is_linked(cella_attuale) == FALSE)
            non_visitate.append(temp)

    # Se non ci sono celle adiacenti non collegate
    # rimuovi la cella attuale (in cima allo stack)
    if (non_visitate is empty)
        visitate.pop()

    # Scegli una cella casuale tra quelle adiacenti,
    # la colleghi a quella attuale,
    # rendendola la nuova cella attuale
    else
        temp = random(non_visitate)
        cella_attuale.link(temp)
        visitate.push(temp)
```

Complessità computazionale: $O(N)$, dove N rappresenta il numero totale di celle del labirinto. L'algoritmo itera su ogni cella (e quelle ad essa collegate) una sola volta, mentre effettua operazioni in tempo costante (al massimo 4 celle adiacenti) all'interno del ciclo.

Efficienza spaziale: $O(K)$, dove K rappresenta la grandezza dello stack *visitate*. La grandezza dello stack nel caso peggiore sarà grande quanto il numero di celle del labirinto.

3.5 Algoritmo Recursive Division

È un algoritmo peculiare rispetto a tutti quelli precedentemente analizzati. È un algoritmo di tipo costruttore, ovvero anziché intagliare i passaggi, costruisce dei muri tra ogni cella e poi scava i passaggi attraverso i muri. Inizialmente costruisce una griglia in cui tutte le celle sono collegate, ovvero non esistono muri, quindi divide il labirinto a metà, crea il muro e un singolo passaggio, poi considera una delle sotto-metà create e così via fino a che non si può più dividere.

La Figura 3.7 permette di vedere chiaramente che ogni regione del labirinto inizialmente vuoto viene divisa a metà.

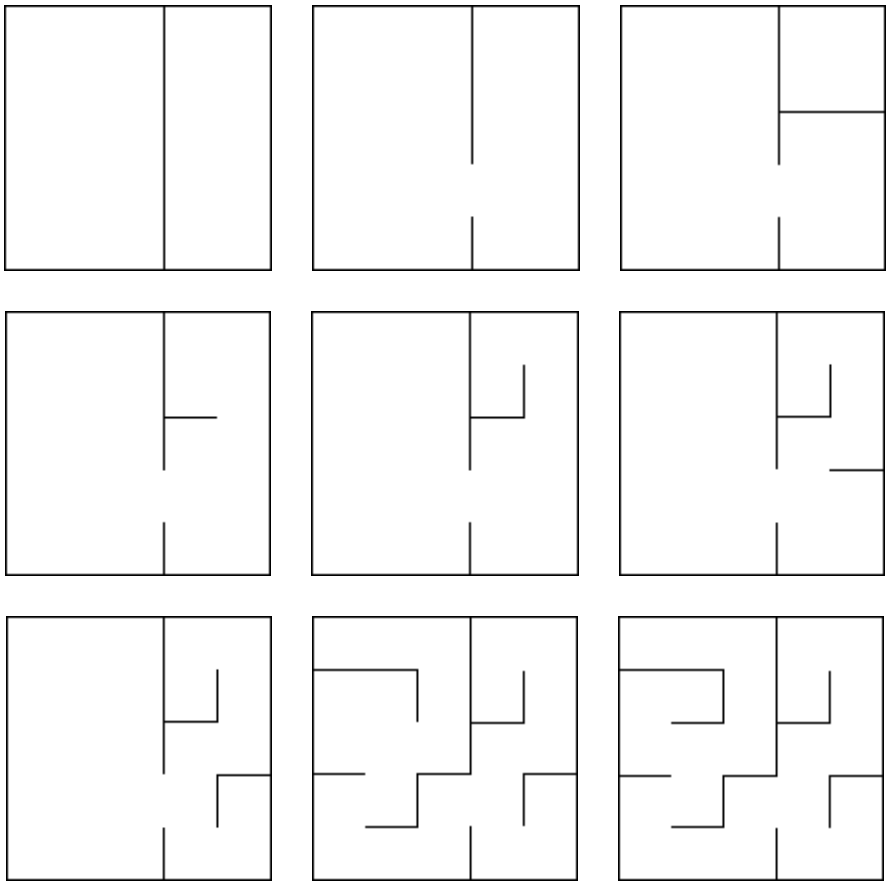

Figura 3.7: Esempio di esecuzione dell'algoritmo Recursive Division. Immagine ottenuta dal sito [3].

Caratteristiche

- Algoritmo di tipo costruttore;
- crea aree all'interno del labirinto simili a stanze, ovvero delle regioni a forma rettangolare con una sola entrata.
- algoritmo con tassellazione frattale, ovvero il labirinto generato può essere considerato come una serie di labirinti più piccoli (con le stesse caratteristiche del labirinto principale) connessi tra loro.

Pseudocodice

```
# Collega ogni cella con le sue adiacenti
# creando un labirinto completamente aperto
for each cell in maze.each_cell()
  for each adiacente in cell.get_adiacenti()
    cell.link(adiacente)

# Inizia divisione ricorsiva
divide(maze, 0, 0, maze.rows, maze.columns)
```

```
# Funzione ricorsiva generica
divide(maze, row, col, height, width)

  # Impossibile dividere ulteriormente
  if (height <= 1 OR width <= 1)
    return

  # Dividi orizzontalmente o verticalmente
  if (height >= width)
    divide_horizontal(maze, row, col, height, width)
  else
    divide_vertical(maze, row, col, height, width)
```

```
# Funzione ricorsiva per la divisione orizzontale
divide_horizontal(maze, row, col, height, width)

  muro_sud = random(0, height) # scegli una riga casuale
  passaggio = random(0, width) # scegli una colonna casuale
```

```
  # Itera su ogni cella lungo il muro
  for each cell in width

    # Ignora la cella su cui tagli il passaggio
    if (passaggio == cell)
      continue

    # Costruisci il muro rimuovendo i collegamenti
    # tra celle
    cell = maze[row+muro_sud, col+passaggio]
    if (cell exists AND cell.sud exists)
      cell.unlink(cell.sud)

    # Ricorsione sulle due nuove regioni create
    divide(maze, row, col, muro_sud, width)
    divide(maze, row+muro_sud, col, height-muro_sud, width)
```

```
# Funzione ricorsiva per la divisione verticale
divide_vertical(maze, row, col, height, width)

  muro_est = random(0, width) # scegli una colonna casuale
  passaggio = random(0, height) # scegli una riga casuale

  # Itera su ogni cella lungo il muro
  for each cell in height

    # Ignora la cella su cui tagli il passaggio
    if (passaggio == cell)
      continue

    # Costruisci il muro rimuovendo i collegamenti
    # tra celle
    cell = maze[row+passaggio, col+muro_est]
    if (cell exists AND cell.est exists)
      cell.unlink(cell.est)

    # Ricorsione sulle due nuove regioni create
    divide(maze, row, col, height, muro_est)
    divide(maze, row, col+muro_est, height, width-muro_est)
```

Complessità computazionale: inizialmente l'algoritmo genera una griglia completamente aperta, ovvero dove tutte le celle sono collegate, tramite un ciclo di $O(4N) = O(N)$ passi (perché ogni cella ha in media 4 celle adiacenti). La parte divide et impera dell'algoritmo è svolta dalla funzione *divide* e dalle sue versioni *divide_horizontal* e *divide_vertical*. Ogni versione effettua una iterazione sulla regione attuale, che diventa sempre più piccola, e poi viene effettuata una operazione a tempo costante $O(1)$ (scollegare una cella per poter creare il passaggio). La somma delle operazioni svolte considerando ogni singola chiamata ricorsiva è proporzionale alla grandezza del labirinto, per cui la funzione *divide* può essere considerata $O(N)$. Quindi le due parti dell'algoritmo impiegano: $O(N)+O(N) = O(N)$ passi.

Efficienza spaziale: $O(1)$. Infatti l'algoritmo non utilizza mai altre strutture dati oltre al labirinto stesso. Se invece consideriamo anche la memoria richiesta per lo stack delle chiamate ricorsive, allora la profondità massima di ricorsione è $\log(R)+\log(C)$, dove R è il numero di righe della griglia e C è il numero di colonne. Questo perché ad ogni passo ricorsivo le dimensioni si dimezzano. Pertanto si può dire che la complessità spaziale vale $O(\log(K))$, dove K è il numero delle celle del labirinto.

3.6 Texture dei labirinti

Per finire il capitolo, mostriamo una serie di immagini dei labirinti generati, di grandezza 50x50 celle, per osservare la differenza di generazione di ogni labirinto.

Il colore dei labirinti parte da celle bianche fino ad arrivare a celle verde scuro, ovvero le più distanti dalla cella iniziale. In particolare la cella di coordinate [0][0], intesa come ingresso del labirinto, viene colorata di giallo, mentre la cella colorata di azzurro viene intesa come obiettivo ed è sempre la cella più distante dalla cella iniziale. Viene anche mostrato il percorso per raggiungere la cella obiettivo.

Come vediamo dalla Figura 3.8, i labirinti generati dall'algoritmo Binary Tree tendono ad avere una texture 'diagonale'. Si possono inoltre notare i due corridoi ininterrotti, in questo caso il corridoio nord e il corridoio est, tipici dell'algoritmo. Come vediamo, il percorso risolutivo è piuttosto semplice.

Come si vede in Figura 3.9, l'algoritmo Sidewinder genera labirinti con una texture 'verticale'. Anche per questo algoritmo si nota facilmente che il corridoio nord è ininterrotto, inoltre ogni vicolo cieco è rivolto verso la direzione opposta

del bias, in questo caso verso sud. Il percorso risolutivo è anche in questo caso particolarmente facile da trovare.

Come vediamo in Figura 3.10, i labirinti generati dall'algoritmo di Aldous-Broder hanno una texture caotica, difficilmente definibile. Il percorso risolutivo è più complicato rispetto ai percorsi risolutivi di Binary Tree e Sidewinder.

La Figura 3.11 mostra che i labirinti generati dall'algoritmo Recursive Backtracker sono anch'essi molto caotici e con percorsi risolutivi particolarmente intricati, dove la lunghezza del percorso risolutivo copre una grande area del labirinto.

Dalla Figura 3.12, invece, si vede chiaramente che i labirinti generati dall'algoritmo Recursive Division possono essere interpretati come sotto-labirinti connessi tra loro.

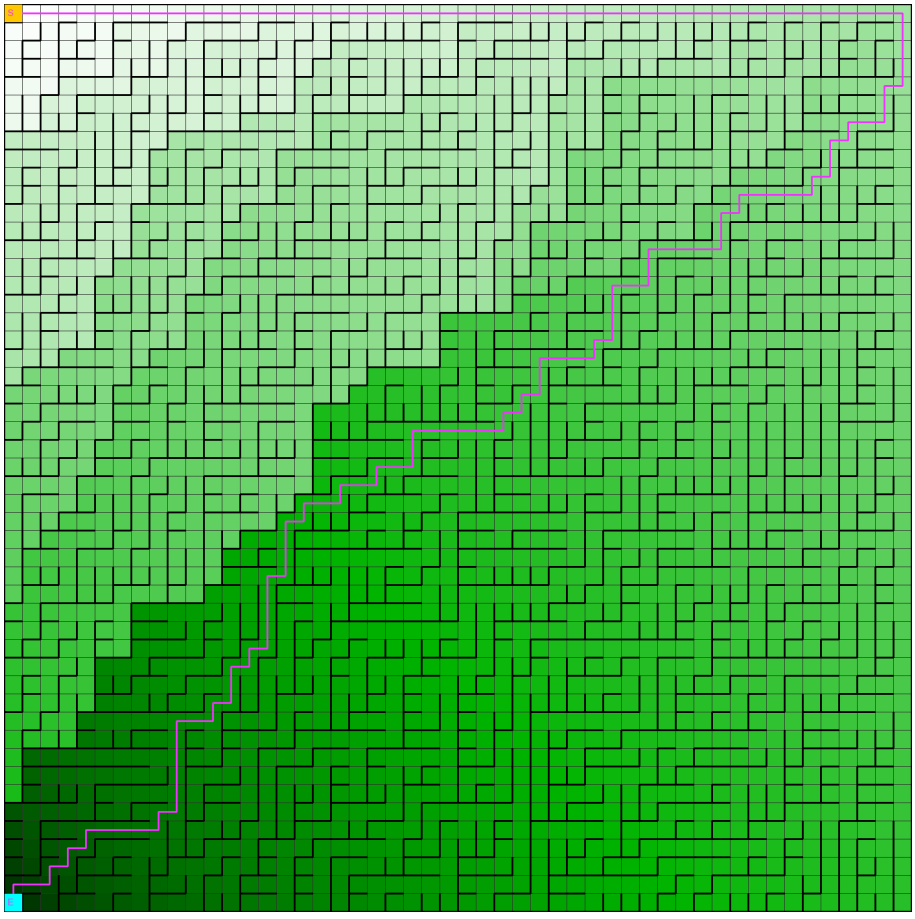

Figura 3.8: Esempio di labirinto di tipo Binary Tree di grandezza 50x50 celle. Immagine generata tramite implementazione Python con la libreria Pillow [12].

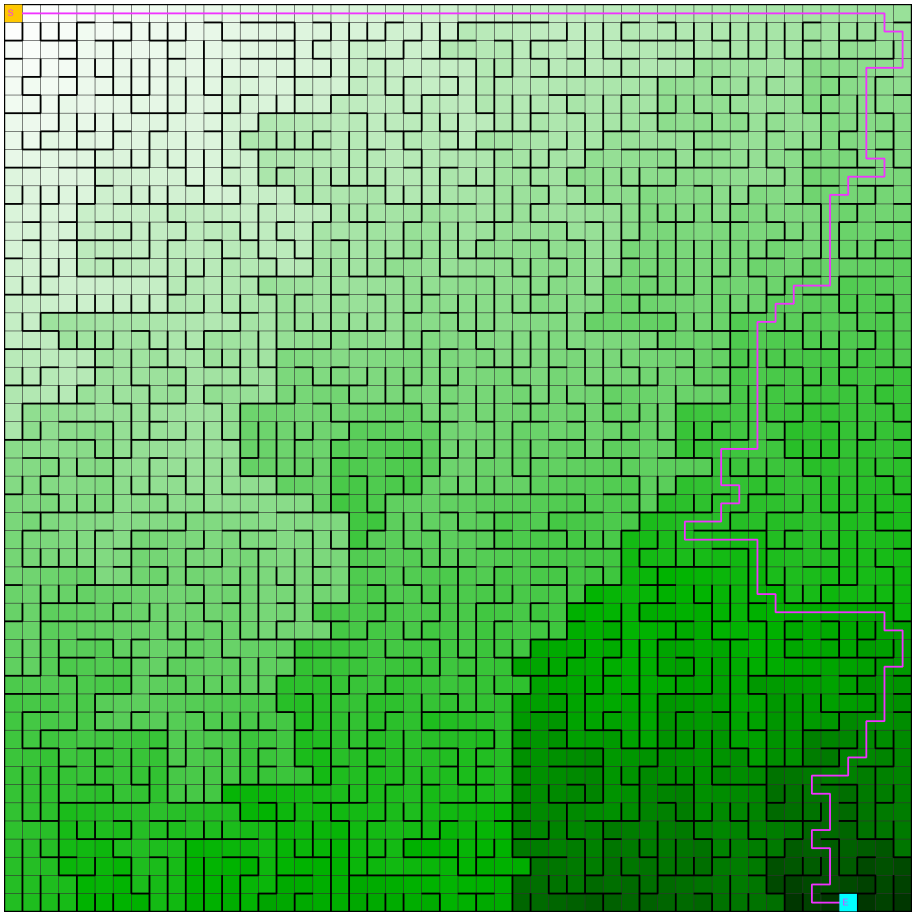

Figura 3.9: Esempio di labirinto di tipo Sidewinder di grandezza 50x50 celle. Immagine generata tramite implementazione Python con la libreria Pillow [12].

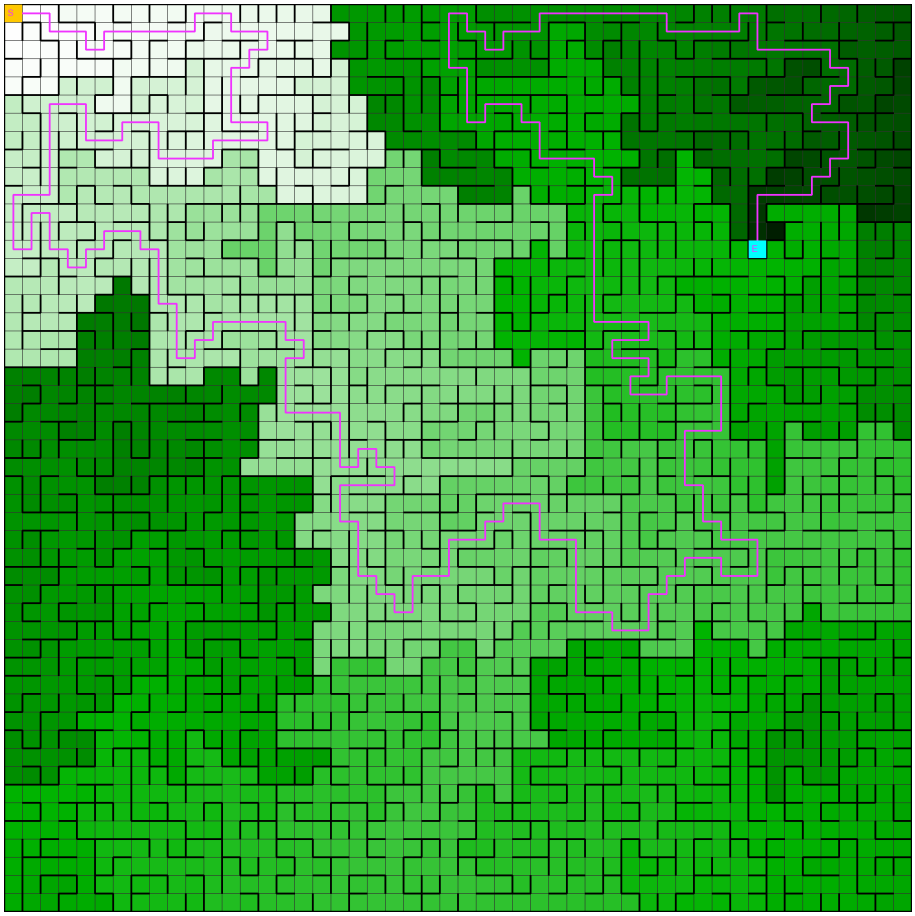

Figura 3.10: Esempio di labirinto di tipo Aldous-Broder di grandezza 50x50 celle. Immagine generata tramite implementazione Python con la libreria Pillow [12].

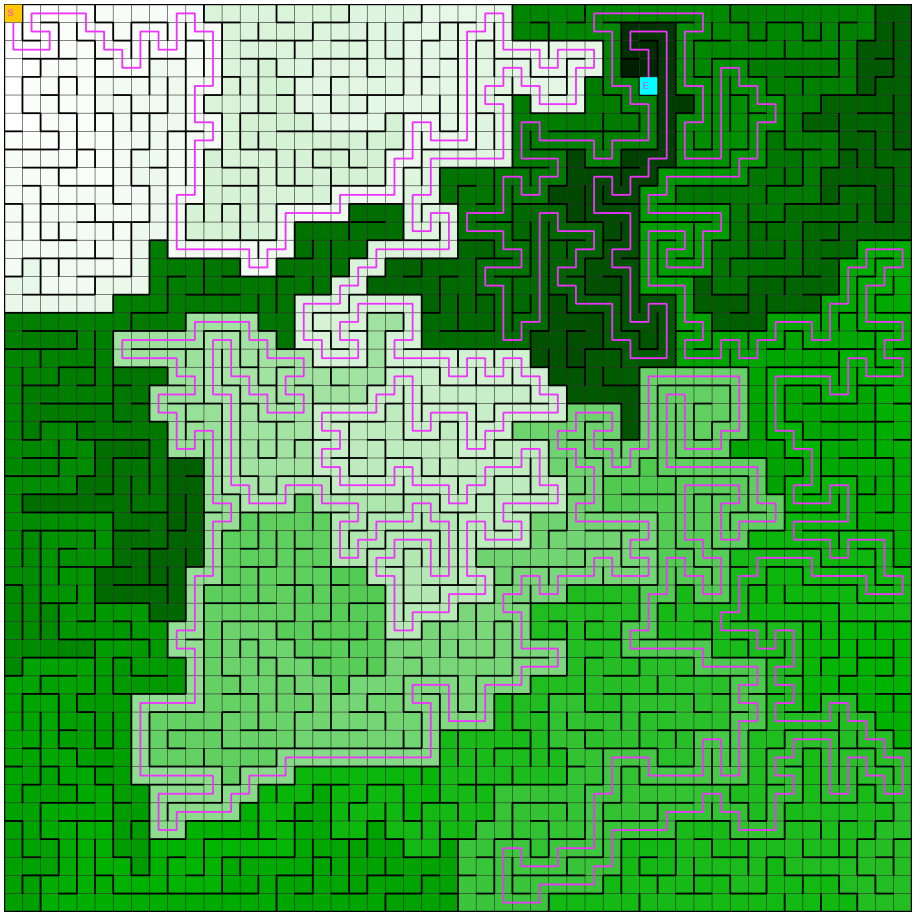

Figura 3.11: Esempio di labirinto di tipo Recursive Backtracker di grandezza 50x50 celle. Immagine generata tramite implementazione Python con la libreria Pillow [12].

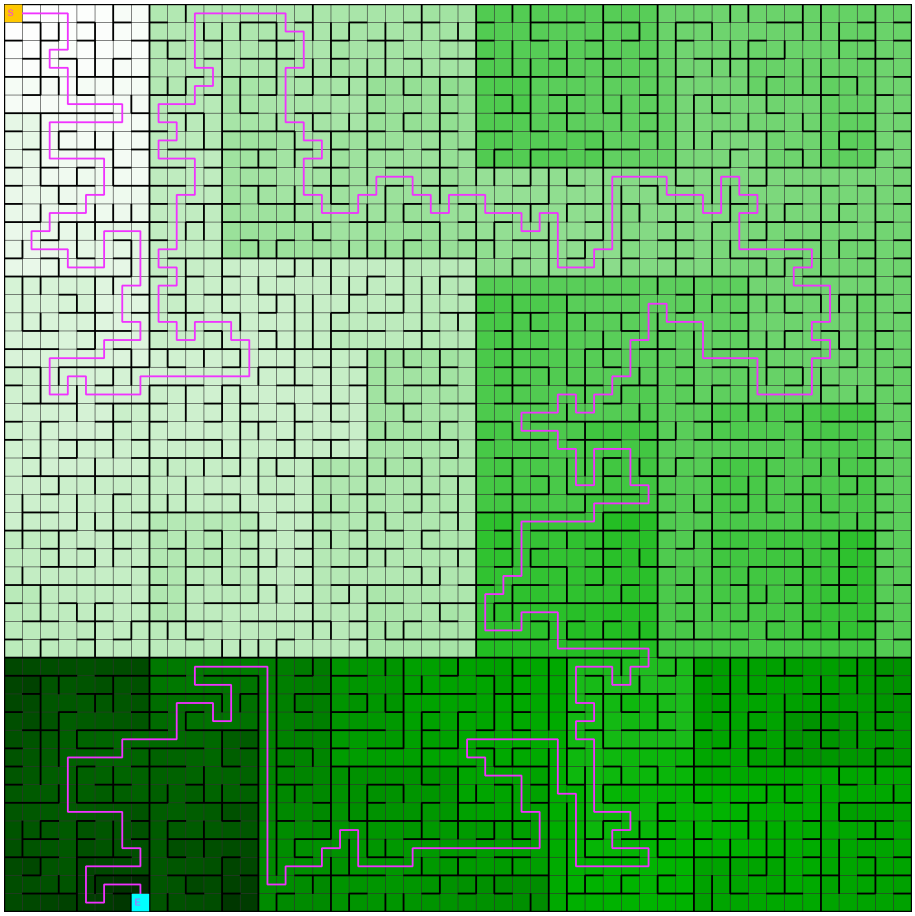

Figura 3.12: Esempio di labirinto di tipo Recursive Division di grandezza 50x50 celle. Immagine generata tramite implementazione Python con la libreria Pillow [12].

Si può concludere che gli algoritmi che generano i labirinti a prima vista più 'realistici' sono l'algoritmo di Aldous-Broder e l'algoritmo Recursive Backtracker. Infatti notiamo che celle di colore verde scuro possono essere vicino a celle di un colore decisamente più chiaro. Questa caratteristica indica il fatto che il percorso che collega quelle parti del labirinti è lungo e tortuoso.

La stessa caratteristica si nota molto di meno nei labirinti generati dagli algoritmi Binary Tree e Sidewinder, in cui la texture mostra che il cambiamento di colore delle celle sembra più graduale.

Caso particolare è invece l'algoritmo Recursive Division, che per la sua peculiarità sembra generare un gradiente più 'sfumato' e diviso in zone facilmente identificabili rispetto agli altri labirinti.

Capitolo 4

Statistiche sugli algoritmi

Ogni algoritmo mostrato ha una serie di caratteristiche: facilità di implementazione, velocità d'esecuzione, bias più o meno evidenti, ecc... Sono state quindi stilate delle statistiche per capire se ci sono algoritmi generativi che, nonostante le loro limitazioni, possono essere considerati oggettivamente migliori in un contesto general purpose, ovvero in un contesto in cui serve generare un labirinto perfetto che sia il più realistico possibile.

Le statistiche sono state sviluppate tramite una serie di script in Python, e comprendono:

- tempo medio di esecuzione degli algoritmi generativi;
- tempo medio di esecuzione dell'algoritmo risolutivo per ogni algoritmo generativo;
- numero di vicoli ciechi;
- lunghezza del percorso più lungo all'interno del labirinto rispetto alla grandezza del labirinto;
- lunghezza della soluzione rispetto alla grandezza del labirinto.

Ogni caratteristica esaminata per gli algoritmi generativi è stata ottenuta analizzando un campione di 1000 tentativi per labirinti 100x100 e 200x200. Per i labirinti di grandezza 500x500 è stato scelto di limitarsi a 500 tentativi per questioni di tempo, questo dovuto soprattutto all'algoritmo di Aldous-Broder, piuttosto lento.

L'algoritmo risolutivo A* richiede che la cella iniziale e la cella obiettivo siano esplicitamente dichiarate. Pertanto, ogni singola iterazione specifica in modo casuale le due celle richieste.

4.1 Tempo medio di esecuzione per algoritmi generativi

Di seguito vengono riportati i tempi medi di esecuzione di ogni algoritmo generativo analizzato. Si può quindi immediatamente osservare come ogni algoritmo si comporta all'aumentare della dimensione del labirinto.

Confrontando la Tabella 4.2 con la Tabella 4.1 si può notare un incremento dei tempi di esecuzione di circa il 400% per gli algoritmi Binary Tree, Sidewinder, Recursive Backtracker e Recursive Division, mentre circa del 500% per l'algoritmo di Aldous-Broder.

Confrontando invece la Tabella 4.3 con la Tabella 4.2 si nota un incremento dei tempi di esecuzione di circa l'800% per gli algoritmi Binary Tree, Sidewinder, Recursive Backtracker e Aldous-Broder, mentre di circa il 900% per l'algoritmo Recursive Division.

Dalla Figura 4.1 si può notare che l'algoritmo di Aldous-Broder è di gran lunga il più lento, a causa della sua natura casuale, mentre il più veloce è Binary Tree, perché richiede praticamente solo una visita riga per riga della matrice del labirinto. Non considerando però algoritmi con bias pesanti, come Binary Tree e Sidewinder, l'algoritmo con tempo migliore risulta essere Recursive Backtracker.

Algoritmo	**Tempo medio (ms) - (s)**
Binary Tree	11,459(ms) - 0,01(s)
Sidewinder	17,244(ms) - 0,02(s)
Aldous-Broder	312,766(ms) - 0,31(s)
Recursive Backtracker	27,841(ms) - 0,03(s)
Recursive Division	32,791(ms) - 0,03(s)

Tabella 4.1: Tempo medio di esecuzione per gli algoritmi generativi su labirinti 100x100 su 1000 tentativi.

Algoritmo	Tempo medio (ms) - (s)
Binary Tree	45,244(ms) - 0,04(s)
Sidewinder	70,163(ms) - 0,07(s)
Aldous-Broder	1613,524(ms) - 1,64(s)
Recursive Backtracker	111,538(ms) - 0,11(s)
Recursive Division	124,225(ms) - 0,12(s)

Tabella 4.2: Tempo medio di esecuzione per gli algoritmi generativi su labirinti 200x200 su 1000 tentativi.

Algoritmo	Tempo medio (ms) - (s)
Binary Tree	390,139(ms) - 0,39(s)
Sidewinder	575,401(ms) - 0,57(s)
Aldous-Broder	13896,16(ms) - 13,89(s)
Recursive Backtracker	897,326(ms) - 0,89(s)
Recursive Division	1135,662(ms) - 1,13(s)

Tabella 4.3: Tempo medio di esecuzione per gli algoritmi generativi su labirinti 500x500 su 500 tentativi.

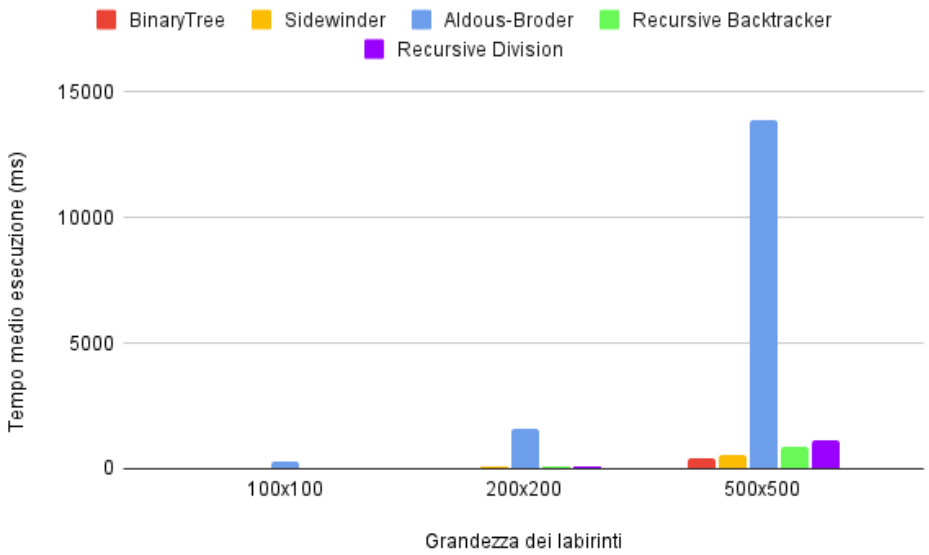

Figura 4.1: Tempo medio di esecuzione per gli algoritmi generativi.

4.2 Tempo medio di esecuzione dell'algoritmo risolutivo

Di seguito vengono riportati i tempi medi di risoluzione, ottenuti applicando l'algoritmo A* sui labirinti generati utilizzando i diversi algoritmi generativi presentati.

Confrontando la Tabella 4.5 con la Tabella 4.4 si può notare un incremento dei tempi di risoluzione fino a 4 volte tanto.

Confrontando la Tabella 4.6 con la Tabella 4.5 si può notare un incremento dei tempi di risoluzione dalle 9 alle 10 volte tanto. Solamente con labirinti di grandi dimensioni, ovvero da 500x500 celle in poi, i tempi di esecuzione iniziano ad avvicinarsi alla soglia di 1 secondo.

Consultando la Figura 4.2 si nota che l'algoritmo A* è considerevolmente più veloce (circa il 30% più veloce) se applicato a labirinti costruiti con Binary Tree piuttosto che labirinti ottenuti con Recursive Division e Recursive Backtracker. Questo è dovuto al fatto che i labirinti generati dagli algoritmi Recursive Backtracker e Recursive Division tendono a creare passaggi tortuosi, che richiedono più tempo per essere risolti. Si può dire quindi che i labirinti generati da algoritmi veloci e semplici sono più velocemente risolvibili.

Algoritmo risolutivo	**Algoritmo generativo**	**Tempo medio (ms) - (s)**
A*	Binary Tree	16,079(ms) - 0,016(s)
	Sidewinder	16,661(ms) - 0,017(s)
	Aldous-Broder	19,051(ms) - 0,019(s)
	Recursive Backtracker	22,177(ms) - 0,022(s)
	Recursive Division	19,581(ms) - 0,019(s)

Tabella 4.4: Tempo medio di esecuzione dell'algoritmo A* per ogni algoritmo generativo su labirinti 100x100 su 1000 tentativi.

Algoritmo risolutivo	**Algoritmo generativo**	**Tempo medio (ms) - (s)**
A*	Binary Tree	66,279(ms) - 0,066(s)
	Sidewinder	71,769(ms) - 0,072(s)
	Aldous-Broder	83,054(ms) - 0,083(s)
	Recursive Backtracker	92,303(ms) - 0,092(s)
	Recursive Division	85,212(ms) - 0,085(s)

Tabella 4.5: Tempo medio di esecuzione dell'algoritmo A* per ogni algoritmo generativo su labirinti 200x200 su 1000 tentativi.

Algoritmo risolutivo	Algoritmo generativo	Tempo medio (ms) - (s)
A*	Binary Tree	611,018(ms) - 0,61(s)
	Sidewinder	681,918(ms) - 0,68(s)
	Aldous-Broder	817,152(ms) - 0,82(s)
	Recursive Backtracker	891,516(ms) - 0,89(s)
	Recursive Division	919,096(ms) - 0,92(s)

Tabella 4.6: Tempo medio di esecuzione dell'algoritmo A* per ogni algoritmo generativo su labirinti 500x500 su 500 tentativi.

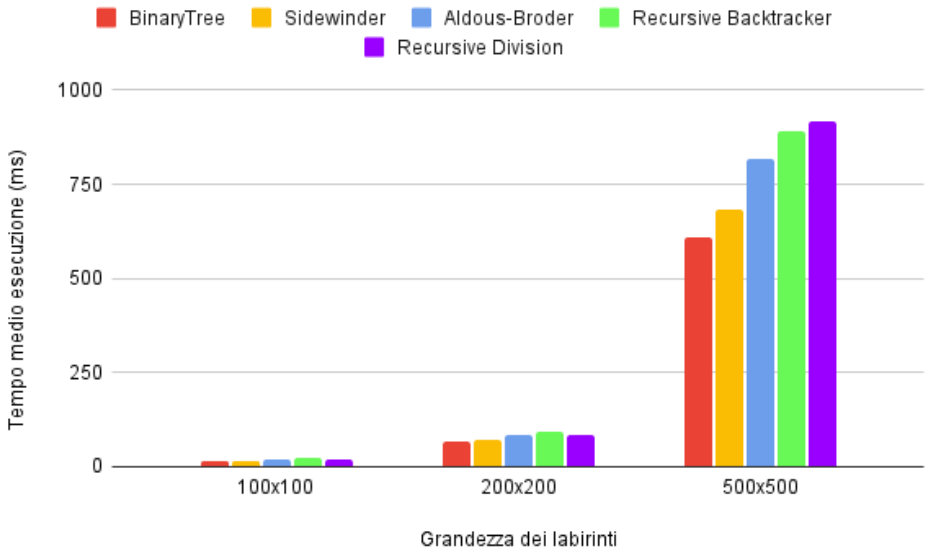

Figura 4.2: Tempo medio di esecuzione dell'algoritmo A* per ogni algoritmo generativo.

4.3 Numero di vicoli ciechi

Di seguito vengono riportate le percentuali di celle definite come vicoli ciechi di ogni algoritmo generativo analizzato. Una cella è considerata un vicolo cieco se è connessa solamente ad una sola altra cella. L'algoritmo (triviale) per ottenere il numero di celle è disponibile all'appendice A.

Le tabelle 4.7, 4.8 e 4.9 riportano tutte la stessa percentuale, con solamente una variazione che va dallo 0.08% allo 0.28% tra la Tabella 4.7 e la Tabella 4.8, mentre abbiamo una variazione che va dallo 0.08% allo 0.18% tra la Tabella 4.8 e la Tabella 4.9.

Per questo motivo si può dire che la percentuale di vicoli ciechi al variare della grandezza del labirinto rimane uguale, come si può vedere chiaramente nella Figura 4.3. Inoltre, si nota che l'algoritmo Recursive Backtracker tende a generare molti meno vicoli ciechi rispetto agli altri algoritmi esaminati, che producono dal 25% al 30% di vicoli ciechi.

Algoritmo	**Percentuale di vicoli ciechi**
Binary Tree	24,98%
Sidewinder	27,70%
Aldous-Broder	29,33%
Recursive Backtracker	9,98%
Recursive Division	26,93%

Tabella 4.7: Percentuale di vicoli ciechi per gli algoritmi generativi su labirinti 100x100 su 1000 tentativi.

Algoritmo	**Percentuale di vicoli ciechi**
Binary Tree	25%
Sidewinder	27,78%
Aldous-Broder	29,37%
Recursive Backtracker	10%
Recursive Division	26,93%

Tabella 4.8: Percentuale di vicoli ciechi per gli algoritmi generativi su labirinti 200x200 su 1000 tentativi.

Algoritmo	**Percentuale di vicoli ciechi**
Binary Tree	25,02%
Sidewinder	27,83%
Aldous-Broder	29,42%
Recursive Backtracker	10,01%
Recursive Division	26,93%

Tabella 4.9: Percentuale di vicoli ciechi per gli algoritmi generativi su labirinti 500x500 su 500 tentativi.

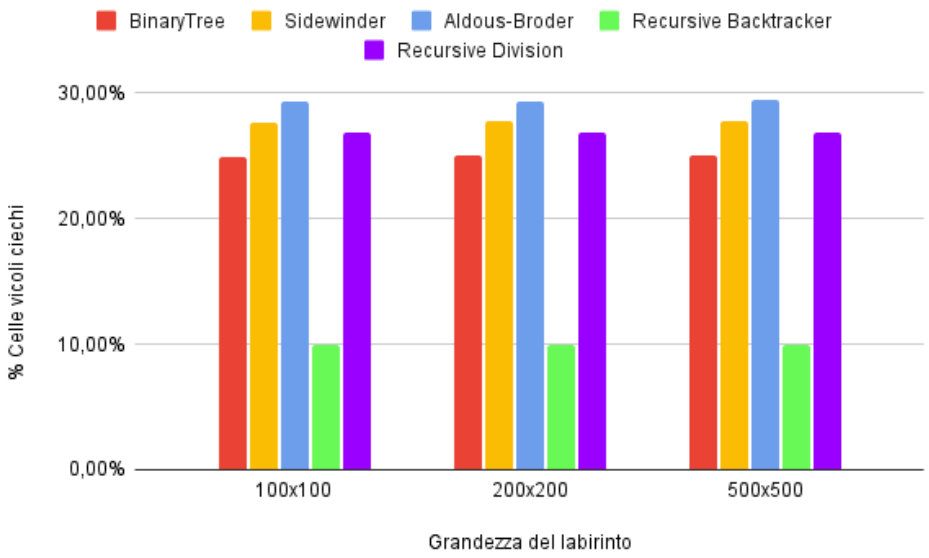

Figura 4.3: Percentuale di vicoli ciechi per gli algoritmi generativi.

4.4 Lunghezza del percorso più lungo

Di seguito viene riportata la lunghezza media del percorso più lungo per ogni labirinto. Questa statistica è stata calcolata applicando due volte l'algoritmo BFS, adattato per pesi unitari. La prima volta viene calcolata la distanza di ogni cella dalla cella root, sempre posta in modo arbitrario come la cella di coordinate [0][0]. A questo punto utilizziamo un algoritmo (triviale) per calcolare la cella con distanza massima, e poi viene riapplicato BFS dalla cella con distanza massima [1], e si trovano quindi le celle con distanza maggiore da quest'ultima. Ovvero si trova uno (tra i molteplici possibili) percorsi più lunghi. Il codice è consultabile all'appendice A.

Dalle tabelle 4.10, 4.11 e 4.12 risulta che all'aumentare della grandezza dei labirinti, aumenta ovviamente la lunghezza del percorso più lungo, ma diminuisce di molto la percentuale della lunghezza rispetto alla grandezza del labirinto.

Consultando la Figura 4.4, risulta evidente che l'algoritmo Recursive Backtracker tende ad avere il percorso più lungo del labirinto di lunghezza molto maggiore rispetto a tutti gli altri labirinti, così come gli algoritmi di Aldous-Broder e Recursive Division, che rispetto agli algoritmi banali come Binary Tree e Sidewinder

hanno un aumento significativo (dalle 2 alle 3 volte tanto) della lunghezza del percorso più lungo.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	385	3,86%
Sidewinder	449	4,49%
Aldous-Broder	707	7,08%
Recursive Backtracker	3775	37,75%
Recursive Division	874	8,74%

Tabella 4.10: Lunghezza del percorso più lungo all'interno del labirinto per gli algoritmi generativi su labirinti 100x100 su 1000 tentativi.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	781	1,95%
Sidewinder	909	2,27%
Aldous-Broder	1694	4,24%
Recursive Backtracker	12617	31,54%
Recursive Division	2206	5,52%

Tabella 4.11: Lunghezza del percorso più lungo all'interno del labirinto per gli algoritmi generativi su labirinti 200x200 su 1000 tentativi.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	1972	0,79%
Sidewinder	2301	0,92%
Aldous-Broder	5331	2,13%
Recursive backtracker	62443	24,98%
Recursive Division	7577	3,03%

Tabella 4.12: Lunghezza del percorso più lungo all'interno del labirinto per gli algoritmi generativi su labirinti 500x500 su 500 tentativi.

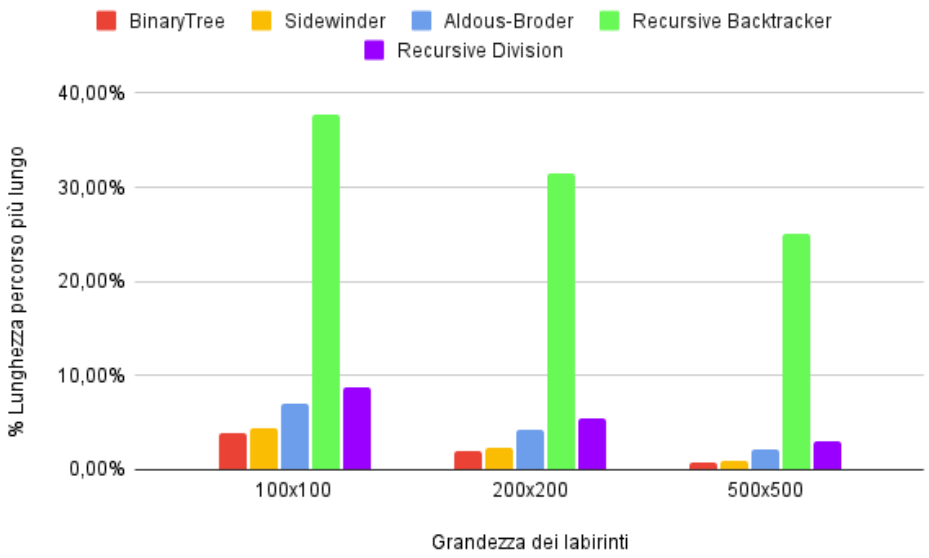

Figura 4.4: Lunghezza del percorso più lungo all'interno del labirinto.

4.5 Lunghezza della soluzione

Di seguito viene riportata la lunghezza media della soluzione di ogni labirinto, che permette di capire quali labirinti a parità di grandezza producono texture di più difficile risoluzione.

Come si vede dalle tabelle 4.13, 4.14 e 4.15, all'aumentare della grandezza dei labirinti ovviamente aumenta la lunghezza della soluzione, ma diminuisce la percentuale della lunghezza rispetto alla grandezza del labirinto.

Inoltre, dalla Figura 4.5 risulta evidente che l'algoritmo Recursive Backtracker tende ad avere la lunghezza della soluzione significativamente maggiore rispetto a tutti gli altri labirinti (dalle 5 alle 10 volte tanto). Questa caratteristica deriva dal fatto che l'algoritmo Recursive Backtracker produce labirinti con percorsi più intricati rispetto ad altri labirinti.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	170	1,71%
Sidewinder	194	1,94%
Aldous-Broder	265	2,66%
Recursive Backtracker	1372	13,73%
Recursive Division	314	3,15%

Tabella 4.13: Lunghezza della soluzione rispetto alla dimensione del labirinto per algoritmi generativi su labirinti 100x100 su 1000 tentativi.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	349	0,87%
Sidewinder	412	1,03%
Aldous-Broder	607	1,52%
Recursive Backtracker	4482	11,21%
Recursive Division	761	1,90%

Tabella 4.14: Lunghezza della soluzione rispetto alla dimensione del labirinto per algoritmi generativi su labirinti 200x200 su 1000 tentativi.

Algoritmo	**Lunghezza media**	**Percentuale rispetto alla dimensione del labirinto**
Binary Tree	878	0,35%
Sidewinder	1044	0,42%
Aldous-Broder	1947	0,78%
Recursive Backtracker	22119	8,85%
Recursive Division	2753	1,10%

Tabella 4.15: Lunghezza della soluzione rispetto alla dimensione del labirinto per algoritmi generativi su labirinti 500x500 su 500 tentativi.

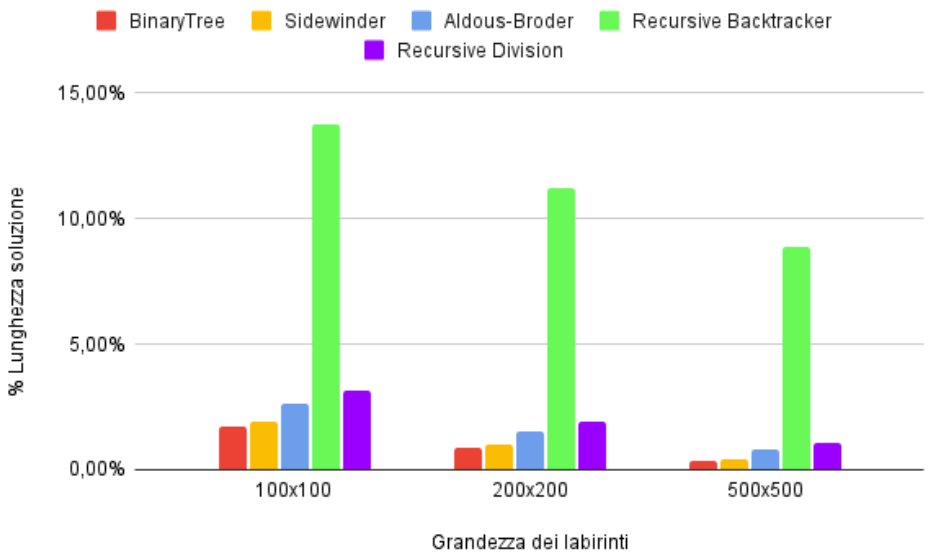

Figura 4.5: Lunghezza della soluzione per gli algoritmi generativi.

Conclusioni e sviluppi futuri

Avendo analizzato gli algoritmi generativi proposti, si può constatare che l'algoritmo Recursive Backtracker è un'ottima scelta in un contesto general purpose per i seguenti motivi:

1. genera labirinti senza bias;
2. genera labirinti con pochi vicoli ciechi, ovvero con percorsi molto lunghi;
3. genera labirinti con percorsi tortuosi, che cambiano spesso direzione;
4. ha complessità computazionale $O(N)$, quindi è molto veloce anche per labirinti molto grandi (500x500, 1000x1000);
5. produce labirinti con soluzioni molto lunghe.

Questo non significa che sia la scelta migliore in ogni caso, infatti la caratteristica di avere pochi vicoli ciechi potrebbe essere un difetto piuttosto che un pregio a seconda dei casi.

Una possibile alternativa all'algoritmo Recursive Backtracker è l'algoritmo Recursive Division. Infatti ha tempi di generazione e risoluzione pressochè identici, ma ha molti più vicoli ciechi, a discapito di una lunghezza del percorso più lungo e della soluzione piuttosto corte rispetto alla grandezza del labirinto, e comunque più corte rispetto a Recursive Backtracker.

Se invece ci si accontenta di labirinti anche con bias evidenti oppure il tempo di esecuzione è una caratteristica ricercata, allora gli algoritmi Binary Tree o Sidewinder sono ideali.

L'unico algoritmo 'da evitare' sarebbe quello di Aldous-Broder, a causa della sua complessità computazionale che risulta proibitiva per labirinti molto grandi, anche se per generare labirinti di medie dimensioni (fino a circa 200x200 celle) è un buon compromesso tra prestazioni e caratteristiche che spesso vengono richieste nei labirinti (senza bias, con molti vicoli ciechi).

Per quanto riguarda il miglioramento del progetto allo stato attuale e di possibili future aggiunte, si potrebbe pensare di implementare algoritmi di generazione per labirinti intrecciati, unicursali, o sparsi e quindi modificare algoritmi risolutivi noti per adattarsi alle caratteristiche richieste. Un'ulteriore modifica potrebbe essere quella di supportare la generazione di labirinti non ortogonali, quindi con tassellazione ad esempio triangolare. Oppure ancora, implementare la rappresentazione delle varie proprietà del labirinto, ad esempio la visualizzazione dei vicoli ciechi, o visualizzare tramite animazioni la costruzione del labirinto per ogni algoritmo generativo.

Appendice A - Codice Python

Algoritmo BFS per il calcolo delle distanze di ogni cella da una cella root

```
def calc_all_distances(self):

  # Crea un'istanza dell'oggetto Distances.
  # La root è la cella attuale (self) su cui
  # viene applicata la funzione
  distances = Distances(self)

  # Lista delle celle di frontiera
  frontier = [self]

  # Finché ci sono celle nella frontiera
  while frontier

    # Frontiera con celle non
    # ancora visitate
    new_frontier = []

    # Itera per ogni cella collegata alla cella attuale
    for each cell in frontier
      for each linked in cell.get_collegate()

        # Se non è gia stata visitata
        if (distances[linked] is None)

          # Aumenta la distanza dalla cella corrente + 1
          distances[linked] = distances[cell] + 1

          # Aggiungi alla frontiera la cella collegata
          new_frontier.append(linked)

    # Aggiorna la frontiera con le celle collegate
    frontier = new_frontier

  # Restituisci un dizionario (cella, distanza) contenente
  # tutte le celle e le relative distanze dalla root
  return distances
```

Algoritmo per il calcolo dei vicoli ciechi del labirinto

```
def deadends(self):

  deadends = []

  for each cell in self.each_cell()

    # Controlla se la cella ha esattamente una sola
    # cella collegata, ovvero se è un vicolo cieco
    if len(cell.all_linked()) == 1
      deadends.append(cell)

  return deadends
```

Algoritmo per ricostruire il cammino minimo per un percorso (root , goal)

```
def shortest_path_to(self, cell_goal):

  # Calcola tutte le distanze dalla root
  path = Distances(self.root)

  current_cell = cell_goal
  path[current_cell] = self[current_cell]

  # Finché non arrivo alla root
  while current_cell != self.root

    # Itera sulle celle collegate
    # alla cella attuale
    for each neighbor in current_cell.get_collegate()

      # Se una cella collegata ha distanza minore
      # della cella attuale, allora è quella
      # da cui si è arrivati
      if self[neighbor] < self[current_cell]
        path[neighbor] = self[neighbor]
        current_cell = neighbor
        break

  # Restituisci un dizionario
  # contenente il cammino minimo (cella, distanza)
  return path
```

Algoritmo per il calcolo della cella più distante dalla root

```
def longest_path_from(self):

  max_distance = 0
  farthest_cell = self.root

  # Itera sulle distanze (cella, distanza)
  for each cell,dist in self._cells.items()

    # Se trovi una cella con distanza maggiore,
    # aggiorna le variabili
    if dist > max_distance
      farthest_cell = cell
      max_distance = dist

  # Restituisci la coppia (cella, distanza)
  return farthest_cell, max_distance
```

Bibliografia

[1] Jamis, Buck. *Mazes for Programmers: Code your Own Twisty Little Passages*. Pragmatic Bookshelf, 2015 (cit. alle pp. 16, 18–20, 22, 43).

[2] Foltin, Martin. *Automated Maze Generation and Human Interaction*. Masaryk University, Faculty of Informatics, Brno, 2008. URL: https://theses.cz/id/k1d3n5/
(visitato il giorno 04/07/2025) (cit. a p. 10).

Siti e risorse online

[3] Jamis, Buck. *Buckblog: Maze Generation: Algorithm Recap*. URL: https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap
(visitato il giorno 06/08/2025) (cit. alle pp. 17, 20, 23, 26, 28).

[4] Jamis, Buck. *Buckblog: Maze Generation: Aldous-Broder algorithm*. URL: https://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm
(visitato il giorno 05/07/2025) (cit. a p. 22).

[5] Walter D. Pullen. *Think Labyrinth: Maze classification*. URL: https://www.astrolog.org/labyrnth/algrithm.htm
(visitato il giorno 14/07/2025) (cit. alle pp. 5, 6, 8–10).

[6] Amit, Patel. *Introduction to A**. URL: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
(visitato il giorno 20/07/2025) (cit. a p. 11).

[7] Amit, Patel. *A*'s Use of the Heuristic*. URL: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
(visitato il giorno 22/07/2025) (cit. a p. 11).

[8] Lisa, Velden. Technischen Universität München. *Der A* Algorithmus*. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-a-star/index_en.html
(visitato il giorno 21/07/2025) (cit. a p. 12).

[9] Nicolò, Pretto. University of Trento, MiRo Lab, *A* project for the course of Robotic Perception and Action. EIT Digital – Autonomous Systems 2018-2019*. URL: https://www.miro.ing.unitn.it/a-1/ (visitato il giorno 14/08/2025).

[10] *Python*. URL: https://www.python.org/ (cit. a p. 2).

[11] *Jetbrains PyCharm*. URL: https://www.jetbrains.com/pycharm/ (cit. a p. 2).

[12] *Pillow library*. URL: https://pypi.org/project/pillow/ (cit. alle pp. 2, 32–36).

Immagini

[13] URL: https://www.shutterstock.com/image-vector/torus-sphere-maze-pattern-set-circular-2107429343 (cit. a p. 6).

[14] URL: https://www.astrolog.org/labyrnth/maze/crack.gif (cit. a p. 7).

[15] URL: https://www.astrolog.org/labyrnth/maze/fractal.gif (cit. a p. 8).

[16] URL: https://weblog.jamisbuck.org/2011/3/4/maze-generation-weave-mazes.html (cit. a p. 9).

[17] URL: https://www.diffen.com/difference/Labyrinth_vs_Maze (cit. a p. 9).