Documentazione Progetto

Università degli Studi di Bergamo

Laurea Magistrale di Ingegneria Informatica

A.A. 2021/2022

Corso di Programmazione Avanzata (6 CFU)

cod. Corso 38090 – Mod. 2

Carne Federico – 1059865

Sommario

Indice delle figure

1 Progetto C++

Il progetto realizzato in C++ si tratta di un semplice gioco da terminale in cui il giocatore deve cercare di uscire da un labirinto, i cui corridoi e mura diventano visibili man mano che l'utente lo esplora. I labirinti, formati da una griglia di "stanze", sono generati casualmente, secondo le opzioni selezionate dal giocatore a inizio partita. Le modalità di gioco offerte sono tre:

1. *Time Trial*: il giocatore, unica entità presente, deve cercare di uscire dal labirinto nel minor tempo possibile, tentando di evitare le trappole che lo rallentano;
2. *Dungeon Mode*: il giocatore deve cercare di sconfiggere tutti i nemici presenti nel labirinto, facendo attenzione alle varie trappole e raccogliendo gli equipaggiamenti presenti nelle stanze. Solo dopo aver sconfitto tutte le altre entità potrà uscire dal labirinto;
3. *Daedalus Mode*: non una vera e propria modalità di gioco, permette al giocatore di creare labirinti, visualizzarne la costruzione passo per passo e salvarli su file (questa funzionalità tornerà utile per il risolutore di labirinti implementato in Haskell);

Durante l'implementazione si è fatto uso nella maggior parte dei casi degli *smart pointers* e utilizzando le *references* solo per parametri di metodi privati quando ritenuto necessario per questioni sintattiche (per esempio, per l'uso di operatori overloaded). Per tutto ciò che necessita di casualità si è fatto uso della classe `std::mt19937` e delle distribuzioni di probabilità presenti nell'header `<random>`, in modo tale da aver maggior controllo e se necessario ripetibilità di una particolare esecuzione. Inoltre, si è cercato di far uso, quando possibile, delle strutture dati più adatte al caso, come `vector`, `set` e `hash_map`, fornite dalla *Standard Template Library*. Un'ultima regola che si è scelto di seguire è l'uso esclusivo di interi a larghezza fissa (per esempio, `uint16_t`, `uint32_t`, …).

1.1 Generazione labirinti

1.1.1 Struttura dati

Un labirinto formato da muri e corridoi può essere modellato, da un punto di vista matematico, come un grafo i cui nodi sono le "stanze" in cui si trova il giocatore, i

passaggi da una stanza all'altra rappresentano gli archi e i muri rappresentano la mancanza di archi. Per questo progetto si è deciso di prendere in considerazione solo labirinti bidimensionali ortogonali, la cui forma è quindi un rettangolo. Un grafo che, quando disegnato, ha un pattern regolare viene detto grafo reticolare. Un grafo reticolare disegnato nel piano, le cui coordinate (x, y) appartengono rispettivamente agli intervalli $1, \dots, n$ e $1, \dots, m$, viene chiamato grafo a griglia quadrata. La classe `GridCell` rappresenta un nodo del grafo mentre la classe `Grid` modellizza il grafo nella sua interezza. Usando un grafo è possibile spostarsi nella griglia conoscendo solo la cella in cui ci si trova e al contempo rispettando i vincoli (per esempio, non attraversare i muri). `GridCell` è un wrapper generico, il cui contenuto può essere anche una classe derivata della classe con cui è stata istanziata (questo grazie all'uso dei puntatori per evitare il fenomeno dello slicing). La cella può anche essere inizializzata a un valore di default, l'unico vincolo posto al contenuto è quello di disporre di un metodo `clone()` (per esempio ereditando dalla *pure abstract class* `Clonable`) per copiare il contenuto in uno `shared_ptr` ed evitare lo slicing nel caso di classi derivate. `Grid` è stato implementato come una matrice con l'operatore di *subscript* come specificato in [1] per velocizzare l'accesso casuale alle celle. Quando viene inizializzato, il grafo può essere già connesso oppure no (questa opzione è utile per alcuni algoritmi di generazione). Ogni nodo può essere connesso solo ai nodi che si trovano nelle quattro direzioni cardinali, rappresentate dalla classe `Direction`. Questa classe simula il comportamento di un enum alla Java, cioè dispone anche di metodi (mentre un enum alla C++ non può avere funzioni membro), attraverso l'uso del costruttore implicito e dell'overriding di alcuni operatori, come l'operatore di uguaglianza. Inoltre, per poter essere usato come chiave di una `hash_map`, dichiara *friend* la struct `std::hash<>`. Come `GridCell`, anche `Grid` è un template rispetto al contenuto delle celle, questo permette di poterlo usare per modellare altri giochi basati su griglie, come gli scacchi o la dama.

La classe `Maze`, che eredita da `Grid` istanziando il contenuto con la classe `Room`, rappresenta il labirinto vero e proprio. `Maze` dispone dei metodi per controllare la mossa vincente, per essere "disegnato" sul terminale e per essere stampato su file (questa differenziazione sarà utile per il solutore in Haskell). `Room` e la sua derivata `DungeonRoom` rappresentano le stanze del labirinto, contenenti trappole, armi e nemici. La mossa vincente viene selezionata scegliendo la cella più lontana da quella di partenza.

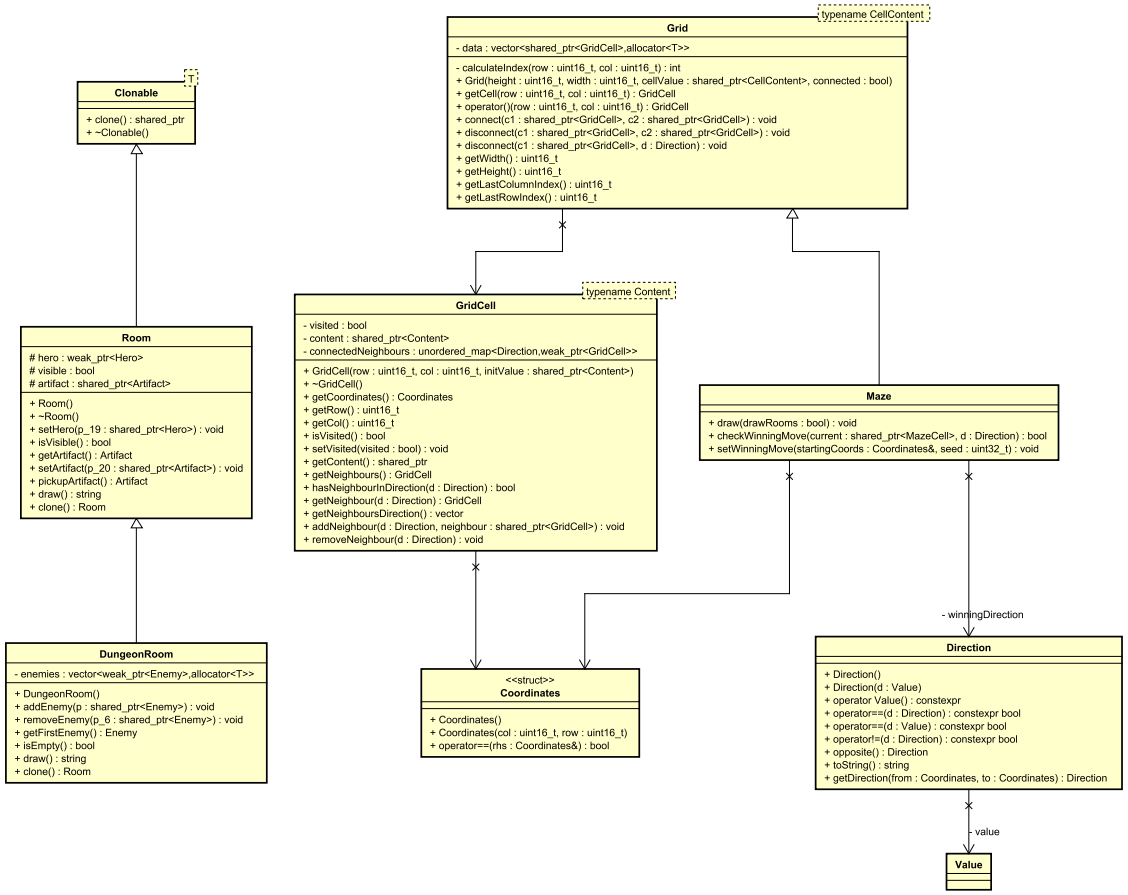


Figura 1: Class diagram – Maze

1.1.2 Algoritmi di generazione

Per la generazione dei labirinti si è deciso di prendere in considerazione solo algoritmi che creano labirinti cosiddetti *perfetti*. La caratteristica principale di questo tipo di labirinti è la presenza di uno e un solo percorso tra qualsiasi coppia di celle. I tre algoritmi implementati, oltre a tanti altri, sono descritti in [2].

1.1.2.1 Randomized Prim

Il primo algoritmo è una versione modificata dell'algoritmo di Prim: a ogni iterazione, invece che l'arco con il costo minore, l'algoritmo seleziona un nodo scelto a caso tra quelli adiacenti alla cella corrente. Lo pseudocodice dell'algoritmo è il seguente:

1) Choose a starting cell in the grid and add it to the path set.
2) While there is cell to be managed in the set:
   a) Randomly connect to one of the already connected neighbour.
   b) Add all unconnected neighbours to the set.

L’algoritmo inizia la creazione da una cella qualsiasi ed espande man mano il labirinto in una direzione casuale, basti che non generi un ciclo. All’inizio il grafo è non connesso, per cui l’algoritmo “scava” un percorso attraverso i muri.

1.1.2.2 Recursive Division

Questo secondo algoritmo invece, è un “costruttore” di muri nel senso che inizia da un grafo connesso (quindi senza muri) e procede ricorsivamente aggiungendo un muro in quella che può essere considerata una stanza gigante, lasciando uno spazio solo per passare da una parte all’altra. Lo pseudocodice è il seguente:

1) Randomly build a wall within this space.
2) Randomly build a path within this wall.
3) Recurse on both sides of the wall.

A ogni passo di ricorsione il labirinto è ancora valido e l’algoritmo può procedere all’infinito, creando ipoteticamente labirinti con stanze infinitamente piccole.

(a) (b) (c)

Figura 2: Labirinti generati da Randomized Prim (a), Recursive Division (b) e SideWinder (c)

1.1.2.3 SideWinder

Il terzo algoritmo permette invece di costruire labirinti grandi a piacere dal momento che ha bisogno di memorizzare solo la riga corrente. Procedendo per righe, l’algoritmo sceglie se aprire un passaggio verso Est, in caso contrario crea un passaggio verso Nord a partire da una cella qualsiasi già visitata nella riga corrente. Lo pseudocodice è il seguente:

For each row in the grid:

For each cell randomly decide whether to carve a passage leading East

a) If the passage is carved add the cell to the current run set.
b) If the passage is not carved, randomly pick one cell from the route set, carve a passage leading North and empty the current run set.

La particolarità di questo algoritmo è che la prima riga del labirinto sarà un lungo corridoio. Anche in questo caso l'algoritmo scava attraverso i muri.

La generazione degli algoritmi è stata implementata attraverso lo *strategy pattern*: le classi derivate da `MazeAlgorithm` implementano le varie strategie di creazione, facendo overloading della funzione *virtual* `generate()`, mentre `MazeGenerator` fa da *context*.

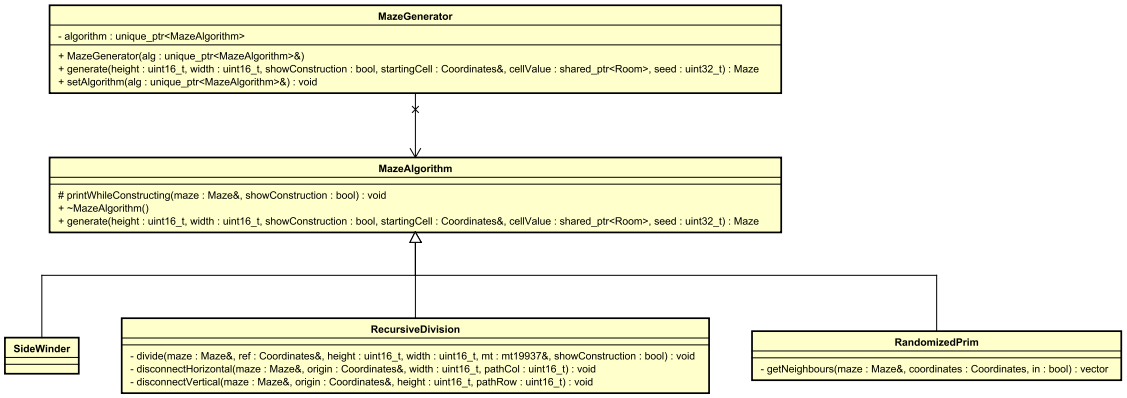


Figura 3: Class diagram – Maze generator

1.2 Entità e artefatti

1.2.1 Entità

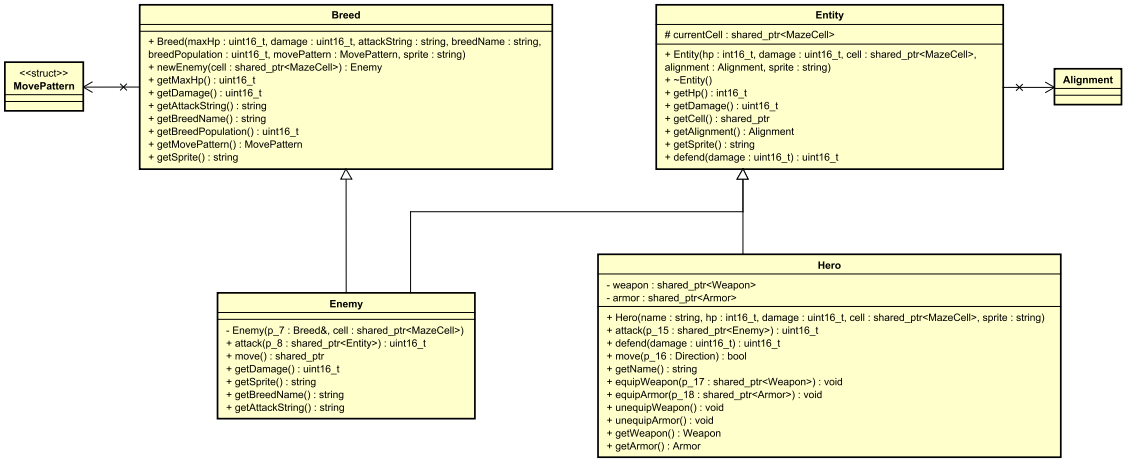


Figura 4: Class diagram – Entities hierarchy

Le entità sono coloro che popolano e si muovono nel labirinto. `Entity` è la classe base e definisce i tratti comuni a tutte le entità come i punti vita, i punti danno e la cella in cui si trova attualmente. `Hero` rappresenta il giocatore mentre `Enemy` rappresenta i mostri da sconfiggere nella modalità Dungeon. Per permettere di definire nuovi nemici senza dover modificare il codice, `Enemy` segue il pattern *Type Object* descritto in [3]. Questo pattern

evita di dover definire una classe per ogni tipologia di mostri che si vuole aggiungere ma consente di definirne solo due: `Enemy`, che rappresenta il *typed object*, e `Breed`, che rappresenta il *type object*. Il *type object* rappresenta un tipo vero e proprio mentre il *typed object* contiene una reference al tipo che lo rappresenta. Dato che ogni nemico ha una specie, la relazione *"has-a"* è ben modellizzabile con l'ereditarietà privata. Il vero vantaggio di usare questo pattern è la possibilità di definire le varie specie in un file json, secondo un approccio *data-driven*, invece che avere una gerarchia hard-coded, introducendo maggior flessibilità e la possibilità di customizzazione da parte degli utenti. Per semplificarne la gestione, `Breed` svolge anche il ruolo di *factory* per la classe `Enemy`. I nemici si muovono casualmente secondo il `MovePattern` definito nel file json.

1.2.2 Artefatti

Gli artefatti rappresentano gli oggetti in cui ci si può imbattere durante l'esplorazione del labirinto, come per esempio trappole o equipaggiamenti per il giocatore. Ogni artefatto ha, tra le altre proprietà, un effetto, definito attraverso una *lambda function*. Questo permette di creare artefatti con diversi effetti senza dover obbligatoriamente definire una classe derivata (ma in questo caso si è scelto di non seguire un approccio data-driven vista la complessità di codificare il comportamento sottoforma di dati, per esempio con il *bytecode pattern*, sempre descritto in [3]).

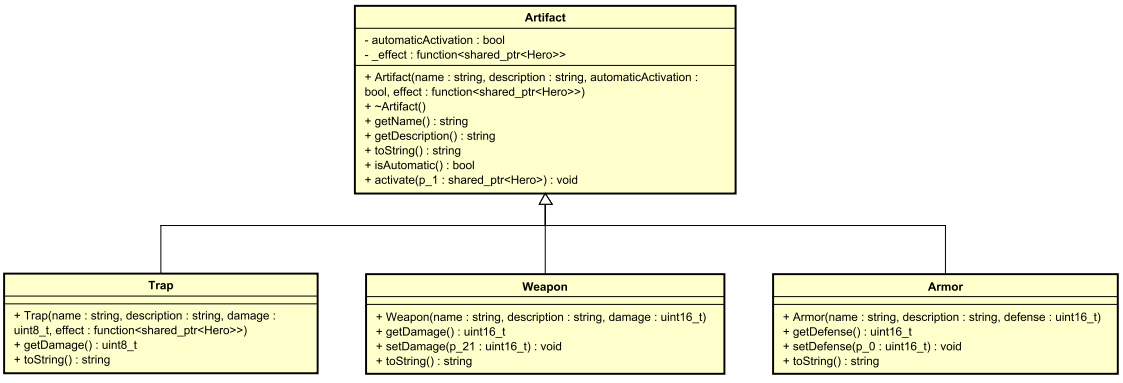


Figura 5: Class diagram – Artifacts hierarchy

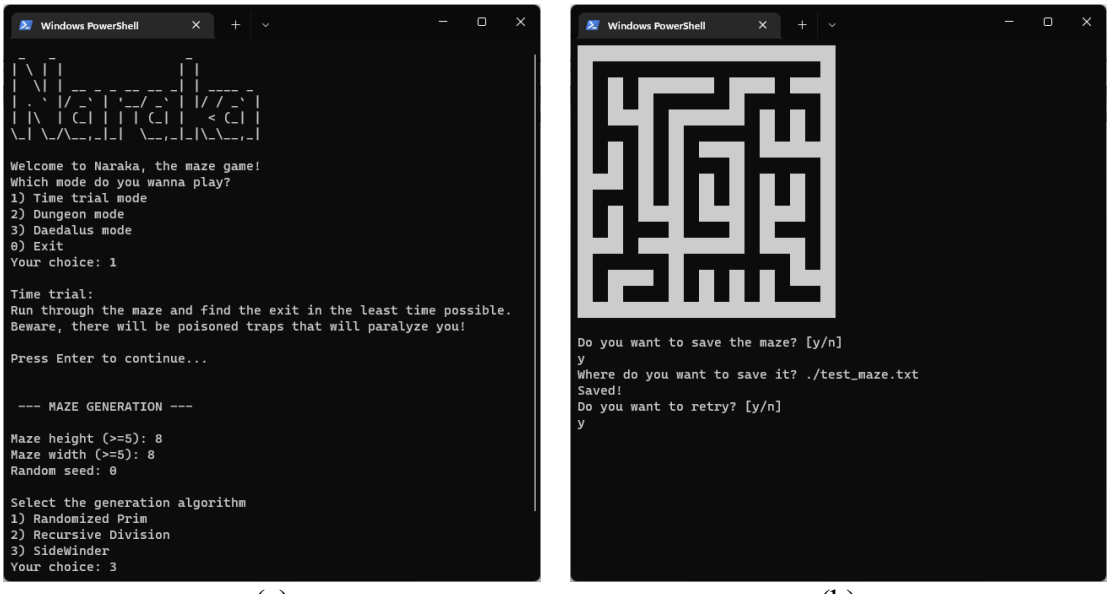
I tre tipi di artefatti definiti sono:

- Le trappole, con effetti negativi per il giocatore.
- Le armi, il cui effetto è aumentare il danno durante un attacco.
- Le armature, il cui effetto è aumentare la difesa del giocatore.

1.3 Funzionamento

Il progetto, denominato Naraka, è disponibile come file executable. Per avviare il programma basta semplicemente fare doppio click sull'eseguibile. L'utente sceglie una tra le tre modalità di gioco e poi inserisce interattivamente i parametri (quali dimensioni, il seed del Mersenne Twister, …) e l'algoritmo di generazione del labirinto. Se la modalità scelta è TimeTrial o Dungeon allora verrà chiesto anche il nome del giocatore, per costruire un oggetto `Hero`, e il numero di trappole da posizionare nelle stanze. La generazione delle trappole avviene grazie a una factory, scegliendo casualmente uno tra i cinque tipi definiti. Nel caso la modalità scelta fosse Dungeon, verrà richiesto anche il percorso dei file json contenenti le specie dei nemici, inserite poi in una `hash_map`, le armi e le armature, inserite in due liste separate. Nemici ed equipaggiamenti verranno poi disseminati casualmente all'interno del labirinto. La lettura da file json avviene utilizzando la libreria JSON for Modern C++ [4].

Completato il setup, la partita ha inizio e il giocatore deve cercare di uscire dal labirinto, inizialmente completamente invisibile, muovendosi e scoprendo man mano la posizione dei muri. Se entra in una stanza in cui è presente una trappola questa viene attivata automaticamente mentre se è presente dell'equipaggiamento lo si potrà raccogliere solo dopo aver sconfitto tutti i nemici nella stanza, se ce ne sono. Una volta terminata la partita è possibile salvare il labirinto e rigiocare.



(a) (b)

Figura 6 : Screen Naraka – Setup (a) e salvataggio (b) di un labirinto 8x8

2 Progetto Haskell

Il progetto realizzato in Haskell si tratta di un risolutore di labirinti in formato testuale, come quelli che possono essere generati e salvati con Naraka. Come già anticipato, i labirinti generati sono *perfetti*, rispettano cioè le due seguenti condizioni:

- Tutte le celle sono parte di un unico spazio connesso.
- Non contiene nessun percorso ciclico.

Questi due vincoli possono essere espressi più semplicemente con la seguente condizione: per ogni coppia di celle, ci può essere uno e un solo percorso che le connette. I labirinti perfetti, anche detti *semplicemente connessi*, possono essere considerati degli alberi in teoria dei grafi. Questo tipo di labirinto ammette un algoritmo di risoluzione molto semplice, applicabile anche nella vita reale: l'algoritmo *wall follower* o *regola della mano destra* (o *sinistra*) richiede di esplorare il labirinto tenendo la mano destra (o sinistra) sempre in contatto con il muro, in questo modo è garantito che il solutore non si perda e che trovi un uscita se questa esiste, altrimenti ritornerà all'ingresso del labirinto. Da un punto di vista algoritmico, tenere la mano destra sempre in contatto con il muro significa che a ogni incrocio si girerà sempre a destra. La soluzione trovata non è sempre la migliore e se il labirinto non è perfetto non è garantito che l'uscita venga raggiunta. Esistono algoritmi migliori che trovano sempre un percorso, che è anche ottimo.

Il labirinto è formato dalle seguenti componenti:

- "`|`", "`---`", "`+`", rispettivamente per i muri verticali, orizzontali e le celle in cui si incrociano più muri;
- I corridoi rappresentati da " " (1 spazio) e " " (3 spazi);
- La cella di partenza è indicata dalla presenza della lettera "`S`";
- La cella di arrivo è indicata dalla presenza della lettera "`G`";

2.1 Parsing del labirinto

La prima operazione svolta dal solutore è il parsing del labirinto da formato testuale a rappresentazione matriciale. Per semplicità, il labirinto non è modellato come un grafo reticolare ma come una matrice di *tiles*: ogni cella (o tile) può essere un muro, un corridoio, la cella di partenza, la cella di arrivo o una cella attraversata dalla soluzione.

Questa differenziazione è utile all'algoritmo per capire quali tiles considerare e quando fermarsi ma consente anche una rappresentazione grafica (sempre in formato testuale) più efficace.

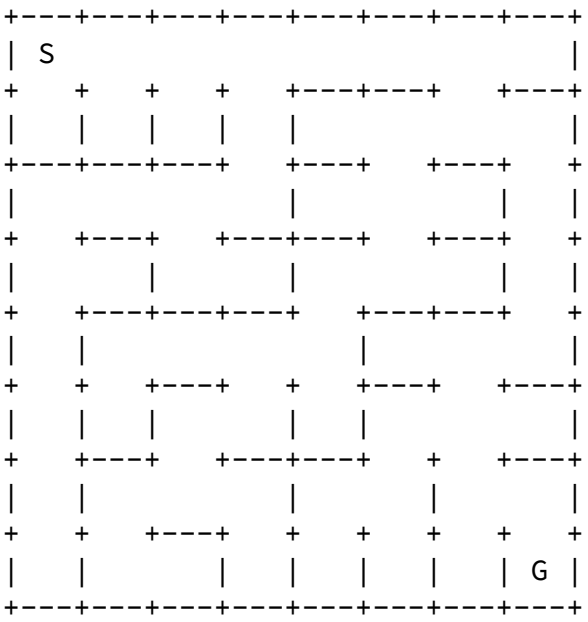


Figura 7: Esempio di labirinto accettato dal solutore

Il parsing del labirinto da file avviene attraverso due *list comprehension* annidate:

```
[[parseTile t | t <- segment str] | str <- lines mazeStr]
```

La list comprehension più esterna considera il labirinto riga per riga, mentre quella più interna fa il parsing di ogni cella, dopo aver spezzato la riga in stringhe di lunghezza 1 e 3 con la funzione `segment`. Il parsing delle celle avviene nel seguente modo:

- se la stringa (lunga 1 o 3 caratteri) è formata solo da spazi, la cella è un corridoio;
- se la stringa contiene una "`S`", la cella è di partenza;
- se la stringa contiene una "`G`", la cella è di arrivo;
- altrimenti, la cella è un muro.

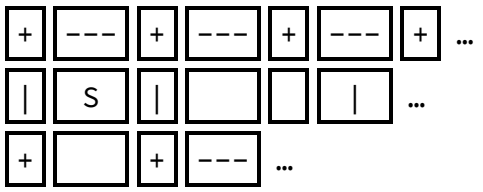


Figura 8: Esempio di suddivisione celle in stringhe di lunghezza 1 e 3

2.2 Risoluzione

L'algoritmo che si è scelto di implementare è **A***, date la sua popolarità, l'efficienza e la garanzia di trovare una soluzione anche quando sono presenti cicli nel labirinto. A* esegue una ricerca di percorso sfruttando alcune informazioni, nel caso della risoluzione dei labirinti la cella che porta all'uscita. A* è sostanzialmente una variazione dell'algoritmo di Dijkstra: mentre Dijkstra cerca indiscriminatamente in tutte le direzioni possibili, sprecando risorse, A* cerca il percorso "più promettente" per arrivare alla cella finale. La principale differenza è la funzione di costo:

- Per Dijkstra, il costo è la distanza dal nodo radice;
- Per A*, il costo è dato dalla somma di due componenti: $g(n)$, cioè la distanza dal nodo radice, e $h(n)$, cioè una stima della distanza dal nodo obiettivo.

Nel caso in esame, $g(n)$ è semplicemente il numero di celle tra il nodo corrente e la cella di partenza, mentre $h(n)$ è *distanza di Manhattan* (o *geometria del taxi*), in riferimento alla sistema stradale tipico americano, in cui le vie di scorrimento sono ortogonali tra di loro. La distanza di Manhattan tra due punti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ si calcola nel seguente modo:

$$h(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

Questa scelta rappresenta un buon trade-off tra approssimazione e velocità di computazione, soprattutto quando le direzioni consentite sono solamente le quattro direzioni cardinali.

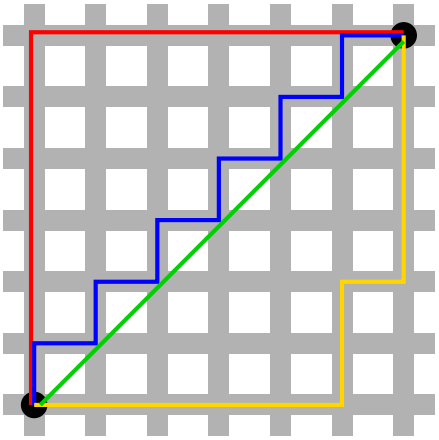


Figura 9: Confronto tra distanza euclidea (in verde) e distanza di Manhattan

Lo pseudocodice di A* è il seguente:

```
function backtrace(previous, current)
  path := {current}
  while current in keys(previous):
    current := previous[current]
    prepend(path, current)
  return path

function A*(start, goal, h)
  openSet := {start}
  closedSet := an empty set
  previous := an empty map

  g_score := map with default value of Infinity
  g_score[start] := 0
  f_score := map with default value of Infinity
  f_score [start] := h(start)

  while openSet is not empty
    current := the node in openSet with the lowest f_score[] value
    if current = goal
      return backtrace(previous, current)

    remove current from openSet
    add current to closedSet
    for each neighbour of current
      if neighbour in closedSet
        continue

      tentative_g_score := g_score[current] + d(current, neighbour)
      if tentative_g_score < g_score[neighbuor]
        previous[neighbour] := current
        g_score[neighbour] := tentative_g_score
        f_score[neighbour] := tentative_g_score + h(neighbour)

        if neighbour not in openSet
          add neighbour to openSet

  return failure
```

Dove h è la distanza di Manhattan dalla cella di arrivo e $d(n_1, n_2)$ è la distanza tra un nodo e il suo vicino, nel caso dei labirinti questa è costante e pari a 1. Se l'euristica è consistente, come è la distanza di Manhattan, quando un nuovo viene rimosso dall'`openSet` allora il percorso che lo raggiunge è già ottimale per cui non sarà mai reinserito;

L'implementazione in Haskell ha richiesto alcuni cambiamenti, come la sostituzione dei cicli con la ricorsione e la definizione di una classe *record* che tenesse traccia dei precedenti, dei valori di g, h e f e delle coordinate della cella. Inoltre, `openSet` e `closedSet` per questioni di efficienza dovrebbero essere implementate rispettivamente come una *priority queue* (o un *min-heap*) e una *hash_map* (o un *hash_set*). Per questioni di semplicità sono state implementate semplicemente come due liste.

2.3 Funzionamento

Il progetto, denominato Samsara, è disponibile sottoforma di eseguibile. Per avviare il programma è necessario digitare nel terminale di Windows il seguente comando:

```
./Samsara.exe path/to/maze_file.ext
```

Il solutore effettua come primo passo il parsing del labirinto presente nel file specificato come parametro. Successivamente controlla la validità del labirinto, ossia cerca se esistono esattamente una cella di ingresso e uno cella di uscita e ne restituisce le coordinate; in caso contrario genera errore. Se il labirinto è valido inizia la ricerca della soluzione con A* e la funzione di ricostruzione del percorso, il cui risultato viene passato come parametro a una funzione che marca le celle attraverso le quali passa la soluzione. Infine, il labirinto risolto viene salvato nel file `path/to/maze_file_solved.ext`.

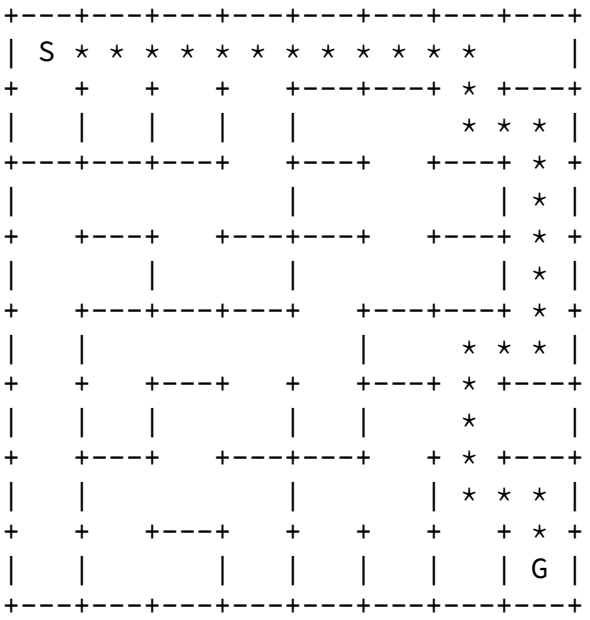


*Figura 10: Labirinto in Figura 7 risolto con A**

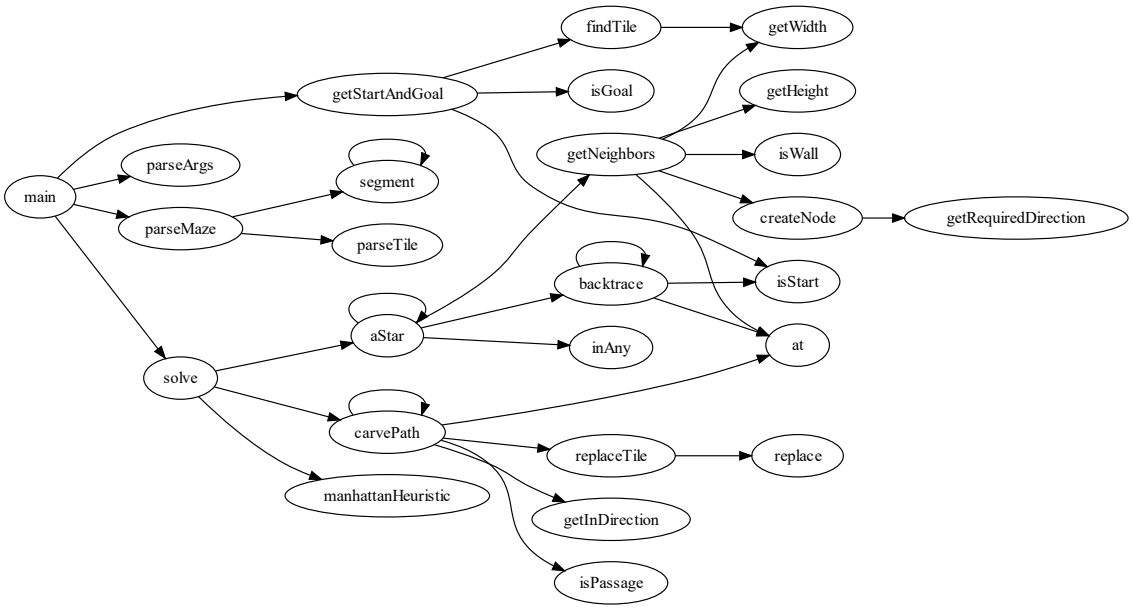


Figura 11: Albero delle chiamate del solutore

3 Riferimenti

[1] Standard C++ Foundation, «Operator Overloading, C++ FAQ,» [Online]. Available: https://isocpp.org/wiki/faq/operator-overloading#matrix-subscript-op. [Consultato il giorno 23 Aprile 2022].

[2] J. Buck, Mazes for programmers: Code your own twisty little passages, The Pragmatic bookshelf, 2015.

[3] R. Nystrom, Game Programming Patterns, Genever Benning, 2014.

[4] N. Lohmann, «JSON for modern C++,» 2013. [Online]. Available: https://json.nlohmann.me/. [Consultato il giorno 24 Aprile 2022].