MazeSolver

Nicola Cappello

2022/2023

Repository Github: https://github.com/CappelloNicola/MazeSolver

Indice

Capitolo 1

Definizione del problema

1.1 Introduzione

Il progetto **MazeSolver** si pone come obiettivo la creazione di un'IA capace di **risolvere labirinti**.

Se consideriamo il labirinto come un insieme di blocchi, allora un labirinto è una matrice di blocchi avente, normalmente, un solo blocco iniziale ed un solo blocco finale. I blocchi hanno dei muri, che permettono o meno il transito da un blocco a quelli adiacenti. Il compito dell'IA è quello di trovare un percorso dal blocco iniziale al blocco finale.

Nel problema preso in esame i labirinti vengono generati in maniera automatica e casuale. Questi hanno le seguenti caratteristiche:

- Dato un blocco, esistono uno o più percorsi verso ogni altro blocco;
- Per la proprietà precedente, esistono uno o più percorsi che portano dal blocco iniziale al blocco finale;
- **NON** è detto che il labirinto sia esente da cammini ciclici (i quali, tuttavia, non impattano sulla completezza degli algoritmi di ricerca adottati).

1.2 Specifica PEAS

- **Performance:** Le misure di prestazione adottate per valutare l'operato di un agente. Viene valutata la capacità dell'agente di trovare o meno una via d'uscita dal labirinto.
- **Environment:** Descrizione degli elementi che formano l'ambiente. L'ambiente in cui opera l'agente è un labirinto. Può essere visto come un insieme di blocchi di forma quadrata aventi un massimo di 3 lati. Esistono un solo blocco di partenza e un solo blocco di destinazione.
- **Actuators:** Gli attuatori disponibili all'agente per intraprendere azioni. L'agente esplora i vari blocchi del labirinto fino a trovare l'uscita. L'esplorazione avviene spostandosi da un blocco all'altro. Dato il blocco che l'agente sta attualmente esplorando, questo può spostarti ad Est, Ovest, Nord o Sud, se i muri del blocco lo permettono.
- **Sensors:** I sensori attraverso i quali l'agente riceve input percettivi. L'agente percepisce l'ambiente sfruttando la rappresentazione virtuale del labirinto.

1.2.1 Caratteristiche dell'ambiente

- **Singolo agente:** All'interno dell'ambiente opera un solo agente, che ha come obiettivo trovare un percorso verso il blocco finale.
- **Completamente osservabile:** L'agente conosce sempre il blocco del labirinto in cui si trova e conosce la struttura dell'intero labirinto grazie alla rappresentazione virtuale.
- **Deterministico:** Il blocco successivo del labirinto che l'agente esplorerà è univocamente determinato dal blocco corrente e dalla direzione in cui l'agente decide di svoltare.
- **Statico:** Il labirinto non cambia la propria struttura mentre l'agente sta esplorando i blocchi.
- **Sequenziale:** La direzione in cui l'agente decide di svoltare dipende anche dal path di blocchi fino ad ora esplorato.
- **Discreto:** In ogni blocco l'agente può scegliere di svoltare in una delle 4 (al massimo) direzioni.

1.3 Analisi del problema

Il problema può essere formalizzato descrivendolo nel seguente modo:

- **Stato iniziale:** Nello stato iniziale l'agente non ha esplorato ancora nessun blocco del labirinto (si trova sul blocco iniziale).
- **Azioni:** L'agente può muoversi a Nord, ad Est, ad Ovest o a Sud, a patto che il blocco non abbia muri che non permettono di spostarsi nella direzione scelta.
- **Modello di transizione:** Ad ogni azione l'agente si troverà su un nuovo blocco inesplorato.
- **Test obiettivo:** L'agente si trova sul blocco finale.
- **Costo di cammino:** Il numero di blocchi da attraversare per raggiungere l'uscita.

Il problema è stato affrontato utilizzando degli algoritmi di ricerca non informata (Ricerca in Ampiezza e Ricerca in Profondità) e informata (A*) visti a lezione.

Capitolo 2

Classi e Interfacce

2.1 *ReturningValues*

La classe *ReturningValues* definisce le istanze di oggetti che vengono restituiti dagli algoritmi di ricerca. Un oggetto di questa classe contiene due strutture dati: *path* e *parents*. La prima struttura dati è un *List* che contiene i nodi che sono stati esplorati durante il processo di ricerca, in ordine. La seconda struttura dati è un *Map* che associa ad ogni nodo esplorato il nodo che lo ha generato. Queste strutture dati sono necessarie al fine di permettere alla classe *ActionListenerCustomized* di visualizzare i nodi esplorati e il path finale su schermo.

2.2 *ActionListerCustomized*

La classe *ActionListenerCustomized* permette di associare ai *JButtons* (i pulsanti che vengono cliccati per avviare un algoritmo di ricerca) un listener che prende in input l'oggetto *ReturningValues* dell'algoritmo e visualizza il path percorso ed il cammino trovato dall'algoritmo.

2.3 *PresetMaze*

La classe *PresetMaze* è un Java Enum che mantiene in memoria la struttura di labirinti definiti dal programmatore in fase di testing degli algoritmi di ricerca, prima che un generatore di labirinti venisse implementato. Un labirinto è rappresentato mediante un array bidimensionale. Ogni posizione dell'array rappresenta un blocco del labirinto. Un valore intero sta ad indicare quali muri sono presenti.

2.4 *Maze*

La classe *Maze* è una *JComponent* ed è la classe responsabile di disegnare il labirinto. Mantiene i riferimenti di tutti i blocchi del labirinto.

2.5 *Block*

La classe *Block* racchiude le informazioni riguardo un singolo blocco del labirinto, tra cui la posizione nell'array bidimensionale rappresentante il labirinto e i muri presenti.

2.6 *BlockPoint*

La classe *BlockPoint* ha due attributi, *row* e *col*. Istanze di questa classe vengono utilizzate quando c'è bisogno di passare come parametro la posizione di un blocco all'interno dell'array bidimensionale, rappresentante il labirinto.

2.7 *GenerateMazeListener*

La classe *GenerateMazeListener* permette di associare al *JButton* utilizzato per generare nuovi labirinti un listener che si occupa di invocare il codice necessario alla generazione di un nuovo labirinto.

2.8 *DrawableMaze*

L'interfaccia *DrawableMaze* definisce i metodi che devono implementare le classi (quali *PresetMaze* e *MazeGenerator*) le cui istanze vengono passate come parametri alla classe *Maze* per permettere il draw del labirinto.

2.9 *MazeGenerator*

La classe *MazeGenerator* è la classe che si occupa di generare dei labirinti dato un numero di righe e colonne. Il processo di generazione avviene nel seguente modo:

Sia *maze[row][col]* l'array bidimensionale di blocchi, rappresentante il labirinto, avente *row* righe ed *col* colonne. Inizialmente ogni blocco ha tutti e 4 i muri;

Sia *visited[row][col]* un'array bidimensionale tale che *visited[i][j] = true* se e solo se *maze[i][j]* è stato visitato dall'algoritmo di generazione. L'algoritmo è implementato dal metodo *createMaze(node)*, che verrà invocato passando come parametro *maze[0][0]*. Il metodo compirà i seguenti passi:

- *visited[node.getRow()][node.getCol()] = true* //marca *node* come visitato
- *nextNode = randomUnvisitedNeighbor(node)* //sceglie a caso un blocco adiacente di *node* non visitato
- finché esiste un *nextNode*:
  - *connectNodes(node, nextNode)* //elimina i muri che separano i nodi
  - *createMaze(nextNode)* // continua l'eliminazione dei muri da *nextNode*
  - *nextNode = randomUnvisitedNeighbor(node)* //quando il metodo in cima allo stack delle chiamate non ha più vicini visitabili, viene restituito il controllo al metodo da cui è partita l'invocazione e viene generato un nuovo vicino visitabile (se esiste);
- *return* //restituisci il controllo al chiamante

Al termine dell'algoritmo, l'array bidimensionale *maze* definirà la struttura di un **Perfect Maze**, ossia di un labirinto avente le seguenti proprietà:

- Dato un blocco, esiste un unico percorso verso ogni altro blocco;
- Per la proprietà precedente, esiste un unico percorso che porta dal blocco iniziale al blocco finale;

Date queste proprietà, gli algoritmi di ricerca, se eseguiti su questo tipo di labirinti, restituirebbero sempre un unico cammino. Per rendere confrontabili gli algoritmi in termini di ottimalità, il processo di generazione dei labirinti prevede una fase di eliminazione random di muri, in quanto:

Dati due blocchi del labirinto A e B adiacenti e separati da un muro, per le proprietà del *Perfect Maze* esiste un unico cammino che parte da A e termina in B. Eliminando il muro che separa A e B esistono ora due percorsi che partono da A e terminano in B. Se il path che parte dal blocco iniziale e termina nel blocco finale del labirinto include i blocchi A e B, allora avremo due percorsi dal blocco iniziale al blocco finale, ed il nuovo percorso è migliore dell'altro in termini di costo di cammino.

Più muri si eliminano, più aumenta la possibilità di generare nuovi path dal blocco iniziale al blocco finale.

Il numero di muri da eliminare è uguale a *max(row, col)*, valore scelto per sperimentazione empirica.

2.10 *Main*

La classe *Main* ospita il metodo *main* in cui viene inizializzata l'interfaccia grafica e i metodi che implementano gli algoritmi di ricerca.

Capitolo 3

Algoritmi di ricerca

3.1 Grafo dello spazio degli stati

Ogni nodo del grafo rappresenta un blocco del labirinto. I figli di un nodo sono tutti i blocchi che è possibile raggiungere muovendosi in una delle direzioni consentite.

3.2 Ricerca non informata

Gli algoritmi di ricerca non informata non dispongono di informazioni aggiuntive circa gli stati ed il problema che permettano di migliorare l'efficienza della ricerca. Gli algoritmi implementati sono la Ricerca in Ampiezza, la Ricerca a Costo Uniforme e la Ricerca in Profondità.

3.2.1 Ricerca in Ampiezza

Con questa strategia di ricerca tutti i nodi a profondità d vengono espansi prima di tutti i nodi a profondità *d+1*. Questa ricerca trova sempre la soluzione avente cammino più breve, se una soluzione esiste. Il cammino più breve è anche ottimo nel caso in cui la funzione costo di cammino sia monotona non decrescente. Nel problema trattato, scendere di un livello nel grafo equivale ad aumentare di uno il costo di cammino. Quindi la Ricerca in Ampiezza restituirà un cammino ottimo.

Discutiamo della complessità: Sia b il fattore di ramificazione (avente valore 4) e d la profondità della soluzione a costo minimo. La Ricerca in Ampiezza ha **complessità temporale** $O(b^d)$, in quanto, supponendo che la soluzione più vicina si trova a profondità d, l'algoritmo genererà $O(b^d)$ nodi. Ogni nodo generato viene memorizzato all'interno della frontiera. La **complessità spaziale** sarà quindi $O(b^d)$

Nella classe Main, il seguente metodo implementa la Ricerca in Ampiezza:

```
public static ReturningValues breadthFirstSearch(Maze maze){
        Map<Block,Block> parents = new HashMap<>();
        Block currentNode = maze.getStartBlock();
        List<Block> path = new ArrayList<>();
        Set<Block> visitedNodes = new HashSet<>();
        Queue<Block> queue = new LinkedList<>();

        if(currentNode.getIsEnd()){
            List<Block> oneBlockPath = new ArrayList<>();
            oneBlockPath.add(currentNode);
            return new ReturningValues(oneBlockPath, parents);
        }

        queue.add(currentNode);

        while(true){
            if(queue.isEmpty()){
                return null;
            }

            currentNode = queue.poll();
            path.add(currentNode);
            visitedNodes.add(currentNode);

            List<Block> neighbors = currentNode.getNeighbors();
            for (Block neighbor : neighbors) {
                if (!visitedNodes.contains(neighbor)) {
                    parents.put(neighbor, currentNode);
                }
                if (!queue.contains(neighbor) && !visitedNodes.contains(
                    neighbor)) {
                    if (neighbor.getIsEnd()) {
                        path.add(neighbor);
                        return new ReturningValues(path, parents);
                    }
                    queue.add(neighbor);
                }
            }
        }
    }
```

- La variabile *parents* associa ad ogni nodo il nodo che l'ha generato;
- La variabile *currentNode* contiene un riferimento al nodo che si sta attualmente espandendo;
- La variabile *path* aggiunge i nodi esplorati in ordine di espansione;
- La variabile *visitedNodes* contiene i riferimenti dei nodi che sono stati esplorati;
- La variabile *queue* rappresenta la frontiera, implementata come una coda FIFO.

Ad ogni iterazione si espande il nodo corrente. Se uno dei figli è un nodo obiettivo, l'algoritmo termina restituendo il risultante percorso, altrimenti si passa alla prossima iterazione estraendo un nodo dalla frontiera (secondo la strategia FIFO). Se ad un'iterazione la coda risulta vuota, non è stata trovata una via d'uscita e viene restituito *null*.

3.2.2 Ricerca in Profondità

Con questa strategia di ricerca viene sempre espanso per primo il nodo in frontiera avente profondità maggiore nell'albero di ricerca corrente.

L'algoritmo è stato implementato nella sua versione iterativa. Nel problema in esame l'algoritmo risulta completo, in quanto non incappa in cammini ridondanti (viene utilizzato l'insieme dei nodi esplorati) e il numero degli stati è un valore finito. L'algoritmo non è ottimo.

Nella classe Main, il seguente metodo implementa la Ricerca in Profondità:

```
public static ReturningValues depthFirstSearch(Maze maze){
        Block currentNode = maze.getStartBlock();
        Deque<Block> queue = new ArrayDeque<Block>();
        Set<Block> visitedNodes = new HashSet<>();
        Map<Block,Block> parents = new HashMap<>();
        List<Block> path = new ArrayList<>();

        queue.push(currentNode);

        if(currentNode.getIsEnd()){
            List<Block> oneBlockPath = new ArrayList<>();
            oneBlockPath.add(currentNode);
            return new ReturningValues(oneBlockPath, parents);
        }

        while(!queue.isEmpty()){

            currentNode = queue.pop();
            path.add(currentNode);
            visitedNodes.add(currentNode);

            List<Block> neighbors = currentNode.getNeighbors();
            for (Block neighbor : neighbors) {
                if (!visitedNodes.contains(neighbor)) {
                    parents.put(neighbor, currentNode);
                    queue.push(neighbor);

                    if (neighbor.getIsEnd()) {
                        path.add(neighbor);
                        return new ReturningValues(path, parents);
                    }
                    visitedNodes.add(neighbor);
                }
            }
        }
        return null;
    }
```

- La variabile *currentNode* contiene un riferimento al nodo che si sta espandendo;
- La variabile *queue* rappresenta la frontiera. Questa è implementata come una coda LIFO;
- La variabile *visitedNodes* contiene i riferimenti di tutti i nodi che sono stati esplorati durante la ricerca;
- La variabile *path* aggiunge i nodi esplorati in ordine di espansione;
- La variabile *parents* associa ad ogni nodo il nodo che l'ha generato.

Ad ogni iterazione si estrae dalla frontiera il nodo avente profondità maggiore nell'attuale albero di ricerca e si procede all'espansione. Se uno dei figli è un nodo obiettivo l'algoritmo restituisce il path calcolato. Se ad un certo punto della ricerca la frontiera risulta vuota, non esiste una soluzione e viene restituito *null*.

3.3 Ricerca informata

Gli algoritmi di ricerca informata sfruttano conoscenze specifiche del problema in esame per essere più efficienti.

L'approccio generale viene chiamato ricerca *best-first*: il nodo n che viene espanso per primo è quello più promettente, e viene scelto in base ad una **funzione di valutazione** *f(n)*.

La funzione di valutazione spesso fa uso di una **funzione euristica** *h(n)*. Questa funzione stima il costo del cammino più conveniente da n al nodo obiettivo.

3.3.1 Ricerca A*

Questo tipo di ricerca utilizza come funzione di valutazione $f(n) = g(n) + h(n)$ dove $g(n)$ rappresenta la funzione costo di cammino (la funzione che restituisce la distanza dal nodo iniziale ad n) e $h(n)$ rappresenta l'euristica adottata per il problema (la funzione che stima il costo del miglior cammino da n al nodo obiettivo).

L'euristica che è stata adottata è rappresentata dalla distanza tra due punti su piano cartesiano. Le coordinate che vengono prese in considerazione sono quelle dei punti di inizio del nodo n e del nodo obiettivo. Con punti di inizio si intendono i punti dai quali inizia il processo di disegno dei blocchi adottando la libreria *javax.swing*.

L'implementazione è identica a quella della Ricerca a Costo Uniforme. Ciò che differisce è la frontiera: questa non è più una coda ordinata per $g(n)$ ma per $f(n)$.

Nella classe Main, il seguente metodo implementa la Ricerca A*:

```
public static ReturningValues aStarSearch(Maze maze){
        Block node = maze.getStartBlock();
        node.setPathCost(0);

        Set<Block> explored = new HashSet<>();
        List<Block> path = new ArrayList<>();
        Map<Block,Block> parents = new HashMap<>();
        PriorityQueue<Block> frontier = new PriorityQueue<>(1000, (o1,
            o2) -> {
            Block endingBlock = maze.getEndBlock();

            double hnO1 =
                    Math.sqrt((endingBlock.getY() - o1.getY()) * (
                        endingBlock.getY() - o1.getY()) + (endingBlock.
                        getX() - o1.getX()) * (endingBlock.getX() - o1.
                        getX()));
            double hnO2 =
                    Math.sqrt((endingBlock.getY() - o2.getY()) * (
                        endingBlock.getY() - o2.getY()) + (endingBlock.
                        getX() - o2.getX()) * (endingBlock.getX() - o2.
                        getX()));

            if(o1.getPathCost()+hnO1 < o2.getPathCost()+hnO2){
                return -1;
            }
            else if(o1.getPathCost()+hnO1 > o2.getPathCost()+hnO2){
                return 1;
            }
            return 0;
        });

        frontier.add(node);

        while(true){

            if(frontier.isEmpty()){
                return null;
            }

            node = frontier.poll();
            path.add(node);
            if(node.getIsEnd()){
                return new ReturningValues(path, parents);
            }

            explored.add(node);

            List<Block> neighbors = node.getNeighbors();
            for (Block e: neighbors) {
                int actualPathCost = node.getPathCost()+1;
                e.setPathCost(actualPathCost);
                if(!explored.contains(e) && !frontier.contains(e)){
                    frontier.add(e);
                    parents.put(e, node);
                }
                else if(frontier.contains(e)){
                    if(e.getPathCost()>actualPathCost){
                        e.setPathCost(actualPathCost);
                        parents.put(e,node);
                    }
                }
            }
        }
    }
```

- La variabile *node* contiene un riferimento al nodo che si sta espandendo;
- La variabile *frontier* rappresenta la frontiera. Questa è implementata come una coda a priorità ordinata per *f(n)*;
- La variabile *explored* contiene i riferimenti di tutti i nodi che sono stati esplorati durante la ricerca;
- La variabile *path* aggiunge i nodi esplorati in ordine di espansione;
- La variabile *parents* associa ad ogni nodo il nodo che l'ha generato.

Ad ogni iterazione si estrae dalla frontiera il nodo n avente valore per *f(n)* minimo. Se questo è un nodo obiettivo l'algoritmo restituisce il path calcolato, altrimenti si procede all'espansione del nodo e all'eventuale aggiornamento di nodi già presenti all'interno della frontiera. Se ad un certo punto della ricerca la frontiera risulta vuota, non esiste una soluzione e viene restituito *null*.

Capitolo 4

Conclusioni

Nonostante il problema analizzato non sia di elevata difficoltà, sono soddisfatto del risultato. Questo progetto ha messo alla prova le conoscenze acquisite durante il corso riguardo gli algoritmi di ricerca. Aspetti che ritenevo chiari sono stati messi in discussione durante l'implementazione, forzandomi a rivedere ed approfondire questi algoritmi. È stata un'esperienza appagante e divertente.