Università di Perugia
Dipartimento di Matematica e Informatica

Artificial Intelligent Systems - Intelligent Models

Agent in a Maze

Professore
Prof. Alfredo Milani

Studente
Nicolò Vescera

Anno Accademico 2021-2022

Indice

1 Obiettivo

L'obiettivo di questa relazione è quello di realizzare un'implementazione di un **ambiente** di **Reinforcement Learning** per allenare un agente, tramite un algoritmo di **Q-Learning**, a muoversi in modo "*intelligente*" all'interno di un labirinto, ovvero l'agente deve essere in grado di non andare a sbattere contro le mura del labirinto e possibilmente arrivare all'uscita.

2 Ipotesi iniziali e vincoli

L'agente si troverà all'interno di un ambiente bidimensionale (griglia 2D) formato da svariate celle che possono essere di due tipi:

1. **Celle Vuote**: in questo tipo di celle l'agente può muoversi liberamente e rappresentano uno spazio vuoto/percorribile del labirinto
2. **Celle Muro**: queste celle rappresentano un muro del labirinto e l'agetne non può attraversarle o fermarcisi sopra, l'unica cosa che può fare è evitarle.

L'insieme di tutte le *Celle Vuote* e *Celle Muro* forma il **labirinto** che può essere visto come una matrice di dimensione $m \times n$, dove m è il numero di righe ed n il numero di colonne.

$$Maze_{m\times n} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \cdots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

Figura 1: Esempio di labirinto di dimensione $m \times n$

L'agente inizierà sempre dalla prima cella in alto a sinistra $a_{0,0}$ (*start position*) e dovrà raggiungere l'ultima cella in fondo a destra $a_{m-1,n-1}$ (*goal position*) muovendosi solo tra le celle a lui vicine (quelle visibili dal vicinato di *Von Neuman*) e mai in diagonale ! Non sarà neanche in grado di spostarsi in celle al di fuori di quelle della matrice e.g. $a_{-1,0}$ e non si verificherà *l'effetto pacman*, cioè l'agente non potrà effettuare mosse del tipo: $a_{1,n-1} \rightarrow a_{1,0}$

$$\begin{matrix} & a_{4,4} & \\ a_{5,3} & a_{5,4} & a_{5,5} \\ & a_{6,4} & \end{matrix}$$

Figura 2: Esempio di vicinato di Von Neuman dove l'agente si trova in posizione $5, 4$

L'ambiente, ad ogni mossa dell'agente, ritornerà *un'osservazione* formata dalle celle a lui vicine (questa volta con il vicinato di Moore) nella sua nuova posizione e un *reward* in base alla bontà dell'azione effettuata:

- -1: una qualunque mossa in una cella vuota
- -5: una mossa contro un muro (equivale a rimanere fermo sul posto)
- $+10$: raggiungere la cella goal

$$\begin{matrix} a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,3} & a_{5,4} & a_{5,5} \\ a_{6,3} & a_{6,4} & a_{6,5} \end{matrix}$$

Figura 3: Esempio di vicinato di Moore dove l'agente si trova in posizione $5, 4$

2.1 Vincoli dell'agente

In breve tutti i vincoli dell'agente:

- Può muoversi solo nelle **Celle Vuote**
- Non può oltrepassare le **Celle Muro** e muoversi contro una di queste equivale a rimanere fermo sul posto
- Non può uscire dalle mura del labirinto (non può andare fuori dalle celle della matrice e.g. $a_{-1,0}$)
- Non esiste *l'effetto pacman*
- Le azioni che può compiere sono: `UP, DOWN, LEFT, RIGHT`
- Può muoversi solo nelle celle a lui vicine e non in diagonale (vicinato di Von Neuman)

2.2 Vincoli dell'ambiente

In breve i vincoli dell'ambiente:

- L'ambiente è rappresentato tramite una griglia 2D (matrice)
- L'ambiente ha dimensione $m \times n$ dove m è il numero di righe ed n il numero di colonne.
- La *starting position* è sempre la prima cella in altro a sinistra: $a_{0,0}$
- La *goal position* è sempre l'ultima cella in fondo a destra: $a_{m-1,n-1}$
- Ritorna all'agente un'*osservazione* data dal vicinato di Moore della posizione dell'agente
- Ritorna un *reward* all'agente in base alla bontà della mossa:
  * -1: una qualunque mossa in una cella vuota
  * -5: una mossa contro un muro (equivale a rimanere fermo sul posto)
  * $+10$: raggiungere la cella goal

3 Struttura e file del progetto

Nota: Tutto il codice mostrato successivamente è disponibile nella seguente repo github: QRL-Maze_in_a_Wall.

La seguente è una breve anteprima di come si presenta la cartella del progetto:

`main.py`	*Script principale*
`MazeEnv.py`	*Rappresentazione dell'ambiente*
`QLearning.py`	*Algoritmo di QLearning*
`GUI.py`	*Responsabile della GUI*
`generate_dataset.py`	*Crea il dataset*
`training.py`	*Script per allenare l'agente*
`testing.py`	*Script per valutare l'agente*
`maze_utils.py`	*Funzioni utilities*
`maze.txt`	*File per rappresentare una labirinto*
`Qmatrix`	*Matrice per QLearning*

- Il file `MazeEnv.py`, che conterrà l'omonima classe, è il responsabile della modellazione dell'ambiente. Questo si occuperà di verificare le mosse dell'agente, di assegnare i reward e di ritornare a quest'ultimo le osservazioni dell'ambiente che lo circonda.
- `QLearning.py` ha il compito di gestire la *matrice* Q : salvarla su apposito file e caricarla in memoria, gestire la fase di allenamento (*training*) e di esecuzione (*testing/execution*). È qui che risiede l'algoritmo di QLearning vero e proprio.
- `GUI.py`, questa classe ha il compito di stampare a video il labirinto e le varie mosse dell'agente. Può farlo in due modi:
  1. **GUI**: tramite una finestra dove il labirinto viene rappresentato a colori
  2. **CLI**: direttamente nel terminale con soli caratteri testuali
- `generate_dataset.py` è responsabile della creazione del dataset: 100 labirinti generati in modo casuale con dimensione massima 10×10, minima di 3×3 ed un numero casuale di mura, per il *training* e 20, costruite allo stesso modo delle precedenti, per il *testing/execution*.
- `training.py` avvia la fase di *training* e permette all'utente di scegliere alcuni parametri come:
  - Numero di Epoche
  - Numero di Step
  - Se utilizzare il dataset di training o una matrice definita dall'utente

  - Se continuare l'allenamento (e quindi caricare la *matrice Q* già esistente andando a modificarla) o iniziarne uno nuovo daccapo (creare una nuova *matrice Q* e popolarla)

  In questa fase non è prevista la stampa *GUI* e se utilizzato il dataset di training, per ogni matrice verrà salvato un grafico che mostra il risultato dell'allenamento.

- `testing.py` permette all'utente di eseguire la fase di *testing/execution* dandogli la possibilità di scegliere:
  - se utilizzare la modalità `step-by-step` (possibilità di eseguire un'azione alla volta e anche di scegliere quale azione fare per ogni stato) o l'esecuzione automatica
  - se utilizzare la stampa *GUI* o *CLI*
  - se avviare l'esecuzione con una matrice generata dall'utente o con il dataset di testing
- `maze_utils.py` insieme di funzioni utili per la gestione del labirinto, come la generazione casuale di quest'ultimi, la possibilità di salvare un labirinto su file o di caricarlo in memoria.
- `main.py` punto iniziale per avviare l'intero progetto, permette all'utente di scegliere quale parte del progetto avviare e testare.
- `maze.txt` file che contiene una matrice definita dall'utente. Tutte le celle delle matrici sono popolate da `0` o `1`, che rispettivamente indicano una **Cella Vuota** e una **Cella Muro**.
- `Qmatrix` la matrice creata durante la fase di addestramento.

3.1 maze.txt

Prima di iniziare ad analizzare il codice in se, è necessario dare uno sguardo a come i labirinti vengono rappresentati. Sia il file `maze.txt` (che viene creato e popolato dall'utente), sia i file generati dallo script per la creazione del dataset, hanno lo stesso formato: un insieme di `0` e `1` separati da spazi disposti su più righe. Questi numeri indicano rispettivamente: una **Cella Vuota** e una **Cella Muro**. Di seguito alcuni esempi di labirinti validi e non:

```
0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1
0 0 1 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0
```

Listing 1: Labirinto 8x9 valido

```
0 0 0 0 0 0 0
0 1 0 1 0 0 0
0 0 0 1 0 0 1
0 0 1 0 1 0 0
0 0 0 0 0 0 0
```

Listing 2: Labirinto 5x7 valido

```
0 1 0 0
1 1 0 0
0 0 0 0
0 1 0 0

```

Listing 3: Labirinto 4x4 NON valido

```
0 0 0 0 0
1 1 0 0 0
0 0 0 1 1
0 1 0 1 0

```

Listing 4: Labirinto 4x5 NON valido

I labirinti 1 e 2 risultano validi in quanto l'agente (che parte dalla prima cella in alto a sinistra) riesce sempre a raggiungere lo stato goal (ultima cella in basso a destra), mentre nel labirinto 3 l'agente risulta bloccato alla partenza ed in nessuno modo riuscirà ad uscire dalla sua "prigione". Stessa cosa vale per il 4 dato che non riuscirà mai a raggiungere il goal.

3.2 maze_utils.py

All'interno di questo file sono presenti alcune funzioni "*utilities*" per la generazione, controllo e caricamento dei labirinti.

3.2.1 is_solvable

È proprio qui che le varie matrici vengono controllate e ottengono lo stato di *valide/risolvibili* (*solvable*). Questo controllo viene fatto tramite un'esplorazione in ampiezza (*BFS*) che percorre tutto il labirinto e ritorna `True` se l'agente riesce ad arrivare allo stato goal, altrimenti `False`. Questo controllo non intacca di molto le performance dell'algoritmo in quanto anche su matrici 10×10 il suo tempo di completamento non è minimamente rilevante:

	3×3	6×6	10×10	50×50	100×100
Tempo (s)	0.0	0.0	0.0	0.0	0.0

Figura 4: Tempi di esecuzione della funzione `is_solvable()` con matrici di diversa dimensione.

Il costo in tempo è $O(mn)$:

$$Tcosto = O(V + E) = O(mn + mn - n + mn - m) = O(3mn - m - n) = O(mn)$$

dove :

- $V = mn$
- $E = (n)(m-1) + (m)(n-1)$

Anche il costo in spazio è identico in quanto la lista `seen` sarà grande al più $m \times n$.

```python
def is_solvable(maze) -> bool:
    x, y = maze.shape

    start_position = (0, 0)
    end_position = (x-1, y-1)

    seen = {start_position}     # elementi visitati
    queue = [start_position]    # coda di elementi da visitare

    while queue:
        i, j = queue.pop(0)
        seen.add((i, j))

        for di, dj in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            ni, nj = i+di, j+dj   # nuova posizione del vicino

            if (ni, nj) in seen:
                continue  # se e' gia' stato visitato

            elif ni < 0 or nj < 0 or ni >= len(maze) or nj >= len(maze[0]):
                continue  # se e' fuori dalla matrice

            elif ni == end_position[0] and nj == end_position[1]:
                return True   # se e' arrivato alla fine

            elif maze[ni][nj] == 1:
                continue  # se e' un muro

            else:   # se e' una casella libera
                seen.add((ni, nj))      # aggiunge alla lista visitati
                queue.append((ni, nj))  # aggiunge alla coda da visitare

    return False
```

Listing 5: Funzione per controllare la risolvibilità dei labirinti

3.2.2 grid_from_file

La funzione che si occupa di caricare in memoria un labirinto da file risulta essere abbastanza banale: tenta di aprire il file `maze.txt`, se non esiste o il suo contenuto è errato o il labirinto al suo interno non è risolvibile ritorna `None` ed un messaggio di errore, altrimenti ritorna la matrice sotto forma di `numpy.ndarray`.

```
def grid_from_file(filename: str) -> tuple[np.ndarray, str]:
    grid = None
    message = ""

    try:
        grid = np.loadtxt(filename, dtype=int)
    except FileNotFoundError:
        grid = None
        message = "File inesistente !"
        return grid, message

    # il file esiste ma risulta vuoto
    if grid is not None and grid.size == 0:
        grid = None
        message = "Il contenuto del file e' errato !"

    elif not is_solvable(grid):
        grid = None
        message = "Il labirinto non e' risolvibile !"

    return grid, message
```

Listing 6: Funzione per il caricamento in memoria di un labirinto da file

3.2.3 generate_grid

La funzione `generate_grid` riesce a creare labirinti, di dimensione e numero di mura date, sempre risolvibili. Ci riesce in quanto controlla sempre se sta generando un muro in una posizione che non dovrebbe (cella *start* e *goal*) (nel caso non crea il muro e lo rigenera casualmente) ed il numero massimo di mura inseribili è pari a:

$$maxWalls = \frac{m * n}{2}$$

Superando $maxWalls$ si potranno andare a generare barriere di mura insuperabili dall'agente, ma restando sotto questo limite è sempre certo che questo fenomeno non accadrà mai.

```python
def generate_grid(m, n, walls=10) -> np.ndarray:
    if walls >= (m * n / 2):
        print("Non e' possibile generare un labirinto esplorabile !!!")
        return None

    solvable = False
    grid = None
    while not solvable:
        grid = np.zeros((m, n), dtype=int)  # matrice piena di 0

        for _ in range(walls):
            # genera a caso una posizione dove inserire il muro
            x = randint(0, m - 1)
            y = randint(0, n - 1)

            # se la cella scelta e' il goal o lo stato iniziale la rigenera
            while (x == 0 and y == 0) or (x == (m - 1) and y == (n - 1)):
                x = randint(0, m - 1)
                y = randint(0, n - 1)

            grid[x][y] = 1  # aggiunge il muro
        solvable = is_solvable(grid)

    return grid
```

Listing 7: Funzione per la generazione di labirinti casuali

3.3 MazeEnv.py

```python
class MazeEnv(object):
    def __init__(self, grid):
    def is_goal(self, cell: int) -> bool:
    def get_agent_row_column(self, new_cell: int = None):
    def moore(self, next_position: int = None) -> np.array:
    def off_grid_move(self, new_cell: int) -> bool:
    def is_wall_colliding(self, cell) -> bool:
    def state_to_int(self, state: np.array) -> int:
    def calculate_next_state(self, action, next_position) -> int:
    def step(self, action: str) -> tuple[int, int, bool, object]:
    def reset(self) -> int:
    def render(self, gui=True) -> bool:
    def actionSpaceSample(self):

```

Listing 8: Metodi di MazeEnv.py

Il precedente è un breve sunto di tutti i metodi presenti all'interno della classe `MazeEnv`. In questa sezione andremo ad analizzare solo quelli più importanti e significativi per ovvie ragioni di spazio e tempo.

3.3.1 Costruttore

```
1     def __init__(self, grid: np.ndarray):
2       self.grid = grid
3       self.m, self.n = self.grid.shape    # dimensione matrice
4
5       # definizione cella goal
6       self.goal = (self.m * self.n) - 1
7
8       # definizione spazio degli stati e goal
9       self.stateSpace = [i for i in range(2**8)]
10      self.stateGoals = [self.goal]
11
12      # definizione delle azioni possibili e spazio delle azioni
13      self.actionSpace = {'U': -self.n, 'D': self.n, 'L': -1, 'R': 1}
14      self.possibleActions = self.actionSpace.keys()
15
16      # posizione dell'agente e stato iniziale
17      self.agentPosition = 0
18
19      # init del reward
20      self.reward = 0
21
22      # inizializzo l'oggetto responsabile della GUI
23      self.gui = None
```

Listing 9: Costruttore di MazeEnv.py

Il costruttore di questa classe prende come parametro un `numpy.ndarray` che rappresenta il labirinto, definisce il goal (riga 6), la posizione iniziale dell'agente (riga 17-18), le possibili azioni che l'agente può compiere (riga 13-14), lo *spazio degli stati* (riga 9) ed utilizza un'inizalizzazione lazy (lo crea quando viene richiesto per la prima volta) dell'oggetto `gui` responsabile della stampa a video del labirinto (riga 24).

Lo *spazio degli stati* è rappresentato da un array monodimensionale `stateSpace` di dimensione pari a $2^8 = 256$ (quindi da 0 fino a 255). Questo numero deriva da tutte le possibili osservazioni che l'ambiente può ritornare all'agente:

$$\begin{matrix} 0 & 0 & 0 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 1 & 0 & 0 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 0 & 1 & 0 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 0 & 0 & 1 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix}$$

$$\begin{matrix} 1 & 1 & 1 \\ 1 & A & 1 \\ 1 & 1 & 1 \end{matrix} \qquad \begin{matrix} 1 & 1 & 0 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 1 & 1 & 1 \\ 0 & A & 0 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} 1 & 1 & 1 \\ 1 & A & 0 \\ 0 & 0 & 0 \end{matrix}$$

Figura 5: Alcune delle possibili osservazioni

Il numero totale di stati possibili risulta quindi essere dato da:

$$maxStateNumber = (possibili\ elementi\ di\ ogni\ cella)^{numero\ totale\ di\ celle}$$

dove :

- $possibili\ elementi\ di\ ogni\ cella = \begin{cases} 0, & se\ CellaVuota \\ 1, & se\ CellaMuro \end{cases}$ quindi 2.
- $numero\ totale\ di\ celle = 8$, viene esclusa la cella dell'agente in quanto risulta superflua ed andrebbe ad aumentare notevolmente il numero degli stati.

Le *posizione* dell'agente non è data dalla normale coppia `(x, y)` di coordinate ma da un numero identificativo univoco di ogni cella. La matrice viene vista come un array monodimensionale (quindi risulta che tutte le righe della matrice sono messe una vicino all'altra) ed ogni cella viene individuata tramite l'indice che avrà in questo array immaginario.

0	1	2
3	4	5
6	7	8

0	1	2	3	4	5	6	7	8

Figura 6: Esempio di una matrice 3×3 (a sinistra) e il suo relativo array immaginario (a destra)

Così facendo, operazioni come controllare se l'agente tenta di uscire dalla matrice risulteranno molto più semplici da implementare. È altresì facile ottenere le coordinate `(x, y)` partendo dall'identificativo della calla (vedi sottosottosezione 3.3.2).

Il dizionario `actionSpace` rappresenta tutte le possibili azioni che l'agente può compiere con il relativo valore da aggiungere alla sua attuale posizione (identificativo univoco della cella) per raggiungere la nuova. Per esempio, tenendo in considerazione la Figura 6, se dalla cella 0 volessimo eseguire l'azione `D`, dobbiamo effettuare il seguente calcolo: $0 + n = 0 + 3 = 3$. Ovviamente azioni come: dalla cella 2 eseguo l'azione `R`, non saranno permesse per i vincoli descritti prima (vedi sottosezione 2.2).

3.3.2 get_agent_row_column

Questo metodo è in grado, tramite calcoli banali, di ritornare le coordinate `(x, y)` di una cella dato il suo identificativo univoco.

```
1  def get_agent_row_column(self, new_cell: int = None):
2    if new_state:
3      x = int(new_cell / self.n)
4      y = new_cell % self.n
5
6    else:
7      x = int(self.agentPosition / self.n)
8      y = self.agentPosition % self.n
9
10   return x, y
```

Listing 10: Metodo che converte identificativo a corrdinate

3.3.3 moore

Qui viene calcolato il vicinato di Moore partendo dalla posizione dell'agente o da una data posizione. Se l'agente si trova vicino ad un bordo della matrice, le celle che fuoriuscirebbero vengono viste come mura:

$$\begin{matrix} A & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \qquad\qquad \begin{matrix} 1 & 1 & 1 \\ 1 & A & 0 \\ 1 & 0 & 1 \end{matrix}$$

Figura 7: Esempio di calcolo del vicinato di Moore quando l'agetne si trova ai bordi della matrice. A sinistra la matrice, a destra il vicinato

È importante notare che il vicinato di Moore viene ritornato da questo metodo come un array e non come una matrice ! Questo array è formato da tutti i vicini dell'agente tranne se stesso:

$$\begin{matrix} 1 & 1 & 1 \\ 1 & A & 0 \\ 1 & 0 & 1 \end{matrix} \qquad\qquad 1 \; 1 \; 1 \; 1 \; 0 \; 1 \; 0 \; 1$$

Figura 8: Esempio di calcolo del vicinato di Moore restituito da questo metodo. A sinistra il vicinato sotto forma di matrice, a destra l'effettivo valore di ritorno

```python
def moore(self, next_position: int = None) -> np.array:
  # prende x,y per calcolare il vicinato
  if next_position is not None:
    x, y = self.get_agent_row_column(next_position)
  else:
    x, y = self.get_agent_row_column()

  vicini = []

  for move in [(-1, -1),(-1, 0),(-1, 1),(0, -1),(0, 1),(1, -1),(1, 0),(1, 1)]:
    new_x, new_y = x + move[0], y + move[1]      # nuova posizione da osservare
    to_add = 0

    if new_x < 0:
      to_add = 1
    elif new_y < 0:
      to_add = 1
    elif new_x > self.m - 1:
      to_add = 1
    elif new_y > self.n - 1:
      to_add = 1
    else:
      to_add = self.grid[new_x][new_y]

    vicini.append(to_add)

  return np.array(vicini)
```

Listing 11: Metodo che calcola il vicinato di Moore

3.3.4 off_grid_move

Questo metodo controlla se la cella data (rappresenta la nuova cella in cui l'agente vuole spostarsi) è valida e quindi l'agente non sta tentando di uscire dalle mura del labirinto. Ricordo che tentare di uscire dalle mura equivale a sbattere contro un muro.

```python
def off_grid_move(self, new_cell: int) -> bool:
  if new_cell not in range(self.n * self.m):
    return True

  else:
    return False
```

Listing 12: Metodo per controllare l'uscita dalla matrice

3.3.5 state_to_int

Qui viene fatta la conversione tra vicinato di Moore (sotto forma di array) e stato/osservazione che l'ambiente andrà a ritornare all'agente. Questa associazione è semplice: ogni vicinato viene visto come un numero binario che deve essere convertito in decimale. Questo numero decimale rappresenta infine lo stato/osservazione.

Figura 9: Esempio di conversione da Moore a stato/osservazione. A sinistra il vicinato di Moore, a destra lo stato equivalente

```
1  def moore_to_state(self, state: np.array) -> int:
2
3      return int("".join([str(x) for x in state]), 2)
4
```

Listing 13: Codice del metodo moore_to_state

3.3.6 calculate_next_state

Questo metodo calcola lo stato successivo dell'agente, controllando la validità delle mosse e assegnando il giusto *reward* ad ogni azione.
La condizione alla riga 8 serve per capire se l'agente sta tentando di uscire dalle mura del labirinto e nel caso, questa azione viene trattata come un normale impatto contro un muro. Nelle righe 12 - 40 controlla la validità della mossa selezionata e, in caso risulti essere una collisione, imposta il reward a -5 (riga 21) e come risultato di ritorno lo stato attuale dell'agente (riga 20 e 22) impedendogli di muoversi. Alla riga 14 controlla se l'agente ha raggiunto lo stato goal e quindi assegna il reward +10 altrimenti -1. Infine restituisce il nuovo stato utilizzando il metodo `state_to_int()` (vedi sottosottosezione 3.3.5).

```
1  def calculate_next_state(self, action, next_position) -> int:
2      state = self.moore()            # ottengo lo stato attuale dell'agente
3
4      to_return = None
5      update_agent_position = True
6
7      # controlla se si e' spostato fuori dalla griglia
8      if self.off_grid_move(next_position):
9          self.reward = -5
10         to_return = state
11
12     else:
13         # reward e risultato uguali per tutti se non impatta contro un muro
14         self.reward = 10 if self.is_goal(next_position) else -1
```

```
15      to_return = self.moore(next_position)
16
17      # controllo se impatta contro un muro
18      if action == 'U':
19        if self.is_wall_colliding(state[1]):
20          update_agent_position = False
21          self.reward = -5
22          to_return = state
23      elif action == 'D':
24        if self.is_wall_colliding(state[6]):
25          update_agent_position = False
26          self.reward = -5
27          to_return = state
28      elif action == 'L':
29        if self.is_wall_colliding(state[3]):
30          update_agent_position = False
31          self.reward = -5
32          to_return = state
33      elif action == 'R':
34        if self.is_wall_colliding(state[4]):
35          update_agent_position = False
36          self.reward = -5
37          to_return = state
38
39      if update_agent_position:
40        self.agentPosition = next_position
41
42    return self.state_to_int(to_return)
```

Listing 14: Codice del metodo calculate_next_state

3.3.7 step

Questo è il metodo vero e proprio che esegue una data azione. Calcola la nuova cella (id) in cui l'agente si troverà, verifica la validità della mossa e calcola il nuovo stato dell'agente (righe 1 - 4) ritornando l'osservazione, il reward ed alcune info utili per il debug (riga 10).

```
1  def step(self, action: str) -> tuple[int, int, bool, object]:
2    # calcola la nuova posizione dell'agente
3    resulting_position = self.agentPosition + self.actionSpace[action]
4    next_int_state = self.calculate_next_state(action, resulting_position)
5
6    actual_state = self.moore()
7    info = {str(actual_state), str(self.state_to_int(actual_state))}
8
9    # effettua la mossa
10   return next_int_state, self.reward, self.is_goal(resulting_position), info
```

Listing 15: Codice del metodo step

3.4 QLearning.py

```
class QLearning(object):
  def __init__(self, env):
  def maxAction(self, Q, state, actions: list[str]) -> str:
  def saveQ(self, Q, fileName):
  def loadQ(self, fileName: str) -> dict[tuple[int, str], float]:
  def execute(self, step_by_step=False, sleep_time=0.5, maxk=30, gui=True):
  def training(self,
               epochs=50000, steps=200, ALPHA=0.1, GAMMA=1.0, EPS=1.0,
               plot=True, resume=False, plot_name: str = None):

```

Listing 16: Metodi di QLearning.py

La classe `QLearning` ha il compito di gestire la fase di *training* e *testing/execution*, implementa l'algoritmo di apprendimento basato su QLearning e gestisce la *matrice Q*.

3.4.1 Matrice Q

Questa matrice è formata da tante righe quanti sono gli stati (quindi 256) e tante colonne quante il numero di azioni (quindi 4).

$$QMatrix = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \vdots & \vdots & \vdots & \vdots \\ a_{255,0} & a_{255,1} & a_{255,2} & a_{255,3} \end{bmatrix}$$

Figura 10: Definizione di matrice Q

In ogni cella è contenuto un valore decimale (e.g. -2.0123451434) che sta ad indicare la bontà di una determinata azione al determinato stato. Per esempio, prendiamo in considerazione la seguente riga della matrice:

$$Qmatrix_{[140,]} = \begin{bmatrix} -5.1234569284 & -5.24234569284 & 7.6204562214 & -2.0204169484 \end{bmatrix}$$

Figura 11: Esempio di una possibile riga della matrice Q

Il primo elemento rappresenta la qualità dell'azione `UP` allo stato 140, il secondo l'azione `DOWN`, il terzo `LEFT` ed il quarto `RIGHT`. È facile intuire che in questo stato l'azione migliore da scegliere è `LEFT` con un valore di 7.6204562214.

Infine è doveroso notare che non tutti gli stati di questa matrice saranno utilizzati, un esempio è lo stato 255 che non si potrà mai verificare per le condizioni descritte in sottosezione 2.2.

$$1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1 \qquad\qquad \begin{matrix} 1 & 1 & 1 \\ 1 & A & 1 \\ 1 & 1 & 1 \end{matrix}$$

Figura 12: Rappresentazione di uno stato inutilizzato della matrice Q (stato 255). A sinistra lo stato sotto forma di array, a destra la visualizzazione a matrice del rispettivo stato.

3.4.2 training

Questo metodo si occupa della gestione della fase di *training*. È possibile specificare i parametri di allenamento oppure verranno utilizzati quelli di default:

- **epochs**: numero di volte che andrà ad analizzare il labirinto
- **steps**: massimo numero di azioni che l'agente può compiere per epoca
- **alpha**: learning rate
- **gamma**: discount factor
- **eps**: epsilon-greedy

Per prima cosa viene inizializzata la matrice Q con tutti 0 (righe 3 - 6), poi viene mostrato a video il labirinto che si sta esplorando (righe 8 - 9) ed infine viene effettuato l'allenamento (righe 11 - 49). Durante questa parte, l'agente inizialmente tende ad esplorare (*exploration phase*) compiendo azioni casuali per vedere "cosa succede", poi, con l'esperienza acquisita, tenderà a convergere sempre nello stato goal (se riesce a trovarlo) passando quindi all'*exploitation phase*. Il tutto è possibile grazie alle righe 38 - 41 che vanno a variare il parametro `eps` al completamento di ogni epoca. Questa "esperienza" si traduce in pratica con l'aggiornamento dei valori di ogni cella della matrice Q (righe 32 - 34).

Al termine dell'allenamento viene mostrato a video un grafico riassuntivo con tutti i punteggi totalizzati (somma di tutti i reward ottenuti) per ogni epoca.

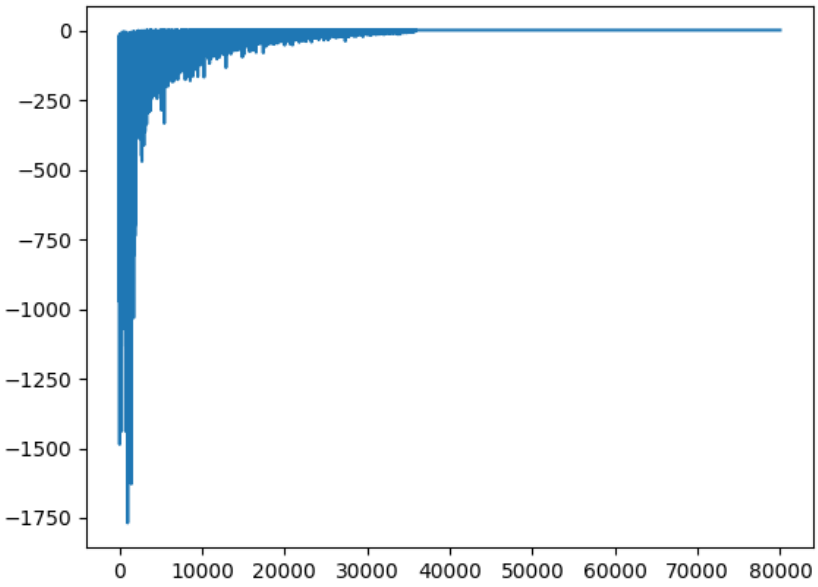


Figura 13: Esempio di grafico ottenuto dopo la fase di training (80000 epoche, 1500 step). Sull'asse x il numero di epoche, sull'asse y il reward totale per epoca.

Dal precedente grafico possiamo notare come nel primo periodo di tempo (da 0 a 40000 epoche) vi è la fase di *exploration*, in quanto l'agente sta esplorando il labirinto con azioni casuali (molte di queste comportano l'impatto contro un muro, perciò inizialmente il reward totale è estremamente basso), successivamente (da 40000 a 80000 epoche) possiamo notare come l'agente è riuscito a trovare la via di fuga e converge sempre alla soluzione passando quindi alla fase di *exploitation*.

```
def training(self, epochs=50000, steps=200, ALPHA=0.1, GAMMA=1.0, EPS=1.0):
  # inizializza la matrice Q
  Q = {}
  for state in self.env.stateSpace:
    for action in self.env.possibleActions:
      Q[state, action] = 0

  self.env.reset()
  self.env.render(gui=False)

  totalRewards = np.zeros(epochs)
  for i in range(epochs):
    if i % int(epochs / 10) == 0:
      print('starting game ', i)

    done = False
    epRewards = 0
    numActions = 0
    observation = self.env.reset()
    while not done and numActions <= steps:
      rand = np.random.random()

      if rand < (1 - EPS):
        action = self.maxAction(Q, observation, self.env.possibleActions)
      else:
        action = self.env.actionSpaceSample()

      observationNext, reward, done, info = self.env.step(action)
      numActions += 1
      epRewards += reward
      actionNext = self.maxAction(Q,observationNext,self.env.possibleActions)
      Q[observation, action] = Q[observation, action] + ALPHA * (reward +
                          GAMMA * Q[observationNext, actionNext] -
                              Q[observation, action])
        observation = observationNext

    # permettere il cambio tra exploration ed exploitation
    if EPS - 2 / epochs > 0:
      EPS -= 2 / epochs
    else:
      EPS = 0

    totalRewards[i] = epRewards

  if plot:
    plt.plot(totalRewards)
    plt.show()

  self.saveQ(Q, "Qmatrix")

```

3.4.3 execute

```
1  def execute(self,step_by_step=False,sleep_time=0.5,maxk=30,gui=True): -> bool
2    self.env.reset()
3    self.env.render(gui=gui)
4    Q = self.loadQ("Qmatrix")
5    ...
6    # esecuzione automatica
7    totReward = 0
8    alive = True
9    action_counter = 0
10   while action_counter < maxk:
11     action = self.maxAction(
12       Q,
13       self.env.state_to_int(self.env.moore()),
14       self.env.possibleActions
15     )
16
17     observationNext, reward, done, info = self.env.step(action)
18     totReward += reward
19     action_counter += 1
20
21     print(f"Action: {action} Reward: {reward}\n")
22     self.env.render(gui=gui)
23
24     if done:
25       print(f"Return: {totReward}")
26       return True
27
28     sleep(sleep_time)  # tempo di attesa per visualizzare lo stato successivo
29     system("clear")
30
31   return False
32
```

Listing 17: Parte di codice del metodo execute

Il metodo `execute` gestisce la fase di *testing/execution* permettendo di visualizzare a schermo il comportamento dell'agente in un dato labirinto. La parte più interessante di questa funzione è quella riportata sopra: possiamo notare come inizialmente viene preparato l'ambente (righe 1 - 4) e caricata la *matrice* Q per poi passare alla fase di esecuzione vera e propria (righe 10 - 30). Qui notiamo come ad ogni iterazione viene scelta la miglior mossa (vedi sottosottosezione 3.4.4) da effettuare nello stato attuale (righe 11 - 15), la quale viene eseguita ottenendo l'osservazione e il reward (riga 17) dall'ambiente, viene mostrata a video (riga 22) ed infine si controlla se l'agente è arrivato allo stato goal e nel caso, l'esecuzione viene interrotta (righe 23 - 25). Possiamo notare come l'agente ha a disposizione un massimo di `maxk` azioni per trovare l'uscita, dopo le quali l'esplorazione terminerà e verrà considerata come fallita (riga 10). Questo metodo ritorna `True` se l'agente è riuscito a trovare la via di fuga, altrimenti `False`.

La rappresentazione della mossa può essere di due tipi: **CLI** o **GUI**.
Nel primo caso verrà stampata la matrice nel terminale, nell'altro una finestra di `pygame` verrà mostrata. Di seguito un esempio:

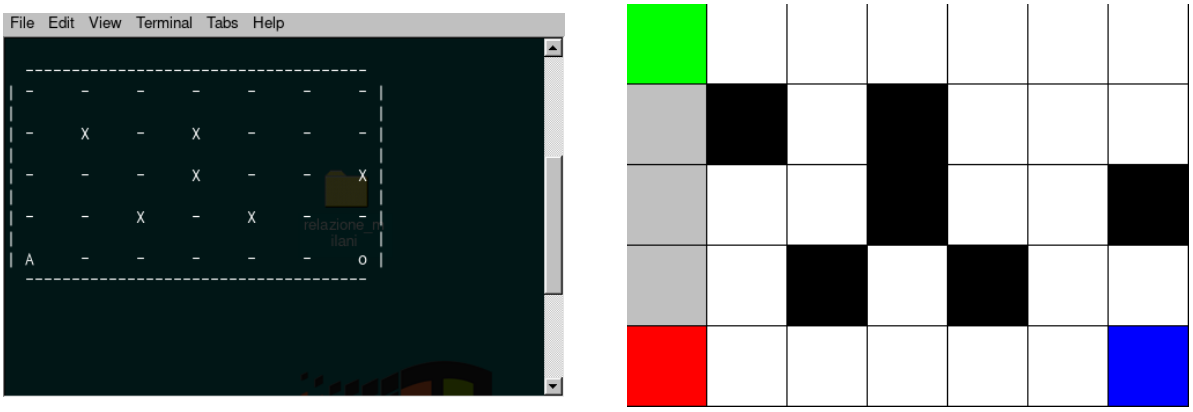


Figura 14: Esempio di otuput CLI (a sinistra) e GUI (a destra) della stessa matrice con 4 azioni `D`

3.4.4 maxAction

Questo metodo si occupa di scegliere la migliore azione da compiere in base alla posizione dell'agente (riga 3). Questa scelta viene effettuata trovando il massimo tra i valori della riga (della *matrice* Q) relativa allo stato attuale in cui l'agente si trova (vedi sottosottosezione 3.4.1).

```
1 def maxAction(self, Q, state, actions: list[str]) -> str:
2   values = np.array([Q[state, a] for a in actions])
3   action = np.argmax(values)
4
5   return actions[action]
6
```

Listing 18: Codice del metodo maxAction

3.5 GUI.py

All'interno di questo file risiede la classe che si occupa della stampa a video dei labirinti. Non andremo ad analizzare il codice ma vedremo solo come vengono rappresentati e cosa indica ogni elemento.

3.5.1 CLI

Questa modalità di stampa mostra la matrice nel terminale. Gli elementi che compongono questo tipo di rappresentazione sono:

- `-` : indica uno spazio vuoto e percorribile dall'agente (da non confondersi con la cinta di mura che delimita il labirinto)
- `X`: indica un muro
- `A`: indica dove si trova l'agente
- `o`: indica l'uscita del labirinto
- `------`: cinta di mura superiore ed inferiore che delimita il labirinto
- |: cinta di mura laterali che delimitano il labirinto

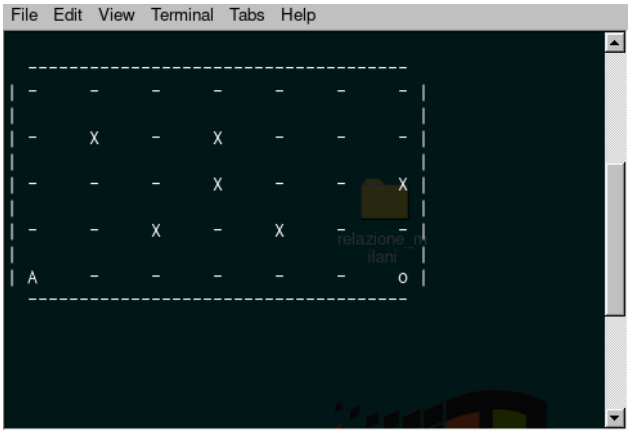


Figura 15: Esempio di rappresentazione CLI di un labirinto.

3.5.2 GUI

Questo tipo di stampa mostra a schermo una finestra di `pygmae` con all'interno una griglia che rappresenta il labirinto. Ogni cella ha un diverso colore che sta ad indicare:

- verde (■): la posizione di partenza dell'agente
- rosso (■): la posizione attuale dell'agente
- bianco ( ): una cella vuota
- blu (■): l'uscita del labirinto
- grigio (■): il cammino effettuato dall'agente
- nero (■): un muro

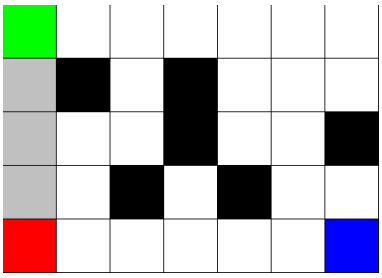


Figura 16: Esempio di rappresentazione GUI di un labirinto.

3.6 training.py

In questo script è presente la funzione che permette all'utente di scegliere i parametri di *training* e dare inizio all'allenamento. È possibile allenare l'agente utilizzando una sola matrice definita dall'utente o utilizzare l'intero dataset di training. C'è la possibilità di iniziare un nuovo allenamento (andando a ricreare la matrice Q) o riprenderne uno già avviato in precedenza (andando a caricare una matrice Q già esistente).

```
...
print(f"Avvio training sul dataset: {dataset}")

QL = QLearning(None)

# ottengo tutti i file nella cartella del dataset
filenames = next(walk(dataset), (None, None, []))[2]
for file in filenames:
  grid, message = grid_from_file(f"{dataset}/{file}")

  if grid is None:
    print(f"Errore in {file}: {message}")

  print(f"Caricato labirinto {grid.shape}")

  env = MazeEnv(grid=grid)
  QL.env = env
  QL.training(
      epochs=epochs, steps=steps,
      ALPHA=alpha, GAMMA=gamma, EPS=eps,
      plot=False, resume=resume, plot_name=file
  )

```

Listing 19: Parte di codice della funzione che avvia la fase di allenamento utilizzando il dataset creato in precedenza

3.7 testing.py

Qui risiede la funzione per avviare la fase di *testing/execution* facendo scegliere all'utente i vari parametri come: avviare la modalità `step-by-step` o quella automatica, se testare l'agente su un labirinto definito dall'utente o su tutti quelli presenti nel dataset di testing, ecc. Una funzione degna di nota è `evaluate()` che permette di "stimare" la precisione dell'agente in un modo abbastanza rudimentale:

- Testa l'agente su ogni labirinto nel dataset di testing.
- Conta quanti labirinti l'agente è stato in grado di risolvere
- Calcola la precisione dell'agente: $^{labirinti\ risolti}/_{labirinti\ totali}$
- Ripete questa procedura per 5 volte ottenendo una media finale

Questo indice permetterà in futuro di confrontare diversi allenamenti (differenti matrici Q) tra di loro e capire quindi qual è il migliore.

```
def evaluate():
  def _evaluate():
    QL = QLearning(None)
    tot = 0
    rew = 0
    filenames = next(walk(dataset), (None, None, []))[2]
    for file in filenames:
      grid, message = grid_from_file(f"{dataset}/{file}")
      env = MazeEnv(grid=grid)
      QL.env = env

      result,reward = QL.execute(step_by_step=False,sleep_time=0.0,gui=False)
      tot += result
      rew += reward
    return tot/len(filenames), rew

  avg_acc = 0
  avg_rew = 0
  max_step = 5

  for _ in range(max_step):
    acc, rew = _evaluate()
    avg_acc += acc
    avg_rew += rew

  avg_acc /= max_step
  avg_rew /= max_step

  print(f"Acc avg: {avg_acc} Rew avg: {avg_rew}")

```

Listing 20: Funzione per la valutazione dell'apprendimento

4 Svolgimento

In questa sezione andrò a descrivere i vari esperimenti, problemi e soluzioni, con anche parti di codice, che ho incontrato durante la realizzazione di questo progetto.

4.1 Primo allenamento

Come primo allenamento ho deciso di utilizzare un labirinto tra quelli presenti nel dataset di training. Ho scelto uno dei più grandi per permettere all'agente di esplorare uno spazio abbastanza ampio in modo da poter riempire più righe della matrice Q possibile. Ho scelto `matrice_69`, una 9×9 con una disposizione delle mura abbastanza complessa.

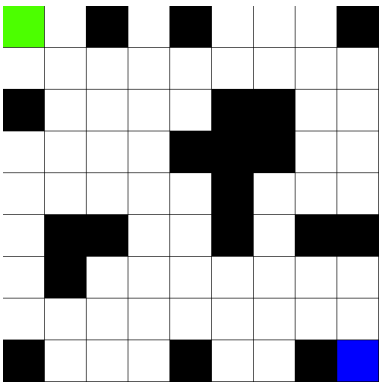


Figura 17: Rappresentazione grafica del labirinto scelto per il primo allenamento.

Ho quindi deciso di allenare l'agente con i seguenti parametri:

Epoche: 50k | **Step**: 400 | **Alpha**: 0.1 | **Gamma**: 1.0 | **EPS**: 0.9

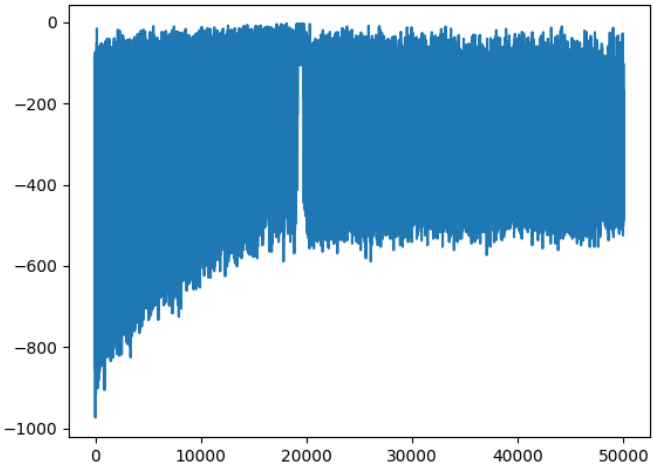


Figura 18: Grafico del risultato del primo allenamento.

Dal grafico sopra mostrato è evidente che l'agente non è riuscito a trovare l'uscita per via dell'insufficiente numero di step ed epoche utilizzato. Anche dalla fase di esecuzione, utilizzando sia la stessa matrice della fase di training che una mai esplorata, si vede chiaramente che l'agente non è in grado di uscire dal labirinto. Nel primo caso l'agente rimane bloccato in un loop di azioni infinito (`L R`), nel secondo impatta sempre contro un muro.

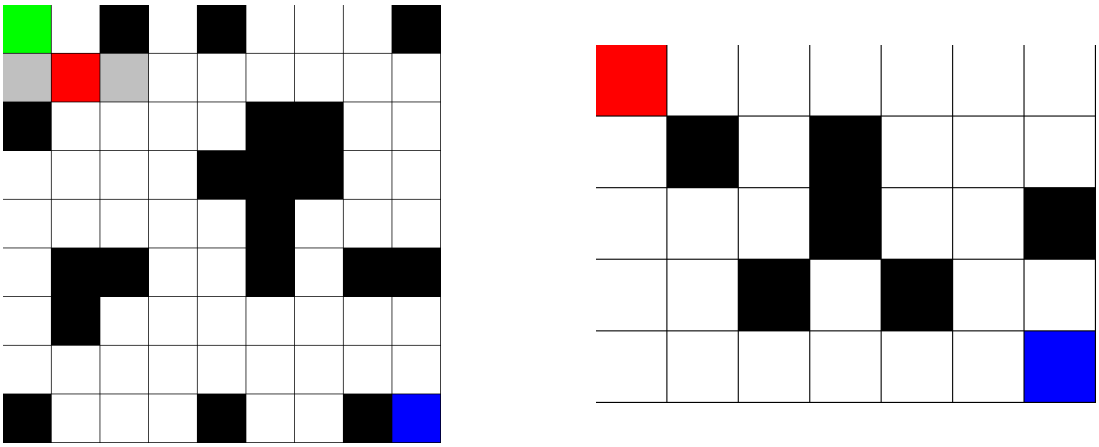


Figura 19: A sinistra l'agente rimane bloccato tra due azioni, a destra l'agente impatta sempre contro il muro.

Inoltre, analizzando meglio la *matrice* Q ed il codice in `QLearning.py` sorge un ulteriore problema: quando l'agente si trova un uno stato mai visto esegue sempre la prima azione (`U`)! Questo per via da della riga 3 della funzione `maxAction` (vedi sottosottosezione 3.4.4), `np.argmax()` sceglie il massimo tra gli elementi in input e nel caso ce ne fossero 2 o più uguali, prende sempre il primo che trova.

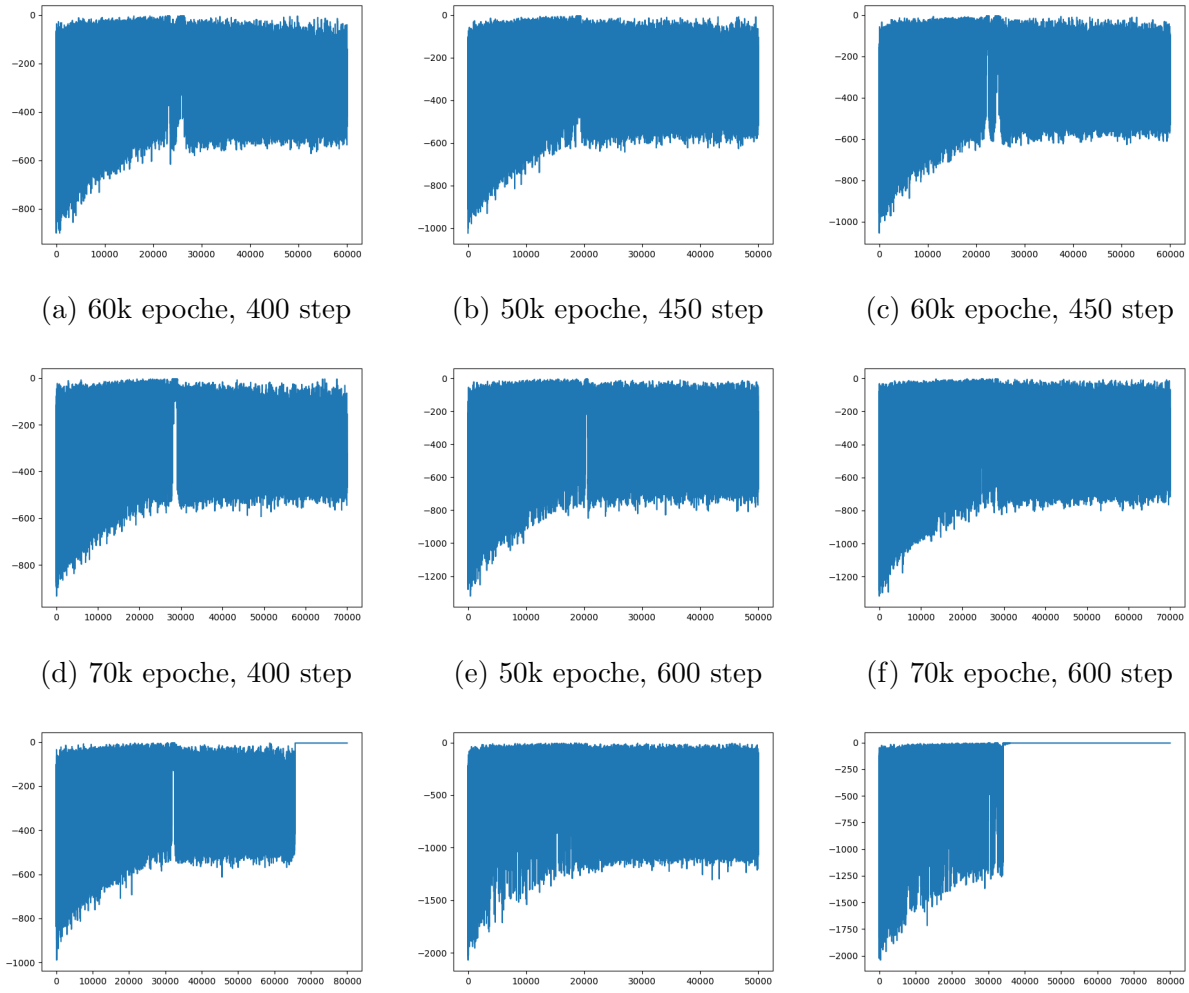
4.1.1 Problema 1: convergenza

Per un labirinto 9×9, 50000 epoche e 400 step non sono sufficienti per trovare la soluzione (vedi sottosezione 4.1). Dato che il dataset di training contiene matrici grandi al massimo 10×10 potrebbe compromettere il risultato della fase di allenamento.

4.1.2 Soluzione a Problema 1

Per risolvere questo problema procedo a tentativi nel seguente modo:

1. Provo ad *aumentare* il numero di *epoche* tenendo *costante* il numero di *step*.
2. Provo ad *aumentare* il numero di *step* tenendo *costante* il numero di *epoche*.
3. Provo ad *aumentare* sia il numero di *epoche* che il numero di *step*.



(a) 60k epoche, 400 step (b) 50k epoche, 450 step (c) 60k epoche, 450 step

(d) 70k epoche, 400 step (e) 50k epoche, 600 step (f) 70k epoche, 600 step

(g) 80k epoche, 400 step (h) 50k epoche, 1000 step (i) 80k epoche, 1000 step

Figura 20: Confronto tra training sulla stessa matrice con epoche e step differenti.

Dai grafici mostrati in precedenza si nota che solo in 2 casi l'agente è riuscito a trovare la soluzione: (g) e (i). Nel caso di (g), si vede come solo vero l'ultima parte della fase di allenamento, l'agente sia riuscito a trovare una soluzione, invece in (i), ha si esplorato inizialmente, ma ha trovato l'uscita decisamente prima. Hanno anche tempi di esecuzione e reward totali differenti, come possiamo notare dalla seguente tabella:

Training	**Tempo di esecuzione (minuti)**	**Reward Totale**
(g)	14 : 52 . 40	-7
(i)	08 : 34 . 95	-5

Figura 21: Confronto trai tempi di esecuzione e reward dei 2 allenamenti.

Deduco quindi che (i) risulta essere la migliore combinazione di epoche e step avendo il minor tempo di esecuzione con il maggior reward totale.

4.1.3 Problema 2: stato sconosciuto

L'agente quando si trova in uno stato sconosciuto andrà sempre ad eseguire l'azione `U` (vedi sottosezione 4.1).

4.1.4 Soluzione a Problema 2

Per risolvere questo problema vado a modificare la funzione `maxAction` (vedi sottosottosezione 3.4.4) facendo ritornare, solo se in fase di esecuzione, un'azione a caso qualora 2 o più elementi della riga presa in considerazione risultino uguali. Questo probabilmente andrà a "sbloccare" l'agente facendolo arrivare in uno stato a lui conosciuto, al costo di eseguire mosse poco intelligenti (potrebbe andare a sbattere contro un muro) ed introdurre aleatorietà nella fase di esecuzione.

$$\begin{bmatrix} 0.000000 & 0.000000 & 0.000000 & 0.000000 \end{bmatrix}$$

Figura 22: Esempio di stato mai visitato.

Se invece, l'agente è "indeciso" tra 2 mosse (solo 2 elementi della riga sono uguali), questa modifica risulterà decisamente più efficacie dato che sceglierà si un'azione casualmente, ma sarà un'azione sensata e valida che molto probabilmente lo condurrà in uno stato a lui noto.

$$\begin{bmatrix} 0.000000 & -1.405382 & 0.000000 & -5.872155 \end{bmatrix}$$

Figura 23: Esempio di indecisione tra 2 mosse.

```python
def maxAction(self, Q, state, actions: list[str], in_execution=False) -> str:
  values = np.array([Q[state, a] for a in actions])
  action = np.argmax(values)

  # prende a caso un'azione se ce ne sono due o piu' uguali
  if in_execution:
    tmp = [i for i, x in enumerate(values) if x == values[action]]
    if len(tmp) > 1:
      action = np.random.choice(tmp)

  return actions[action]

```

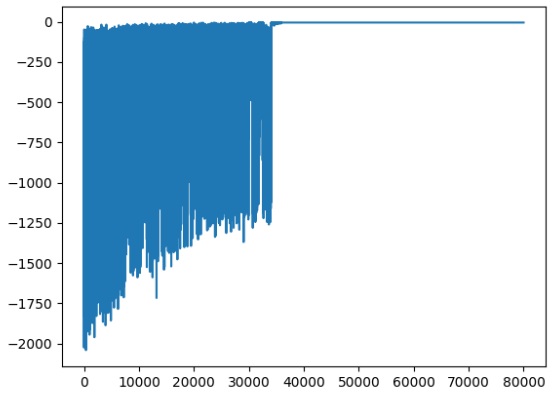
Listing 21: Modifica per risoluzione a Problema 2

4.2 Secondo Allenamento

Avendo applicato Soluzione a Problema 1 e Soluzione a Problema 2 effettuo un secondo allenamento, sempre sulla matrice scelta in sottosezione 4.1 e con i seguenti parametri:

Epoche: 80k | **Step**: 1000 | **Alpha**: 0.1 | **Gamma**: 1.0 | **EPS**: 0.9

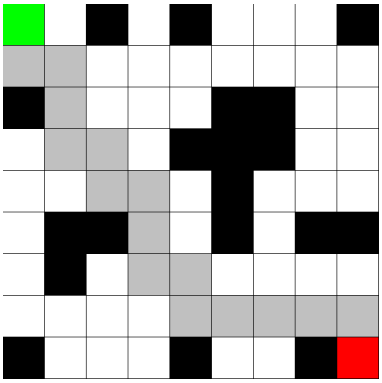
Successivamente vado prima a testarlo sul labirinto con cui è stato allenato, poi con una matrice mai vista, infine effettuo la valutazione come descritto in sottosezione 3.7.



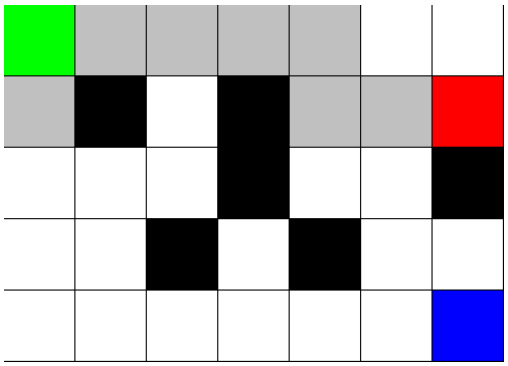
(a) Grafico di training

Acc (avg)	Rew (avg)
0.12	-1378.8

(b) Valutazione finale



(c) L'agente nella matrice di training



(d) L'agente in una matrice sconosciuta

Figura 24: Risultati secondo allenamento

Dall'analisi dei risultati sopra mostrati, si può notare come l'agente risulta essere molto bravo a risolvere il labirinto su cui è stato allenato (c), ma appena viene messo all'interno di uno mai esplorato non sa come comportarsi: in (d) si vede come quest'ultimo sta procedendo in modo casuale sperando di arrivare, prima o poi, in una serie di stati conosciuti che lo porteranno al goal. Anche il risultato della valutazione generale (mostrata in (b)) è molto scadente: l'agente riesce, in media, a risolvere solo 2 labirinti tra i 20 presenti nel dataset di testing.

4.2.1 Problema 1

Il problema di questi scarsi risultati risiede nella matrice Q: moltissimi stati (in questo specifico caso 213) non sono mai stati esplorati e quindi l'agente sarà portato a scegliere azioni casuali invece che basarsi sulla "conoscenza" acquisita durante l'allenamento.

```
-2.255275423663378547 -2.121039500176469517 -4.693440727996912032 -2.366805664424987867
-2.374627887715270447 -2.373328666128976181 -2.372542805941218944 -2.372503147618498076
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
-2.371282481079700517 -2.368817406376339818 -2.368644643610111871 -2.328848871442608868
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
0.0000000000000000000 0.0000000000000000000 0.0000000000000000000 0.0000000000000000000
```

Listing 22: Alcune righe della matrice Q

Risulta quindi essere molto bravo nel trovare la fuga solo nella matrice con cui è stato allenato, negli altri labirinti non sa quasi mai come muoversi. In pratica la matrice Q è poco popolata.

4.2.2 Soluzione a Problema 1

La soluzione a questo problema è abbastanza intuitiva: cercare di "mostrare" all'agente il maggior numero di stati possibili, in modo tale da accrescere la sua "esperienza". Ciò equivale a dire: popolare più righe possibili della matrice Q.

Allenerò quindi l'agente utilizzando tutto il dataset di training (vedi sezione 3, sottosezione 3.6) con i seguenti parametri:

Epoche: 80k | **Step**: 1500 | **Alpha**: 0.1 | **Gamma**: 1.0 | **EPS**: 0.9

Per poi valutare i risultati e confrontarli con quelli ottenuti dall'allenameto precedente.

4.3 Terzo allenamento

Procedo quindi come descritto in Soluzione a Problema 1. Questo allenamento è risultato estremamente oneroso in termini di tempo: circa 6 ore. Dovendo analizzare così tante matrici, di dimensioni abbastanza elevate, è normale che questa fase sia molto lunga. Questo costo elevato è anche dato dall'impiego di un'architettura single thread durante la fase di allenamento. Il passaggio al multi threading appare effettivamente molto difficile in quanto ogni epoca dipende dall'esperienza appresa nell'epoca passata, in più, va implementata una gestione efficiente e sincronizzata della matrice Q.

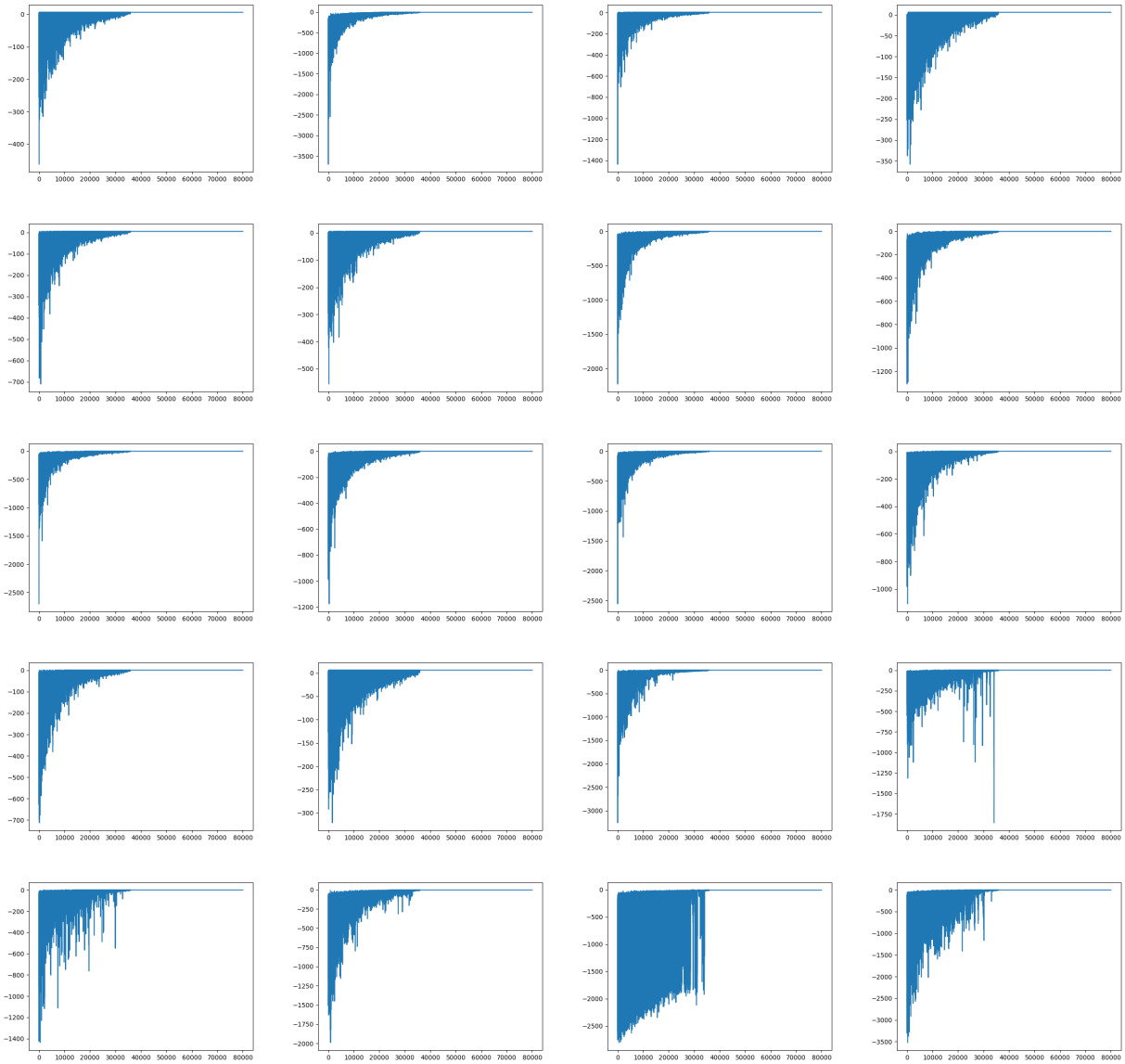
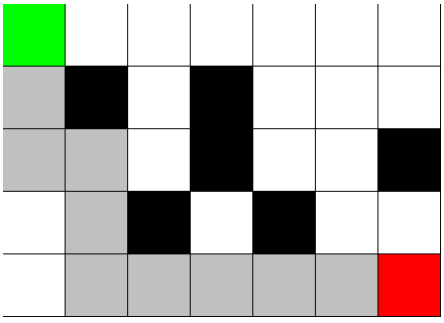


Figura 25: Alcuni dei grafici ottenuti durante la fase di training



(a) L'agente in una matrice mai vista

Acc (avg)	**Rew (avg)**
0.4	-338.8

(b) Valutazione finale

Dai dati sopra riportati possiamo notare come l'agente ora è in grado di risolvere una matrice mai vista prima (a) ed i valori medi di *accuracy* e *reward totale* sono migliorati notevolmente. Da un confronto più accurato possiamo notare che l'accuracy media è migliorata del 200% circa e il reward totale medio del 300% circa. L'agente ora è in grado di uscire da quasi la metà dei labirinti presenti nel dataset di training contro solo i 2 precedenti !

	Secondo Allenamento	**Terzo Allenamento**	
Accracy (avg)	0.12	0.4	$\sim +200\%$
Reward (avg)	-1378.8	-338.8	$\sim +300\%$

Figura 27: Confronto tra Secondo Allenamento e Terzo Allenamento

Mettendo a confronto le due matrici Q possiamo vedere come quella attuale risulta decisamente più completa (e quindi migliore) della precedente in quanto solo 41 stati sono inesplorati (righe vuote) contro i 213 precedenti.

```
 −2.25527542 −2.12103950 −4.69344072 −2.36680566
 −2.37462788 −2.37332866 −2.37254280 −2.37250314
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 −2.37128248 −2.36881740 −2.36864464 −2.32884887
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000
 0.000000000 0.000000000 0.000000000 0.000000000

```

Listing 23: Alcune righe della matrice Q di Secondo Allenamento

```
−3.6719095 −3.6719095 −3.7019095 −3.6519095
−5.4704139 −3.8491211 −3.7531035 1.33333333
−3.6419015 −3.6719087 −3.6719024 −3.6219095
−3.6821233 −3.6821233 −3.7321233 −3.7321233
−3.6221233 −3.6021233 −3.6521233 −3.6521233
0.00000000 0.00000000 0.00000000 0.00000000
−3.9537487 −4.0453708 −4.1297552 −4.1437734
−3.6416716 −3.6319088 −3.6808753 −3.6811229
−3.6273465 −3.6109095 −3.6615790 −3.6614216
−9.1105364 −9.1449290 −9.1322054 −9.1326386
−3.7019095 −3.7520104 −3.7520385 −3.7520236
−2.0637738 −1.8209197 −1.7467445 −6.9174127
−3.6819095 −3.6519095 −3.6519095 −3.6319095

```

Listing 24: Alcune righe della matrice Q di Terzo Allenamento

5 Riflessioni finali

Con Terzo Allenamento l'agente sa muoversi abbastanza bene all'interno dei labirinti, se questi non risultano troppo complessi, riuscirà quasi sicuramente a trovare la soluzione. Ci sono però alcuni casi che non è in grado di affrontare che verranno elencati di seguito:

5.1 Stati Inesplorati

Grazie alle modifiche apportate in sottosottosezione 4.1.3 l'agente sa affrontare abbastanza bene gli stati inesplorati ma non è sempre così:

- Nel caso più fortunato sceglie la mossa che lo porta in uno stato successivo a lui noto e continua.
- A volte può succedere che non sceglie immediatamente la mossa più appropriata portandolo, per esempio, a sbattere contro un muro (peggiorando il reward finale) per poi riuscire a sbloccarsi effettuando un'altra azione.
- Nel caso peggiore però, può capitare che l'agente scelga sempre la stessa mossa per molte volte (è raro ma piò accadere) oppure che rimanga bloccato tra 2 o più stati mai visti impedendogli di raggiungere l'uscita.

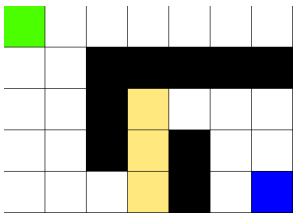
Prendiamo, per esempio, la seguente situazione: tra l'agente ed il goal ci sono 4 stati a lui sconosciuti e che la sequenza di azioni giusta da scegliere sia `R, R, R, R`. La probabilità che scelga la giusta combinazione di azioni è data da:

$$\frac{1}{4} * \frac{1}{4} * \frac{1}{4} * \frac{1}{4} = \frac{1}{4}^4 = \frac{1}{64} = 0.015 = 1.5\%$$

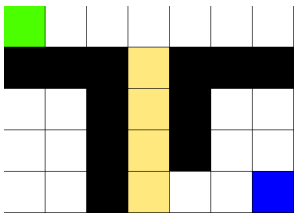
La probabilità di superare con successo 4 stati inesplorati consecutivi è estremamente bassa ! Le modifiche apportate alla funzione `maxAction()` servono per gestire stati inesplorati isolati e impedire all'agetne di bloccarsi, ma quando abbiamo troppi stati inesplorati consecutivi (già 4 risultano eccessivi) queste non possono fare molto.

5.2 Stati Ambigui

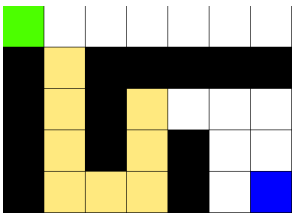
Testando l'agente su vari labirinti ho notato che ci sono alcune disposizioni di mura o determinati stati, conosciuti e ben esplorati, che "rompono" l'agente e lo intrappolano in un loop infinito di mosse uguali. Chiamerò questi stati: "*Stati Ambigui*". Il problema risale alla fase di allenamento e nello specifico nel dataset di training: esistono alcune matrici che, seppur generate casualmente, contengono delle "discese" o delle "salite": specifiche disposizioni delle mura per cui l'agente si trova circondato ai lati da Celle Muro e le uniche azioni tra cui può scegliere sono `U` e `D`.



(a) Esempio di salita



(b) Esempio di discesa

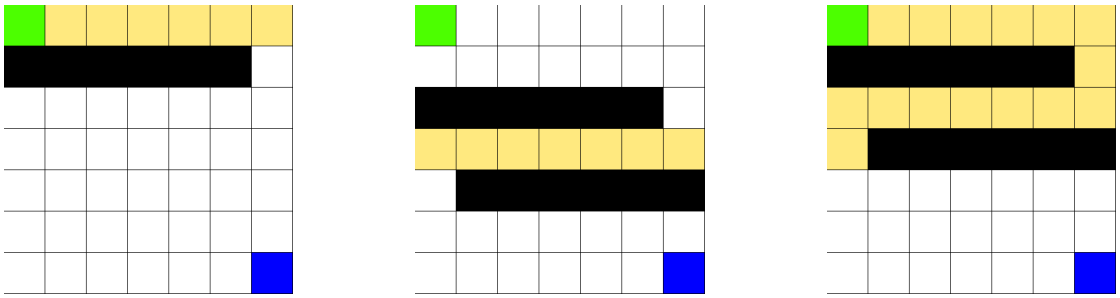


(c) Esempio di discesa e salita

Se l'agente viene allenato "a scendere" o "a salire" (allenamento con matrici che hanno uno solo dei precedenti pattern di mura) non ci sono problemi, ma se nel dataset di training compaiono entrambi, l'agente non saprà più come comportarsi: gli stati che compongono la "discesa" e la "salita" sono gli stessi, ma in base al tipo di pattern, deve compiere azioni differenti. Il risultato finale sarà il seguente: l'agente rimarrà bloccato tra i primi due stati

della "discesa/salita".

Lo stesso discorso vale anche per pattern di mura del tipo: "tunnel a destra" e "tunnel a sinistra". Un "tunnel" è come una "discesa/salita", soltanto che le uniche azioni che l'agetne può compiere sono `L` e `R`. Un "tunnel a destra" è un tunnel in cui si devono effettuare molteplici azioni `R` per uscirne. Un "tunnel a sinistra" è un tunnel nel quale si devono scegliere azioni `L` per uscirne.



(a) Esempio di tunnel a destra (b) Esempio di tunnel a sinistra (c) Esempio di tunnel misto

6 Conclusioni

Dopo aver eseguito un primo allenamento iniziale, aver preso visione e corretto i vari problemi con i successivi due allenamenti, aver confrontato le performance dell'agente per oguno di questi, posso concoludere che, tramite QLearning, sono riuscito a far apprendere all'agente come evadere da labirinti di modeste dimensioni e con una disposizione delle mura abbastanza basilare, ma con evidenti limiti quando la complessità delle matrici aumenta e con la completa incapacità di questultimo in presenza di "salite/discese" o "tunnel" (vedi sezione 5). In conclusione il QLearning si mostra si, un ottimo punto di partenza per studiare il RL, ma presenta evidenti limiti a risolvere questa tipologia di problemi.