Università di Perugia
Dipartimento di Matematica e Informatica

Esame di Artificial Intelligent Systems

Progetto Agent Cleaning Leaves

Professore
Prof. Alfredo Milani

Studente
Nicolò Posta

Anno Accademico 2021-2022

Indice

1 Obiettivo

L'obiettivo di questa relazione è quello di realizzare un'implementazione di un **ambiente** di **Reinforcement Learning** nel quale allenare un agente, tramite un algoritmo di **Q-Learning**, a muoversi in modo autonomo all'interno di una griglia nella quale sono disposte casualmente delle **foglie** che dovrà andare a risucchiare per "pulire" l'ambiente ed in fine raggiungere l'uscita.

2 Ipotesi iniziali e vincoli

L'agente si troverà in un ambiente bidimensionale, ovvero una griglia, con due differenti tipi di celle:

1. **Celle Vuote**: in queste celle l'agente può spostarsi liberamente o decidere di usare l'azione di risucchio per le foglie.

2. **Celle Con Foglie**: in queste celle vi è presente una foglia, sono normalmente percorribili dall'agente ma in più se vi viene fatta l'azione di risucchio l'agente raccoglierà quella foglia rimuovendola permanentemente dall'ambiente e aggiungendola al suo "zaino" che poi verrà svuotato non appena arriverà nello stato Goal ottenendo un reward maggiore più alto è il numero di foglie risucchiate durante il tragitto.

L'insieme di queste celle va a creare il terreno nel quale l'agente si muoverà che può essere visto come una matrice $m \times n$ con m il numero di righe ed n quello delle colonne.

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

La cella iniziale dell'agente sarà sempre quella in alto a sinistra ($c_{0,0}$) dovendo sempre raggiungere come goal la cella in basso a destra $c_{m-1,n-1}$. I movimenti saranno limitati alle quattro direzioni cardinali impedendo uno spostamento in diagonale tra le celle ed ovviamente l'agente non potrà uscire dalla griglia, infatti quest'ultima verrà trattata come se fosse circondata da mura che ne delimitano il perimetro impedendone all'agente la fuoriuscita o un eventuale effetto "pacman".
Il tipo di osservazione effettuabile dall'agente è legata al vicinato a 9 di Moore, ovvero l'agente sarà i grado di osservare tutti ed 8 i vicini della propria posizione e potrà sapere se in essi vi è presente una foglia o meno, ed ovviamente potrà anche conoscere la casella nella quale è fermo in quel momento.
Ad ogni azione dell'agente saranno assegnati diverti reward a seconda della casella nella quale si trova:
(I reward sono stati modificati dal progetto di base in quanto durante la fase di esamina del comportamento dell'agente ho riscontato che cambiando i reward con questa configurazione ho ottenuto dei risultati migliori)

- -1: una qualunque mossa verso una cella vuota
- -10: se tenta di muoversi fuori dalla griglia impattando su un muro
- -5: l'utilizzo dell'azione suck in una casella non avente una foglia
- $+5$: l'utilizzo dell'azione suck in una casella avente una foglia
- $+5 \times foglie\ raccolte$: per il raggiungimento della casella goal

2.1 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 *posizione iniziale* è sempre la prima cella in altro a sinistra: $c_{0,0}$
- La *posizione goal* è sempre l'ultima cella in fondo a destra: $c_{m-1,n-1}$
- Ritorna all'agente un'*osservazione* data dal vicinato di 9-Moore della posizione dell'agente
- Ogni 2 azioni effettuate dall'agente una **raffica di vento** andrà a spostare le foglie dell'ambiente in altre caselle non occupate in maniera casuale.
- Ritorna un *reward* all'agente in base alla mossa:
    * -1: una qualunque mossa verso una cella vuota
    * -10: se tenta di muoversi fuori dalla griglia impattando su un muro
    * -5: l'utilizzo dell'azione suck in una casella non avente una foglia
    * $+5$: l'utilizzo dell'azione suck in una casella avente una foglia
    * $+5 \times foglie\ raccolte$: per il raggiungimento della casella goal

2.2 Vincoli dell'agente

- Può muoversi in qualsiasi cella della griglia
- Non può uscire dai lati dell'ambiente perciò non potrà avvenire l'effetto "pacman"(non può andare fuori dalle celle della matrice e.g. $c_{0,-1}$ o passare dalla cella $c_{0,n-1}$ alla $c_{0,0}$)
- Le azioni che può compiere sono: `UP, DOWN, LEFT, RIGHT, SUCK`.

3 Struttura e composizione del Progetto

Il progetto si struttura in quattro file distinti:

- **main.py**: Script principale del progetto
- **LeavesWorld.py**: Rappresentazione dell'ambiente del progetto
- **QLearning.py**: Algoritmo di QLearning del progetto
- **GUI.py**: Funzioni per la rappresentazione della GUI
- **Q Matrix**: Matrice salvata dal QLearning

3.1 main.py

Essendo il file principale contiene l'omonima funzione **main** che fa partire un piccolo menu per scegliere cosa fare:

- **Training**: Permette il training dell'agente su una griglia creata dall'utente via riga di comando.
- **Execute step-by step**: Permette l'esecuzione dell'agente usando la Q Matrix su una griglia creata dall'utente via riga di comando.
- **Training with dataset**: Permette il training dell'agente su un dataset di griglie i quali parametri sono inseriti dall'utente.
- **Execute with dataset**: Permette l'esecuzione dell'agente su un dataset di griglie i quali parametri sono inseriti dall'utente.

La funzionalità di esecuzione non nel dataset può essere impostata in semiautomatico ovvero sarà possibile impartire mosse all'agente per farlo spostare a nostro piacimento oppure scegliere di far eseguire la prossima mossa seguendo la Q Matrix di quest'ultimo.
Questa funzionalità è stata molto utile nelle fasi di test per poter controllare il corretto funzionamento del programma e il controllo degli stati visibili dall'agente.

3.2 LeavesWorld.py

Questo è il file dove ho implementato la classe del nostro ambiente, di seguito andrò a spiegare le diverse funzioni che vi ho inserito ed spiegherò in dettaglio il funzionamento delle funzioni più importanti:

`isTerminalState()`	*Controlla se l'agente è nello stato finale*
`getAgentRowAndColumn()`	*Ritorna x ed y dell'agente*
`getNeighbours()`	*Ritorna il vicinato visto dall'agente*
`offGridMove()`	*Controlla se l'agente esce dalla griglia*
`neighboursToInt()`	*Trasforma il vicinato dell'agente in un intero*
`calculateNextState()`	*Calcola l'ambiente dopo la mossa dell'agente*
`step()`	*Ritorna il reward e modifica l'ambiente*
`randomGrid()`	*Genera casualmente la griglia di partenza*
`moveLeaves()`	*Muove le foglie nell'ambiente*
`reset()`	*Riporta l'ambiente allo stato iniziale*
`printMatrix()`	*Stampa a video la griglia dell'ambiente*
`render()`	*Aggiorna la GUI*
`actionSpaceSample()`	*Genera un azione casuale tra quelle possibili*

3.2.1 offGridMove

```
# controllo se il movimento è off grid
def offGridMove(self, newState: int, oldState: int) -> bool:
# se ci muoviamo su una cella non nella griglia
    if newState not in range(self.n*self.m):
        return True
    # controlliamo se x_new e y_new sono entrambi variati
    x_old = int(oldState / self.n)
    y_old = oldState % self.n
    x_new = int(newState / self.n)
    y_new = newState % self.n
    if x_old != x_new and y_old != y_new:
        return True
```

In questa funzione vado a controllare se l'agente va a spostarsi al di fuori della matrice andando ad uscire dall'ambiente stesso. Questo è possibile calcolando le coordinate dell'agente prima e dopo la mossa che vuole compiere e con un semplice controllo matematico si può vedere se esce o meno dall'ambiente.
Per farlo ci basta pensare che i 4 movimenti possibili dall'agente (`UP, DOWN, LEFT, RIGHT`) portano a spostarsi o nell'asse delle X o in quello delle Y ma non in entrambi contemporaneamente, perciò se al termine della mossa abbiamo una cambiamento di entrambe le coordinate sapremo che quella mossa non sarà valida. Questo controllo è stato aggiunto perchè non basta controllare se il nuovo stato è presente in quelli possibili dato che per come è salvata la matrice in memoria spostarsi dal lato destro verso destra risulterebbe in uno stato presente

nella matrice ma effettivamente saremmo nella prima casella della riga successiva, andando di fatto a cambiare la nostra X ed Y contemporaneamente.

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

$$AgentPosition = 3 \qquad AgentPosition = 4$$

Infatti come si può vedere dall'esempio l'agente andando a destra andrebbe a finire nella cella numero 4 ovvero uno stato presente nella matrice ma effettivamente avrebbe compiuto un azione illegale per le regole dell'ambiente e perciò è importante controllare anche questi casi.

3.2.2 getNeighbours e neighboursToInt

```
# restituisce lo stato attuale basandosi sulle coordinate dell'agente
def getNeighbours(self,  nextPosition = "default") -> np.ndarray:
    if nextPosition != "default":
        x = int(nextPosition / self.n)
        y = nextPosition % self.n
    else:
        x, y = self.getAgentRowAndColumn()

    neighbours = []
    #               UL        U       UR       L       NM       R       DL        D       DR
    for move in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)]:
        new_x, new_y = x + move[0], y + move[1]
        neighbour = 0

        if new_x < 0:
            neighbour = 2
        elif new_y < 0:
            neighbour = 2
        elif new_x > self.m - 1:
            neighbour = 2
        elif new_y > self.n - 1:
            neighbour = 2
        else:
            neighbour = self.grid[new_x][new_y]

        neighbours.append(neighbour)
    return np.array(neighbours, dtype=np.int8)
```

La funzione getNeighbours va a riportare effettivamente il vicinato di 9-Moore che poi verrà passato alla funzione neighboursToInt. Questo viene fatto controllando ogni cella adiacente alla posizione dell'agente e controllando se quella cella sia all'interno dei confini della matrice dell'ambiente, se questo non accade allora il vicino viene considerato come muro ed aggiunto il numero 2 alla matrice dei vicini per contrassegnarlo.

Questo metodo permette di non avere la griglia dell'ambiente più grande del dovuto e allo stesso tempo permette di comunque distinguere i limiti dell'ambiente da delle semplici caselle vuote altrimenti questo avrebbe portato a dei casi nei quali l'agente non sarebbe stato in grado di risolvere il problema.
Come ad esempio il vicinato vuoto:

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

Se non inserissimo le mura esterne ed utilizzassimo una codifica binaria non potremmo avere la distinzione fra foglie e mura, oppure semplicemente utilizzando celle vuote al posto delle mura dell'ambiente, come nella matrice del vicinato inserita sopra, avremmo che l'agente non sarebbe in grado di arrivare a convergere verso una chiara mossa da fare dato che ogni cella vuota nel suo vicinato potrebbe essere in verità un muro dell'ambiente.
Questo porta ad un risultato non ottimale in quanto solo in casi fortuiti sarebbe possibile risolvere il problema, ovvero quando l'allenamento finisca con lo scegliere la mossa `RIGHT` come mossa migliore in questo stato e in fase di esecuzione si arrivi a questo stato solo quando l'agente si trovi già nel bordo inferiore dell'ambiente dato che lo stato goal si trova per definizione sempre nell'angolo in basso a destra della matrice dell'ambiente.

```python
# converte lo stato in un intero in base 3
def neighboursToInt(self, state: int) -> int:
    tmp_state = np.array2string(state, separator="")
    return int(tmp_state[1:-1], 3)
```

$$\begin{matrix} 0 & 1 & 0 \\ 0 & A & 1 \\ 2 & 2 & 2 \end{matrix} \qquad 0\ 1\ 0\ 0\ 0\ 1\ 2\ 2\ 2$$

$$10001222_3 = 2240_{10}$$

Figura 1: Esempio di codificazione dello stato

Per ovviare ai problemi sopracitati nella funzione neighboursToInt ho deciso di codificare il vicinato di 9-Moore visibile dall'agente tramite la funzione getNeighbours in numeri in base 3 che sarebbero poi corrisposti alla rispettiva riga dello stato nella Q Matrix.
Utilizzando la codifica in base 3 si va ad ampliare non di poco la grandezza della Q Matrix e la va a rendere una matrice sparsa, ovvero solo poche delle righe effettivamente codificano uno stato possibile nell'ambiente a cui viene sottoposto l'agente perchè esistono delle combinazioni di vicini irraggiungibili. Di seguito ne ho riportate alcune.

2 2 2 2 2 1 2 1 1	2 2 2 2 1 2 2 1 1
0 2 0 0 1 0 0 1 1	2 2 2 2 1 1 2 2 2

Figura 2: Esempio di stati dei vicini impossibili da verificarsi

Di seguito ho riportato le due uniche configurazioni di vicini contenenti delle mura ottenibili nell'ambiente sottoposto all'agente. Di queste configurazioni vi si possono verificare anche tutte e 4 le rotazioni che hanno come punto di rotazione la casella centrale della matrice del vicinato di 9-Moore.

2 2 2 2 1 1 2 1 1	2 1 1 2 1 1 2 1 1

Figura 3: Esempio degli stati dei vicini contenenti dei 2 nella codifica

3.2.3 step

```
# esegue uno step in base all'azione passatogli
def step(self, action: str) -> tuple[int, int, bool, dict]:
    # incremento il contatore per muovere le foglie
    if self.iterator == 2:
        self.iterator = 0
        self.moveLeaves()
    self.iterator += 1
    current_neighbours = self.getNeighbours()
    resultingPosition = self.agentPosition + self.actionSpace[action]
    x, y = self.getAgentRowAndColumn()
    if action == 'S' and self.grid[x][y] == 1:
        self.grid[x][y] = 0
        reward = 5
        self.leaves_sucked += 1
```

```
    elif action == 'S' and self.grid[x][y] != 1:
        reward = -5
    elif not self.isTerminalState(resultingPosition):
        reward = -1
    else:
        reward = 5*self.leaves_sucked
    # controllo se il movimento è off grid
    if not self.offGridMove(resultingPosition, self.agentPosition):
        resultingState = self.calculateNextState(action, resultingPosition)
        self.agentPosition = resultingPosition
        return resultingState, reward, \
        self.isTerminalState(resultingPosition), None
    else:
        reward = -10
        return self.neighboursToInt(current_neighbours), reward, \
        self.isTerminalState(resultingPosition), None
```

Questa funzione va a calcolare il reward da assegnare all'agente e lo stato dei vicini nel quale si verrà a trovare dopo l'azione che ha compiuto, inoltre se l'azione compita è quella di risucchiare una foglia in una casella avente una foglia al suo interno la va a rimuovere dalla griglia, in fine controlla se l'agente è arrivato nello stato goal e se il controllo è positivo glielo notifica.
Oltre a questo la funzione va a invocare lo spostamento delle foglie ogni 2 movimenti effettivi dell'agente andando così ad attivare la **raffica di vento** richiesta nell'implementazione del progetto.

3.2.4 moveLeaves e randomGrid

```
# genera una composizione random delle foglie nella griglia
def randomGrid(self) -> None:
    for i in range(self.leaves):
        a = random.randint(0, self.m - 1)
        b = random.randint(0, self.n - 1)
        while self.grid[a][b] == 1: # controllo che la cella sia vuota
            a = random.randint(0, self.m - 1)
            b = random.randint(0, self.n - 1)
        self.grid[a][b] = 1
    self.grid_for_reset = self.grid

# muove le foglie
def moveLeaves(self) -> None:
    newgrid = np.zeros((self.m,self.n))
```

```
    for i in range(self.m):
        for j in range(self.n):
            # cerco le foglie nella griglia
            # e ne genero la nuova posizione
            if self.grid[i][j] == 1:
                a = random.randint(0, self.m - 1)
                b = random.randint(0, self.n - 1)
                if newgrid[a][b] == 1:
                    if newgrid[i][j] == 1:
                        while newgrid[a][b] == 1:
                            a = random.randint(0, self.m - 1)
                            b = random.randint(0, self.n - 1)
                        newgrid[a][b] = 1
                    else:
                        newgrid[i][j] = 1
                else:
                    newgrid[a][b] = 1
    self.grid = newgrid
```

La funzione randomGrid va a posizionare casualmente le foglie iniziali all'interno della griglia di partenza le quali verranno poi spostate utilizzando la funzione moveLeaves, quest'ultima va a spostare casualmente le foglie ancora presenti nell'ambiente in altre posizioni tenendo conto però che se lo spostamento va a comportare una collisione con una foglia, ovvero che nella casella nuova vi sia già presente una foglia, essa non andrà a spostarsi ma resterà nella casella di partenza.
Questo però può portare ad un problema di fondo, ovvero dato che nell'andare a cercare le foglie da spostare si va a sfogliare la matrice dall'inizio alla fine potrebbe succedere che arrivati ad un certo punto avremo spostato abbastanza foglie da riscontrate una collisione e quindi dovremo riposizionare la foglia al suo posto iniziale che però potrebbe già essere occupato, questo potrebbe farci perdere questa foglia "sovrascrivendola" su quella casella, allora per ovviare a questo problema ho aggiunto un controllo che se questo accade va a spostare la foglia in una casella vuota dell'ambiente.
Questo problema è molto rilevante in quanto con l'aumentare della densità di foglie nella griglia va di conseguenza ad aumentare la probabilità che questo specifico evento accada e che possa andare a far perdere informazioni preziose come appunto sovrascrivere una foglia sulla stessa casella diminuendo di fatto le foglie presenti nella griglia in maniera impropria.

0	1	0	0	0	1
0	1	1	1	0	0
1	0	0	1	1	0
0	0	1	0	1	0
0	1	0	0	1	0
1	0	0	1	0	0

Matrice : 1

0	0	0	0	0	1
0	1	0	1	0	0
1	0	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	1	0

Matrice : 2

Figura 4: Esempio di errore di sovrascrittura di una foglia

In questo esempio ho visualizzato meglio il problema, la foglia in rosso (1,6) della matrice n.1 al momento dello spostamento viene messa nella posizione (5,4), in blu nella matrice n.2, ma avendo già una foglia allora viene reinserita nella posizione iniziale (1,6), la quale però era già stata occupata da una foglia spostata in precedenza, in rosso nella matrice n.2, e allora per evitare di sovrascriverla la vado a posizionare in una casella vuota della nuova matrice, non curandomi dell'eventualità di una collisione dato che continuerò a generare la nova posizione fino a che non si trovi una cella vuota, in questo esempio è la cella (3,6), in verde nella matrice n.2.

3.2.5 render e printMatrix e GUI

```
# stampa la matrice a schermo
def printMatrix(self) -> None:
    elem = "-------"
    print(elem*(self.n-1)+"-"*self.n)
    x, y = self.getAgentRowAndColumn()
    for row in range(self.m):
        for col in range(self.n):
            if row == x and col == y:
                print ('X', end='\t')
            elif  self.grid[row][col]==1:
                print ('F', end='\t')
            else:
                print('-', end='\t')
        print('\n')
    print(elem*(self.n-1)+"-"*self.n)
```

```python
# renderizza l'ambiente sulla GUI
def render(self, print_matrix=False) -> None:
    if print_matrix:
        self.printMatrix()
    x, y = self.getAgentRowAndColumn()
    self.gui.draw((x,y), self.grid)
    tmp = self.grid.ravel()
    total_sum = tmp.tolist().count(1)
    print(f"Leaves in the grid: {total_sum}")
    print(f"Leaves sucked: {self.leaves_sucked}")
```

Queste sono le due funzioni utilizzate per visualizzare l'ambiente graficamente, una stampa sulla console la griglia e l'altra si interfaccia con la classe GUI e va a creare una finestra con stampata al suo interno una griglia con diversi colori in base allo stato dell'ambiente.

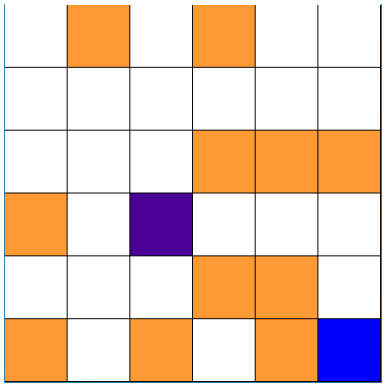


Figura 5: Esempio di GUI

Legenda colori:

- **Viola**: agente
- **Arancione**: foglia
- **Blu**: stato goal

3.3 QLearning.py

In questo file ho inserito le funzioni per allenare e testare l'agente utilizzando un ambiente creato al momento dall'utente o tramite la creazione di un DataSet di ambienti creati casualmente.
Nelle sezioni seguenti andrò a spiegare le funzioni presenti nel file e porterò degli esempi di test che ho fatto.

`maxAction()`	*Calcola l'azione da eseguire basandosi sulla Q Matrix*
`printQ()`	*Stampa a schermo la Q Matrix*
`saveQ()`	*Salva su file la Q Matrix*
`loadQ()`	*Carica in memoria la Q Matrix presa da un file esterno al programma*
`createDataSet()`	*Crea un dataset di dimensione variabile di matrici $n \times m$ con una percentuale di foglie scelta dall'utente*
`trainingWithDataset()`	*Effettua il training del modello su differenti matrici prese dal dataset*
`blankQMatrix()`	*Salva un file con una matrice vuota*
`newMatrix()`	*Genera una nuova matrice vuota*
`executeOnDataset()`	*Richiama l'esecuzione del modello su tutti gli ambienti del dataset*
`execute()`	*Fa eseguire il modello in maniera automatica o meno sull'ambiente in memoria*
`training()`	*Esegue l'allenamento sull'ambiente caricato in memoria*

3.3.1 Matrice Q

Questa matrice è formata da tante righe quanti sono gli stati (ovvero 19750) e tante colonne quante il numero di azioni (quindi 5).

$$QMatrix = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{19570,0} & a_{19570,1} & a_{19570,2} & a_{19570,3} & a_{19570,4} \end{bmatrix}$$

Figura 6: Definizione della Q Matrix

Di seguito andrò a trattare del perchè il numero di stati massimi necessari sia 19750, per iniziare riporto il vicinato con codifica maggiore, ovvero l'ultima riga della QMatrix.

$$\begin{matrix} 2 & 2 & 2 \\ 2 & 1 & 1 \\ 2 & 1 & 1 \end{matrix} \qquad \begin{matrix} 2 & 2 & 2 & 2 & 1 & 1 & 2 & 1 & 1 \end{matrix}$$

$$222211211_3 = 19570_{10}$$

Figura 7: Vicinato con codifica in stato più grande in base 10

Controllando lo stato con codifica più grande ottenibile si può notare che sia inferiore della codifica più grane ottenibile con 9 cifre in base 3, ovvero 19570 è minore di $3^9 = 19.683$. Con questa accortezza possiamo rimuovere circa 100 righe dalla tabella la quale purtroppo sarà una matrice sparsa e quindi prevalentemente piena di righe che non codificano in uno stato effettivo dell'ambiente perchè non tutti gli stati della matrice codificano con vicinati effettivamente riscontrabili nell'ambiente di questo progetto.
La codificazione del vicinato di 9-Moore in stati, infatti, è calcolata basandosi sul vicinato visto dall'agente e trasformando le celle della matrice in 0 se sono prive di foglie, in 1 quando ne hanno una al loro interno ed in 2 se la cella sarebbe fuori dall'ambiente andando così in fine a creare il numero in base 3 richiesto.
In ogni cella della matrice è contenuto un valore decimale (ad esempio 45.529975) che va ad indicare la qualità dell'azione da compiere nel determinato stato. Per esempio, prendiamo in considerazione la seguente riga della matrice:

$$Qmatrix_{[19570]} = \begin{bmatrix} 45.529975 & 70.065169 & 49.731448 & 51.950491 & 64.803483 \end{bmatrix}$$

Figura 8: Esempio di una possibile riga della Q Matrix

Il primo elemento rappresenta la qualità dell'azione `UP` allo stato 19570, 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 `DOWN` con un valore di 70.065169.

3.3.2 maxAction

```python
# calcola l'azione migliore per uno stato basandosi sulla Q matrix
def maxAction(self,Q: dict, state: int, actions: list, in_execution=False):
    values = np.array([Q[state,a] for a in actions])
    action = np.argmax(values)

    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]
```

Questa funzione utilizza la Q Matrix e il vicinato corrente dell'agente per calcolare l'azione da compiere controllando qual'è il valore massimo della riga corrispondente alla codifica del vicinato in stato.
Considerazioni sull'implementazione:
La funzione argmax di numnpy per come è implementata se nell'array passatogli trova due

valori massimi uguali va a ritornare sempre l'indice del primo valore rendendo l'allenamento e l'esecuzione dell'agente non corretto in quanto verrà preferita un'azione a discapito di un'altra anche se entrambe con lo stesso valore.
Questo è particolarmente importante nelle primissime fasi di allenamento in quanto se l'agente si trova di fronte ad uno stato mai visto i suoi valori nella Q Matrix saranno tutti uguali a 0, e perciò la funzione argmax ritornerà sempre l'azione codificata nella prima colonna (`UP`) invece di effettivamente provare un azione random come spiega la teoria del Q-Learning.
Oltre al problema precedentemente citato vi si aggiunge il fatto che se durante l'esecuzione l'agente va da imbattersi in uno stato mai incontrato durante l'allenamento quest'ultimo andrà sempre a compiere la mossa `UP` invece di effettivamente preferire una mossa random in quanto nessun valore è più grande di un altro.
Per ovviare a questi problemi ho deciso di implementare un controllo sull'esistenza di più valori massimi uguali e nel caso ci siano venga scelto uno di essi casualmente e non sempre il primo.

3.3.3 training e trainingWithDataset

```python
# esegue il training dell'agente su un ambiente passato dall'utente
def training(self,epochs=80000, steps=1500, ALPHA=0.1, GAMMA=1.0, EPS=1.0, plot=True, dataset=False) -> None:
    # inizializzo la Q matrix o la carico se esiste già
    starting_eps = EPS
    if not dataset:
        Q = self.newMatrix()
    else:
        Q = self.Q.copy()

    self.env.reset()
    if not dataset:
        self.env.printMatrix()
    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()
            action = self.maxAction(Q,observation, self.env.possibleActions) if rand < (1-EPS) \
                                                    else 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
        if EPS - 2 / epochs > 0:
            EPS -= 2 / epochs
        else:
            EPS = 0
        totalRewards[i] = epRewards

    if not dataset:
        if plot:
            plt.plot(totalRewards)
            plt.savefig(f"imgs/n_{self.env.n}_m_{self.env.m}_percentuale_foglie_{self.env.p_leaves}.png)

    if dataset:
        self.Q = Q.copy()
        return Q
    else:
        self.saveQ(Q,"Qmatrix")
```

```python
# training della Q matrix usando un dataset di esempio
def trainingWithDataset(self, dataset: list, max_steps: int) -> None:
    self.blankQMatrix()
    self.Q = self.loadQ(fileName="Qmatrix")
    num_training = 1
    Q = {}
    for elem in dataset:
        self.env = LeavesWorld.LeavesWorld(elem[0],elem[1],elem[2])
        print(f"Training number: {num_training}")
        Q = self.training(epochs = 10000, steps = max_steps, dataset=True)
        num_training += 1
    self.saveQ(Q,"Qmatrix")
```

La funzione training può eseguire l'allenamento dell'agente inizializzando una nuova Q Matrix se l'ambiente è stato generato dall'utente, oppure se viene specificato l'uso di DataSet va ad utilizzare una nuova matrice solo nel primo allenamento, andando così a migliorare ogni volta la matrice precedentemente generate, questa opzione è utilizzata nella funzione trainWithDataset che va ricorsivamente a chiamare la funzione training utilizzando sempre la Q Matrix allenata sull'ambiente dell'allenamento precedente.
Di default ho scelto di utilizzare 80000 epoche per permettere la convergenza, 1500 step massimi per permettere all'agente di muoversi liberamente il più possibile e poter vedere più stati possibili prima di ogni nuova epoca, solitamente questo porta le prime iterazioni ad avere un agente che si muove quasi casualmente ma poi inizia a convergere verso l'utilizzo di pochi step per arrivare al goal.
I parametri utilizzati nell'equazione si Bellman di default sono impostati a:

- **ALPHA: 0.1**
  Il learning rate del Q learning
- **GAMMA: 1.0**
  Il discount factor
- **EPS: 1.0**
  La EPS è legata al valore di greedy dell'algoritmo

3.3.4 execute

```python
# esegue l'agente usando la Q matrix su un ambiente passato dall'utente
def execute(self, auto=True, fast=False, max_steps=1500) -> None:
    # se l'esecuzione è in modalità auto l'agente esegue le azioni basandosi
    # sulla Q matrix caricata senza poter eseguire azioni passatogli dall'utente
    if auto:
```

```
        # se l'esecuzione è in modalità fast, non viene mostrato l'ambiente
        if not fast:
            self.env.render()
        Q=self.loadQ("Qmatrix")
        totReward=0
        while max_steps > 0:
            action=self.maxAction(Q, self.env.neighboursToInt(self.env.getNeighbours()), \
            self.env.possibleActions, in_execution=True)
            observationNext, reward, done, info = self.env.step(action)
            max_steps -= 1
            totReward += reward
            if not fast:
                print("Action: "+str(action)+" Reward: "+str(reward)+"\n")
                self.env.render()
                print(f"Total reward: {totReward}")
            if done:
                percentage_leaves_sucked = self.env.percentageLeavesSucked
                print(f"Percentage of leaves sucked: {round(percentage_leaves_sucked, 2)}%")
                exit()
            if not fast:
                sleep(0.2)
            clear()
        print("Max step reached")
        exit()
    else:
        # se l'esecuzione è in modalità non auto, l'agente esegue le azioni passate
        # dall'utente oppure basandosi sulla Q matrix caricata se non è stata passata una azione
        clear()
        self.env.render(print_matrix=True)
        Q=self.loadQ("Qmatrix")
        totReward=0
        command=input("Total reward: "+str(totReward)+"\nExecuteNext?(y/n/uP/dOWN/lEFT/riGTH/sUCK):")
        while  command != 'n':
            if max_steps >= 0:
                if command =='y':
                    action=self.maxAction(Q, self.env.neighboursToInt(self.env.getNeighbours()), \
                    self.env.possibleActions, in_execution=True)
                elif command == 'u':
                    action='U'
                elif command == 'd':
                    action='D'
                elif command == 'l':
                    action='L'
                elif command == 'r':
                    action='R'
                elif command == 's':
                    action='S'
                observationNext, reward, done, info = self.env.step(action)
                max_steps -= 1
                totReward += reward
                clear()
                print("Action:"+str(action)+" Reward:"+str(reward)+"\n")
                self.env.render(print_matrix=True)
                if not done:
                    command=input("Total reward: "+str(totReward)+"\n+
                    "ExecuteNext?(y/n/uP/dOWN/lEFT/rIGTH/sUCK):")
                if done:
                    percentage_leaves_sucked = self.env.percentageLeavesSucked
                    print(f"Percentage of leaves sucked: {round(percentage_leaves_sucked, 2)}%")
                    exit()
            else:
                percentage_leaves_sucked = self.env.percentageLeavesSucked
                print(f"Percentage of leaves sucked: {round(percentage_leaves_sucked, 2)}%")
                print("Max step reached")
                exit()
```

Questa funzione va a caricare la Q Matrix da file e la utilizza per far eseguire l'agente sull'ambiente passatogli in due possibili modi:

- **Esecuzione automatica**: stampa la GUI a schermo e nella griglia ne riproduce le mosse scelte dall'agente tramite l'utilizzo della Q matrix caricata in precedenza, nel mentre stampa a schermo le azioni scelte dall'agente e il reward ottenuto, in fine riporta il numero di foglie risucchiate e il reward totale dell'agente.

- **Esecuzione semi automatica**: stampa la GUI a schermo ma stavolta aspetta un input da tastiera da parte dell'utente, questo permette di scegliere se eseguire la prossima azione seguendo la Q Matrix dell'agente oppure far eseguire un azione scelta dall'utente e quindi ignorando la Q Matrix per quell'azione, anche questa funzione stampa il reward ottenuto dall'azione eseguita e al raggiungimento del goal stampa il numero totale di foglie risucchiate ed il reward totale ottenuto dall'agente.

3.3.5 executeOnDataset

La funzione executeOnDataset riporta un

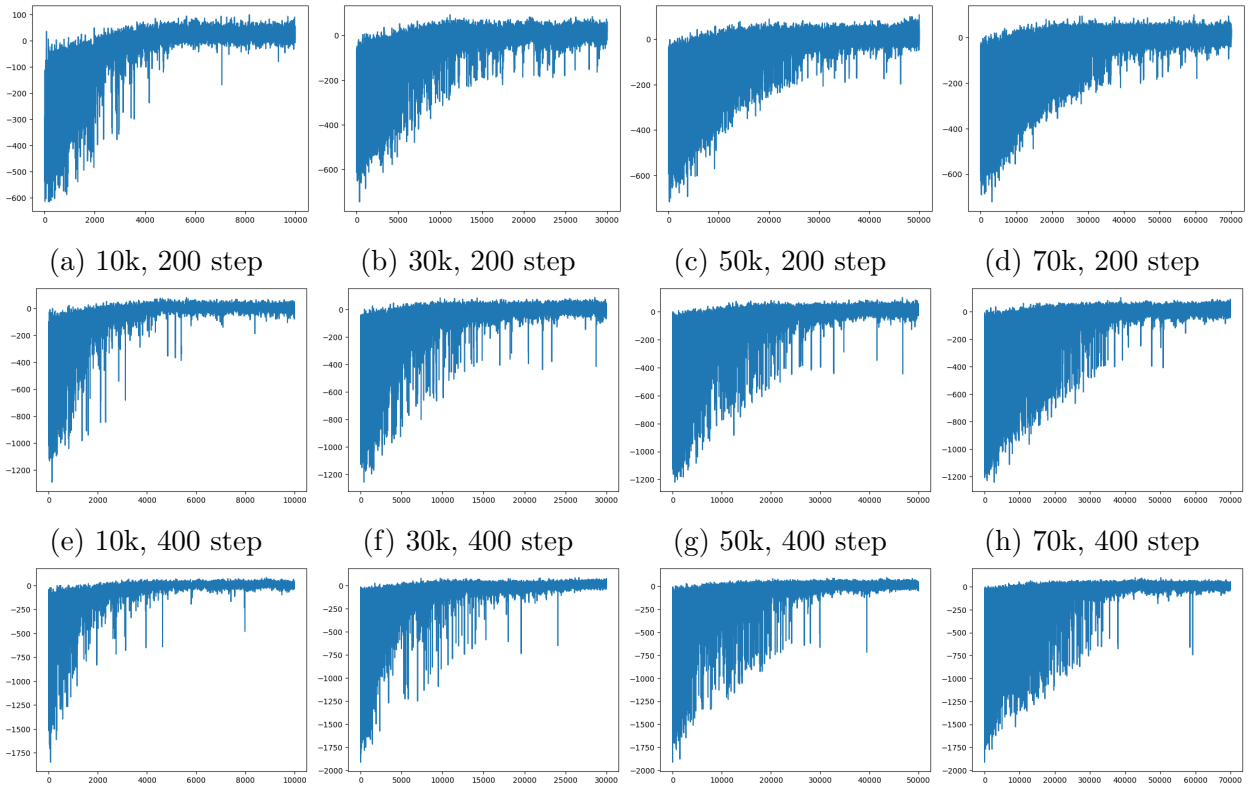
```
# esegue l'agente usando la Q matrix caricata su un dataset di esempio
def executeOnDataset(self, dataset: list,, max_steps=1500) -> None:
    Q=self.loadQ("Qmatrix")
    evaluation = []
    iterator = 0
    times_stuck = 0
    totLeaves = 0
    reward_list = []
    step_list = []
    for elem in dataset:
        print(f"Execution on the {iterator+1} element of the dataset")
        iterator += 1
        self.env = LeavesWorld.LeavesWorld(elem[0],elem[1],elem[2])
        totLeaves += self.env.leaves
        totReward=0
        steps=0
        while steps < max_steps:
            action=self.maxAction(Q, self.env.neighboursToInt(self.env.getNeighbours()), \
            self.env.possibleActions, in_execution=True)
            observationNext, reward, done, info = self.env.step(action)
            steps += 1
            totReward += reward
            if done:
                step_list.append(steps)
                evaluation.append(round(self.env.percentageLeavesSucked, 2))
                reward_list.append(totReward)
                break
        if steps == max_steps:
            evaluation.append(0.0)
            reward_list.append(totReward)
            times_stuck += 1
            print("Agent got stuck, skip iteration")
    mean_steps = np.mean(step_list)
    mean_reward = np.mean(reward_list)
    mean_percentage = np.mean(evaluation)
    print(f"Times stuck: {times_stuck}")
    print(f"Mean reward: {round(mean_reward, 2)}")
    print(f"Mean steps: {round(mean_steps, 2)}")
    print(f"Total number of leaves in the dataset: {totLeaves}")
    print(f"Mean percentage of leaves sucked in the experiment: {round(mean_percentage, 2)}%")
```

4 Svolgimento Test

In questa sezione andrò a spiegare i vari test eseguiti. Per rendere i test comparabili e rieseguibili ho deciso di impostare il seed delle funzioni random utilizzando la classe creata da me nel file **Randomizer.py**.

4.1 Allenamento su una singola Matrice

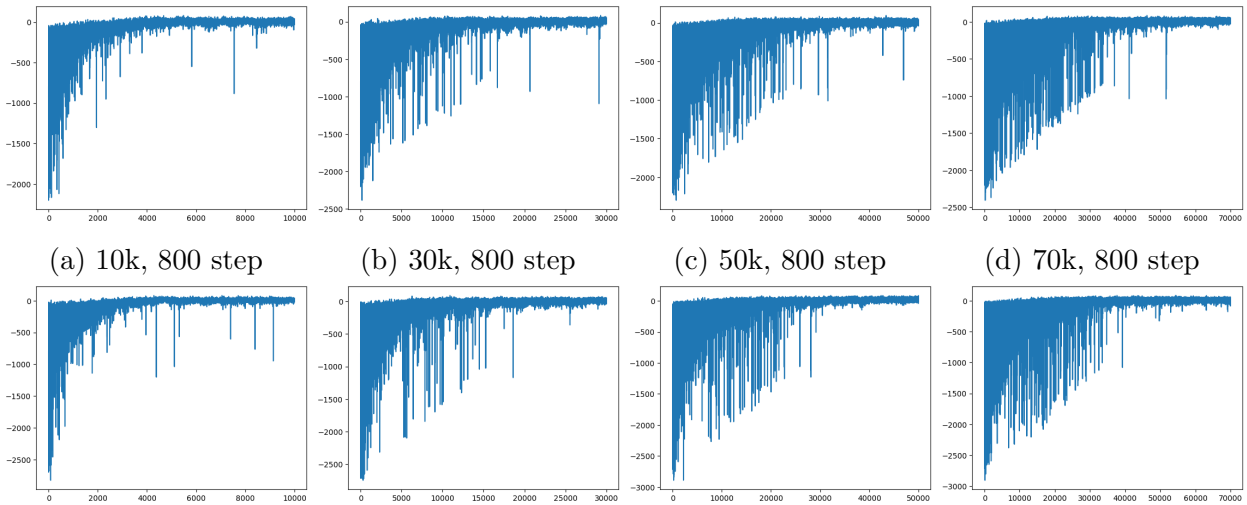
Dato che il problema da risolvere è governato da movimenti casuali delle foglie ho iniziato allenando l'agente solo su una matrice con una densità di foglie del 40% per dare la possibilità all'agente di vedere la maggior parte degli stati possibili.
Questi sono i risultati variando il numero di Epoche e Step utilizzati:



(a) 10k, 200 step (b) 30k, 200 step (c) 50k, 200 step (d) 70k, 200 step

(e) 10k, 400 step (f) 30k, 400 step (g) 50k, 400 step (h) 70k, 400 step

(i) 10k, 600 step (j) 30k, 600 step (k) 50k, 600 step (l) 70k, 600 step



(a) 10k, 800 step (b) 30k, 800 step (c) 50k, 800 step (d) 70k, 800 step

(e) 10k, 1000 step (f) 30k, 1000 step (g) 50k, 1000 step (h) 70k, 1000 step

Figura 10: Confronto tra training sulla stessa matrice con epoche e step differenti.

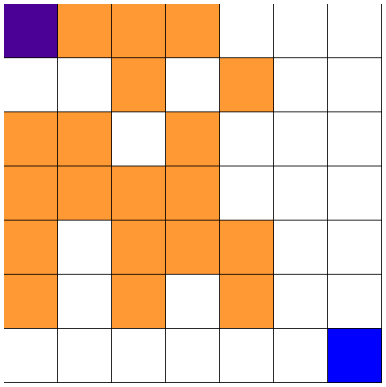


Figura 11: Matrice 7×7 usata per allenamento e test

Sia per l'allenamento che per l'esecuzione ho utilizzato lo stesso posizionamento delle foglie ma differenti seed così da ottenere movimenti casuali differenti fra allenamento ed esecuzione.

Runs	Total steps	Total reward	Total leaves in env. = 19	
			Leaves sucked	Percentage of leaves
10k, 200 step	47	59	11	57.89%
10k, 400 step	20	17	4	21.05%
10k, 600 step	34	32	7	36.84%
10k, 800 step	52	29	8	42.11%
10k, 1000 step	36	38	7	36.84%
30k, 200 step	39	2	4	21.05%
30k, 400 step	44	22	7	36.84%
30k, 600 step	33	45	7	36.84%
30k, 800 step	20	47	6	31.58%
30k, 1000 step	111	38	16	84.21%
50k, 200 step	42	9	6	31.58%
50k, 400 step	45	3	5	26.32%
50k, 600 step	35	57	9	47.37%
50k, 800 step	57	5	7	36.84%
50k, 1000 step	51	4	6	31.58%
70k, 200 step	31	24	6	31.58%
70k, 400 step	27	28	6	31.58%
70k, 600 step	27	21	5	26.32%
70k, 800 step	27	14	4	21.05%
70k, 1000 step	31	10	4	21.05%

Tabella 1: Risultati allenamento su matrice 7×7 con 40% di foglie

Come si può vedere dalla tabella aumentando il numero di Step solitamente si va ad aumentare il numero di foglie raccolte prima del raggiungimento dello stato goal a discapito degli step utilizzati per raggiungerlo.
Dall'elaborazione dei dati raccolti ho deciso di scegliere come modello migliore quello che va a raccogliere più foglie prima ri raggiungere lo stato goal e quindi il modello allenato con 30k epoche a 1000 step.

4.2 Variazione del valore Gamma

Gamma	Total steps	Total reward	Total leaves in env. = 19	
			Leaves sucked	% of leaves
0.0	24	21	4	20.05%
0.2	19	7	3	15.79%
0.4	17	17	3	15.79%
0.6	22	30	5	26.32%
0.8	20	32	5	26.32%
1.0	111	38	16	84.21%

Tabella 2: Confronto risultati esecuzione su matrice 7×7 con 40% di foglie allanando con 30k epoche e 1000 step

In questo test ho modificato il discount factor dell'equazione di Bellman per controllare se rendendo l'agente più incentivato ad un reward istantaneo avrebbe portato a raccogliere più foglie, ma questo non ha portato a miglioramenti nell'allenamento dell'agente.

4.3 Allenamento su più matrici

L'allenamento su più matrici con densità fissa del 40% di foglie può migliorare il modello in maniera sostanziale in quanto sarà più probabile che incontrerà più volte tutti i possibili stati e quindi potrà aggiornare la Q Matrix in maniera accurata.
Questo test lo ho svolto su 100 differenti matrici con lati anch'essi variabili da un minimo di 2 celle ad un massimo di 10.
Per il test ho deciso di utilizzare 10k epoche per matrice in quanto avendo 100 matrici avrei comunque avuto 100×10 epoche totali con le quali allenare il modello.

Runs	Total steps	Total reward	Total leaves in env. = 19	
			Leaves sucked	% of leaves
10k, 1000 s., 100 m.	25	31	5	26.32%
30k, 1000 s., 1 m.	111	38	16	84.21%

Tabella 3: Confronto risultati esecuzione su matrice 7×7 con 40% di foglie

Il test sulla matrice di testing non ha riscontrato un notevole miglioramento dell'agente.
Osservazioni:
Questo potrebbe essere dato dal fatto che durante il training con matrici di grandezza inferiore l'agente possa aver imparato a raggiungere il lato inferiore dell'ambiente e semplicemente scorrere a destra fino al raggiungimento dello stato goal non prendendo molte foglie.
Questo è comunque considerabile un comportamento ottimo in matrici di piccole dimensioni in quanto la distanza percorsa fino a goal è notevolmente inferiore a quella in matrici più

grandi andando così ad ottenere un reward più elevato anche senza risucchiare molte foglie nel tragitto.

4.4 Test su più matrici

I test sono stati effettuati su 100 matrici con lati variabili tra la lunghezza di 2 e 10 caselle, e con il 40% di foglie.

Runs	Mean steps	Mean reward	Total leaves in env. = 19	
			Times stuck	Mean % of leaves
10k 1000 s. 100 m.	162	-130.17	14	24.46%
30k 1000 s. 1 m.	209	-163	15	43.00%

Tabella 4: Risultati esecuzione su 100 matrici variabili con 40% di foglie

Anche in questo caso possiamo osservare come l'agente allenato su una singola matrice dia comunque risultati di gran lunga migliori di quello allenato su un Data Set di matrici.
Test con 100 matrici di grandezza variabile tra 5 e 10, e con il 40% di foglie al loro interno.

Runs	Mean steps	Mean reward	Total leaves in dataset = 2181	
			Times stuck	Mean % of leaves
10k 1000 s. 100 m.	34	9.88	0	23.20%
30k 1000 s. 1 m.	97	-14.25	0	46.57%

Tabella 5: Risultati esecuzione su 100 matrici variabili con 40% di foglie

4.5 Ultimo allenamento

Notando come utilizzando una singola matrice abbia portato a risultati di gran lunga superiori dell'approccio multimatrice ho deciso di effettuare un ultimo test controllando come la grandezza della matrice vada ad influire sulla resa dell'agente. Per fare questo ho allenato l'agente in una matrice 20×20 con una densità di foglie del 40% e 1500 step massimi per ogni epoca. Questi sono i risultati ottenuti sulla matrice ed il dataset di Test:

Runs	Steps	Reward	Total leaves	Leaves sucked	% of leaves
30k 1500 s. 1 m.	39	-25	19	2	10.53%
Runs	Mean steps	Mean reward	Times stuck	Toal Leaves	% of leaves
30k 1500 s. 100 m.	165	-144.61	7	1288	18.69%

Tabella 6: Risultati esecuzione su 100 matrici variabili con 40% di foglie

Come si può ben vedere l'utilizzo di una matrice molto grande va a rendere l'allenamento inefficace perchè durante la fase dello spostamento delle foglie, avendo una matrice molto grande e non estremamente piena, quest'ultime non vanno a distribuirsi attorno all'agente e non gli permettono di vedere abbastanza stati portandolo a non convergere abbastanza strettamente.

5 Conclusioni

5.1 Limitazioni del Q Learning

Grazie ai test effettuati in precedenza ho riscontrato dei problemi intrinsechi dell'applicazione del Q Learning in questo problema.
Il problema principale è dato dall'algoritmo, in quanto l'applicazione di esso in matrici con bassa densità di foglie, o quando l'agente risucchia abbastanza foglie prima del raggiungimento del goal da diminuire abbastanza la densità di foglie nella matrice, con valori che variano dal 5% fino anche allo 0%, l'agente può incappare in loop che non gli permettono il raggiungimento del goal.
Il problema che emerge è basati sul funzionamento stesso del Q Learning, il quale utilizzando la Q Matrix per scegliere sempre la miglior mossa in un determinato stato può portare a bloccare l'agente in un ciclo di azioni che non gli permettono di raggiungere il goal anche se tecnicamente non ha ancora raggiunto il limite massimo di azioni compibili in quell'ambiente.
Vi sono due interessanti casi di studio:

- **Percentuale bassa di foglie**
- **Nessuna foglia nell'ambiente**

Nel caso di una percentuale bassa di foglie nell'ambiente l'agente può incappare in un loop in quanto la probabilità che una foglia si sposti in una delle caselle ad esso adiacente diventa circa $9/(n-1) \times (m-1) - 1$ per ogni foglia e questo in matrici di grandi dimensioni va a diminuire così tanto da renderlo un problema perchè l'agente solitamente si troverà nello stato senza alcuna foglia vicina, ovvero lo stato vuoto.
Perciò per come funziona il Q Learning avendo l'agente sempre nello stesso stato la Q Matrix ritornerà sempre la stessa mossa migliore così facendo l'agente arriverà ad uno dei lati dell'ambiente e solo allora vedrà un nuovo stato in quanto nei vicini si troveranno delle mura, come illustrato nell'esempio riportato sotto.

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \qquad\qquad \begin{matrix} 0 & 0 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 2 \end{matrix}$$

Figura 12: Esempio dell'azione `RIGHT` come scelta dalla Q Matrix nello stato vuoto

Arrivati al bordo della matrice avremo due possibilità, la prima è che durante l'allenamento l'agente è stato in grado di scegliere come mossa migliore lo spostarsi lungo il muro sino ad arrivare al goal, se invece questo non è accaduto, in quanto tutte le mosse possibili a parte lo scontrarsi contro il muro hanno lo stesso reward, se l'agente scegliesse di tornare indietro allora andrebbe a cadere proprio nel loop di azioni che non gli permetterebbe di raggiungere il goal se non fosse per un fortuito spostamento di una foglia nel suo vicinato così da fargli cambiare azione e possibilmente rompere il loop e fargli raggiungere il goal.
Purtroppo nel caso in cui nella matrice non vi sia più nessuna foglia se l'agente va a cadere nel loop sopra descritto non vi saranno modi per spostarlo da esso e perciò non potrà mai raggiungere il goal, anche con step infiniti a sua disposizione.
Per i due problemi sopra citati sono arrivato alla conclusione che l'utilizzo del Q Learning, per quanto efficacie fino ad una certa soglia, non è l'algoritmo consigliabile in questo problema in quanto l'algoritmo stesso potrebbe impedire all'agente di raggiungere il goal.
Una miglioria possibile sarebbe quella di introdurre una mossa casuale se l'agente va cadere in un loop di azioni come spiegato sopra, così da tentare di farlo comunque arrivare al goal entro il numero massimo di stap consentito.

5.2 Pensieri finali

Dopo aver eseguito differenti test, apportato delle modifiche all'agente ed al modello ed aver constatato i limiti dell'algoritmo di Q Learning posso concludere che sono riuscito a far apprendere all'agente come muoversi e risucchiare un considerevole numero di foglie prima di raggiungere lo stato goal con dei discreti risultati in quanto il problema assegnatomi essendo governato da movimenti casuali delle foglie non è possibile risolverlo completamente utilizzando questo algoritmo perchè anche il miglior agente ottenuto potrebbe non essere efficace in matrici di grandi o piccole dimensioni con una bassa densità di foglie in quanto può sempre bloccarsi in loop che lo portano a non raggiungere il goal finale.