Progetto Introduzione Intelligenza Artificiale
A.A. 2020/2021
Docente: Prof. Valentina Poggioni
Assistenti alla didattica: Dr. Alina Elena Baia, Dr. Gabriele Di Bari

Federici Alessandro
Dipartimento: Ingegneria Informatica
Matricola: 302326

Marchetti Edoardo
Dipartimento: Ingegneria Informatica
Matricola: 312879

Il progetto consiste nella realizzazione di un'applicazione di Intelligenza Artificiale completa di aspetti di gestione di: sensori per l'acquisizione dei dati dall'esterno relativi a stati e obiettivi, ragionamento/ricerca della soluzione per i goal acquisiti, esecutori per la realizzazione delle azioni che conducono alla soluzione.

Progetto Battaglia navale

Battaglia navale è un classico gioco a turni in cui lo scopo è schierare le proprie navi su un campo di battaglia e cercare di affondare quelle dell'avversario prima che lui affondi le tue senza sapere dove quest'ultimo le abbia posizionate. In questo progetto proponiamo una versione nella quale il programma dovrà riconoscere e ricreare il campo di battaglia per poi risolverlo affondando tutte le navi con il minor numero di tentativi possibili, restituendo in tempo reale le probabilità che una qualsiasi nave sia in una determinata casella. Lo stato iniziale del gioco verrà passato al sistema attraverso una foto di una griglia quadrata di grandezza libera(minima 3x3) rappresentante il campo di battaglia e la posizione delle navi le quali saranno rappresentati attraverso dei numeri secondo la leggenda seguente:

- 0 (Spazio vuoto)
- 2 (Nave da due)
- 3 (Nave da tre)
- 4 (nave da quattro)
- 5 (nave da cinque)

1. Descrizione dei vincoli:

- La griglia che si vuole far leggere deve essere una griglia quadrata (non ci sono vincoli per quanto riguarda il numero di righe e colonne in quanto il codice è scritto parametricamente rispetto alle linee che vengono rilevate).
- La griglia deve essere di dimensione minima 3x3.
- Le navi devono essere rappresentate attraverso degli indici (“2”, “3”, “4”, “5”) i quali vanno ad indicare anche il numero di blocchi che la nave occupa in griglia.
- Gli indici non per forza devono essere tutti presenti. Ad esempio è possibile schierare anche solo la nave da 3 e da 5.
- Una nave può essere disposta in orizzontale o in verticale.
- Due navi non possono essere adiacenti.

2. Acquisizione e classificazione degli input, stato iniziale e goal.

E’ stato realizzato un programma che, passata in input un’immagine contenente la griglia del campo di battaglia e la configurazione delle navi per lo stato iniziale:

a) Interpreti l’immagine individuando attraverso la libreria OpenCv la grandezza della griglia, le varie caselle e le navi attraverso un modello di classificazione basato sul dataset MNIST e la libreria keras in grado di leggere i numeri. Tale modello di classificazione ha un'architettura di tipo rete neurale convoluzionale. Inizialmente la scelta era ricaduta su un modello di NN classico con uno strato di input, uno o due strati nascosti e uno strato di output. Poichè però le predizioni fatte su le immagini di test provate dagli studenti si è deciso di provare ad implementare una rete di CNN con due strati convolutivi, un layer Flatten e infine uno strato di output. Nel tentativo di migliorare ancora le prestazioni, in quanto tramite delle verifiche grafiche si era notato che i livelli di accuracy e di loss andavano a peggiorare in fase di test, sono state effettuate alcune ricerche tramite forum e video su YouTube. Il modello finale che è stato prodotto è riportato nella figura sottostante:

```
cnn_model3 = models.Sequential()

# Feature Learning block:
cnn_model3.add(layers.Conv2D(30, (3,3), activation='relu', input_shape=(28,28,1)))
cnn_model3.add(layers.MaxPooling2D((2,2)))
cnn_model3.add(layers.Conv2D(15, (3,3), activation='relu'))
cnn_model3.add(layers.MaxPooling2D((2,2)))

#Dropuot layer
cnn_model3.add(layers.Dropout(rate=0.2))

# Classification block:
cnn_model3.add(layers.Flatten())
cnn_model3.add(layers.Dense(128, activation='relu'))
cnn_model3.add(layers.Dense(10, activation='softmax'))
```

Figura 2.1

La cosa che si ritiene più importante da sottolineare è l'utilizzo dello strato Dropout. Infatti tale strato permette in fase di training di andare a disattivare alcune connessioni all'interno della rete in modo che il modello si alleni a riconoscere le immagini con meno informazioni a disposizione. Nelle immagini riportate qua sotto si può vedere la differenza delle delle capacità predittive del modello con (Figura 2.2) e senza (Figura 2.3) il Dropout layer.

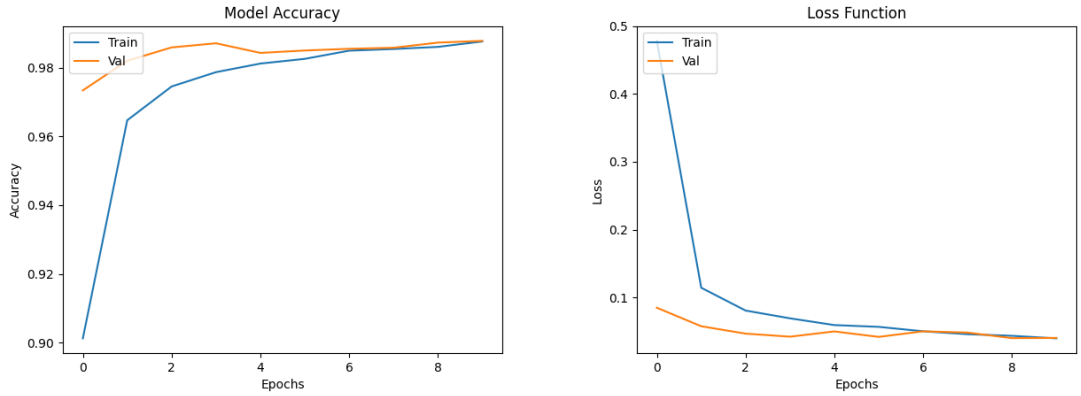


Figura 2.2

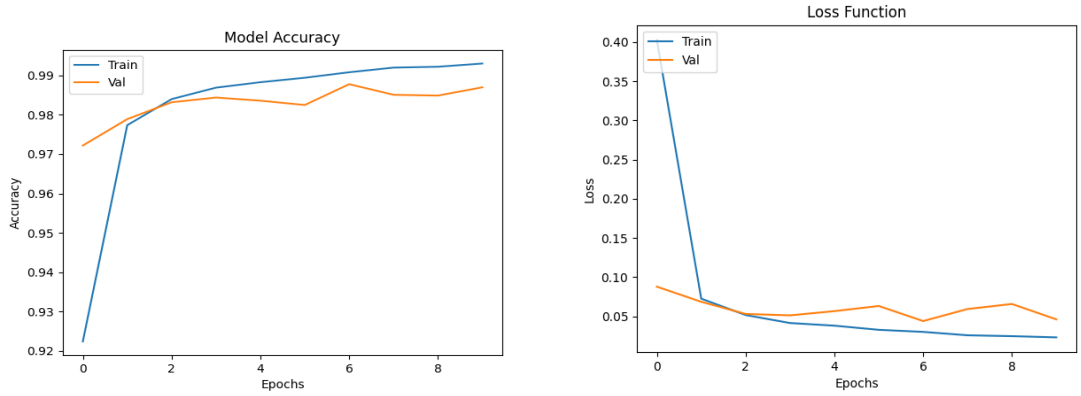


Figura 2.3

Nel progetto allegato sono comunque presenti tutti e tre i modelli utilizzati nelle varie fasi di sviluppo se si vuole fare un paragone delle prestazioni. Per indicare quale modello utilizzare al ImageReader è sufficiente andare nel file utilities e inserire il nome del modello nella funzione illustrata di seguito.

```
##Initialize model
def initialiazePredictionModel(): #Permette di caricare un modello per il riconoscimento dell'immagine
    model = tf.keras.models.load_model('cnn_model_3')
    return model
```

Figura 2.4

Tramite queste implementazioni però non si ottiene sempre una corretta lettura dell'immagine, infatti il programma non sempre legge correttamente i numeri presenti in griglia. Per tanto si è affiancato al modello di classificazione un correttore che in base alle letture del modello aggiusta la griglia per ottenere un risultato migliore.

Tale metodo scandaglia la griglia ottenuta dalla lettura dell'immagine e controlla ogni casella. Se la predizione fatta dal modello restituisce un valore non contenuto nei vincoli (ex 1, 6, 7 etc etc) il contenuto della casella viene impostato a 0(figura 2.5). Lo stesso accade per i numeri isolati o che non potrebbero rappresentare una nave, per esempio se sulla casella controllata è presente un 3, tale casella dovrebbe avere altri due 3 in altrettante caselle consequenziali per non essere impostato a 0, poiché il numero 3 rappresenta una nave occupante tre caselle.

```
if grid[i][j] == 1 or grid[i][j] > 5:
    grid[i][j]=0
```

Figura 2.5

b) Invochi il solutore (3. Implementazione della classe AI per la ricerca delle navi), il quale calcola le varie configurazioni possibili in base alla grandezza della griglia, al numero e tipo delle navi presenti, in modo da poter completare il gioco nel minor numero di mosse possibili. Il solutore creato per questo progetto, a fronte di una corretta lettura dell'immagine, è sempre in grado di portare a termine il gioco affondando tutte le navi presenti nel campo di battaglia.

3. Implementazione della classe AI per la ricerca delle navi.

La parte di ricerca delle navi all'interno della griglia viene svolta dalla classe AI.py. La classe Battlefield crea un'istanza di AI passandole il numero delle righe e delle colonne e l'array in cui sono contenuti gli oggetti Ship che rappresentano le navi.

```
ai = AI(self.num_row_column, self.ship)
[matrix_sum , maxProb] = ai.calcola_probabilita_iniziali()
grid.setMaxProb(maxProb)
grid.draw_probabilities(screen, matrix_sum)
```

Figura 3.1

```
def __init__(self, num_row_column, shipArray):
    self.matrix_sum = np.zeros((num_row_column, num_row_column))
    self.num_row_column = num_row_column
    self.num_ships = len(shipArray)
    self.ship_array = shipArray
    self.crea_matrici()
    self.reference_coordinates = np.array([-1,-1])
    self.frontier = np.array([])
    self.explored = []
```

Figura 3.2

Di seguito vengono illustrati le variabili inizializzate:

- *matrix_sum:* matrice di float. L'elemento alle coordinate (i,j) indica il numero di configurazioni che una casella può ospitare
- *num_row_column:* numero di righe e di colonne
- *num_ships:* numero di navi iniziali
- *ship_array:* array di oggetti Ship, usato per recuperare la dimensione delle navi e la cella del primo blocco della nave
- *reference_coordinates:* coppia di numeri in cui viene indicata la cella dell'ultimo colpo andato a segno; (-1,-1) indica che nessuna nave è stata ancora agganciata
- *frontier:* lista di celle che devono essere colpite prima di tentare un colpo basato sulla probabilità
- explored: lista di celle che sono già state colpite

La funzione `crea_matrici(self)` (il cui codice è illustrato nella figura 3.3) permette di andare ad inizializzare la tabella *matrix_list*. Una riga i matrix_list è costituita da: tag della nave (rappresentato dal numero di blocchi occupati), una matrice per le configurazioni orizzontali e una matrice per le configurazioni verticali.

```
def crea_matrici(self):
    self.matrix_list = np.empty((0,3)) #3 colonne = , indice nave, matrice_o, matrice_v

    for ship in self.ship_array:
        num_nave = ship.getTag()       #Poichè l'identificativo della nave è il numero di blocchi occupati
        matrice_o = np.zeros((self.num_row_column , self.num_row_column ), int)
        matrice_v = np.zeros((self.num_row_column , self.num_row_column ), int)

        self.matrix_list  = np.append(self.matrix_list , np.array([[ num_nave, matrice_o, matrice_v]]), axis=0)
```

Figura 3.3

Una volta terminata l'inizializzazione viene invocato il metodo `calcola_probabilita_iniziali(self)` (Figura 3.4) tramite il quale la classe AI calcola tutte le possibili configurazioni che ogni nave può assumere nella griglia popolando così le matrici *matrice_o* e *matrice_v*. Il metodo restituisce la matrice somma iniziale e il contenuto della cella che presenta la probabilità più alta di contenere una nave. La matrice somma viene aggiornata ogni volta che viene eseguito un colpo tramite la funzione `aggiorna_matrice_somma(self)`, il quale restituisce i due parametri appena descritti.
Terminato il calcolo delle probabilità iniziali vengono passati dei parametri all'oggetto grid per la visualizzazione grafica della griglia e dalla distribuzione delle configurazioni.

```python
def calcola_probabilita_iniziali(self):

    #Ciclo per scorrere le navi
    i = 0
    for ship in self.ship_array:
        dim_ship = ship.getTag()
        matrice_o = self.matrix_list[i][1]

        #Ciclo per inizializzare l'orizzontale
        for j in range(self.num_row_column):
            for k in range(self.num_row_column):
                if(k+dim_ship-1 < self.num_row_column):
                    z = 0
                    while z < dim_ship:
                        matrice_o[j][k+z] = matrice_o[j][k+z] + 1
                        z = z+1

        self.matrix_list[i][1] = matrice_o


        #Ciclo per inizializzare verticale
        matrice_v = self.matrix_list[i][2]
        for j in range(self.num_row_column):
            for k in range(self.num_row_column):
                if(j+dim_ship-1 < self.num_row_column):
                    z = 0
                    while z < dim_ship:
                        matrice_v[j+z][k] = matrice_v[j+z][k] + 1
                        z = z+1

        self.matrix_list[i][2] = matrice_v

        [matrix_sum, max] = self.aggiorna_matrice_somma()
        i = i+1


    return [matrix_sum, max]
```

Figura 3.4

```python
def aggiorna_matrice_somma(self):
    self.matrix_sum = np.zeros((self.num_row_column, self.num_row_column))

    for i in range(self.matrix_list.shape[0]):
        self.matrix_sum = self.matrix_sum + self.matrix_list[i][1]
        self.matrix_sum = self.matrix_sum + self.matrix_list[i][2]

    self.stampa_matrice_somma()

    #Ricerca del massimo
    max = 0.0
    for i in range(self.matrix_sum.shape[0]):
        for j in range(self.matrix_sum.shape[1]):
            if self.matrix_sum[i][j] > max:
                max = self.matrix_sum[i][j]

    return [self.matrix_sum, max]
```

Figura 3.5

Ogni volta che durante il main loop per la visualizzazione della finestra di PyGame viene premuta la barra spaziatrice viene invocato il metodo *shot()* tramite il quale l'istanza di AI decide quali coordinate sceglie di colpire:

- se la lista *Frontier* è **vuota** allora viene scelta la cella che contiene il valore più alto. In caso di più celle con lo stesso valore allora viene scelta la prima che si incontra nella scansione.
- se la lista *Frontier* **non** è **vuota** allora viene scelto il primo elemento della lista.

```
def shot(self):
    if(not self.frontier.size == 0): #Devo scegliere in base a dove ho colpito
        print(".............Scelgo da frontier.............")
        #[row, column] = self.frontier[0].get_coordinates()
        [row, column] = self.frontier[0].get_coordinates()
        return [row, column]
    else:
        print(".............Scelgo prob.................")
        max = np.array([-1,-1, -1]) #[riga,colonna,probabilità]
        for j in range(self.num_row_column):
                for k in range(self.num_row_column):
                    if(max[2] < self.matrix_sum[j][k]):
                        max = [j,k, self.matrix_sum[j][k]]

        return [max[0], max[1]]
```

Figura 3.6

Definite le coordinate in cui tentare il colpo viene fatto un check tramite l'oggetto grid per verificare se la coppia (riga,colonna) non era già stata scelta in precedenza. Se ciò è vero, il colpo viene notificato alla griglia e AI pone a 0 il valore della cella (riga,colonna) della matrice_somma tramite `set_to_zero_hit_cell(self, hit_row, hit_column)` (Figura 3.7).

```
if(grid.available_square(row, column)):
    esito = grid.setHit(row,column)
    ai.set_to_zero_hit_cell(row, column)
```

Figura 3.7a

```
def set_to_zero_hit_cell(self, hit_row, hit_column):
    for i in range(self.matrix_list.shape[0]):
        matrice_o = self.matrix_list[i][1]
        matrice_v = self.matrix_list[i][2]

        matrice_o[hit_row][hit_column] = 0
        matrice_v[hit_row][hit_column] = 0

        self.matrix_list[i][1] = matrice_o
        self.matrix_list[i][2] = matrice_v


        self.aggiorna_matrice_somma()
```

Il ciclo for permette di aggiornare la matrice orizzontale e verticale di ogni nave ancora in gioco.

Al termine viene aggiornata la matrice somma.

Figura 3.7b

Successivamente si verifica l'esito del colpo (ottenuto da grid.setHit() della figura 3.7a). Se è stata colpita una nave allora la cella in cui si trova viene posta a -1 nelle matrici di tutte le navi tramite il metodo `set_to_minus_one_cell(self, hit_row, hit_column)` (il quale opera in maniera simile a set_to_zero_hit_cell) , viene ridotta la vita della nave colpita (riga

3 di figura 3.8) e quindi si verifica se è stata affondata. Nel caso in cui la nave risulti essere affondata allora la classe AI rimuove le matrici della nave tramite la procedura `remove_matrix(self, sunk_ship)`. Inoltre vengono "simulati" dei colpi nelle celle adiacenti alla nave in quanto, come illustrato nei vincoli, non è possibile schierare due navi vicine.

```
if(esito == 'hit'):
    ai.set_to_minus_one_cell(row, column)
    for i in range(self.ship.size):
        sunk = self.ship[i].checkHit(self.ship[i].getIsVertical(), row, column)
        if (sunk):
            esito = 'sunk'
            ai.remove_matrix(self.ship[i])
            grid.set_hit_around_ship(self.ship[i])
            self.ship = np.delete(self.ship, i)

            break
```

Figura 3.8

```
def remove_matrix(self, sunk_ship):


    #Rimuovo la riga associata alla nave affondata
    removed = False
    i = 0
    while i < self.matrix_list.shape[0] and (not removed):
        if (self.matrix_list[i][0] == sunk_ship.getTag()):
            self.matrix_list = np.delete(self.matrix_list, i, 0)
            removed = True
        i = i+1

    #Caratteristiche nave affondata
    sr = sunk_ship.getStartRow()
    sc = sunk_ship.getStartColumn()
    bo = sunk_ship.getTag()

    #Aggiorno le matrici delle altre navi poichè non possono stare attacate
    #Simulo dei colpi con esito WATER intorno alla nave

    if(sunk_ship.getIsVertical()):
            #Nave affondata in verticale
            for row in range(sr-1, sr+bo+1):
                if(row >= 0 and row < self.num_row_column):
                    for column in range (sc-1, sc+2):
                        if(column >= 0 and column<self.num_row_column and self.matrix_sum[row][column] > 0):
                            self.set_to_zero_hit_cell(row,column)
                            self.aggiorna_matrici(row,column)

    else:
        #Nave affondata in orizzontale
        for row in range(sr-1, sr+2):
            if (row >= 0 and row < self.num_row_column):
                for column in range (sc-1, sc+bo+1):
                    if(column >= 0 and column<self.num_row_column and self.matrix_sum[row][column] > 0):
                        self.set_to_zero_hit_cell(row,column)
                        self.aggiorna_matrici(row,column)
```

Figura 3.9

Rimosse la riga relativa alla nave affondata vengono ricavate le informazioni sulla nave necessarie per "colpire" correttamente tutte le celle adiacenti.

La simulazione di colpi sulle caselle che fanno da contorno alla nave viene eseguita tramite set_to_zero_hit_cell e `aggiorna_matrici(self, hit_row, hit_column)`,metodo che verrà spiegato tra poco.

Successivamente, come mostrato dalla figura 3.8, viene notificato il tutto anche all'oggetto grid così da avere anche un riscontro grafico dell'evoluzione delle possibili configurazioni.

Terminato il check sull'esito della nave vengono ricalcolate le nuove probabilità tramite `calcola_nuove_probabilita(self, hit_row, hit_column, esito)`.

```
matrix_sum = ai.calcola_nuove_probabilita(row, column, esito)
grid.draw_figures(screen)
grid.draw_probabilities(screen, matrix_sum)
```

Figura 3.10

```
def calcola_nuove_probabilita(self, hit_row, hit_column, esito):

    if(esito == 'water'):
        self.aggiorna_matrici(hit_row, hit_column)

    elif(esito == 'hit'):
        #Se non ho agganciato ancora nessuna nave salvo dove l'ho trovata
        if(self.reference_coordinates[0] == -1 and self.reference_coordinates[1] == -1):
            self.reference_coordinates = [hit_row, hit_column]

        self.aggiorna_frontier(hit_row, hit_column)

    elif(esito == 'sunk'):
        self.frontier = np.array([])
        self.reference_coordinates = np.array([-1, -1])


    self.aggiorna_matrice_somma()
    self.remove_node_from_frontier(hit_row, hit_column)
    return  self.matrix_sum
```

Figura 3.11

Come si può vedere in Figura 3.11 ci sono tre casi possibili:

- esito = 'water', in questo caso vengono aggiornate semplicemente le matrici
- esito = 'hit', se le reference_coordinates valgono (-1,-1) significa che fino a quel momento si stava sparando seguendo i valori della matrix_sum. Si imposta quindi reference_coordinates = (hit_row, hit_column) così da poter aggiornare correttamente la lista di frontier tramite la procedura `aggiorna_frontier(self, hit_row, hit_column)`invocata subito dopo. Se invece già si era agganciata la nave allora i frontier verranno aggiornati prendendo come riferimento reference_coordinates per capire se la nave è in orizzontale o in verticale. Non vengono aggiornate le matrici in quanto fino a quando non si ricade nel caso tre il metodo shot non usa la matrix_sum

- esito = ‘sunk’, viene svuotata la lista dei frontier e si resetta reference_coordinates. Anche qui l’aggiornamento delle matrici non è richiesto in quanto è già stato fatto da remove_matrix.

Infine si aggiorna la matrice somma e viene rimossa da frontier la coppia (hit_row, hit_column) se presente.

Di seguito sono illustrate le funzioni `aggiorna_matrici(self, hit_row, hit_column)`(Figura 3.12), `aggiorna_frontier(self, hit_row, hit_column)` (Figura 3.13 e 3.14) e `remove_node_from_frontier(self, row_to_remove ,column_to_remove)` (Figura 3.15) .

```
def aggiorna_matrici(self, hit_row, hit_column):

    #Scadisco la lista delle matrici nave per nave
    for i in range(int (self.matrix_list.shape[0])):

        blockOccupied = self.matrix_list[i][0] #indice nave = blocchi occupati
        matrice_o = self.matrix_list[i][1]
        matrice_v = self.matrix_list[i][2]

        #Aggiorno orizzontali
        for j in range(hit_column - blockOccupied +1 , hit_column +1):
            if(j>=0 and j+blockOccupied-1 < self.num_row_column and (not (hit_row, j) in self.explored) ):
                good = True

                for k in range(blockOccupied):
                    if(j+k != hit_column and matrice_o[hit_row][j+k] == 0 ):
                        good = False

                if good:
                    for k in range(blockOccupied):
                        if(matrice_o[hit_row][j+k] > 0):
                            matrice_o[hit_row][j+k] = matrice_o[hit_row][j+k] - 1


        #Aggiorno verticali
        for j in range(hit_row-blockOccupied+1 , hit_row+1):
            if(j>=0 and j+blockOccupied-1 < self.num_row_column and (not (j, hit_column) in self.explored)):
                good = True
                for k in range(blockOccupied):
                    if(j+k != hit_row and matrice_v[j+k][hit_column] == 0 ):
                        good = False

                if good:
                    for k in range(blockOccupied):
                        if(matrice_v[j+k][hit_column] > 0):
                            matrice_v[j+k][hit_column] = matrice_v[j+k][hit_column] - 1



        self.matrix_list[i][1] = matrice_o
        self.matrix_list[i][2] = matrice_v

    self.explored.append((hit_row, hit_column))
```

Figura 3.12

In *aggiorna_matrici* vengono aggiornate le matrici orizzontali e verticali di ogni singola nave. La logica di base è quella di prendere la cella colpita e andare a decrementare di uno tutte le celle che sono coinvolte in una disposizione della nave valida che comprende le coordinate (hit_row, hit_column). Ad esempio per la nave da due blocchi, se il colpo è stato effettuato in

(2,2) si verificano le configurazioni [(2,1), (2,2)] e [(2,2),(2,3)] nella matrice orizzontale e [(1,2)(2,2)] e [(2,2)(2,3)] nella matrice verticale. Infine si aggiunge (hit_row, hit_column) a explored per evitare con in un momento successivo si decrementino delle celle per delle configurazioni non valide.

In *aggiorna_frontier* si possono avere due casi:

- la lista di frontier è **vuota** (Figura 3.13), allora si inseriscono le 4 celle adiacenti a quella colpita.
- la lista di frontier **non** è **vuota** (Figura 3.14), allora si deve osservare reference_coordinates per capire se la nave è orizzontale o in verticale e quindi aggiungere e togliere le celle corrette.

```
if(self.frontier.size == 0): #Aggiungo le caselle adiacenti con prob != 0
        #Sopra
        if (hit_row - 1 >= 0 and self.matrix_sum[hit_row-1][hit_column] > 0 ):
            self.frontier = np.append(self.frontier,  Node(hit_row-1, hit_column, self.matrix_sum[hit_row-1][hit_column]))
            self.printFrontier()

        #Sotto
        if(hit_row + 1 < self.num_row_column and self.matrix_sum[hit_row+1][hit_column] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row+1, hit_column, self.matrix_sum[hit_row+1][hit_column]))
            self.printFrontier()

        #Destra
        if(hit_column + 1 < self.num_row_column and self.matrix_sum[hit_row][hit_column+1] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row, hit_column+1, self.matrix_sum[hit_row][hit_column+1]))
            self.printFrontier()

        #Sinistra
        if(hit_column-1 >= 0 and self.matrix_sum[hit_row][hit_column-1] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row, hit_column-1, self.matrix_sum[hit_row][hit_column-1]))
            self.printFrontier()
```

Figura 3.13

```
else:
    #Aggiungo le caselle adiacenti a quella che ho colpito con prob != 0 e con hit_row == reference_row (oppure con hit_column == reference_column)

    #verifico se hit_row == reference row
    if (hit_row == self.reference_coordinates[0]):

        #Se SI allora la nave è in orizzontale

        #Tolgo i nodi con row != reference_row
        self.remove_node_from_frontier(self.reference_coordinates[0] - 1, self.reference_coordinates[1])
        self.remove_node_from_frontier(self.reference_coordinates[0] + 1, self.reference_coordinates[1])

        #Aggiungo i nodi adiacenti sulla stessa riga e con prob != 0
        #Destra
        if(hit_column + 1 < self.num_row_column and self.matrix_sum[hit_row][hit_column+1] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row, hit_column+1, self.matrix_sum[hit_row][hit_column+1]))
            self.printFrontier()

        #Sinistra
        if(hit_column-1 >= 0 and self.matrix_sum[hit_row][hit_column-1] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row, hit_column-1, self.matrix_sum[hit_row][hit_column-1]))
            self.printFrontier()

    else:
        #Se NO allora nave in verticale
        #Tolgo i nodi con column != reference_column
        self.remove_node_from_frontier(self.reference_coordinates[0], self.reference_coordinates[1] - 1)
        self.remove_node_from_frontier(self.reference_coordinates[0], self.reference_coordinates[1] + 1)

        #Aggiungo i nodi adiacenti sulla stessa riga e con prob != 0
        #Sopra
        if (hit_row - 1 >= 0 and self.matrix_sum[hit_row-1][hit_column] > 0 ):
            self.frontier = np.append(self.frontier,  Node(hit_row-1, hit_column, self.matrix_sum[hit_row-1][hit_column]))
            self.printFrontier()

        #Sotto
        if(hit_row + 1 < self.num_row_column and self.matrix_sum[hit_row+1][hit_column] > 0):
            self.frontier = np.append(self.frontier, Node(hit_row+1, hit_column, self.matrix_sum[hit_row+1][hit_column]))
            self.printFrontier()
```

Figura 3.14

Infine con *remove_matrix* si va semplicemente a rimuovere la coppia di coordinate richiesta.

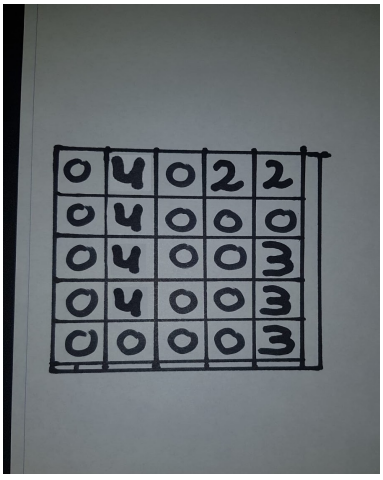
```
def remove_node_from_frontier(self, row_to_remove ,column_to_remove):
    #Devo rimuovere il nodo con coordinate passate
    i = 0
    removed = False
    while (i < self.frontier.size and (not removed)):
        node = self.frontier[i]
        if node.get_row() == row_to_remove and node.get_column() == column_to_remove:
            self.frontier = np.delete(self.frontier, i)
            removed = True
        i = i+1

    self.printFrontier()
```

Figura 3.15

4. Esecuzione

1. Si disegni una griglia rispettando i vincoli imposti dal progetto

Figura 4.1



2. Si faccia una foto alla griglia e la si passi al programma
3. Una volta avviato il programma, quando avrà visualizzato e ricreato lo stato iniziale, premendo spazio si potrà visualizzare il completamento dell'obiettivo passo dopo passo

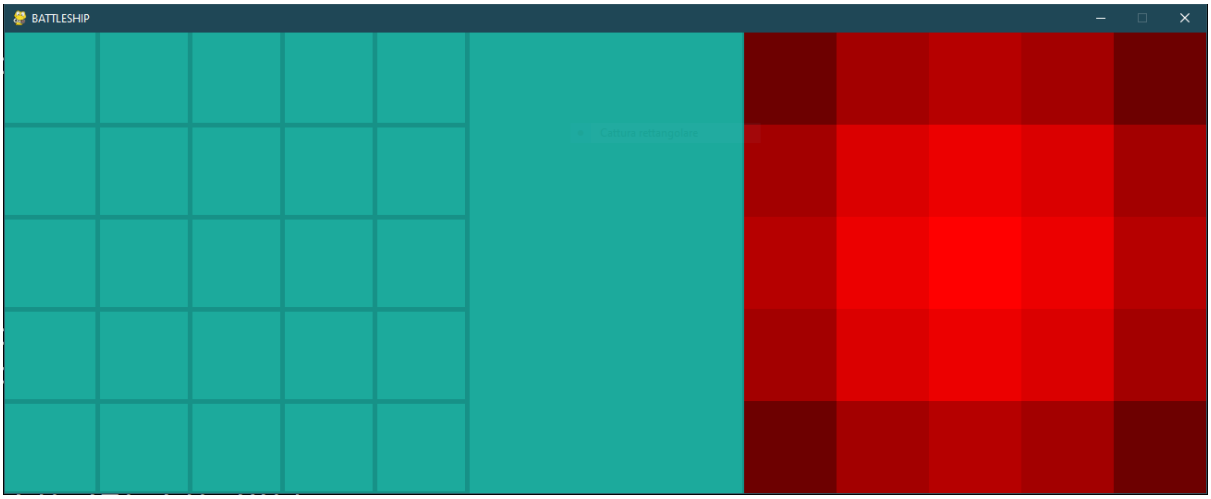


Figura 4.3a

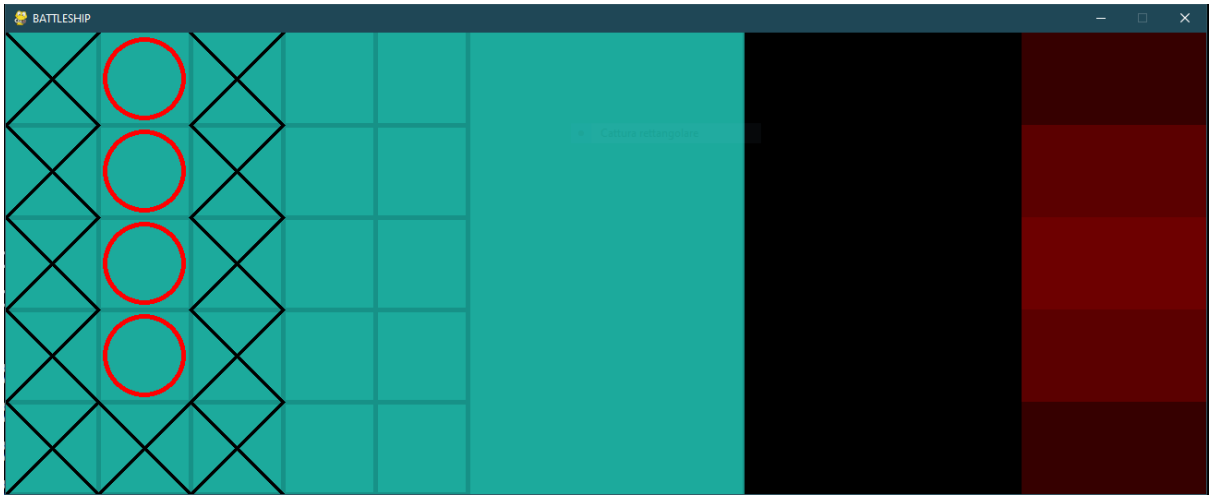


Figura 4.3b

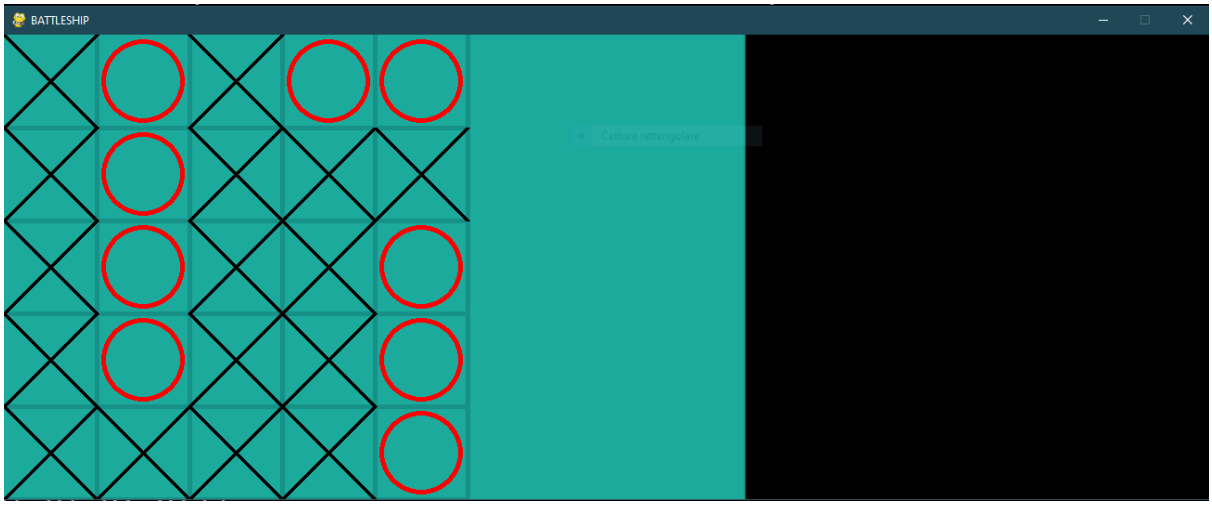


Figura 4.3c