

# Documentazione AI - 1 assignment

Piccirilli, Addario

16 ottobre 2023

## 1 Introduzione

Inizialmente, abbiamo iniziato a studiare il codice in modo tale da comprendere cosa facesse ogni modulo del progetto: difatti, il codice si presentava gerarchicamente diviso tra agenti ed ambiente, dove ogni sotto-gerarchia faceva da ambiente per l'agente della gerarchia superiore.

## 2 Creare un nuovo "Environment"

Piuttosto che creare una nuova classe che ereditasse le caratteristiche della classe Environment, si è scelto di modificare i parametri passati a **Rob.env** per modificare le coordinate dei muri rendendo più difficoltoso il passaggio del bot all'interno del suo ambiente.

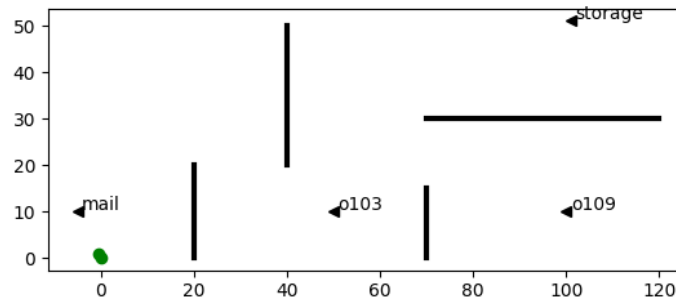


Figura 1: Immagine di come si presentano i muri inizialmente

## 3 Controller opportunistico

*"The current controller visits the locations in the **to\_do** list sequentially. Change the controller so that it is opportunistic; when it selects the next location to visit, it selects the location that is closest to its current position. It should still visit all the locations."*

La consegna di chiedeva di rendere il controllore del movimento opportunistico in modo tale che scegliesse di visitare per prima la locazione più vicina alla sua posizione corrente. Pertanto, abbiamo modificato il metodo **do** della classe **Rob\_top\_layer** in modo tale che calcolasse la distanza euclidea

dalla posizione corrente del bot verso le altre locazioni da raggiungere. Successivamente, viene scelta di visitare la locazione più vicina, ossia quella con la distanza minore. Una volta raggiunta tale locazione, questa viene rimossa dalla lista **to\_do** e si passa alla successiva iterazione dove vengono ricalcolate le distanze.

```

1     def do(self, plan):
2         """carry out actions.
3         actions is of the form {'visit':list_of_locations}
4         It visits the locations in turn.
5         """
6
7         to_do = plan['visit']
8         to_do_length = len(to_do)
9         for i in range(to_do_length):
10            rob_x, rob_y = self.middle.env.rob_x, self.middle.env.
11                rob_y
12            print(f"Rob_x:_{rob_x},_Rob_y:_{rob_y}")
13            distance_list = [euclidean_distance(rob_x, rob_y, *self.
14                locations[pos]) for pos in to_do]
15            print("Distance_List:", distance_list)
16            next_pos_index = distance_list.index(min(distance_list))
17            print("Position", self.locations[to_do[next_pos_index]], "
18                To_do:", to_do, "Locations:", self.locations)
19            arrived = self.middle.do({'go_to': self.locations[to_do[
20                next_pos_index]], 'timeout': self.timeout})
21            self.display(1, "Arrived_at", to_do[next_pos_index],
22                arrived)
23            if arrived:
24                to_do.pop(next_pos_index)

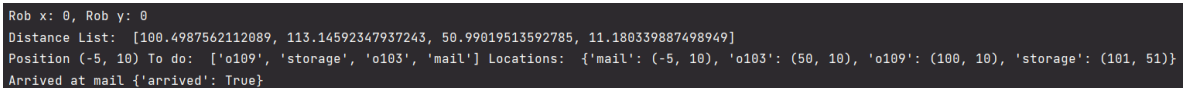
```

Nel file **Distance.py** è presente il metodo **euclidean\_distance**:

```

1     def euclidean_distance(x1, y1, x2, y2):
2         return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

```



```

Rob x: 0, Rob y: 0
Distance List: [100.4987562112089, 113.14592347937243, 50.99819513592785, 11.180339887498949]
Position (-5, 10) To do: ['o109', 'storage', 'o103', 'mail'] Locations: {'mail': (-5, 10), 'o103': (50, 10), 'o109': (100, 10), 'storage': (101, 51)}
Arrived at mail {'arrived': True}

```

Figura 2: Immagine della stampa della console

In questo modo, anche se passiamo le locazioni da visitare in modo non ordinato, il bot andrà sempre a visitare quella più vicina a lui.

```

1 top.do({'visit': ['o109', 'storage', 'o103', 'mail']})

```

## 4 Environment dinamico

*"Optional: create an environment in which the walls (obstacles) are dynamic, means they change their position overtime (dynamic environment)."*

Abbiamo deciso di implementare due pattern di muri: in particolare, il primo pattern viene disegnato e applicato fisicamente all'environment all'inizio dell'esecuzione del programma, mentre il secondo pattern viene applicato dopo che un timer, precedentemente scelto, arriva a 0. Infatti, il timer viene decrementato ogni *tick* durante l'esecuzione.

Di seguito la dichiarazione dell'environment con i due pattern dei muri:

```

1  env = Rob_env(walls=
2      {((20, 0), (20, 20)),
3        ((40, 20), (40, 50)),
4        ((70, 30), (120, 30)),
5        ((70, 0), (70, 15))},
6      other_walls=
7      {((40, 0), (50, 30)),
8        ((80, 40), (120, 30))})

```

Tutto il codice successivo è presente all'interno del file **agentMiddle.py**.  
Codice che gestisce l'aggiornamento dei pattern:

```

1  self.timer = 50 # timer to count the number of ticks between an
   env to the other
2
3  ...
4
5  if self.timer == 0:
6      switch_env()
7      plot_new_env()
8      self.timer = 50
9
10 self.timer -= 1

```

Infine, il codice che gestisce il cambio dell'environment e la funzione che cancella e disegna i nuovi muri:

```

1  def switch_env():
2      # Exchanging the walls
3      temp = self.env.env.walls
4      self.env.env.walls = self.env.env.other_walls
5      self.env.env.other_walls = temp
6
7  def plot_new_env():
8      # Print the actual walls
9      print("Changed environment", self.env.env.walls)
10
11     # Remove old walls
12     for wall in self.env.env.other_walls:
13         a, b = wall
14         x1, y1 = a
15         x2, y2 = b
16
17         print(f"x1: {x1}, y1: {y1}, x2: {x2}, y2: {y2}")
18         plt.plot((x1, x2), (y1, y2), "-w", linewidth=4)
19
20     # Draw the new wall
21     for wall in self.env.env.walls:
22         a, b = wall
23         x1, y1 = a
24         x2, y2 = b
25
26         print(f"x1: {x1}, y1: {y1}, x2: {x2}, y2: {y2}")
27         plt.plot((x1, x2), (y1, y2), "-k", linewidth=3)

```