Andrea Augello
Department of Engineering, University of Palermo, Italy

Agenti

Università degli Studi di Palermo

Definizione di agente

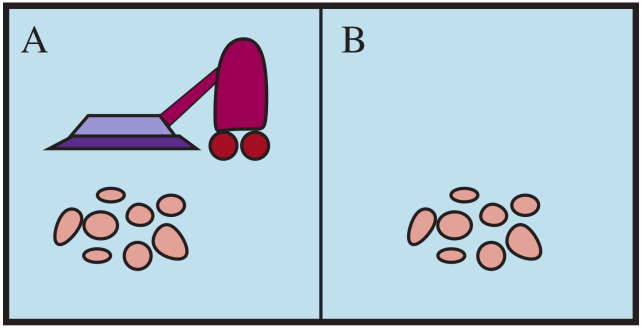
Agente

Un **agente** è un'entità che **percepisce** il suo **ambiente** attraverso dei **sensori** e agisce su di esso attraverso degli **attuatori**.

Caso studio

Caso studio

Prendiamo in considerazione un aspirapolvere automatico in un mondo a blocchi.



- **Sensori**: percepisce la posizione e la presenza di sporcizia
- **Attuatori**: può muoversi e pulire
- **Obiettivo**: pulire tutta la stanza

Modellazione dell'ambiente

L'ambiente del problema può essere modellata attraverso una classe `Environment` che contiene:

- Un dizionario che associa ad ogni posizione la condizione `Dirty` o `Clean`
- Metodo `perceive`: restituisce la condizione della posizione dell'agente
- Metodo `clean`: pulisce la posizione in cui si trova l'agente
- Metodo `__str__`: restituisce una stringa riassuntiva dello stato dell'ambiente

Modellazione dell'ambiente

L'ambiente del problema può essere modellata attraverso una classe `Environment` che contiene:

- Un dizionario che associa ad ogni posizione la condizione `Dirty` o `Clean`
- Metodo `perceive`: restituisce la condizione della posizione dell'agente
- Metodo `clean`: pulisce la posizione in cui si trova l'agente
- Metodo `__str__`: restituisce una stringa riassuntiva dello stato dell'ambiente

```
class Environment():
    def __init__(self):
        self.locationCondition = {'A':'Dirty', 'B':'Dirty'}

    def perceive(self, agent):
        return self.locationCondition[agent.location]

    def clean(self, agent):
        self.locationCondition[agent.location] = 'Clean'

    def __str__(self):
        return str(self.locationCondition)
```

Modellazione dell'agente

L'agente può essere modellato attraverso una classe `Vacuum` che contiene:

- Attributi:
  - Un riferimento all'ambiente in cui si trova
  - La posizione in cui si trova
- Metodi:
  - `perceive`: chiama il metodo `perceive` dell'ambiente
  - `move`: muove l'agente nella direzione specificata

Modellazione dell'agente

```
class Vacuum():
    def __init__(self, environment):
        self.environment = environment
        self.location = "A"

    def perceive(self):
        return self.environment.perceive(self)

    def get_location(self):
        return self.location

    def move(self, direction):
        if direction == 'Right' and self.location == 'A':
            self.location = 'B'
        elif direction == 'Left' and self.location == 'B':
            self.location = 'A'
        else:
            pass

    def clean(self):
        self.environment.clean(self)
```

Agente controllato manualmente

```
python -i vaacum.py
```

```
env = Environment()
vacuum = Vacuum(env)
print(env)                  # {'A': 'Dirty', 'B': 'Dirty'}
vacuum.perceive()
vacuum.clean()
print(env)                  # {'A': 'Clean', 'B': 'Dirty'}
vacuum.move('Right')
vacuum.perceive()
vacuum.clean()
print(env)                  # {'A': 'Clean', 'B': 'Clean'}
```

Agente controllato manualmente

```
python -i vaacum.py
```

```
env = Environment()
vacuum = Vacuum(env)
print(env)                  # {'A': 'Dirty', 'B': 'Dirty'}
vacuum.perceive()
vacuum.clean()
print(env)                  # {'A': 'Clean', 'B': 'Dirty'}
vacuum.move('Right')
vacuum.perceive()
vacuum.clean()
print(env)                  # {'A': 'Clean', 'B': 'Clean'}
```

Non è un agente in grado di agire autonomamente!
Possiamo definire un agente che agisce in modo autonomo?

Funzione agente esplicita

Funzione agente esplicita

I possibili stati sono abbastanza pochi da poter essere elencati esplicitamente, determinando per ciascuno di essi l'azione da compiere.

Posizione agente	Stato cella	Azione
A	Clean	Right
A	Dirty	Clean
B	Clean	Left
B	Dirty	Clean

Funzione agente esplicita

Implementiamo la funzione agente esplicita come un metodo della classe `TableVacuum`, che estende la classe `Vacuum`.

```
class TableVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)
```

Funzione agente esplicita

Implementiamo la funzione agente esplicita come un metodo della classe `TableVacuum`, che estende la classe `Vacuum`.

```
class TableVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)

    def agentFunction(self):
        status = self.perceive()
        if    self.get_location() == 'A' and status == 'Clean':
            self.move('Right')
        elif self.get_location() == 'A' and status == 'Dirty':
            self.clean()
        elif self.get_location() == 'B' and status == 'Clean':
            self.move('Left')
        elif self.get_location() == 'B' and status == 'Dirty':
            self.clean()
```

Funzione agente esplicita — Limitazioni

- La funzione agente esplicita può essere definita solo per problemi con un numero finito di stati
- Gli stati possono essere anche molto numerosi, rendendo difficile la definizione della funzione agente
- Non necessariamente siamo in grado di stabilire a priori quale sia l'azione corretta per ogni stato
- Se l'agente deve essere dotato di memoria, la tabella cresce in modo esponenziale

L'intelligenza artificiale si pone come obiettivo quello di definire agenti in grado di produrre comportamenti razionali da un programma relativamente piccolo invece che da una vasta tabella di decisione.

Agente basato su riflesso

Agente basato su riflesso

Un agente basato su riflesso sceglie le azioni in base allo stato corrente dell'ambiente, senza tener conto degli stati precedenti.

L'agente implementato dalla tabella precedente è un agente basato su riflesso.

Agente basato su riflesso

Un agente basato su riflesso sceglie le azioni in base allo stato corrente dell'ambiente, senza tener conto degli stati precedenti.

L'agente implementato dalla tabella precedente è un agente basato su riflesso.

```
class ReflexVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)

    def agentFunction(self):
        status = self.perceive()
        if status == 'Dirty':
            self.clean()
        else:
            self.move( 'Right' if self.get_location() == 'A' else
                'Left')
```

Agente basato su riflesso — Limitazioni

- L’agente funziona correttamente solo se l’ambiente è completamente osservabile
- L’agente può facilmente entrare in loop
- La randomizzazione può essere utile per evitare i loop, ma non è una soluzione generale. Algoritmi deterministici possono essere più efficienti e sicuri.

Agente cieco

Agente cieco

Analizziamo un agente “cieco” che non è in grado di percepire la propria posizione

```
class BlindVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)

    def get_location(self):
        return ""
```

Come possiamo definire una funzione agente per questo agente?

Agente cieco

```
import random

class BlindVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)

    def get_location(self):
        return ""

    def agentFunction(self):
        status = self.perceive()
        if status == 'Dirty':
            self.clean()
        else:
            self.move( 'Right' if random.randint(0,1) == 0 else
                'Left')
```

Agente basato su modello

Agente basato su modello

- Il modo più efficace di gestire l'osservabilità parziale è quello di mantenere un modello interno dell'ambiente.
- Il modello dipende dalle percezioni precedenti e dalle azioni effettuate dall'agente.
- In generale, potrebbe essere necessario tenere traccia di come l'ambiente evolve nel tempo.

Agente basato su modello

Implementiamo `ModelVacuum` come una classe che estende `Vacuum` e che contiene un attributo `model` che rappresenta il modello interno dell'agente. Questo attributo può essere usato come condizione di stop.

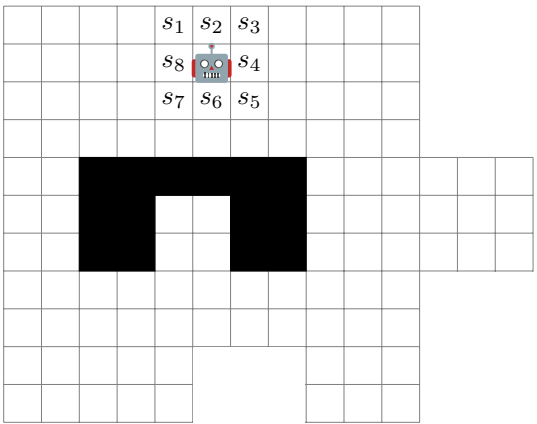
Agente basato su modello

Implementiamo `ModelVacuum` come una classe che estende `Vacuum` e che contiene un attributo `model` che rappresenta il modello interno dell'agente. Questo attributo può essere usato come condizione di stop.

```
class ModelVacuum(Vacuum):
    def __init__(self, environment):
        super().__init__(environment)
        self.model = {'A':None, 'B':None}

    def agentFunction(self):
        status = self.perceive()
        self.model[self.get_location()] = status
        if self.model['A'] == self.model['B'] == 'Clean':
            print('Done')
        elif status == 'Dirty':
            self.clean()
        else:
            self.move( 'Right' if self.get_location() == 'A' else
                'Left')
```

Un caso più complesso

Un caso più complesso



Vogliamo che il robot esegua il seguente compito:

- andare verso una cella adiacente ad un confine della stanza o ad un oggetto
- percorre il confine o il perimetro dell'oggetto

Inoltre:

- Il robot può percepire solo le 8 celle adiacenti $s_1, \ldots, s_8$
- Il robot non ha nessuna informazione sulla forma della stanza e sulla posizione degli oggetti
- Il robot si può muovere nelle 4 direzioni cardinali, a meno che non si trovi in una cella adiacente ad un oggetto o ad un confine

Modellare l'ambiente

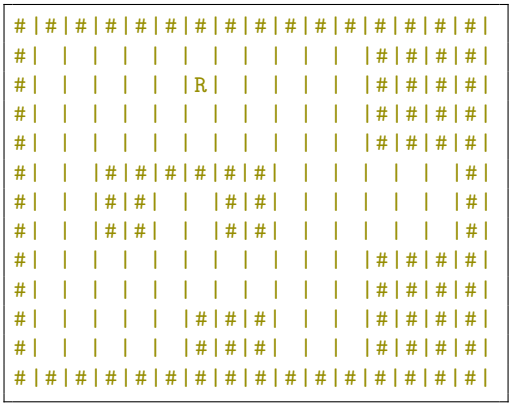
```
import numpy as np
class Environment():
    def __init__(self):
        #11x14 grid with 1 cell thick border
        self.grid = np.zeros((13,16))
        #set border cells to 1
        self.grid[0,:] = 1
        self.grid[12,:] = 1
        self.grid[:,0] = 1
        self.grid[:,12:] = 1
        self.grid[11,6:9] = 1
        self.grid[10,6:9] = 1
        self.grid[5:8,12:15] = 0
        #add obstacle
        self.grid[5:8,3:9] = 1
        self.grid[6:8,5:7] = 0
        #place robot
        self.robot = (2,6)
```

La griglia è rappresentata come una matrice di valori binari, dove 1 indica una cella occupata e 0 una cella libera.

Memorizziamo la posizione del robot come una tupla di coordinate (x, y).

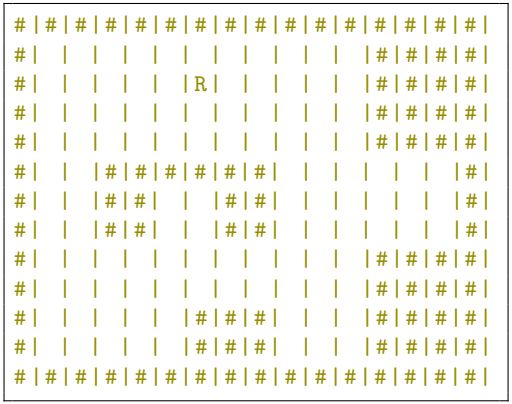
Modellare l'ambiente

Vogliamo anche poter stampare la griglia con dell'ASCII art in modo da poter visualizzare lo stato dell'ambiente.



Modellare l'ambiente

Vogliamo anche poter stampare la griglia con dell'ASCII art in modo da poter visualizzare lo stato dell'ambiente.




```
def __str__(self):
    s = ""
    for i,row in
        enumerate(self.grid):
        for j,cell in
            enumerate(row):
            s += "R|" if (i,j) ==
                self.robot else
                "#|" if cell == 1
                else " |"
        s += "\n"
    return s
```

Le percezioni del robot

L’ambiente deve restituire un vettore di percezioni che rappresenta lo stato dell’ambiente intorno al robot.

Le percezioni del robot

L'ambiente deve restituire un vettore di percezioni che rappresenta lo stato dell'ambiente intorno al robot.

Magia nera con Numpy:

```
def percieve(self):
    x,y = self.robot
    #3x3 grid around the robot
    contour =
        self.grid[x-1:x+2,y-1:y+2]
    contour = contour.flatten()
    # remove the center cell
    contour =
        np.delete(contour,4)
    return contour
```

Le percezioni del robot

L'ambiente deve restituire un vettore di percezioni che rappresenta lo stato dell'ambiente intorno al robot.

Magia nera con Numpy:

```
def percieve(self):
    x,y = self.robot
    #3x3 grid around the robot
    contour =
        self.grid[x-1:x+2,y-1:y+2]
    contour = contour.flatten()
    # remove the center cell
    contour =
        np.delete(contour,4)
    return contour
```

Metodo tradizionale

```
def percieve(self):
    x,y = self.robot
    s = []
    for i in range(-1,2):
        for j in range(-1,2):
            if (i == 0 and j == 0):
                continue
            s.append(self.grid[x+i,y+j])
    return s
```

Le percezioni del robot

L'ambiente deve restituire un vettore di percezioni che rappresenta lo stato dell'ambiente intorno al robot.

Magia nera con Numpy:

```
def percieve(self):
    x,y = self.robot
    #3x3 grid around the robot
    contour =
        self.grid[x-1:x+2,y-1:y+2]
    contour = contour.flatten()
    # remove the center cell
    contour =
        np.delete(contour,4)
    return contour
```

Metodo tradizionale

```
def percieve(self):
    x,y = self.robot
    s = []
    for i in range(-1,2):
        for j in range(-1,2):
            if (i == 0 and j == 0):
                continue
            s.append(self.grid[x+i,y+j])
    return s
```

```
def percieve(self):
    x,y = self.robot
    return [ self.grid[x+i,y+j] for i in range(-1,2) for j in range(-1,2)
        if not (i == 0 and j == 0) ]
```

Muovere il robot

Il robot deve potersi muovere nelle 4 direzioni cardinali, `Up`, `Down`, `Left` e `Right`.

Muovere il robot

Il robot deve potersi muovere nelle 4 direzioni cardinali, `Up`, `Down`, `Left` e `Right`.

Se il robot si trova in una cella adiacente ad un oggetto o ad un confine, non può muoversi in quella direzione e rimane fermo.

Muovere il robot

Il robot deve potersi muovere nelle 4 direzioni cardinali, `Up`, `Down`, `Left` e `Right`.

Se il robot si trova in una cella adiacente ad un oggetto o ad un confine, non può muoversi in quella direzione e rimane fermo.

```
def move(self,action):
    new_pos = list(self.robot)
    if   action == "Up":
        new_pos[0] -= 1
    elif action == "Down":
        new_pos[0] += 1
    elif action == "Left":
        new_pos[1] -= 1
    elif action == "Right":
        new_pos[1] += 1
    new_pos = tuple(new_pos)
    if self.grid[new_pos] != 1:
        self.robot = new_pos
```

Modellare il robot

Il robot avrà come attributo l'ambiente in cui si trova e il metodo `action` che implementa la funzione agente.

L'azione di default, se non vengono percepiti oggetti o confini, è quella di muoversi verso l'alto.

Modellare il robot

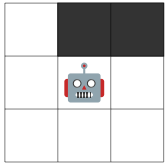
Il robot avrà come attributo l'ambiente in cui si trova e il metodo `action` che implementa la funzione agente.

L'azione di default, se non vengono percepiti oggetti o confini, è quella di muoversi verso l'alto.

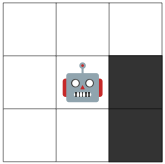
```
class Robot():
    def __init__(self, environment):
        self.environment = environment

    def action(self):
        perception = self.environment.percieve()
        if sum(perception) == 0:
            environment.move("Up")
        #...
```

Ragioniamo su feature composte

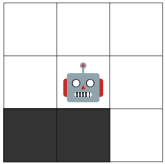
Le feature assumono valore `True` quando almeno una delle celle è occupata.



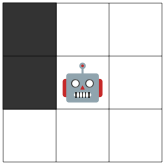
x_1



x_2



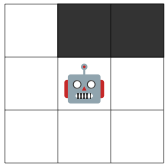
x_3



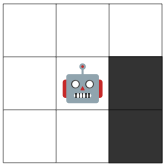
x_4

Ragioniamo su feature composte

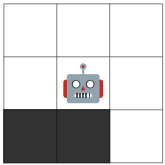
Le feature assumono valore `True` quando almeno una delle celle è occupata.



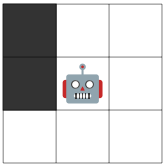
x_1



x_2



x_3



x_4

```
    def action(self):
    #Le percezioni vanno riordinate per essere in senso orario
        perception = [perception[i] for i in [0,1,2,4,7,6,5,3]]
        x1 = sum(perception[1:3] ) > 0
        x2 = sum(perception[3:5] ) > 0
        x3 = sum(perception[5:7] ) > 0
        x4 = perception[0]+ perception[7] > 0
```

La funzione agente completa

- Sempre verso l'alto finché non si incontra un oggetto o un confine
- Se sopra è ostruito e a destra è libero, destra
- Se la destra è ostruita e sotto è libero, giù
- Se sotto è ostruito e a sinistra è libero, sinistra
- Se a sinistra è ostruito e sopra è libero, su

La funzione agente completa

- Sempre verso l'alto finché non si incontra un oggetto o un confine
- Se sopra è ostruito e a destra è libero, destra
- Se la destra è ostruita e sotto è libero, giù
- Se sotto è ostruito e a sinistra è libero, sinistra
- Se a sinistra è ostruito e sopra è libero, su

```
def action(self):
    perception = self.environment.percieve()
    if sum(perception) == 0:
        self.environment.move("Up")
    perception = [perception[i] for i in
        [0,1,2,4,7,6,5,3]]
    x1 = sum(perception[1:3] ) > 0
    x2 = sum(perception[3:5] ) > 0
    x3 = sum(perception[5:7] ) > 0
    x4 = perception[0]+ perception[7] > 0
    if x1 and not x2:
        self.environment.move("Right")
    elif x2 and not x3:
        self.environment.move("Down")
    elif x3 and not x4:
        self.environment.move("Left")
    elif x4 and not x1:
        self.environment.move("Up")
```