Andrea Augello (andrea.augello01@unipa.it)
Department of Engineering, University of Palermo, Italy

Agenti

Università degli Studi di Palermo

Definizione di agente

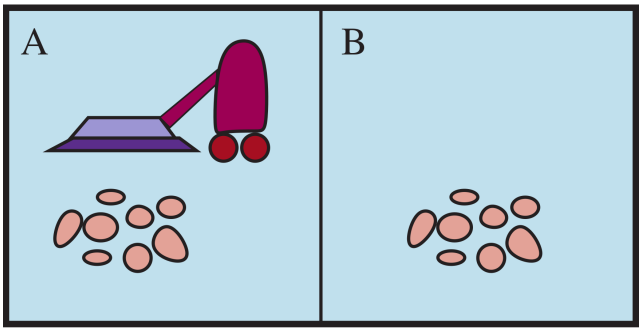
Agente

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 nella posizione corrente (Ambiente parzialmente osservabile)
- **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`, inizialmente entrambe `Dirty` (dall'esterno può essere osservato solo lo stato della posizione in cui si trova l'agente, implementare la property `status`)
- Un attributo che rappresenta la posizione dell'agente (con relativi metodi per implementare la property)
- Metodo `_step`: manda avanti il tempo nell'ambiente (utile per ambienti dinamici)
- Metodo `move_vacuum`: muove l'agente in una direzione specificata
- Metodo `clean`: pulisce la posizione in cui si trova l'agente
- Metodo `__str__`: restituisce una stringa riassuntiva dello stato dell'ambiente

Modellazione dell'ambiente

```
class Environment():
    def __init__(self):
        pass
    def _step(self):
        pass
    @property
    def status(self) -> str:
        pass
    @property
    def vacuum_location(self) -> str:
        pass
    @vacuum_location.setter
    def vacuum_location(self, location : str):
        pass
    def move_vacuum(self, direction: str):
        pass
    def clean(self):
        pass
    def __str__(self) -> str:
        pass
```

Modellazione dell'agente

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

- Attributi:
  - Un riferimento all'ambiente in cui si trova
- Metodi:
  - `perceive`: restituisce la property `status` dell'ambiente
  - `get_location`: restituisce la property `vacuum_location` dell'ambiente
  - `apply_action`: applica un'azione all'ambiente (muovi a destra, muovi a sinistra, pulisci)
  - `agent_function`: funzione agente che determina l'azione da compiere (per ora vuota)

Modellazione dell'agente

```
class Vacuum():
    def __init__(self, environment : Environment):
        pass

    def perceive(self) -> str:
        pass

    def get_location(self) -> str:
        pass

    def apply_action(self, action : str):
        pass

    def agent_function(self):
        pass
```

Agente controllato manualmente

`ManualVacuum` è una classe che estende `Vacuum` e che permette di controllare manualmente l'agente. La funzione `agent_function` sarà un loop che chiede all'utente di inserire un'azione da compiere.

Agente controllato manualmente

`ManualVacuum` è una classe che estende `Vacuum` e che permette di controllare manualmente l'agente. La funzione `agent_function` sarà un loop che chiede all'utente di inserire un'azione da compiere.

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

    def agent_function(self):
        print("Possible actions: 'R' (Right), 'L' (Left), 'C'
            (Clean), 'P' (Perceive), 'Q' (Quit)")
        while (x := input('\tEnter action: ').upper()) != 'Q':
            if x == 'P':
                print(f"\t\tLocation: {self.get_location()}, Status:
                    {self.perceive()}")
            else:
                self.apply_action(x)
```

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
A, A	Dirty, Clean	Right
A, B	Clean, Dirty	Clean
B, A	Clean, Dirty	Clean
B, B	Dirty, Clean	Left
A, A, B	Dirty, Clean, Dirty	Clean
B, B, A	Dirty, Clean, 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 : Environment):
        super().__init__(environment)
        self.table = {
            (('A','Clean'),):'R',
            (('A','Dirty'),):'C',
            (('B','Clean'),):'L',
            (('B','Dirty'),):'C',
            (('A','Dirty'),('A','Clean')):'R',
            (('A','Clean'),('B','Dirty')):'C',
            (('B','Clean'),('A','Dirty')):'C',
            (('B','Dirty'),('B','Clean')):'L',
            (('A','Dirty'),('A','Clean'),('B','Dirty')):'C',
            (('B','Dirty'),('B','Clean'),('A','Dirty')):'C'
        }
        self.perception_sequence = []
```

Funzione agente esplicita

`agent_function` dovrà percepire lo stato dell'ambiente e aggiungerlo alla sequenza delle percezioni, quindi dovrà cercare la sequenza di percezioni nella tabella e applicare l'azione corrispondente.

Se la sequenza di percezioni non è presente nella tabella, l'agente non sa cosa fare e si ferma (idealmente lasciando tutto pulito).

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.

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.

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

    def agent_function(self):
        while input('Continue? (Y/n): ').lower() != 'n':
            if self.perceive() == 'Dirty':
                self.apply_action('C')
            elif self.get_location() == 'A':
                self.apply_action('R')
            else:
                self.apply_action('L')
```

Agente basato su riflesso — Limitazioni

- L’agente funziona correttamente solo se l’ambiente è 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 : Environment):
        super().__init__(environment)

    def perceive(self) -> str:
        return ""

    def get_location(self) -> str:
        return ""
```

Come possiamo definire una funzione agente per questo agente?

Agente cieco

```
import random

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

    def perceive(self) -> str:
        return ""

    def get_location(self) -> str:
        return ""

    def agent_function(self):
        while input('Continue? (Y/n): ').lower() != 'n':
            self.apply_action('C')
            self.apply_action( 'R' if random.randint(0,1) == 0 else
                'L')
```

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 : Environment):
        super().__init__(environment)
        self.model = {'A':None, 'B':None}

    def agent_function(self):
        while self.model != {'A':'Clean', 'B':'Clean'}:
            pass
```

Diversi tipi di ambienti

Proviamo adesso a creare varianti dell'ambiente con le seguenti caratteristiche:

- `NonDeterministicEnvironment`: le azioni di pulizia e di movimento potrebbero non avere successo
- `DynamicEnvironment`: l'ambiente evolve nel tempo, ad ogni step c'è la possibilità che una cella a caso diventi sporca
- `NoisyEnvironment`: l'agente percepisce lo stato dell'ambiente in modo non accurato, lo stato della cella potrebbe risultare pulito anche se è sporco, e la posizione dell'agente potrebbe essere percepita in modo errato.

Come si comportano gli agenti in questi ambienti?