# Documentazione Progetto AI

Analisi degli Algoritmi e dell'Ambiente

7 febbraio 2026

## 1 Esplorazione dell'Ambiente e Dinamiche

Prima di applicare gli algoritmi risolutivi, è fondamentale analizzare le proprietà del *Markov Decision Process* (MDP) definito dall'ambiente `AquaticEnv-v0`. Il codice di ispezione ci permette di comprendere la struttura delle ricompense e delle transizioni.

### 1.1 La Struttura delle Ricompense (`env.RS`)

L'oggetto `env.RS` rappresenta la **Funzione di Ricompensa** (Reward Function) $R(s)$. In questo ambiente, la ricompensa è modellata come funzione del solo stato di arrivo. `env.RS` è un vettore di dimensione pari al numero di stati $|\mathcal{S}|$, dove l'elemento $i$-esimo contiene la ricompensa immediata ottenuta entrando nello stato $i$.

Matematicamente:

$$R(s) = \texttt{env.RS}[s]$$

Ad esempio, per lo stato obiettivo (Goal, indicato con "G"), ci si aspetta un valore positivo, mentre per gli stati ostacolo un valore negativo o nullo.

### 1.2 Matrice di Transizione (`env.T`)

La matrice (o tensore) `env.T` definisce la dinamica del sistema, ovvero la probabilità di transizione $P(s'|s, a)$:

$$P(s'|s, a) = \texttt{env.T}[s, a, s']$$

Dove $s$ è lo stato corrente, $a$ l'azione e $s'$ lo stato successivo. L'analisi del codice mette a confronto due tipologie di stati:

1. **Stato Deterministico** (es. `state = 1`): L'azione scelta porta quasi sicuramente allo stato desiderato. La probabilità di successo è vicina a 1.
2. **Stato Stocastico** (es. `state = 13`): Qui l'ambiente introduce incertezza (simulando ad esempio correnti acquatiche). Anche scegliendo un'azione precisa, la probabilità di finire nello stato target potrebbe essere bassa, distribuendosi su altri stati adiacenti.

## 2 Algoritmo: Value Iteration

L'obiettivo di questo algoritmo è calcolare la funzione valore ottimale $V^*(s)$ per ogni stato $s$ dell'ambiente, risolvendo iterativamente l'Equazione di Bellman per l'ottimalità.

## 2.1 Inizializzazione

La funzione `acquaticValueIteration` inizia inizializzando due vettori di dimensione pari al numero di stati (`env.observation_space.n`):

- `U_1`: rappresenta i valori allo step corrente, matematicamente $V_{k+1}(s)$.
- `U`: rappresenta i valori allo step precedente, matematicamente $V_k(s)$.

Inizialmente, $V_0(s) = 0$ per tutti gli stati.

## 2.2 Il Ciclo Principale

L'algoritmo entra in un ciclo `while True` che continua finché i valori non convergono o si raggiunge il limite `maxIterazione`. Ad ogni iterazione $k$, per ogni stato $s$, viene eseguito un aggiornamento basato sulla natura dello stato.

### 2.2.1 Gestione degli Stati Terminali

Se lo stato corrente è un obiettivo (indicato da `"G"` nella griglia):

$$\text{if } \texttt{env.grid[state]} == \texttt{"G"}$$

Il valore dello stato viene fissato semplicemente alla sua ricompensa immediata, poiché non ci sono transizioni future:

$$V_{k+1}(s) = R(s)$$

Dove `env.RS[state]` corrisponde a $R(s)$.

### 2.2.2 Aggiornamento di Bellman (Stati Non Terminali)

Per tutti gli altri stati, l'algoritmo calcola il valore atteso per ogni possibile azione. Questo corrisponde al blocco di codice:

```
insiemePunteggiAzioni[action] +=
    env.T[state, action, nextState] * U[nextState]
```

Matematicamente, per ogni azione $a$, si calcola il valore $Q$ (Quality):

$$Q_k(s, a) = \sum_{s'} P(s'|s, a) \cdot V_k(s')$$

Dove:

- `env.T[state, action, nextState]` è la probabilità di transizione $P(s'|s, a)$.
- `U[nextState]` è il valore dello stato successivo all'iterazione precedente $V_k(s')$.

Successivamente, si applica l'operatore di Bellman per trovare il valore ottimale dello stato massimizzando sulle azioni:

$$V_{k+1}(s) = R(s) + \gamma \cdot \max_a \{Q_k(s, a)\}$$

Nel codice Python:

$$\texttt{U\_1[state]} = \texttt{env.RS[state]} + \texttt{discount} \cdot \max(\texttt{insiemePunteggiAzioni})$$

### 2.3 Condizione di Terminazione

Alla fine di ogni scansione degli stati, viene calcolata la massima differenza ("norma infinito") tra il vettore dei valori nuovi e vecchi:

$$\delta = \max_s |V_{k+1}(s) - V_k(s)|$$

Nel codice: `delta = max(delta, abs(U_1[state] - U[state]))`.

L'algoritmo termina se:

$$\delta < \epsilon \cdot \frac{1-\gamma}{\gamma}$$

Questa specifica condizione garantisce che l'errore sulla policy finale sia limitato. Se la condizione è vera, il ciclo si interrompe (`break`).

### 2.4 Output

La funzione restituisce:

1. La **Policy Ottimale** $\pi^*(s)$, ottenuta chiamando la funzione ausiliaria `values_to_policy` sul vettore dei valori finali.
2. Il vettore dei valori `U` che approssima $V^*(s)$.

## 3 Algoritmo: Policy Iteration

Mentre la Value Iteration cerca di trovare direttamente la funzione valore ottimale, la **Policy Iteration** separa il problema in due sotto-fasi distinte che vengono ripetute ciclicamente: la valutazione della policy corrente e il suo miglioramento.

La funzione `acquaticPolicyIteration` implementa questo approccio.

### 3.1 Inizializzazione

L'algoritmo inizia con una policy arbitraria (in questo caso, l'azione 0 per tutti gli stati):

```
policy = [0 for _ in range(env.observation_space.n)]
```

Anche i valori degli stati `U` vengono inizializzati a zero.

### 3.2 Fase 1: Policy Evaluation (Valutazione)

All'interno del ciclo principale, il primo blocco di codice (`while True`) serve a calcolare il valore $V^\pi(s)$ della policy corrente $\pi$. A differenza della Value Iteration dove cerchiamo il *massimo* tra le azioni, qui l'azione è fissata dalla policy corrente:

$$a = \pi(s) = \texttt{policy[state]}$$

L'aggiornamento del valore diventa quindi un'equazione lineare (risolta qui iterativamente):

$$V^\pi_{k+1}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) \cdot V^\pi_k(s')$$

Nel codice Python:

```
u_somma += env.T[state, policy[state], nextState] * U[nextState]
U_1[state] = env.RS[state] + discount * u_somma
```

Questa fase termina quando i valori convergono per la policy fissata (condizione su `delta`).

### 3.3 Fase 2: Policy Improvement (Miglioramento)

Una volta calcolati i valori accurati per la policy corrente, l'algoritmo verifica se esiste un'azione che fornisce un risultato migliore rispetto a quella attualmente scelta.

Per ogni stato $s$, viene calcolato il valore Q per tutte le possibili azioni $a$:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a) \cdot V^{\pi}(s')$$

Nel codice, questo vettore è chiamato `insiemePunteggiAzioni`.

Successivamente, si confronta la migliore azione possibile con quella attuale:

$$\text{Se } \max_a Q^{\pi}(s,a) > Q^{\pi}(s,\pi(s))$$

Allora la policy viene aggiornata avidamente (*Greedy Update*):

$$\pi_{new}(s) = \arg\max_a Q^{\pi}(s,a)$$

Nel codice:

```
if max(insiemePunteggiAzioni) > sommaPolicy:
    policy[state] = argmax(insiemePunteggiAzioni)
    unchanged = False
```

### 3.4 Terminazione

L'algoritmo termina quando la policy diventa **stabile**. La variabile booleana `unchanged` (inizializzata a `True` all'inizio del ciclo esterno) rimane vera solo se, durante la fase di miglioramento, non è stata trovata nessuna azione migliore per nessuno stato.

if unchanged: break

Questo garantisce che la policy trovata sia quella ottimale $\pi^*$.

## 4 Algoritmi visti