Documentazione Progetto AI

Analisi degli Algoritmi e dell'Ambiente

8 febbraio 2026

1 Modellazione del Problema

Il problema è modellato come un Markov Decision Process (MDP) con:

- Spazio degli stati finito (griglia $10 \times 10 \rightarrow 100$ stati),
- Spazio delle azioni discreto (L, R, U, D),
- Transizioni stocastiche (celle con correnti),
- Ricompense immediate (costo di movimento, bonus energia, goal).

2 Confronto tra gli Algoritmi

Perché la scelta di usare entrambi gli algoritmi?

Il **Value Iteration** converge alla funzione valore ottimale V^* solo in modo asintotico: ci si ferma quando il cambiamento tra due iterazioni è inferiore alla soglia `maxDifferenza`, tuttavia non ho mai la certezza matematica assoluta di aver raggiunto il valore esatto, ma solo un'approssimazione molto stretta.

Il **Policy Iteration** converge alla policy ottimale in un numero finito di iterazioni. La PI esplora sistematicamente le opzioni fino a trovare quella perfetta. Il calcolo costa però al computer maggior tempo e maggior spazio, in correlazione a quanto grande è il problema da affrontare o l'ambiente preso in considerazione.

3 VALUE ITERATION

La Value Iteration ha come parametri:

- `env`: ambiente MDP;
- `maxIterazione`: limite per evitare loop infiniti;
- `discount` (γ): fattore di sconto;
- `maxDifferenza` (ϵ): soglia di convergenza citata in precedenza.

3.1 Alcune precisazioni

Il fattore di sconto, `discount`, serve a dare un valore attuale ai premi futuri. Senza questo valore l'algoritmo non sarebbe in grado di distinguere una strategia che guadagna poco ma per molto molto tempo e una strategia che guadagna tanto subito. Per visualizzarlo meglio immaginiamo un labirinto in cui la meta vale 100 punti; se la raggiungo in un passo, il valore è 100, se la raggiungo in due passi il valore muta in $100 \times 0.9 = 90$ e, seguendo questo andamento, se la raggiungessi in dieci passi, il valore diventerebbe circa 34. Analogamente, i malus che si possono trovare nell'ambiente sono tanto più "a valore pieno" quanto più vicini al premio. In definitiva, i valori che vengono calcolati avranno indicativamente, proprio come un'onda, un valore maggiore in vicinanza dell'obiettivo e sempre minore man mano che ci si allontana perché i valori calcolati allontanandocisi saranno il prodotto di sconti già subiti dai valori prima calcolati. Come un'onda che parte dall'obiettivo, i valori si scremano man mano che questa si diffonde.

3.2 Inizializzazione

Inizializzo due funzioni di valori (liste):

- `U[]`: funzione di valore alla iterazione precedente;
- `U_1[]`: funzione di valore aggiornata.

che avranno inizialmente tutti gli stati a 0.

3.3 Ciclo Principale

Inizia il ciclo principale `while True`, dove all'interno ho:

- `delta`: misura la massima variazione tra due iterazioni; servirà per verificare la convergenza
- copiamo `U_1` in `U`

Ora `U` è la "vecchia" funzione di valore perché da ora in avanti verrà aggiornata solo `U_1`.
Per ogni stato dell'ambiente, `for state`, io controllo:

- Se lo stato è finale, in questo caso solo se è Goal, io gli do il valore +20 pescandolo dai reward dell'ambiente.
- Altrimenti, per tutti gli altri stati:
  1. Creo una lista `insiemePunteggiAzioni[]`, inizializzato a zero;
  2. Per ogni azione, `for action`, e per ogni stato prossimo, `for nextState`, aggiorno la lista di `insiemePunteggiAzioni[]` sommando la probabilità, dato un certo stato, di finire in quello stato finale utilizzando quell'azione; tutto questo moltiplicato per il valore dello stato prossimo contenuto in `U`.

La probabilità di transizione gestisce automaticamente la stocasticità delle correnti e si usa la vecchia funzione di valore `U`. Quindi `insiemePunteggiAzioni[]` alla fine avrà 4 valori corrispondenti a [valore utilizzando LEFT, valore utilizzando RIGHT, valore utilizzando UP, valore utilizzando DOWN].

Fatto ciò, devo inserire in `U_1` i valori aggiornati e quindi ogni stato assumerà il valore che aveva prima sommato al discount moltiplicato per il valore massimo contenuto in `insiemePunteggiAzioni[]`, questo per far sì che venga scelta l'azione che mi aspetto sia migliore.

3.4 Convergenza

Grazie al `delta` vedo a quanto risale la differenza tra lo stato aggiornato contenuto in `U_1` ed il suo corrispettivo vecchio stato contenuto in `U`. Se la variazione massima `delta` tra due iterazioni successive della funzione di valore è minore di

$$\texttt{maxDifferenza} \times \frac{(1 - \texttt{discount})}{\texttt{discount}}$$

oppure se è stato raggiunto il numero massimo di iterazioni consentite, allora l'algoritmo uscirà dal ciclo `while` e verrà ritornata la policy greedy. In breve, se `delta` è sufficientemente piccolo, la distanza tra la funzione di valore stimata e quella ottima è limitata superiormente.

Dalla funzione di valore finale, `return`, si estrae la policy greedy e per ogni stato si seleziona l'azione che massimizza il valore atteso. In particolare, `np.asarray(U)` converte la lista Python `U` in un array NumPy; serve solo per compatibilità con la funzione `values_to_policy`, non cambia il contenuto, solo il tipo.

4 POLICY ITERATION

La Policy Iteration ha come parametri principali:

- `env`: ambiente MDP;
- `maxIterazione`: limite per evitare loop infiniti;
- `discount` (γ): fattore di sconto che, come nella Value Iteration, serve a dare un valore attuale ai premi futuri;
- `maxDifferenza` (ϵ): soglia di convergenza per la valutazione della policy.

4.1 Inizializzazione

Inizializzo due funzioni di valori (liste) e la policy casuale:

- `U[]`: funzione di valore alla iterazione precedente;
- `U_1[]`: funzione di valore aggiornata;
- `policy`: azione da eseguire nello stato s; inizialmente tutte le azioni sono settate a 0.

4.2 Ciclo esterno

Il ciclo principale, `while True`, continua finché la policy cambia oppure non viene raggiunto il numero massimo di iterazioni. `cont` tiene traccia delle iterazioni esterne, cioè dei cicli Policy Evaluation + Policy Improvement.

4.3 Policy Evaluation – valutare la policy corrente

La policy corrente viene valutata calcolando i valori di tutti gli stati seguendo quella policy fissa.

Per ogni stato dell'ambiente, `for state`, io controllo:

- Se lo stato è finale, in questo caso solo se è Goal, io gli do il valore +20 pescandolo dai reward dell'ambiente.
- Altrimenti, per tutti gli altri stati:
  1. Inizializzo una variabile `u_somma` a zero, che conterrà per ogni stato prossimo, `for nextState`, la somma della probabilità, dato un certo stato, di finire in quello stato finale utilizzando l'azione della policy di quello stato; tutto questo moltiplicato per il valore dello stato prossimo contenuto in `U`.
  2. Fatto ciò, devo inserire in `U_1` i valori aggiornati e quindi ogni stato assumerà il valore che aveva prima sommato al discount moltiplicato per `u_somma`.

Al contrario del VI, non ho da scegliere tra i valori delle quattro mosse perché sto seguendo la policy.

Grazie al `delta` vedo a quanto risale la differenza tra lo stato aggiornato contenuto in `U_1` ed il suo corrispettivo vecchio stato contenuto in `U`. Se la variazione massima `delta` tra due iterazioni successive della funzione di valore è minore di

$$\texttt{maxDifferenza} \times \frac{(1 - \texttt{discount})}{\texttt{discount}}$$

oppure se è stato raggiunto il numero massimo di iterazioni consentite, allora l'algoritmo uscirà dal ciclo `while` e verrà ritornata la policy greedy. In breve, se `delta` è sufficientemente piccolo, la distanza tra la funzione di valore stimata e quella ottima è limitata superiormente.

4.4 Policy Improvement – aggiornare la policy

4.4.1 `for` che apre nuovi orizzonti utilizzando tutte le mosse

Per ogni stato dell'ambiente, `for state`, si calcolano i valori attesi di tutte le possibili azioni nello stato attuale, il classico `insiemePunteggiAzioni[]`, ora a zero, contenente i valori di LEFT, RIGHT, UP e DOWN.

Quindi per ogni azione, `for action`, e per ogni stato prossimo, `for nextState`, aggiorno la lista di `insiemePunteggiAzioni[]` sommando la probabilità, dato un certo stato, di finire in quello stato finale utilizzando quell'azione; tutto questo moltiplicato per il valore dello stato prossimo contenuto in `U`.

4.4.2 `for` che ricalcola i valori utilizzando solo la mossa della policy

Inizializzo `sommaPolicy`, il valore atteso dello stato seguendo la policy corrente, per ora a zero; piccola curiosità: il nostro `sommaPolicy` ha la stessa funzione del `u_somma` precedentemente incontrato.

Per ogni stato prossimo, `for nextState`, `sommaPolicy` assumerà il valore della probabilità, dato un certo stato, di finire in quello stato finale utilizzando quell'azione; tutto questo moltiplicato per il valore dello stato prossimo contenuto in `U`.

4.4.3 `if` che controlla se la mossa della policy può essere migliorata

Se un valore della migliore mossa calcolata su uno stato è maggiore della mossa della policy sul medesimo stato, devo aggiornare la mossa della policy e segnare che è avvenuto un cambiamento, `unchanged = False`.

In caso `unchanged` risulti vero, e quindi la policy non è stata aggiornata, oppure il contatore ha superato le iterazioni, il ciclo esterno termina, permettendo di ritornare il risultato.

Dalla funzione di valore finale, `return`, si estrae la policy ottima e per ogni stato si seleziona l'azione che massimizza il valore atteso. In particolare, `np.asarray(U)` converte la lista Python `U` in un array NumPy; serve solo per compatibilità con la funzione `values_to_policy`, non cambia il contenuto, solo il tipo.

5 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.

5.1 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.

5.2 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.

6 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.

6.1 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.

6.2 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.

6.2.1 Gestione degli Stati Terminali

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

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

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 $R(s)$.

6.2.2 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:

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

6.3 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`).

6.4 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 $V^*(s)$.

7 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.

7.1 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.

7.2 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 π. 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`).

7.3 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
```

7.4 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.

$$\text{if unchanged: break}$$

Questo garantisce che la policy trovata sia quella ottimale π^*.

8 Algoritmi visti

UNINFORMED SEARCH

9 Analisi dell'Algoritmo: BFS Tree Search

Il codice implementa la *Breadth-First Search* (BFS) nella variante **Tree Search**. A differenza della *Graph Search*, questa versione non tiene traccia degli stati già visitati (non utilizza un insieme *explored*), il che la rende adatta solo ad ambienti privi di cicli o dove la ridondanza è gestibile.

L'algoritmo esplora lo spazio degli stati livello per livello, garantendo di trovare la soluzione più breve (in termini di numero di passi) se ne esiste una.

9.1 Inizializzazione

La funzione inizia creando il nodo radice corrispondente allo stato iniziale del problema:

```
node = Node(problem.startstate, None)
```

Vengono inizializzati i contatori per le metriche di performance:

- `time_cost = 1`: Conta il numero di nodi generati (inizialmente solo la radice).
- `space_cost = 1`: Traccia il numero massimo di nodi in memoria simultaneamente.

9.2 Goal Test Preliminare

Prima di avviare il ciclo di ricerca, viene effettuato un controllo immediato: se lo stato iniziale è già lo stato obiettivo, la ricerca termina con successo a costo zero:

```
if node.state == problem.goalstate:
    return build_path(node), time_cost, space_cost
```

9.3 Gestione della Frontiera

La frontiera viene inizializzata come una **Coda FIFO** (*First-In, First-Out*), tipica della BFS:

```
frontier = NodeQueue()
frontier.add(node)
```

L'uso di una coda assicura che i nodi vengano espansi nell'ordine in cui sono stati generati (prima i nodi a profondità d, poi quelli a $d+1$).

9.4 Ciclo di Esplorazione (Loop)

Il ciclo `while` continua finché la frontiera non è vuota. Ad ogni iterazione:

1. **Estrazione**: Il nodo in testa alla coda viene rimosso (`frontier.remove()`) per essere espanso.
2. **Espansione**: Si itera su tutte le azioni possibili definite dallo spazio delle azioni (`problem.action_space.n`). Per ogni azione:
   - Viene generato il nuovo stato successore tramite `problem.sample(node.state, action)`.
   - Viene creato un nuovo oggetto `child` (figlio), collegandolo al padre (`node`).
   - Si incrementa il `time_cost` poiché è stato generato un nuovo nodo.
3. **Goal Test all'Espansione**: Una caratteristica fondamentale di questa implementazione efficiente della BFS è che il controllo dell'obiettivo avviene al momento della generazione del figlio, non alla sua estrazione:

   ```
   if problem.goalstate == child.state:
       return build_path(child), time_cost, space_cost
   ```

   Questo risparmia tempo e memoria evitando di inserire il nodo goal nella frontiera per poi doverlo estrarre successivamente.
4. **Aggiornamento Frontiera e Space Cost**: Se il nodo non è l'obiettivo, viene aggiunto alla coda. Dopo ogni ciclo di espansione, si aggiorna lo `space_cost` se la dimensione corrente della frontiera supera il massimo registrato in precedenza:

   ```
   space_cost = max(space_cost, len(frontier))
   ```

9.5 Conclusione

Se il ciclo termina e la frontiera si svuota senza aver trovato l'obiettivo, la funzione restituisce `None` (fallimento), insieme ai costi computazionali sostenuti.

BFS versione GraphSearch:

10 Analisi dell'Algoritmo: BFS Graph Search

Questa funzione implementa la variante **Graph Search** della BFS. A differenza della *Tree Search*, questa versione è progettata per gestire ambienti contenenti cicli o percorsi ridondanti, evitando di ri-esplorare stati già visitati.

10.1 Inizializzazione e Strutture Dati

Come nella versione precedente, viene creato il nodo radice e verificato se è immediatamente l'obiettivo. La differenza sostanziale risiede nelle strutture dati utilizzate:

1. **Frontiera**: Una coda FIFO (`NodeQueue`) per gestire l'ordine di esplorazione in ampiezza.
2. **Insieme Esplorati**: Viene inizializzato un insieme vuoto:

   ```
   explored = set()
   ```

   Questo set (spesso chiamato *Closed List*) memorizzerà gli stati che sono già stati espansi, permettendo controlli di appartenenza in tempo costante $O(1)$ (o quasi, a seconda dell'implementazione dell'hashing).

10.2 Ciclo di Esplorazione

Il ciclo `while` procede finché la frontiera non è vuota. Le operazioni chiave sono:

10.2.1 Estrazione e Marcatura

Quando un nodo viene estratto dalla frontiera (`frontier.remove()`), il suo stato viene immediatamente aggiunto all'insieme `explored`:

```
node = frontier.remove()
explored.add(node.state)
```

Questo "chiude" lo stato: da questo momento in poi, sappiamo di averlo già analizzato.

10.2.2 Espansione e Filtraggio (Graph Search)

Durante la generazione dei figli, l'algoritmo applica il filtro fondamentale che distingue la Graph Search dalla Tree Search. Per ogni nodo figlio generato (`child`):

```
if child.state not in explored and child.state not in frontier:
```

Questa condizione verifica due cose:

- **Non in Explored**: Assicura che non torniamo su uno stato già visitato (evita cicli infiniti).
- **Non in Frontier**: Assicura che non aggiungiamo alla coda un nodo che è già in attesa di essere esplorato (evita ridondanza).

10.2.3 Goal Test e Inserimento

Se il nodo figlio supera il filtro (è un nuovo stato), viene effettuato il test dell'obiettivo (Early Goal Test):

```
if child.state == problem.goalstate:
    return build_path(child), time_cost, space_cost
```

Se non è l'obiettivo, viene aggiunto alla frontiera per la futura espansione.

10.3 Analisi dei Costi

- **Time Cost**: Incrementato ogni volta che viene generato un nodo figlio (`child = Node(...)`).
- **Space Cost**: In questa versione, la complessità spaziale deve tenere conto di tutti i nodi mantenuti in memoria. La formula utilizzata somma sia i nodi in attesa (frontiera) sia quelli già visitati (explored):

  ```
  space_cost = max(space_cost, len(frontier) + len(explored))
  ```

  Questo riflette il fatto che, in una Graph Search, la memoria non viene mai liberata per gli stati visitati, portando a un'occupazione di memoria lineare rispetto al numero di stati del grafo ($O(|V|)$).

11 Ricerca a Profondità Limitata (DLS)

La *Depth-Limited Search* (DLS) è una variante della *Depth-First Search* (DFS) che impone un limite massimo alla profondità di esplorazione. Questo risolve il problema principale della DFS: il rischio di perdersi in percorsi infiniti o eccessivamente profondi.

11.1 Struttura della Funzione Ricorsiva

La funzione `Recursive_DLS_TreeSearch` implementa la logica core dell'algoritmo. Riceve in input il nodo corrente e un budget residuo (`limit`).

11.1.1 Casi Base

La ricorsione si arresta in due situazioni:

1. **Goal Test**: Se il nodo corrente è lo stato obiettivo:

   ```
   if problem.goalstate == node.state:
       return build_path(node), 1, node.depthcost
   ```

   Restituisce la soluzione trovata.

2. **Cutoff (Limite Raggiunto)**: Se il `limit` scende a 0 e non siamo sul goal:

   ```
   elif limit == 0:
       return "cut_off", 1, node.depthcost
   ```

   Restituisce il valore speciale `"cut_off"`, che segnala che la ricerca è stata interrotta arbitrariamente, non perché l'albero sia finito, ma perché è finito il budget.

11.1.2 Passo Ricorsivo

Se non siamo in un caso base, l'algoritmo espande i nodi figli:

1. Itera sulle azioni possibili.
2. Chiama se stessa ricorsivamente riducendo il limite: `limit - 1`.
3. Aggrega i risultati:
   - `time_cost`: Somma i nodi esplorati dai figli.
   - `space_cost`: Calcola la profondità massima raggiunta (max tra i figli).

11.1.3 Gestione del Risultato (Backtracking)

Durante la risalita dalla ricorsione, l'algoritmo deve distinguere tra un fallimento reale (vicolo cieco) e un taglio (cutoff):

- Se un figlio restituisce una soluzione, questa viene propagata immediatamente verso l'alto.
- Se un figlio restituisce `"cut_off"`, si imposta un flag `cut_off_occurred = True`.
- Alla fine del ciclo sui figli:

  ```
  if(cut_off_occurred): return "cut_off", ...
  else: return "failure", ...
  ```

  Se non abbiamo trovato soluzioni ma abbiamo incontrato almeno un cutoff, restituiamo `"cut_off"` (potrebbe esserci una soluzione più in profondità). Se non c'è stato nessun cutoff, significa che abbiamo esplorato tutto il sotto-albero senza trovare nulla: è un `"failure"`.

12 Iterative Deepening Search (IDS)

L'*Iterative Deepening Search* (IDS) combina i vantaggi della BFS (completezza e ottimalità) con quelli della DFS (bassa occupazione di memoria). L'idea è eseguire una serie di DLS con limiti di profondità crescenti: $limit = 0, 1, 2, \ldots, \infty$.

12.1 Implementazione

La funzione `IDS` funge da wrapper che gestisce il ciclo delle profondità.

12.1.1 Ciclo Principale

Il codice utilizza un generatore o un ciclo infinito per incrementare la profondità i:

```
for i in zero_to_infinity():
```

Ad ogni iterazione, viene lanciata una nuova ricerca DLS ripartendo da zero (nuovo nodo radice) con il nuovo limite i:

```
node = Node(problem.startstate, None)
solution_dls, time, space = DLS_Function(node, problem, i, set())
```

12.1.2 Aggregazione dei Costi

Un aspetto cruciale dell'IDS è come vengono calcolati i costi:

- `total_time_cost += time_cost`: Poiché IDS rigenera i nodi delle profondità precedenti ad ogni iterazione, il costo temporale è cumulativo. Se la soluzione è a profondità d, i nodi a profondità 1 sono stati generati d volte.
- `total_space_cost = max(...)`: Lo spazio richiesto è determinato solo dall'ultima iterazione (quella più profonda), poiché la memoria viene liberata tra un'iterazione e l'altra.

12.1.3 Terminazione

L'algoritmo termina quando la DLS restituisce un risultato diverso da `"cut_off"`:

```
if solution_dls != "cut_off":
    return solution_dls, total_time, total_space, i
```

Questo significa che:

- O ha trovato una **Soluzione** valida.
- O ha restituito **Failure** (il che significa che l'intero spazio degli stati è stato esplorato fino alla fine senza trovare soluzioni, quindi è inutile aumentare il limite).

13 Uniform-Cost Search (UCS)

La *Uniform-Cost Search* (UCS) è un algoritmo di ricerca non informata che esplora il grafo espandendo sempre il nodo con il costo di cammino cumulativo $g(n)$ più basso. A differenza della BFS, che espande in base alla profondità (numero di passi), la UCS garantisce di trovare la soluzione a costo minimo anche in grafi con costi degli archi variabili (o pesati).

13.1 Funzione Ausiliaria: `present_with_higher_cost`

Prima di analizzare il ciclo principale, osserviamo la funzione helper:

```
def present_with_higher_cost(queue, node):
    if node.state in queue:
        if queue[node.state].pathcost > node.pathcost:
            return True
    return False
```

Questa funzione è cruciale per l'ottimizzazione. Verifica se il nodo che stiamo per inserire è già presente nella frontiera (coda di priorità), ma con un costo $g(n)$ **peggiore** (maggiore) di quello attuale. Se restituisce `True`, significa che abbiamo trovato un percorso più breve per raggiungere uno stato già noto, e quindi dobbiamo aggiornarlo.

13.2 Struttura dell'Algoritmo

13.2.1 Inizializzazione

L'algoritmo utilizza una **Priority Queue** invece di una semplice FIFO queue.

```
queue = PriorityQueue()
queue.add(Node(environment.startstate))
```

La frontiera ordina i nodi in base al loro `pathcost`. Il nodo con il costo minore sarà sempre in testa.

13.2.2 Il Ciclo Principale e il Goal Test

Un punto fondamentale che distingue la UCS dalla BFS è la posizione del Goal Test.

```
node = queue.remove()
if node.state == environment.goalstate:
    return build_path(node), time_cost, space_cost
```

Il controllo se siamo arrivati all'obiettivo viene effettuato **dopo l'estrazione** dalla coda, non durante la generazione.

- *Perché?* Se facessimo il controllo alla generazione (come in BFS), potremmo accettare un percorso sub-ottimale solo perché lo abbiamo trovato per primo. Controllando all'estrazione, siamo sicuri che il nodo estratto abbia il costo $g(n)$ minore possibile tra tutti quelli nella frontiera.

13.2.3 Espansione e Aggiornamento

Per ogni azione possibile, viene generato un nodo figlio calcolando il nuovo costo cumulativo:

$$g(child) = g(parent) + cost(action)$$

Nel codice:

```
child = Node(..., node.pathcost + 1, node.pathcost + 1)
```

(Qui si assume costo unitario +1 per ogni passo).

Successivamente, si gestiscono tre casi per il nodo figlio:

1. **Nuovo nodo**: Se lo stato non è mai stato esplorato né è in coda, viene aggiunto.
2. **Miglioramento (Decrease Key)**: Se lo stato è già in coda, ma il nuovo percorso è più economico (verificato da `present_with_higher_cost`), il nodo vecchio viene sostituito con quello nuovo:

   ```
   elif present_with_higher_cost(queue, child):
       queue.replace(child)
   ```

3. **Scarto**: Se lo stato è già in `explored` o è in coda con un costo minore, il nuovo percorso viene ignorato.

13.3 Analisi dei Costi

- **Time Cost**: Numero di nodi estratti dalla coda di priorità.
- **Space Cost**: Dato dalla somma della dimensione della frontiera e dell'insieme dei visitati:

  ```
  space_cost = max(space_cost, len(queue) + len(explored))
  ```

  Questo riflette la complessità spaziale tipica di UCS che può essere elevata se i costi degli archi sono molto piccoli.