Explainable AI: Rocksample

Francesco Cecconello

30 giugno 2021

1 POMCP

Partially Observable Monte-Carlo Planning (POMCP) [1] è un algoritmo online utilizzato nel campo dell'intelligenza artificiale e della robotica, il cui scopo è generare in maniera efficiente delle policy approssimate per i Partially Observable Markov Decision Processes (POMDP).
Un POMDP è un framework, basato sui Markov Decision Process, in cui un agente è chiamato a generare una policy di azioni ottimale, massimizzando il reward a lungo termine. Per decidere quali azioni compiere, l'agente può osservare lo spazio circostante, inferendo informazioni la cui accuratezza può essere espressa sotto forma di probabilità. Gli MDP, infatti, si basano sulle catene di Markov, ovvero dei processi in cui gli stati non sono direttamente osservabili, ma si evolvono generando eventi (l'unica parte effettivamente osservabile) con una distribuzione di probabilità che dipende solo dallo stato corrente.
POMCP genera le policy online, evitando di incorrere nel *curse of dimensionality*, ovvero esplorando l'ambiente a partire dallo stato corrente senza dover visitare l'intero spazio degli stati, occupando quindi solo la memoria necessaria a rappresentare la policy, ovvero l'azione migliore da compiere dato il belief attuale.
Tuttavia, pur essendo una soluzione più scalabile rispetto alle principali presenti nella letteratura, POMCP restituisce risultati poco interpretabili dall'utente, poiché evita la rappresentazione diretta di una policy.

2 XPOMCP

Per sfruttare al meglio le potenzialità di POMCP e fornire contemporaneamente delle informazioni facilmente interpretabili dall'utente, si ricorre all'utilizzo di XPOMCP[2], il quale sfrutta l'espressività delle formule logiche per rappresentare proprietà specifiche della policy considerata. Tramite una formula logica detta *rule template*, si stabiliscono delle relazioni fra le variabili del belief: tale formula si concretizza in una regola con l'instanziazione delle variabili libere mediante l'analisi delle tracce e risolvendo un problema di *max-SMT*.
La formula restituisce una rappresentazione della policy suddivisa in due classi principali, ovvero quella degli step che hanno soddisfatto la regola nella traccia e quelli che non l'hanno fatto, segnalando le azioni che hanno violato la regola e rendendo più facile all'utente la comprensione del motivo per cui si sono verificate delle violazioni della formula.

3 Rocksample

Il problema Rocksample[1] simula il movimento di un robot su una griglia quadrata $n \times n$, all'interno della quale sono disposte delle rocce. Alcune di queste rocce restituiscono un reward all'agente se decide di effettuare un sampling, mentre le altre non restituiscono alcun valore; l'obiettivo è effettuare il sampling solamente delle rocce che hanno valore e, in seguito, uscire dal lato destro della mappa. Le azioni di base sono sei, ovvero *north,south,east,west,sample* e *check*, dove le prime quattro codificano il movimento deterministico del robot all'interno della griglia, l'azione di *sample* permette al robot di scavare quando si trova sopra a una roccia e *check* consente al robot di raccogliere dati su una roccia per valutarne la qualità. Durante queste ultime due azioni l'agente non si muove.
L'osservazione di una roccia è disturbata da rumore, il quale cresce esponenzialmente con la distanza tra il robot e la roccia stessa. L'agente riceve un reward positivo per l'uscita dalla griglia, un altro reward positivo se la roccia su cui ha effettuato il sampling è una roccia di valore e un reward nullo se, invece, la roccia era di scarsa qualità. Il reward riguardante l'uscita dalla mappa subisce un processo di decadimento nel tempo scontato di un fattore γ.

4 Rappresentazione delle informazioni

La posizione in cui si trova il robot all'interno della griglia viene definita da due variabili, ovvero la coordinata x e la coordinata y, rispettivamente mappate in *coord x* e *coord y*; sono note anche le posizioni *rock 1, rock 2, ...* delle rocce, anch'esse codificate da due coordinate, alle quali viene aggiunta un'ulteriore variabile *rock rel*, la quale ritornerà le coordinate della roccia su cui si troverà eventualmente il robot. Tale accorgimento permetterà all'utente di capire quale roccia abbia generato un'anomalia nell'esecuzione quando è stato effettuato il sampling su di essa.
Oltre alle posizioni del robot e delle rocce, viene tenuta traccia anche dei sampling effettuati tramite l'array binario *sampled*, in cui un 1 in posizione i indica che la roccia i è già stata scavata. Confrontando *sampled* con *rock rel*, sarà possibile determinare se *rock rel* sia già stata scavata o meno, ponendo in questo caso a 0 la probabilità che sia una roccia di qualità.
Per memorizzare queste informazioni durante l'esecuzione di ogni singola traccia, si inizializzano i tre array *self.coord_x_in_runs, self.coord_y_in_runs* e *self.sampled_in_runs*, i quali verranno riempiti mano a mano ad ogni run.

5 Parsing

Le informazioni sul sistema vengono ricavate da tracce di tipo XES, uno standard basato su XML per event logs. In particolare, nel caso di Rocksample vengono specificati la taglia della griglia *Size*, il numero di rocce *NumRocks* e le coordinate di ogni roccia all'interno della lista *rocks* così implementata

```
<list key="rocks">
    <list key="rock">
        <int key="number" value="0"/>
        <int key="coord x" value="0"/>
        <int key="coord y" value="3"/>
    </list>
```

dove ad ogni chiave *rock* sono associate altre tre chiavi: *number*, che identifica la roccia, *coord x* e *coord y*, che indicano la posizione di tale roccia. In ogni traccia le coordinate x e y potranno cambiare per ogni roccia, per cui per immagazzinare tale informazione si inizializzano i due array *self.coord_x_in_runs* e *self.coord_y_in_runs*. Per contribuire all'espressività di XPOMCP, è utile ricavare tali informazioni dalla traccia, ovvero effettuare un parsing per memorizzare le posizioni delle rocce all'interno della mappa. In questo caso, bisognerà estrarre la lista *rocks* dalla traccia e allocare tutte le posizioni in un array *rocks*.

```
for rock in node_from_key(log,'rocks'):
    rocks.append([int(node_from_key(rock,'coord x').attrib['value']),
        int(node_from_key(rock,'coord y').attrib['value'])])
```

In seguito, raccolte le informazioni *coord_x* e *coord_y* sulla posizione attuale del robot e memorizzate all'interno dei rispettivi array *self.coord_x_in_runs* e *self.coord_y_in_runs*, si eseguirà l'azione prevista dalla traccia. A ogni configurazione del belief sono associati un numero da $0_{10} = 00000000000_2$ a $2047_{10} = 11111111111_2$ e un numero intero di particles: il primo, convertito in binario, rappresenta i valori delle rocce, dove 1 indica una roccia probabilmente di valore e 0 una roccia senza valore, mentre il secondo quantifica l'attendibilità della configurazione, a seconda di quante particles ne confermino la possibile correttezza; intuitivamente, più particles vengono associate a una certa configurazione, più è probabile che questa sia affidabile.
Durante l'esecuzione della traccia, a ogni roccia all'interno del belief verrà associata la somma delle particles delle configurazioni in cui tale roccia veniva considerata di qualità, ovvero si otterrà una valutazione complessiva della bontà di ogni roccia data dalle indicazioni di ogni configurazione.

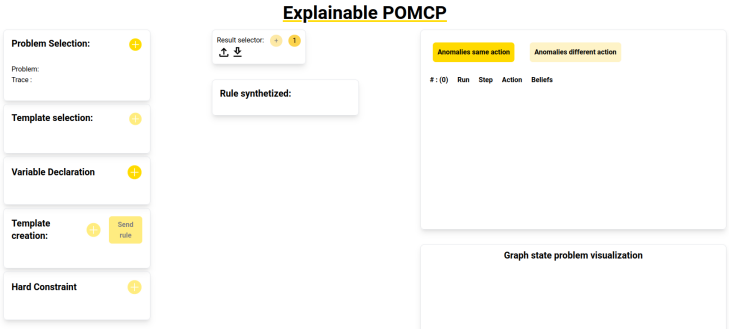
```
for i in node_from_key(event, 'belief'):
    state = i.attrib['key']
    particles = int(i.attrib['value'])

    rocks_set =str(int(bin(int(state))[2:]))[::-1]
    total += particles
    for rock in range(len(rocks_set)):
        if rocks_set[rock] == "1":
            self.belief_in_runs[-1][-1]['rock '+str(rock)] += particles
for state in self.states_prova:
    self.belief_in_runs[-1][-1][state] /= total
```

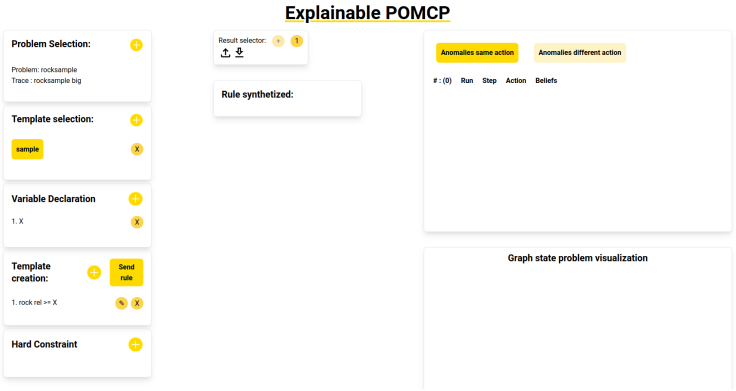
Infine si farà ricorso all'array *sampled*, il quale terrà traccia delle rocce già scavate in ogni run: nel caso in cui la roccia su cui si effettui l'azione di sampling non fosse ancora stata scavata, *rock rel* assumerà il valore probabilistico di tale roccia, altrimenti otterrà una probabilità pari a 0 di essere preziosa.
Durante ogni run, *rock rel* indicherà la roccia scavata durante una certa iterazione: nel caso in cui si verifichi un'anomalia, ovvero se venisse eseguito il sampling su una roccia con un valore al di sotto della treshold calcolata automaticamente da POMCP, *rock rel* permetterà velocemente di identificare quale roccia abbia generato l'anomalia.

6 Interfaccia grafica

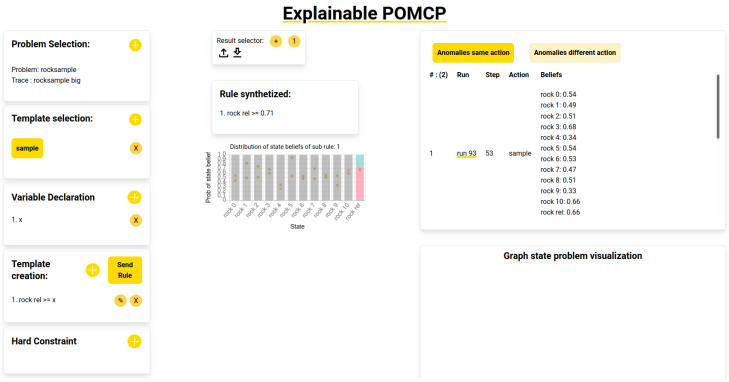
Per rappresentare l'esecuzione della traccia è stato scelto Gatsby, un framework front end che garantisce alte performance e scalabilità, basato sul linguaggio React. La mappatura delle tracce, degli stati e delle azioni viene estratta direttamente dal backend, per cui è risultato necessario solamente aggiungere il mapping di Rocksample a quello dei problemi già presenti.
Terminata la fase di preparazione del problema dal punto di vista del backend, sarà possibile avviarlo spostandosi nella directory in cui è presente il file *run.sh* e lanciare il comando *bash run.sh*, il quale aprirà una connessione in ascolto sulla porta 8001. In seguito, si potrà avviare Gatsby all'interno della directory del progetto con il comando *gatsby develop* , il quale aprirà una connessione all'indirizzo *http://localhost:8000/*: cliccando sul link si verrà reindirizzati all'home page sul browser predefinito.



Per controllare l'esecuzione di Rocksample, si selezioneranno il problema *rocksample* e la traccia *rocksample_small.xes* o la traccia *rocksample_big.xes*, a seconda delle preferenze; in seguito, si potrà scegliere quale azione monitorare per verificarne le anomalie. Nel nostro caso si è scelto di utilizzare *rocksample_big.xes*.
Per il problema Rocksample, l'azione più sensibile è quella di sampling, poiché è quella che può fornire reward positivi o negativi: scegliendo quindi l'azione *sample* e creando una variabile X, si potrà controllare quante volte la roccia correntemente scavata, ovvero *rock rel*, genererà delle anomalie. In questo caso X rappresenterà la threshold, ovvero la soglia di confidenza minima affinché il robot decida di effettuare il sampling di una roccia; un'anomalia corrisponderà a un *sample* effettuato con una confidenza minore della threshold.



Cliccando su *Send rule* verrà generato un grafico, nel quale i pallini gialli indicheranno le anomalie rilevate dal sistema, le quali compariranno nel riquadro in alto a destra.



7 Conclusioni

Il risultato dell'esecuzione del codice indica la presenza di due anomalie, dove la prima riguarda la roccia 10, mentre la seconda la roccia 7: nonostante la threshold calcolata dall'analisi delle tracce sia di 0.71, infatti, è stato effettuato il sampling su entrambe le rocce con una confidenza rispettivamente di 0.66 e 0.69. Per capire quale delle due anomalie sia più grave, solitamente si calcola la severity tramite la distanza di Hellinger ma, considerando ogni roccia come una distribuzione di probabilità a sé, il belief totale non sommerà mai a uno, per cui non sarà possibile calcolare la severity tramite il metodo previsto dal template utilizzato precedentemente per altri POMDPs su cui si basa il codice per Rocksample.
Quello che si può fare, invece, è calcolarla manualmente, generando mille punti casuali che rispettino la regola (ovvero che abbiano un valore compreso tra la threshold e 1), calcolare la distanza di Hellinger fra ognuno di essi e l'anomalia e memorizzare la distanza minima, ripetendo tale procedimento per ogni anomalia rilevata da XPOMCP.

```python
import random
import math

def hellinger_distance(array1,array2):
    argument = 0
    for i in range(len(array1)):
        argument += (math.sqrt(array1[i]) - math.sqrt(array2[i]))**2
    sqr = math.sqrt(argument)
    final = sqr/math.sqrt(2)
    return argument


anomalies = [0.69,0.66]
threshold = 0.71
hellinger_distances = [1]*len(anomalies)
for i in range(len(anomalies)):
    for j in range(1000):
        generated_point = random.uniform(threshold,1)
        artificial_distr = [generated_point,1-generated_point]
        anomaly_distr = [anomalies[i],1-anomalies[i]]
        hd = hellinger_distance(artificial_distr,anomaly_distr)
        if hd<hellinger_distances[i]:
            hellinger_distances[i] = hd
print(hellinger_distances)
```

In questo modo, le severity associate alla roccia 10 e alla roccia 7, calcolate manualmente, saranno rispettivamente di 0.0005 e 0.003, cioé relativamente piccole, ad indicare che la gravità delle anomalie è piuttosto bassa, poiché si effettua il sampling con confidenze non troppo distanti dalla threshold.

Bibliografia

[1] David Silver and Joel Veness. “Monte-Carlo planning in large POMDPs”. In: Neural Information Processing Systems. 2010. URL: http://hdl.handle.net/1721.1/100395.

[2] Giulio Mazzi, Alberto Castellini, and Alessandro Farinelli. “Identification of Unexpected Decisions in Partially Observable Monte-Carlo Planning: A Rule-Based Approach”. In: *AAMAS ’21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*. Ed. by Frank Dignum et al. ACM, 2021, pp. 889–897. URL: https://dl.acm.org/doi/10.5555/3463952.3464058.