Tesi di Laurea Magistrale

Reinforcement Learning applicato alla robotica collaborativa.

Relatore/i

prof. Antonelli Dario

Candidato

Filippo Crosa

Anno Accademico 2021 - 2022

Indice

Introduzione

Una panoramica sul mondo HRC.

Negli ultimi anni negli ambienti industriali moderni è sempre più pervasiva la presenza dei robot che stanno progressivamente prendendo il posto degli operatori umani. La loro capacità di spostare carichi pesanti, manipolare oggetti e sostanze potenzialmente pericolosi ed effettuare alcuni tipi di operazioni che richiedono grande accuratezza li rendono preferibili in molte applicazioni; inoltre sollevano l'uomo dall'onere di dover effettuare procedure ripetitive e tediose.

La produzione industriale non può comunque prescindere dalla presenza del fattore umano: i robot non sono in grado di pensare e possono solo eseguire movimenti e pattern preimpostati. Inoltre, i manipolatori robotici hanno tipicamente sei o sette gradi di libertà, mentre il braccio umano ne ha circa trenta[1]. La robotica collaborativa (Human Robot Collaboration, HRC) consente di beneficiare dei vantaggi dei robot in tutti quei compiti che richiedono comunque la presenza del personale umano.

In *Skill-based Dynamic Task Allocation in Human-Robot-Cooperation with the Example of Welding Application*[2] si propone una classificazione sulle diverse metodologie su come uomo e robot collaborativo (cobot) possono lavorare insieme.

- Coesistenza, dove uomo e cobot agiscono nello stesso ambiente ma non interagiscono tra di loro (separazione spaziale e temporale).
- Lavoro sincronizzato, quando i due operatori condividono lo stesso spazio di lavoro ma in tempi diversi (separazione temporale).
- Cooperazione, avviene se umano e cobot lavorano nello stesso spazio e nello stesso momento ma su oggetti diversi.
- Collaborazione, quando uomo e robot eseguono un lavoro insieme e le azioni del primo hanno conseguenze dirette ed immediate sull'azione del secondo.

[1] ALES VYSOCKY, PETR NOVAK, *Human – robot collaboration in industry*, «MM Science Journal», giugno 2016, p. 903
[2] RAINER MÜLLER, MATTHIAS VETTE, AARON GEENEN, *Skill-based Dynamic Task Allocation in Human-Robot-Cooperation with the Example of Welding Application*, «Procedia Manufacturing», Vol. 11, 2017, pp. 13-21.

Ciascun tipo di collaborazione deve prevedere delle specifiche di sicurezza adeguate al tipo di interazione specifica che avviene tra umano e cobot. Tali specifiche sono raccolte e spiegate nella norma di sicurezza ISO/TS 15066[3] che definisce quattro classi di requisiti di sicurezza per i cobot:

- Safety-rated Monitored Stop (SMS): dove il robot sospende il proprio movimento quando l'operatore umano entra nel suo spazio di lavoro.
- Hand-guiding (HG): l'operatore umano utilizza un dispositivo localizzato sul robot per spostarlo e guidarlo nello spazio di lavoro.
- Speed and separation monitoring (SSM): lo spazio di lavoro è diviso in aree costantemente monitorate con sensori e dispositivi di visione. Il robot può eseguire i suoi compiti al massimo della sua velocità fintanto che l'uomo non violi lo spazio di lavoro del robot.
- Power and force limiting (PFL): in questo caso si prevede che umano e cobot lavorino a stretto contatto. I parametri di movimento vengono monitorati in real-time come anche i sensori di coppia nei vari giunti. Il robot è capace di rilevare le collisioni e reagire immediatamente.

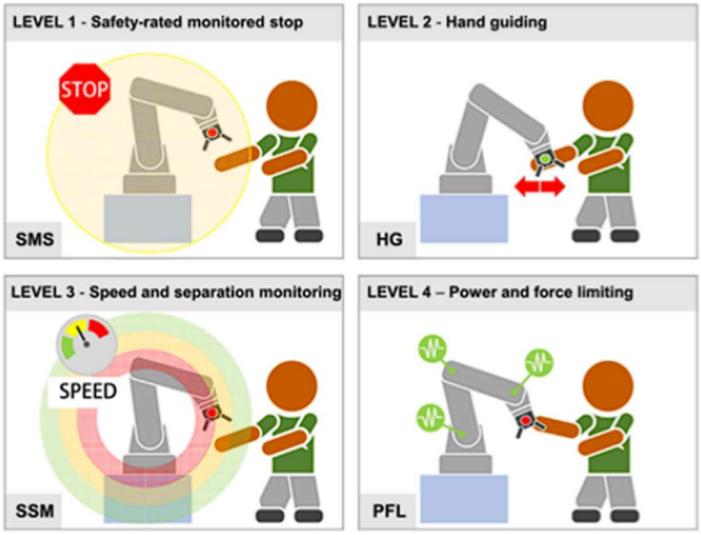


***Figura 1**: classi di sicurezza definiti nella norma ISO/TS 15066 per la robotica collaborativa.*

[3]ISO/TS 15066:2016 Robots and robotic devices – Collaborative Robots.

Nell'articolo *Human–Robot Collaboration in Manufacturing Applications: A Review*[4] vengono analizzati i lavori di ricerca inerenti alla robotica collaborativa nel decennio 2009-2018 (fig.2) e si osserva che il trend di interesse in tale ambito è in forte crescita, in particolare quello relativo ai processi di assemblaggio.

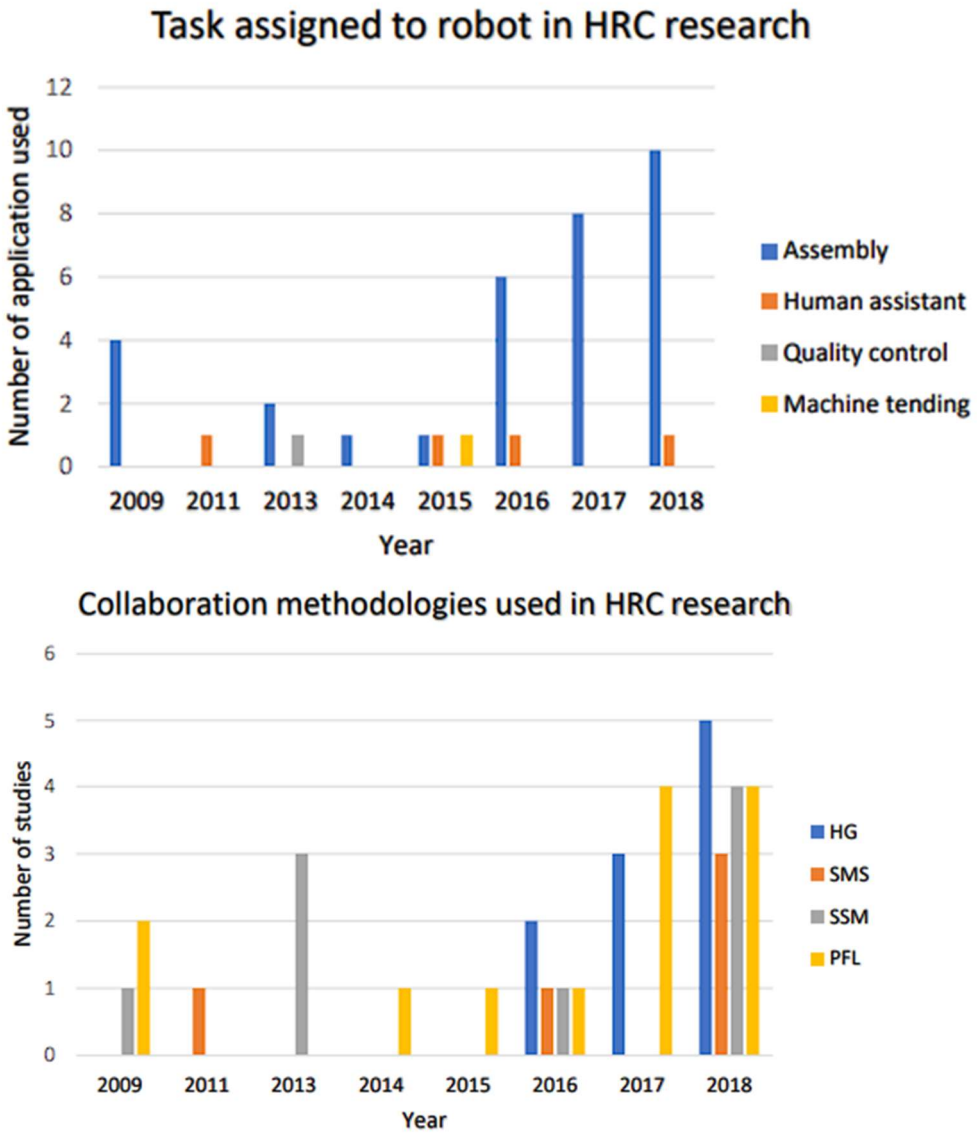


Figura 2

[4] ELOISE MATHESON, RICCARDO MINTO, EMANUELE G. G. ZAMPIERI, MAURIZIO FACCIO, GIULIO ROSATI, *Human–Robot Collaboration in Manufacturing Applications: A Review*, «MDPI Journal Robotics», 2019, p. 8.

Questo elaborato si inserisce direttamente nella tendenza dominante, prendendo in considerazione la metodologia PFL nel contesto del lavoro di assemblaggio; tale metodologia trova una naturale applicazione in aziende di medio-piccole dimensioni che si occupano della fabbricazione di prodotti che possono essere realizzati in diverse varianti. Il volume di produzione è tipicamente basso e diviso in piccoli lotti.

Obiettivi.

Nell'ottica di una collaborazione stretta, due operatori devono agire in sinergia, con unicità di intenti, proattività e flessibilità. In particolare, è proprio la flessibilità che determina il successo di un lavoro collaborativo ed è l'aspetto più indigesto per il robot, che, in quanto macchina programmata, è adatta ad eseguire pattern precisi e statici.

Rendere i robot intelligenti e flessibili significa renderli capaci di osservare il presente e di scegliere l'azione migliore in previsione del futuro prossimo. Ed è proprio sul concetto di migliore che deve potersi esprimere l'intelligenza del robot e sarà il focus principale di questa ricerca.

Il processo di assemblaggio può essere modellato tramite MDP[5]. Sulla MDP è possibile eseguire algoritmi di RL tramite i quali il robot deve imparare a scegliere le proprie azioni. Nei primi due casi di studio verrà utilizzato un algoritmo standard (Q-learning) per risolvere dei casi semplici ed "insegnare" al robot la sequenza di assemblaggio più veloce e a correggere alcune scelte non previste da parte dell'umano. Nel terzo e ultimo caso di studio si cercherà di generalizzare uno schema di assemblaggio generato casualmente e verrà applicato un algoritmo originale di adversarial reinforcement learning la cui idea di base è di mettere in competizione l'agente umano e l'agente robot: durante i vari episodi di apprendimento l'umano imparerà quali sono le azioni peggiori e cercherà di attuarle, mentre il robot tenterà, all'opposto, di impedire che l'umano faccia tali scelte. Al termine del training il robot otterrà la capacità di scegliere le proprie azioni in modo tale da ridurre il più possibile l'allungamento della sequenza d'assemblaggio che l'umano potrebbe causare.

[5] DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learning*, «Science Direct», Vol. 8, 2022.

Sviluppo del codice e scelte implementative.

Siccome l'applicazione di adversarial learning che si vuole implementare non è standard, non sono state trovate librerie sufficientemente elastiche per poterla realizzare in maniera agevole; di conseguenza, si è scelto di scrivere gli algoritmi da zero dopo un attento studio ed analisi della teoria su cui si basano le principali tecniche di RL.

La scelta del linguaggio di programmazione è ricaduta inevitabilmente su Python. La decisione è stata dettata non tanto dalla volontà di sfruttare a fondo la libreria PyTorch (nei nostri esempi non useremo l'accelerazione hardware della GPU), quanto dai vantaggi peculiari che offre. Infatti, Grazie alla sua espressività e flessibilità[6] è stato possibile accelerare enormemente la creazione degli algoritmi; inoltre, la sua popolarità ha reso possibile reperire alcune librerie esterne molto utili per la produzione di strutture dati quali alberi e grafi e la loro rappresentazione grafica.

Con lo scopo di poter garantire la riproducibilità dei risultati e compatibilità, si riporta la versione delle librerie/tools utilizzati per scrivere tutti gli script in Python:

- Versione Python: 3.9.7
- Versione Anaconda: 4.12.0
- Versione PyTorch: 1.11.0

[6] Marco Buttu, Python, Guida completa, Milano, LSWR edizioni, 2020, pp. 15-17

Impostazione dell'ambiente e passaggi preliminari

Questo capitolo ha lo scopo di fornire le indicazioni necessarie per poter installare l'IDE e le varie librerie esterne che serviranno per eseguire i vari script Python.

É stato preso come riferimento un manuale[7] di RL, la cui sezione iniziale è proprio dedicata ad impostare tutto il necessario per poter sviluppare le prime implementazioni con PyTorch[8].

Prima di tutto è necessario installare Anaconda. Al link https://docs.anaconda.com/anaconda/install/ è possibile seguire le istruzioni specifiche per il proprio sistema operativo.

Il passo successivo è installare PyTorch. Al link https://pytorch.org/get-started/locally è possibile selezionare le caratteristiche del proprio sistema e il tool metterà a disposizione un comando che dovremo copiare:

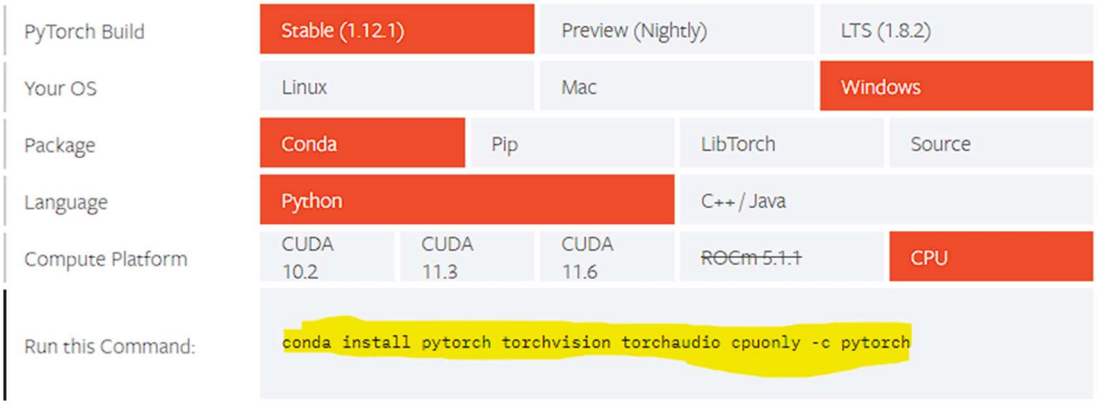


***Figura 3**: Esempio di caratteristiche della macchina di destinazione della libreria PyTorch.*

Per la macchina su cui sono stati implementati i programmi sono state selezionate le opzioni presenti in figura.

Come specificato nell'introduzione, non è stato necessario utilizzare la GPU per effettuare i vari training degli agenti; pertanto, si consiglia di selezionare 'CPU' nella sezione 'Computer Platform', in quanto la selezione di CUDA non comporterebbe alcun vantaggio in termini di tempo di esecuzione.

[7] YUXI LIU, *PyThorch 1.x, Reinforcement Learning, Cookbook*, Birmingha, Packt, 2019, pp. 10-12.
[8] https://pytorch.org/ consultato in data 16 Ottobre 2022.

Eseguire come amministratore ‘Anaconda Powershell Prompt’ (è un programma integrato in Anaconda) ed eseguire il comando precedentemente copiato dalla pagina web di Pytorch:

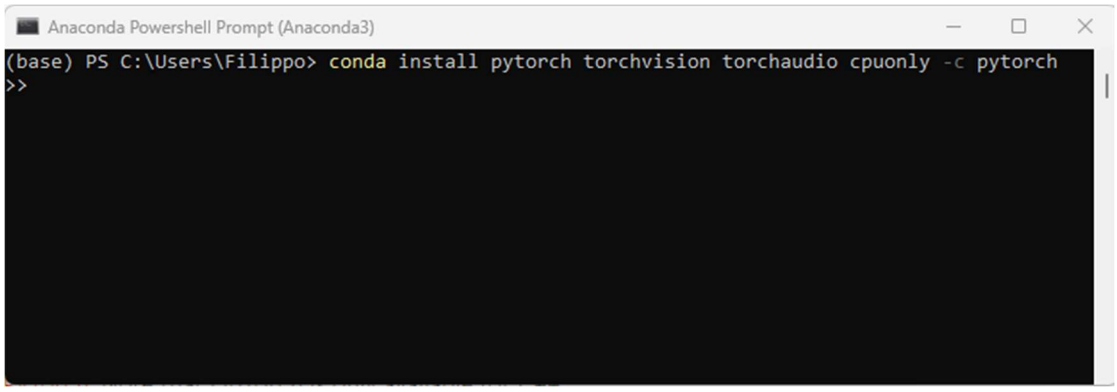


***Figura 4**: Esecuzione del comando prelevato dalla pagina web di PyTorch.*

Per poter correttamente eseguire tutti gli script utilizzati per la stesura della tesi è necessario installare altre librerie esterne: sempre nel prompt di anaconda, inserire i comandi:

- pip install anytree
- pip install networkx

La prima libreria consente di costruire gli alberi utilizzati nel capitolo 2; la seconda viene intensivamente impiegata nel capitolo 3.

L’installazione dell’IDE e delle librerie è terminato, ora è possibile aprire Spyder (che, analogamente alla Powershell, viene installata automaticamente insieme ad Anaconda) da cui si possono aprire e mandare in esecuzione i vari script.

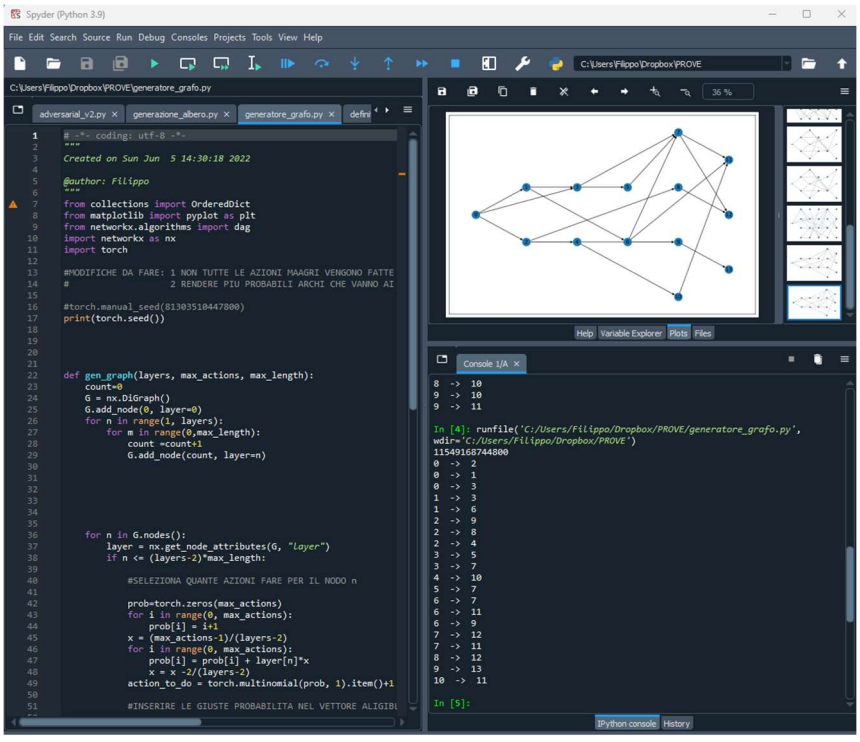


***Figura 5**: Esempio di esecuzione script in ambiente Spyder.*

Primo caso di studio

1.1. Introduzione.

Il primo caso di studio prende in esame un processo di assemblaggio collaborativo ideato in laboratorio e presentato nel lavoro *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learning*[9]. Verranno proposte due soluzioni che condurranno allo stesso risultato:

- La prima basata sull'algoritmo di Q-learning[10] che sostanzialmente cerca di replicare quanto già ottenuto nel testo di riferimento con, in aggiunta, l'ottimistico obiettivo di risolvere la criticità presentata nella sua conclusione[11];
- Una basata su programmazione dinamica[12] (DP) che sfrutta le caratteristiche del particolare assemblaggio presentato.

Nel caso specifico, l'obiettivo è montare un oggetto costituito da due flange identiche (F1 e F2) e una flangia quadrata (S) su una base (B).

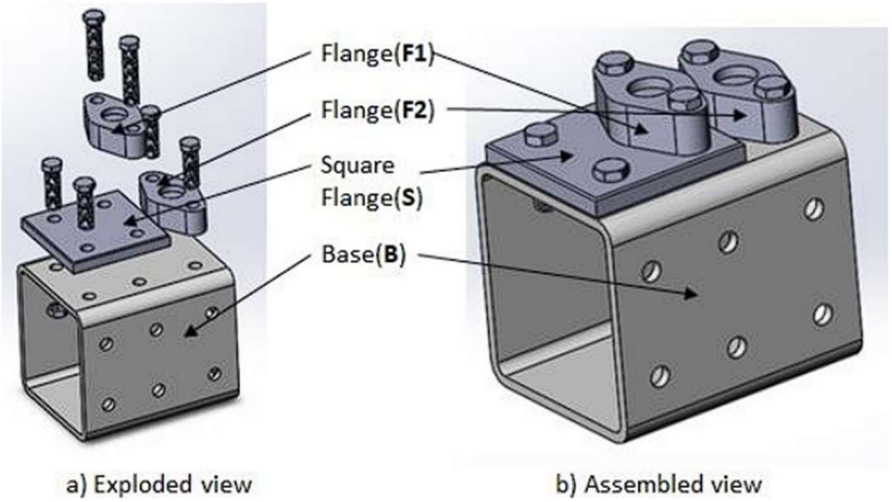


Figura 6: *Schema di assemblaggio.*

[9] DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learning*, «Science Direct», Vol. 8, 2022.
[10] DAVID SILVER, *Lecture 5: Model-Free Control*, in «Introduction To Reinforcement Learning with David Silver», 2015, https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver, consultato in Aprile 2022.
[11] DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation*, «Science Direct», Vol. 8, 2022, p. 6.
[12] DAVID SILVER, *Lecture 3: Planning By Dynamic Programming*, in «Introduction To Reinforcement Learning with David Silver», 2015.

Sebbene possa sembrare un compito banale, il processo di assemblaggio in realtà nasconde al suo interno diversi fattori che nell'ottica dell'HRC possono rappresentare delle vere e proprie criticità: poiché F1 e F2 sono identiche e B è simmetrica, sono possibili più configurazioni e ciascuna di esse può essere realizzata con diversi pattern. Quelle che per noi umani possono sembrare azioni o configurazioni equivalenti, nell'ottica del robot rappresentano scenari completamente inediti, tali da poter mettere in crisi la sua capacità di collaborare nel procedimento di assemblaggio. Questo caso di studio rappresenta, nella sua semplicità, un ottimo spunto per ragionare sulle possibili soluzioni e sulle loro conseguenti estensioni.

Il primo passo per poter strutturare una soluzione basata su RL e DP è di modellizzare il problema sottoforma di MDP[13] (Markov Decision Process).

Ciascuno stato della MDP deve inglobare la 'storia' dell'assemblaggio che ha condotto ad esso. Nel nostro caso, infatti, S11 e S17 (fig.7) sono stati che rappresentano lo stesso sub-assemblato, ma esso è stato ottenuto tramite due sequenze diverse e quindi vengono rappresentati da due stati diversi:

- S11 è ottenuto tramite la sequenza base → flangia ovale in posizione 3 → flangia quadrata in posizione 1-2 → flangia ovale in posizione 1;
- S17 è ottenuto tramite base → flangia quadrata in posizione 1-2 → flangia ovale in posizione 1 → flangia ovale in posizione 3.

Viene riportata la tabella rappresentante la lista degli stati e delle azioni abilitate in ciascuno stato (tab.1).

[13] *Ivi*., *Lecture 2: Markov Decision Processes*.

Tabella 1: *Tabella degli stati-task-azioni possibili.*

State	Task	Slot n.	Action
1	B	-	Ass F, ass S
2	B U F1	1	Ass F, ass S, disass
3	B U F2	2	Ass F, ass S, disass
4	B U F3	3	Ass F, ass S, disass
5	B U S12	1,2	Ass F, ass S, disass
6	B U S23	2,3	Ass F, ass S, disass
7	B+F1 U S23	2,3	Ass F, ass S, disass
8	B+F3 U S12	1,2	Ass F, ass S, disass
9	B+F1+(S23 U F02)	2	Wrong state
10	B+F1+(S23 U F03)	3	Terminal
11	B+F3 + (S12 U F01)	1	Terminal
12	B+F3 + (S12 U F02)	2	Wrong state
13	B+ (S12 U F01)	1	Ass F, ass S, disass
14	B+ (S12 U F02)	2	Ass F, ass S, disass
15	B+ (S23 U F02)	2	Ass F, ass S, disass
16	B+ (S23 U F03)	3	Ass F, ass S, disass
17	B+ (S12 + F01) U F3	3	Terminal
18	B+ (S12 + F02) U F3	3	Wrong state
19	B+ (S23 + F02) U F1	1	Wrong state
20	B+ (S23 + F03) U F1	1	Terminal

Da cui si può ricavare direttamente la rappresentazione grafica sottoforma di grafo (fig.7).

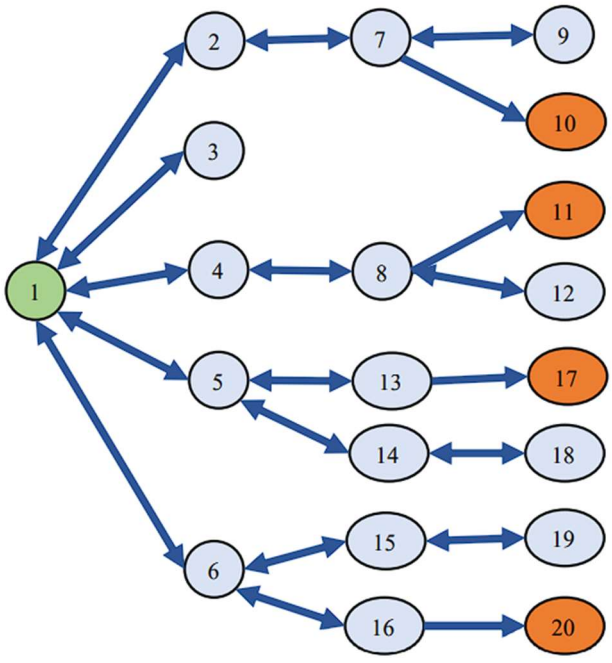


Figura 7: *Grafo dell'MDP, con stato iniziale in verde e stati finali in arancione.*

Per penalizzare la lunghezza dei percorsi è stato deciso di applicare un reward negativo (-1) ad agni azione che non conduce a uno degli stati finali (nel qual caso il reward è stato impostato a +2). Le sequenze ottimali risultano essere {S1, S2, S7, S10}, {S1, S5, S13, S17}, {S1, S4, S8, S11}, {S1, S6, S16, S20}. La soluzione proposta nella ricerca è in grado di trovare come cammino ottimale {S1, S5, S13, S17}, ma è anche in grado di adattare il proprio percorso anche in casi in cui alcuni stati vengono forzati dall'umano, tranne nella situazione in cui viene forzato lo stato S12 (tab.2).

Tabella 2: *Sequenze degli stati con i vari interventi dell'umano in grassetto.*

MDP Observations						
Decision Steps	1	2	3	4	5	6
Base Sequence	S1	S5	S13	S17		
Human -> S6	S1	**S6**	S16	S20		
Human -> S14	S1	S5	**S14**	S5	S13	S17
Human -> S2	S1	**S2**	S7	S10		
Human -> S2,S9	S1	**S2**	S7	**S9**	S7	S10
Human -> S3	S1	**S3**	S1	S6	S16	S20
Human –> S4	S1	**S4**	S8	S11		
Human -> S4,S12	S1	**S4**	S8	**S12**	S12	S12

Quando l'umano impone la transizione S8 → S12, il robot non è in grado di correggere la sequenza e raggiungere uno degli stati finali, limitandosi a riproporre continuativamente lo stato S12.

Nella prossima sezione tenteremo di replicare lo stesso algoritmo e verificheremo se siamo riusciti ad ottenere gli stessi risultati.

1.2. Tentativo con Q-learning.

Prima di analizzare nel dettaglio i risultati ottenuti, viene proposto di seguito una breve inciso sul Q-learning utile per poter avere una visione chiara dei dati prodotti dall'algoritmo in questo caso di studio e, soprattutto, nei due successivi.

Lo scopo ora è quello di trovare per ogni stato S (rappresentante un certo step dell'assemblaggio) qual è la successiva azione che consente di avvicinarsi il più possibile allo stato finale (il prodotto finito). Questo obiettivo sarebbe immediatamente conseguibile se avessimo una mappa (o matrice) capace di associare ad ogni stato un valore tanto maggiore quanto tale stato è vicino a quello finale. Si prenda ad esempio la matrice 4x4 che rappresenta 16 stati numerati da sinistra a destra e dall'alto al basso a partire da 0. La prima cella in alto a sinistra S=0 è lo stato finale. Qualsiasi sia il nostro stato iniziale, bisogna arrivare allo stato finale sapendo che ci si può muovere

solamente in direzione nord, est, sud e ovest (N, E, S, O). Se riuscissimo ad ottenere la mappa in fig.8A, il compito risulterebbe banale: basterebbe infatti muoversi nella casella col valore più alto. Se partissimo in posizione 11, potremmo facilmente tracciare il percorso migliore per arrivare a S=0 procedendo step-by-step scegliendo la mossa che ci porta nella casella adiacente con valore più alto. In caso di pari merito, la scelta è equivalente (fig.8B).

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

Figura 8A

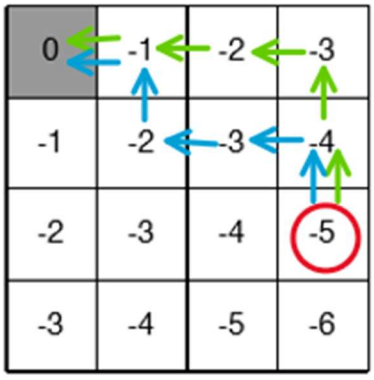


Figura 8B

Tale matrice viene chiamata V-table (tabella dei valori). L'algoritmo Q-learning non tenta di imparare la V-table; si basa invece sulla Q-table. Questa è una tabella (o struttura dati equivalente, come una mappa o un dizionario) in cui si assegna un valore a tutte le coppie stato-azione (S,A).

Ad esempio, lo stato 11 avrà i valori Q(7,N)=-5, Q(7,S)=-7, Q(7,O)=-5; la scelta da fare è N oppure O. Di conseguenza, se si conosce la Q-table dell'MDP è possibile nuovamente trovare il percorso ottimale (fig.9).

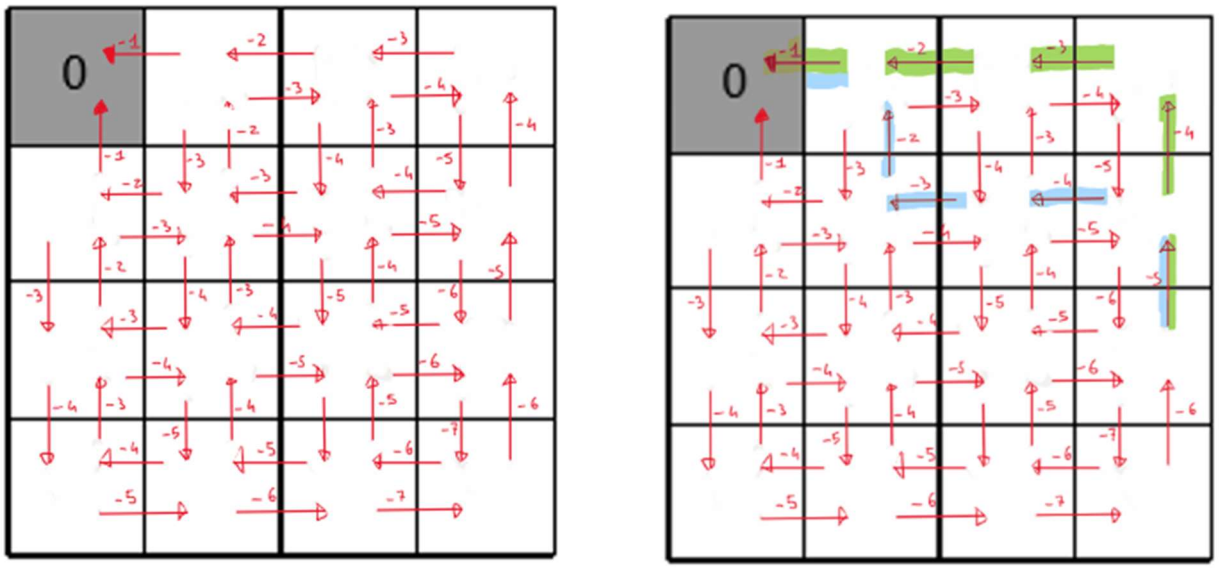


Figura 9

L’apprendimento tramite Q-value ha proprio l’obiettivo di creare un’approssimazione quanto più veritiera possibile della Q-table e da essa ricavare l’insieme di azioni (policy) che consentono di ottenere il percorso ottimale. Nel caso dell’esempio, tale approssimazione viene ottenuta ponendo un agente nella griglia e facendogli fare numerosi percorsi tramite la scelta delle proprie azioni, ciascuna delle quali riceverà come feedback dell’ambiente un reward positivo o negativo. Grazie a tale feedback l’agente costruirà la propria esperienza (la Q-table) che andrà ad affinarsi con lo scorrere degli episodi di apprendimento.

Tornando al nostro caso di studio, è necessario codificare tutte le azioni possibili prima di poter implementare l’algoritmo:

- Azione 0: assembla flangia in slot 1;
- Azione 1: assembla flangia in slot 2;
- Azione 2: assembla flangia in slot 3;
- Azione 3: assembla flangia quadrata negli slot 1 e 2;
- Azione 4: assembla flangia quadrata negli slot 2 e 3;
- Azione 5: smonta flangia da slot 1;
- Azione 6: smonta flangia da slot 2;
- Azione 7: smonta flangia da slot 3;
- Azione 8: smonta flangia quadrata da slot 1 e 2;
- Azione 9: smonta flangia quadrata da slot 2 e 3;

Integrando tale convenzione e lo schema mostrato in tab.1, possiamo numerare gli archi del grafo:

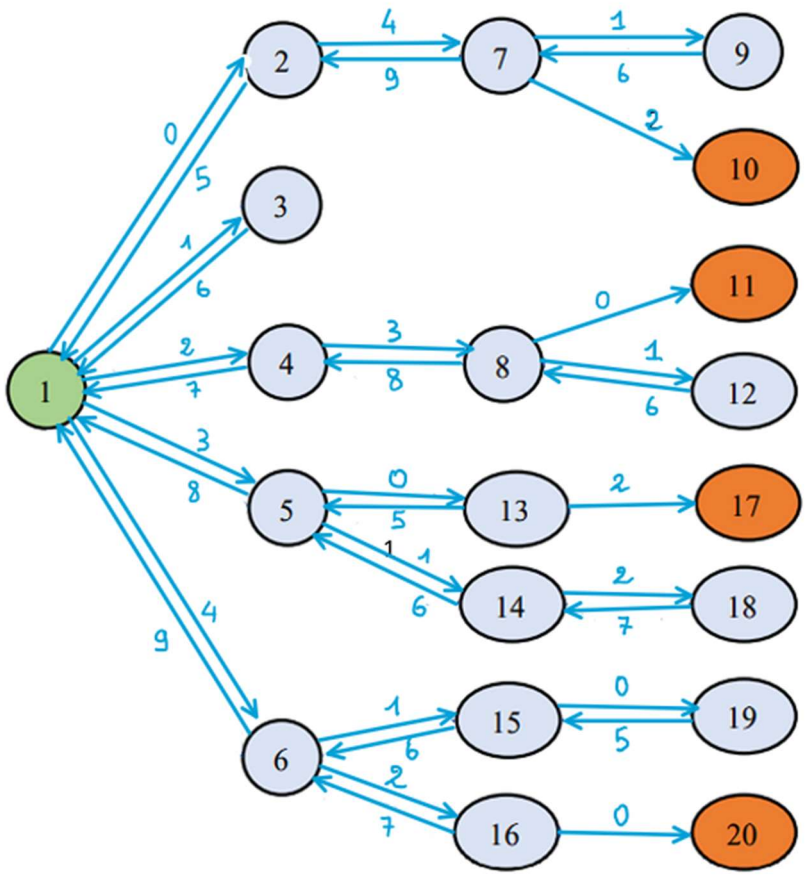


Figura 10: *Grafo dell'MDP con archi in blu rappresentanti le possibili azioni eseguibili in ciascuno stato.*

Per formalizzare la dinamica di transizione tra uno stato ed un altro è stata definita una struttura dati piuttosto complessa: si tratta di un dizionario di dizionari che ad ogni stato associa ad ogni possibile azione una lista di tuple. Ciascuna tupla definisce la probabilità di terminare in uno degli stati raggiungibili. Ad esempio P[0][0] contiene le tuple (1/3, 1, -1, False), (1/3, 2, -1, False), (1/3, 1, -1, False) che definiscono la probabilità che dato lo stato 1 e l'azione 1 si termini rispettivamente negli stati 1, 2 e 3; -1 è il reward e False indica che è stato raggiunto uno stato non terminale.

Il programma (attempt_Q_learning.py) viene mandato in esecuzione con ε inizialmente impostato a 0.9 per incentivare l'esplorazione nella fase iniziale; ad ogni episodio viene decrementato di 0.001 fino a diventare costante sul 10%. L'addestramento viene portato a termine in 1000 episodi. Di seguito si riporta la policy appresa (si tenga presente che per semplicità di realizzazione nel programma gli stati vengono numerati a partire da 0):

Tabella 3: *Policy appresa.*

STATO	AZIONE
1	3
2	4
3	6
4	3
5	0
6	2
7	2
8	0
9	6
12	6
13	2
14	6
15	6
16	0
18	7
19	5

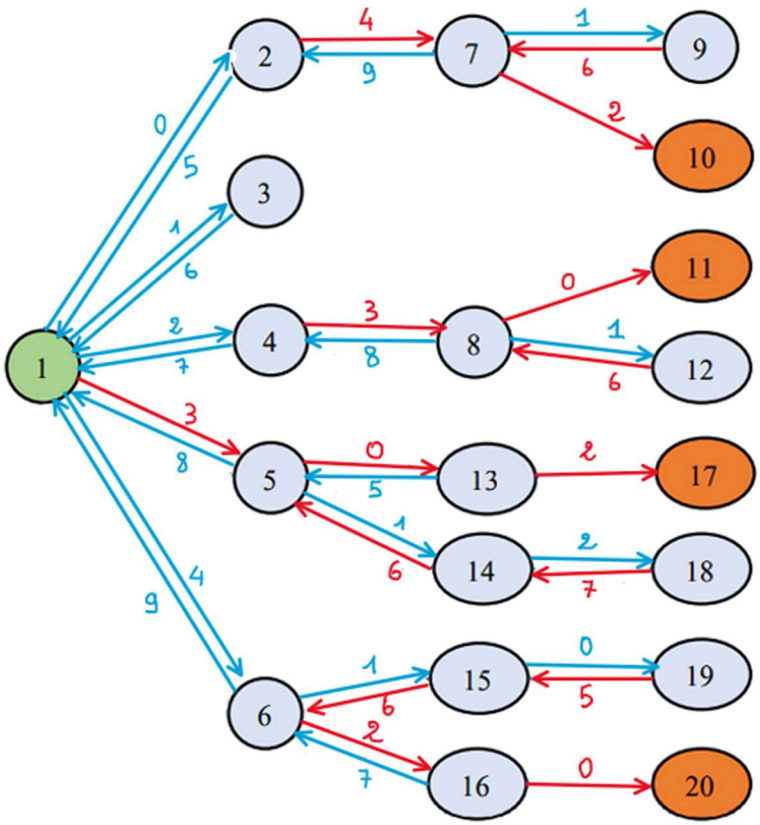


Figura 11: *Grafo dell'MDP con azioni in rosso rappresentanti la policy appresa.*

Dalla policy appresa possiamo notare che il robot opta per la sequenza {S1, S5, S13, S17}.

Allo stesso modo di come è stato fatto nella ricerca[14] possiamo ipotizzare che l'intervento umano tenti di deviare la sequenza verso stati non appartenenti alla sequenza ottimale del robot, ma questa volta in tutti i casi il robot riesce ad intraprendere le scelte necessarie per portare a

Tabella 4: *Sequenze degli stati con intervento dell'uomo in grassetto.*

MDP Observations						
Decision Steps	1	2	3	4	5	6
Base Sequence	S1	S5	S13	S17		
Human -> S6	S1	**S6**	S16	S20		
Human -> S14	S1	S5	**S14**	S5	S13	S17
Human -> S2	S1	**S2**	S7	S10		
Human -> S2,S9	S1	**S2**	S7	**S9**	S7	S10
Human -> S3	S1	**S3**	S1	S6	S16	S20
Human –> S4	S1	**S4**	S8	S11		
Human -> S4,S12	S1	**S4**	S8	**S12**	S8	S11

[14] DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation*, «Science Direct», Vol. 8, 2022, p. 6.

termine l'assemblaggio. Anche l'imposizione dello stato S8 che nello studio precedente conduceva al blocco della sequenza sullo stato S12 ora viene risolto correttamente:

Il migliore risultato comunque può essere semplicemente conseguenza di piccole variazioni nell'implementazione dell'algoritmo o dei parametri del training.

Possiamo notare che i reward guadagnati dall'agente convergono nei primi 500-600 episodi.

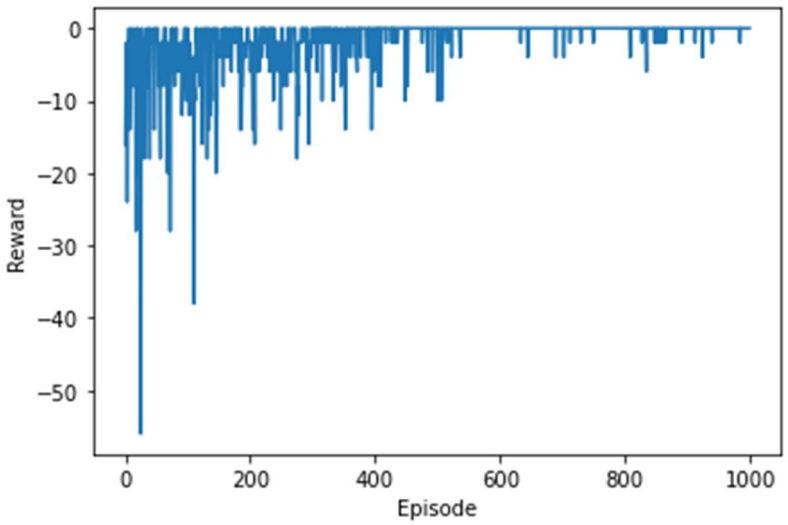


Figura 12: *Grafico episodi-reward*

1.3. Tentativo DP.

Alternativamente alla soluzione mediante Q-learning si può pensare di procedere alla soluzione del problema tramite programmazione dinamica[15].

Tra le motivazioni che hanno condotto alla scelta di usare il Q-learning sicuramente la più rilevante è che tale algoritmo è capace di adattarsi a tutte quelle situazioni in cui non è possibile esplorare interamente lo spazio degli stati[16] perché magari non si conosce la MDP che regola l'ambiente di studio oppure perché la loro cardinalità renderebbe il problema computazionalmente intrattabile. Tuttavia, in questo particolare caso di studio, la MDP è nota e il numero di stati è tutto sommato esiguo. Questo porta a considerare come una valida alternativa l'utilizzo della DP.

[15] DAVID SILVER*, Lecture 3: Planning By Dynamic Programming*, in «Introduction To Reinforcement Learning with David Silver», 2015.
[16] IVI.*, Lecture 4: Model-Free Prediction*.

Data una policy si può trovare la V-table dell'MDP e, una volta ottenuta la V-table, è possibile raffinare la precedente policy. Questo passaggio si bassa sull'equazione di Bellman[17] e viene eseguito un numero fisso di volte oppure fino a quando si ritiene che si sia giunti a convergenza: infatti alla fine del ciclo di iterazioni dovremmo aver ottenuto la policy ottimale e la conseguente V-table.

$$V^*(s) := max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

L'equazione rappresenta un passo di iterazione dell'algoritmo. $V^*(s)$ rappresenta il valore della policy ottimale; $T(s,a,s')$ è la probabilità di transizione dallo stato s allo stato s' effettuando l'azione a; $R(s,a)$ è il reward ottenuto dallo stato s che ha fatto l'azione a.

Nel programma (attempt_dynamic_programming.py) tutti questi elementi sono già interamente contenuti nel dizionario di dizionari P utilizzato per l'implementazione del Q-learning nel paragrafo precedente.

Tale calcolo deve essere fatto per tutti gli stati e per tutte le iterazioni fino a convergenza, da qui si può già intuire il motivo per cui MDPs con un'alta cardinalità di stati non possono essere trattate con questo algoritmo.

Una volta ottenuta $V^*(s)$ è possibile calcolare la policy ottimale:

$$\pi^*(s) := argmax_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

L'implementazione prevede i seguenti passaggi:

1) Inizializzare la V-table a zero;
2) Aggiornare la i valori V(s) tramite l'equazione di Bellman;
3) Calcolare la massima variazione di valore tra tutti gli stati;
4) Se la variazione massima è minore della soglia 0.0001 si termina, altrimenti tornare al punto 2;
5) Calcolare la policy ottimale.

[17] DAVID SILVER*, Lecture 3: Planning By Dynamic Programming*, in «Introduction To Reinforcement Learning with David Silver», 2015.

Di seguito vengono proposti i risultati ottenuti:

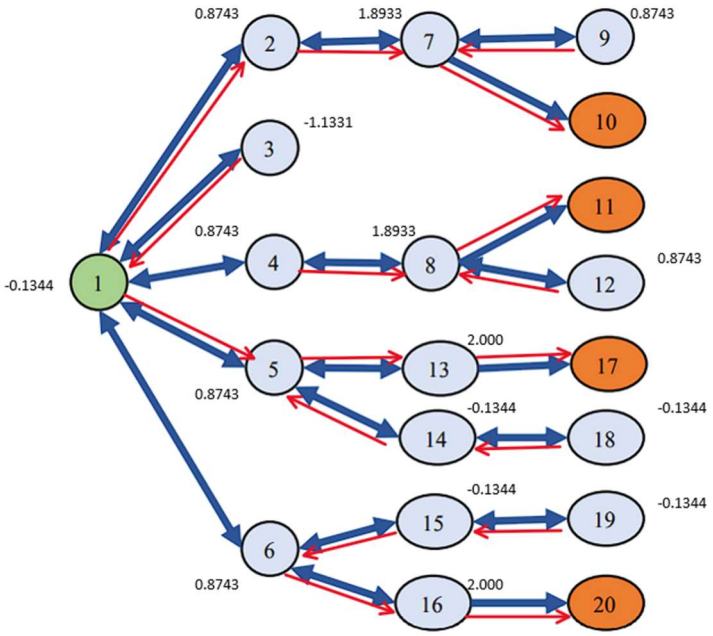


Figura 13: *Grafo dell'MDP con in rosso le azioni intraprese seguendo la policy appresa.*

È possibile notare come la policy ottenuta sia identica a quella appresa tramite Q-learning, a conferma della validità di entrambi i metodi per risolvere questo semplice caso di studio.

Secondo caso di studio

2.1. Introduzione.

Nel precedente caso di studio si è preso in esame l'albero rappresentante un semplice assemblaggio reale e siamo riusciti ad ottenere una policy tale per cui il robot è sempre in grado di scegliere il percorso che conduce allo stato finale. Questo obiettivo è stato raggiunto utilizzando sia un algoritmo basato su programmazione dinamica sia uno basato su RL.

Ora l'obiettivo sarà di cercare di estendere la soluzione a qualsiasi albero, anche di dimensioni molto maggiori. Per questo motivo verrà presa in considerazione solamente la soluzione basata su RL. Questa scelta è dettata dal fatto che l'approccio con programmazione richiede che:

1) il processo di Markov MDP sia conosciuto;
2) la forma matriciale dell'equazione di Bellman[18] non abbia dimensioni tali da rendere troppo gravosi i calcoli da effettuare.

Se il punto 1 non desta particolare preoccupazione poiché il processo di composizione di un prodotto deve essere noto a priori, il punto 2 è potenzialmente fonte di problemi: un assemblaggio complesso potrebbe comportare una matrice intrattabile.

2.2. Costruzione dell'albero.

Per testare l'efficacia del Q-learning con alberi di diverse dimensioni è stato prodotto lo script tree_attempt.py. In particolare, al suo interno è presente la funzione createTree (n1, n2), i cui parametri in input n1 e n2 rappresentano rispettivamente l'altezza dell'albero e numero di figli di ciascun nodo. Viene restituita una struttura dati che rappresenta un albero completo[19].

Un generico nodo foglia può rappresentare l'assemblaggio completato oppure uno stato non valido; la classe di appartenenza finale è il risultato di un'estrazione casuale equiprobabile. Ai nodi sono stati dati i nomi nel seguente modo:

- il nodo radice è 0;
- i figli di ciascun nodo sono numerati in ordine crescente da 0;
- ciascun figlio eredita il nome del padre e vi appende il proprio numero.

[18] *Ibidem*.
[19] MAURIZIO GIACCI, *Appunti Laboratorio di Algoritmi e Strutture Dati*, pp. 15-18

Questo criterio di nomenclatura permette, dato un qualsiasi nodo, di poter intuire i suoi antenati a colpo d'occhio. Per quel che riguarda i reward, al raggiungimento dei nodi foglia validi è stato scelto di attribuire un punteggio positivo uguale all'altezza dell'albero, mentre un -1 per tutti gli alti step. Utilizzeremo inoltre tale convenzione: se un nodo ha N figli, le azioni 0, 1, …, N-1 servono a raggiungere i figli, mentre l'azione N consiste nel tornare al padre. Ad esempio, se 010110 non è un nodo foglia e ha due figli, sapremo automaticamente che:

- non è il nodo radice (che è sempre 0);
- i suoi antenati sono 0, 01, 010, 0101, 01011 (padre);
- è il primo figlio di 01011;
- i figli si chiameranno 0101100, 0101101, 010112
- l'azione 0 condurrà al figlio 0101100
- l'azione 1 condurrà al figlio 0101101
- l'azione 2 condurrà al padre 01011

2.3. Risultati.

Nel caso di una chiamata della funzione createTree(4,2) abbiamo ottenuto questo grafo:

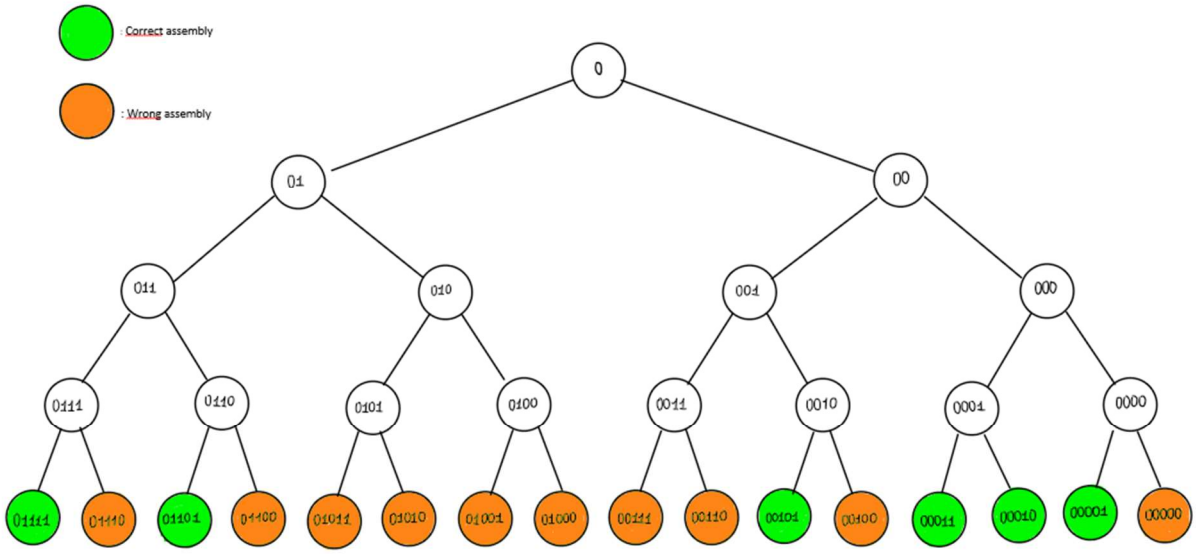


Figura 14.

Il procedimento di addestramento tramite Q-learning ha prodotto la seguente Q-table:

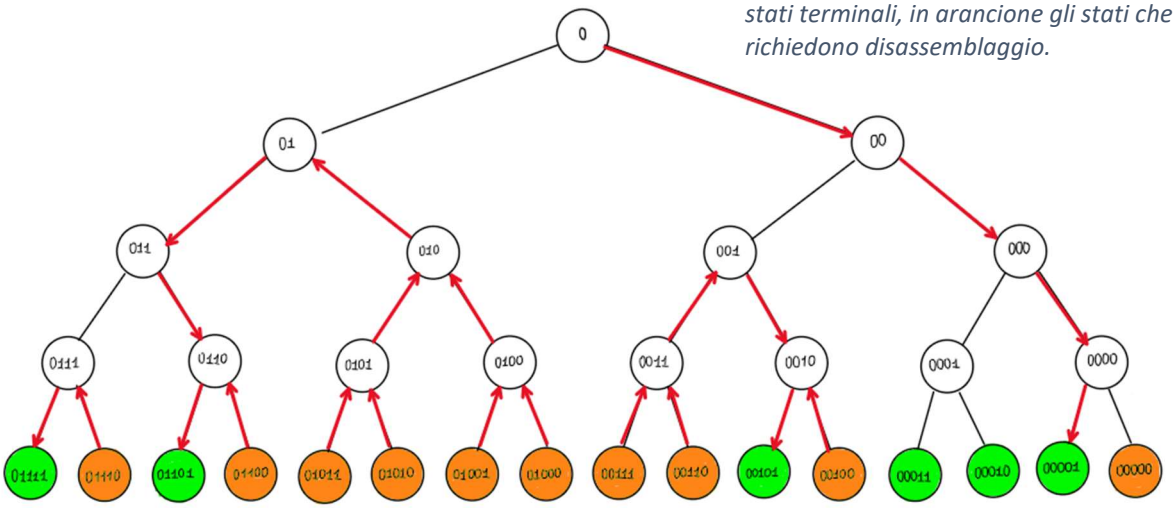
Tabella 5: *Q-Table che a ciascuna coppia stato-azione associa il valore appreso durante il training. L'azione scelta per la policy è sottolineata in giallo.*

Q(stato,azione)	AZIONE=0	AZIONE=1	AZIONE=2
0	1	1	\
00	2	2	-6.5565e-07
001	3	1	1
0010	2	4	2
000	3	3	1
00100	\	\	3
00101	\	\	\
01	-6.5565e-07	2	-6.5565e-07
0001	4	4	1.9999
00011	\	\	\
011	3	3	1
0111	1.9981	4	2
01110	\	\	
01111	\	\	\
0110	2	4	2
01100	\	\	3
01101	\	\	\
00010	\	\	\
010	-1	-1	1
0011	-6.1658e-04	-3.6895e-04	2
00111	\	\	1
00110	\	\	0.9999
0000	2	4	2
00000	\	\	3
00001	\	\	\
0100	--2.0015	-2	-6.5574e-07
01001	\	\	-1
01000	\	\	-1.0002
0101	-2.0021	-2.0002	-6.5565e-07
01010	\	\	-1.0002
01011	\	\	-1

: policy

Si riporta la policy sul grafico dell'albero:

Figura 15: *Grafo dell'MDP con policy segnata con frecce rosse. In verde gli stati terminali, in arancione gli stati che richiedono disassemblaggio.*



È possibile notare che, se si lascia assoluta libertà al robot, il percorso proposto sarà sempre 0 → 00 → 000 → 0000 → 00000. Tuttavia, se viene forzato un altro stato dall'uomo, verrà proposto un percorso alternativo per giungere a uno degli stati finali validi.

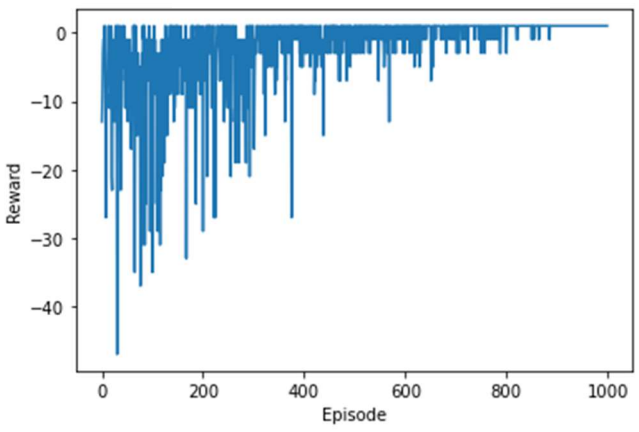
Esempio:

- l'umano impone inizialmente 0 → 01 → 010;
- il robot propone 010 → 01 → 011 → 0110 → 01101.

Sostanzialmente il robot reagisce in maniera adattiva, cercando di riportare il cammino lungo la direzione corretta nel caso in cui l'umano faccia scelte errate o non ottimali.

Tale risultato è sicuramente positivo, ed è stato ottenuto portando a termine 1000 episodi di addestramento.

Figura 16: *Grafico episodi-reward.*



Il grafico di fig.16 mostra che sono stati effettivamente sufficienti per giungere a convergenza nonostante il valore di ε sia stato inizialmente impostato a 0.9. In questa maniera la convergenza risulta senz'altro più lenta, ma viene incentivata la fase di esplorazione durante i primi episodi.

Si può verificare la correttezza della Q-table e della policy generate comparandole direttamente con quelle che ci possiamo calcolare manualmente (poiché stiamo valutando un albero ancora relativamente semplice). Le due Q-table (tab.5 e tab.6) risultano essere sostanzialmente identiche al netto di variazioni a partire dalla quarta cifra decimale; questo è indice del fatto che, durante l'addestramento, l'esplorazione è stata efficace e tutti i nodi sono stati valutati frequentemente allo scorrere degli episodi. In altri termini: non siamo mai rimasti imbottigliati nell'esplorazione di una sola porzione dell'albero, abbiamo avuto modo di costruirci esperienza su tutto il grafo in modo da imparare qual è il comportamento ottimale in ogni situazione possibile.

Q(stato,azione)	AZIONE=0	AZIONE=1	AZIONE=2
0	1	1	\
00	2	2	0
001	3	1	1
0010	2	4	2
000	3	3	1
00100	\	\	3
00101	\	\	\
01	0	2	0
0001	4	4	2
00011	\	\	\
011	3	3	1
0111	2	4	2
01110	\	\	3
01111	\	\	\
0110	2	4	2
01100	\	\	3
01101	\	\	\
00010	\	\	\
010	-1	-1	1
0011	0	0	2
00111	\	\	1
00110	\	\	1
0000	2	4	2
00000	\	\	3
00001	\	\	\
0100	-2	-2	0
01001	\	\	-1
01000	\	\	-1
0101	-2	-2	0
01010	\	\	-1
01011	\	\	-1

: policy

Tabella 6: *Q-Table ottimale che a ciascuna coppia stato-azione associa il valore. L'azione scelta per la policy ottimale è sottolineata in giallo.*

Si riporta ora un tentativo di addestramento nel caso in cui si passi alla funzione un albero di altezza 4 e con 3 figli per nodo.

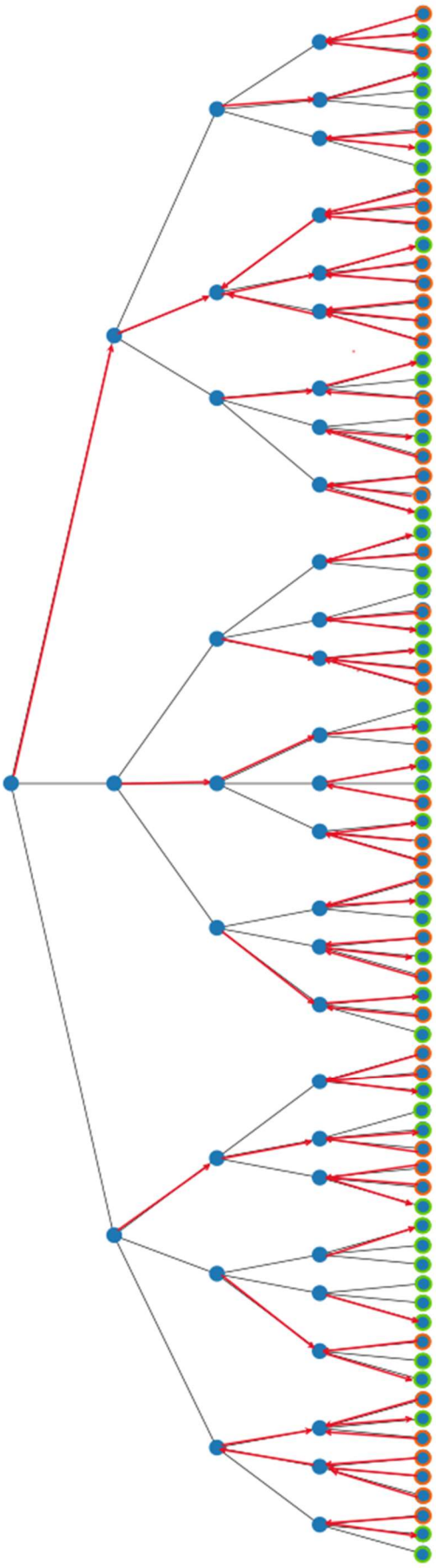


Figura 17: *MDP modellato tramite albero di altezza 4 e 3 figli per nodo.*

Anche se la policy appresa è corretta, analizzando la Q-table calcolata (tab.7) e confrontandola coi valori reali si nota già una certa discrepanza che, tuttavia, risulta non ancora sufficiente ad invalidare la policy. Quindi ci si aspetta che aumentando ancora il numero di nodi, l'affidabilità della stima della Q-table possa calare a tal punto da rendere sbagliata la policy trovata, anche a causa dell'insufficiente esplorazione dell'albero. Il fatto che policy e reward convergano in circa 1000 episodi non implica che la policy appresa sia efficace o anche solo corretta: possono addirittura esserci stati che non sono mai stati visitati e quindi dai quali non è stato possibile apprendere nulla. Qualora il robot, in fase di deployment, si ritrovasse in tali stati, non saprebbe come comportarsi e la modalità più conservativa di procedere sarebbe quella di non fare alcuna azione, di fatto bloccando il processo di assemblaggio.

Si può intuire che in questo caso la convergenza è indice non tanto del fatto che il robot ha imparato tutto ciò che era necessario imparare dall'esperienza, quanto del fatto che sta continuando a reiterare gli stessi percorsi facendo le stesse scelte: la fase di exploitation è troppo sbilanciata e predominante rispetto a quella di exploration. Tale situazione può essere mitigata impostando $\varepsilon = 0.4$ e mantenendolo fisso: sostanzialmente si impone alta la probabilità di fare la scelta che sembra non essere ottimale in modo tale da esplorare anche altri possibili percorsi (tab.8 e fig.18). Oppure si può ridurre il decremento di ε dopo ogni episodio (es. -0.00001 anziché -0.001) per sfruttare una fase di exploration più lunga (tab.9 e fig.19). Si vede che con queste soluzioni la Q-map calcolata torna ad approssimare molto bene quella reale, anche se ovviamente nel primo caso non sarà più possibile osservare convergenza durante gli episodi di addestramento.

Tabella 7: *Q-Value appresa con $\varepsilon = 0.9$ con decremento di 0.001 in 1000 episodi di training per l'MDP di fig.17.*

F0	-4.1723e-07	-4.1729e-07	-4.1724e-07	\
F01	1.0000	0.9915	0.9916	-1.0001
F012	0.1222	1.4844	1.9988	-0.2050
F0121	0.0000	2.3520	2.7667	-0.1030
F01211	0.	0.	0.	0.
F0120	-0.4000	1.9200	1.2000	-0.4673
F01202	0.	0.	0.	0.
F00	1.0000	1.0000	1.0000	-1.0000
F001	1.9477	2.0000	1.6920	-0.0025
F0011	3.0000	2.9961	0.8242	0.9206
F00112	0.0000	0.0000	0.0000	1.9690
F00110	0.	0.	0.	0.
F000	2.0000e+00	1.9999e+00	1.9991e+00	-1.5381e-05
F0002	-0.3566	-1.3952	2.9999	0.8414
F00020	0.0000	0.0000	0.0000	1.3809
F0000	3.0000	0.9958	0.9694	0.9997
F0001	3.0000	2.3520	-0.3428	0.2444

F00010	0.0000	0.0000	0.0000	1.4788
F00010	0.	0.	0.	0.
F0012	1.9200	-0.8000	2.8600	0.1767
F00111	0.	0.	0.	0.
F010	2.0000	1.9860	1.9666	-0.0025
F0101	0.0482	-0.5390	2.9961	0.8464
F01010	0.0000	0.0000	0.0000	1.7279
F01012	0.	0.	0.	0.

Tabella 8: *Q-Value appresa con $\varepsilon = 0.4$ costante in 500000 episodi di training per l'MDP di fig.17.*

F0	-4.1723e-07	-4.1723e-07	-4.1723e-07	\
F01	1.0000	1.0000	1.0000	-1.0000
F012	2.0000e+00	2.0000e+00	2.0000e+00	-4.1723e-07
F0121	1.0000	3.0000	3.0000	1.0000
F01211	0.	0.	0.	0.
F0120	1.0000	3.0000	3.0000	1.0000
F01202	0.	0.	0.	0.
F00	1.0000	1.0000	1.0000	-1.0000
F001	2.0000e+00	2.0000e+00	2.0000e+00	-4.1723e-07
F0011	3.0000	3.0000	1.0000	1.0000
F00112	0.0000	0.0000	0.0000	2.0000
F00110	0.	0.	0.	0.
F000	2.0000e+00	2.0000e+00	2.0000e+00	-4.1723e-07
F0002	1.0000	1.0000	3.0000	1.0000
F00020	0.0000	0.0000	0.0000	2.0000
F0000	3.0000	1.0000	1.0000	1.0000
F0001	3.0000	3.0000	1.0000	1.0000
F00012	0.0000	0.0000	0.0000	2.0000
F00010	0.	0.	0.	0.
F0012	3.0000	1.0000	3.0000	1.0000
F00111	0.	0.	0.	0.
F010	2.0000e+00	2.0000e+00	2.0000e+00	-4.1723e-07
F0101	1.0000	1.0000	3.0000	1.0000
F01010	0.0000	0.0000	0.0000	2.0000
F01012	0.	0.	0.	0.

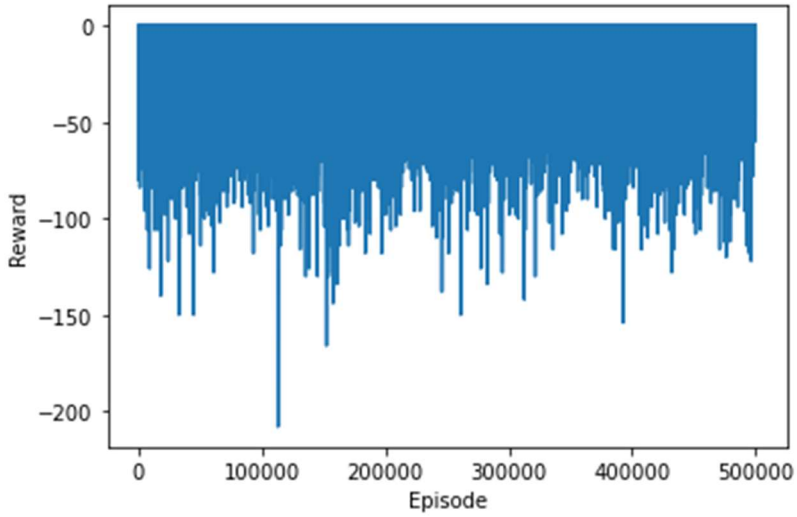


Figura 18: *Grafico episodi-reward per $\varepsilon = 0.4$ costante in 500000 episodi di addestramento.*

Tabella 9: *Q-Value appresa con $\varepsilon = 0.9$ con decremento di 0.00001 in 500000 episodi di training per l'MDP di fig.17.*

F0	-4.1723e-07		-4.1723e-07	-4.1723e-07	\
F01	1.0000		1.0000	1.0000	-1.0000
F012	2.0000e+00		2.0000e+00	2.0000e+00	-4.1723e-07
F0121	1.0000		3.0000	3.0000	1.0000
F01211	0.		0.	0.	0.
F0120	1.0000		3.0000	3.0000	1.0000
F01202	0.		0.	0.	0.
F00	1.0000		1.0000	1.0000	-1.0000
F001	2.0000e+00		2.0000e+00	2.0000e+00	-4.1723e-07
F0011	3.0000		3.0000	1.0000	1.0000
F00112	0.0000		0.0000	0.0000	2.0000
F00110	0.		0.	0.	0.
F000	2.0000e+00		2.0000e+00	2.0000e+00	-4.1723e-07
F0002	1.0000		1.0000	3.0000	1.0000
F00020	0.0000		0.0000	0.0000	2.0000
F0000	3.0000		1.0000	1.0000	1.0000
F0001	3.0000		3.0000	1.0000	1.0000
F00012	0.0000		0.0000	0.0000	2.0000
F00010	0.		0.	0.	0.
F0012	3.0000		1.0000	3.0000	1.0000
F00111	0.		0.	0.	0.
F010	2.0000e+00		2.0000e+00	2.0000e+00	-4.1723e-07
F0101	1.0000		1.0000	3.0000	1.0000
F01010	0.0000		0.0000	0.0000	2.0000
F01012	0.		0.	0.	0.

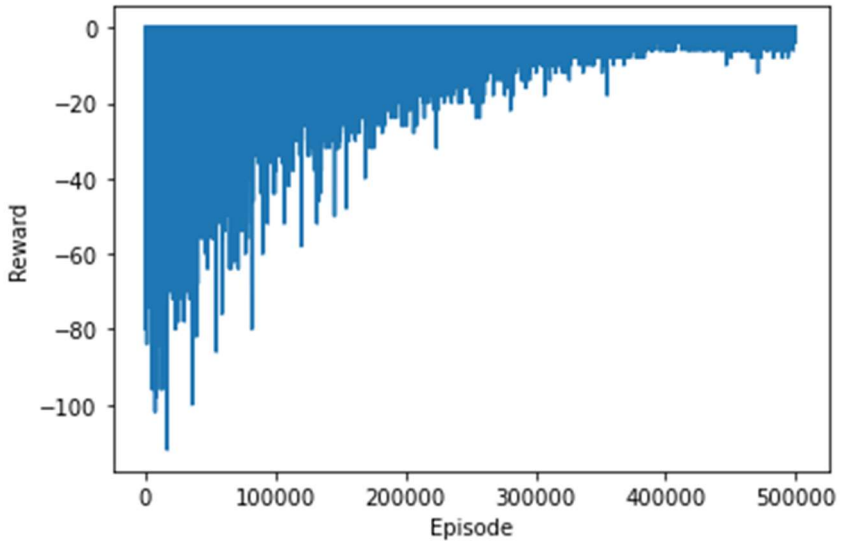


Figura 19*: Grafico episodi-reward per $\varepsilon = 0.9$ con decremento di 0.00001 in 500000 episodi di addestramento.*

Terzo caso di studio

3.1. Introduzione.

Nei primi due casi presi in esame siamo riusciti ad ottenere il risultato voluto: dato un qualsiasi albero rappresentante uno schema di assemblaggio e un sufficiente numero di episodi di allenamento, il robot ha scoperto per ciascuno stato qual è il successivo step per avvicinarsi il più velocemente possibile al prodotto finale. È il momento ora di stressare il concetto di "più velocemente possibile" avendo ben saldo in mente che stiamo considerando il punto di vista di un robot che sta operando in collaborazione con un umano; questo può sbagliare o non avere le idee chiare su quale possa essere la sequenza di assemblaggio ottimale e, più in generale, ha un comportamento non deterministico e non conoscibile a priori.

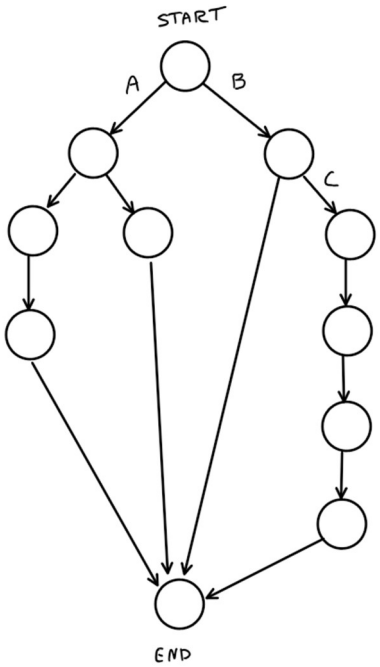


***Figura 20**: Grafo rappresentante l'MDP di esempio.*

Consideriamo l'MDP in fig.20 e supponiamo che robot e umano debbano compiere un'azione a testa e che inizi il robot. Quale tra A e B rappresenta l'opzione migliore?

L'azione B consentirebbe potenzialmente di portare a terminare l'assemblaggio in soli 2 step, ma se successivamente l'umano decidesse di fare l'azione C saremmo vincolati a proseguire lungo

il percorso più lungo (lunghezza = 6). All'opposto, l'azione A non permette di selezionare il percorso globalmente più breve ma fa sì che, qualsiasi sia la successiva scelta dell'uomo, la lunghezza massima del cammino sia 4. Questa appare quindi come l'opzione migliore, nel senso che minimizza la possibilità che l'uomo possa fare scelte deleterie durante il processo di assemblaggio.

In quest'ottica può essere utile immaginare che tra uomo e robot si stia disputando una partita: da una parte il robot cerca di arrivare al termine del grafo il prima possibile, dall'altra l'umano tenta di rallentare il più possibile questo processo. Questo tipo di dinamica può essere modellata ed affrontata utilizzando tecniche di adversarial reinforcement learning.

3.2. Adversarial Reinforcement Learning.

> In an adversarial setting there are multiple (at least two) agents in the world. In particular, in a game with two players, when an agent wins a game it is given a positive reinforcement and its opponent is given negative reinforcement. Maximizing reward corresponds directly to winning games. Over time the agent is learning to act so that it wins the game.[20]

Nel terzo e ultimo caso di studio si cercherà di adottare un punto di vista diverso rispetto ai precedenti: l'obiettivo ora è di addestrare sia l'agente-robot, sia l'agente-umano. Il robot dovrà imparare a fare scelte che impediscano all'umano di allungare il procedimento di assemblaggio, mentre l'umano verrà allenato a fare le scelte peggiori. Durante il training entrambi gli agenti diventeranno più abili nel selezionare le proprie azioni modificando opportunamente le proprie policy ed esplorando vari cammini del grafo, stimolati ciascuno dal miglioramento dell'altro. Il training terminerà quando le policy non subiranno più variazioni significative oppure dopo un numero fissato di episodi (partite).

In ogni partita avremo:

- i giocatori: un agente-robot e un agente-umano in competizione tra loro;
- l'arena di gioco: è un grafo diretto aciclico[21] (DAG) con alcune peculiarità che verranno spiegate in dettaglio in seguito;

[20] WILLIAM T. B UTHER, MANUELA M. VELOSO, *Adversarial Reinforcement Learning, Gennaio* 2003, p.2.
[21] K. THULASIRAMAN, M. N. S. SWAMY, *Graphs: Theory and Algorithms*, John Wiley and Son, 1992, p. 118.

- l'obiettivo del robot: giungere a uno dei nodi finali il prima possibile;
- l'obiettivo dell'umano: tardare il più possibile l'arrivo ai nodi finali;
- le regole:
    - le partite iniziano dallo stato 0;
    - inizia il robot;
    - robot e umano hanno a disposizione un turno a testa fino al termine della partita;
    - la partita termina quando si giunge a uno dei nodi finali;

L'applicazione proposta si differenzia dal tradizionale adversarial RL su un aspetto rilevante: nel nostro caso l'esito della partita è univoco ed ineluttabile. Si giungerà sempre e comunque al termine del grafo, quindi non ha senso parlare di vittoria o sconfitta per i due agenti. In una plausibile analogia con un videogioco, per i due giocatori l'obiettivo non è di vincere la partita, ma bensì di raccogliere (o perdere) il maggiore (o minore) numero di punti-rewards possibile prima che la partita termini.

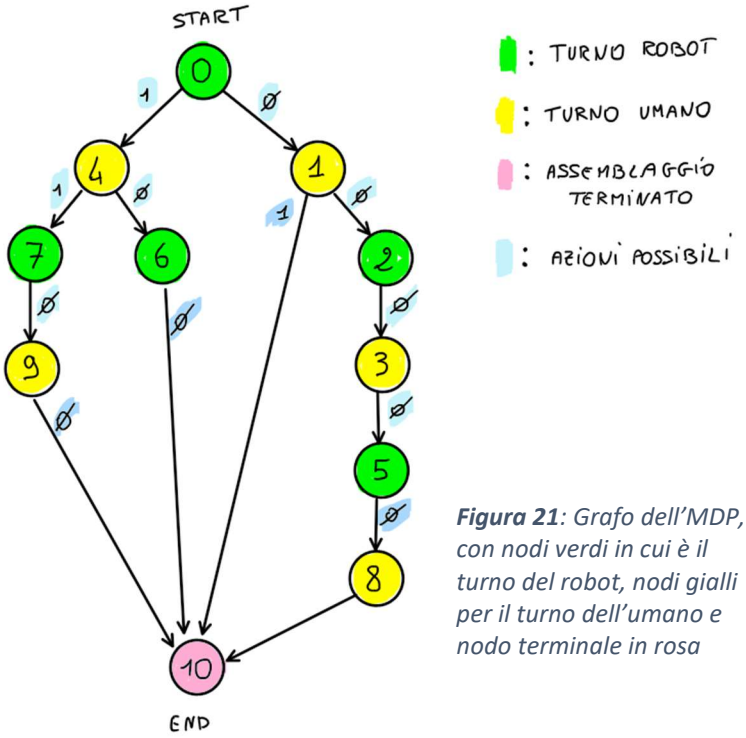


***Figura 21**: Grafo dell'MDP, con nodi verdi in cui è il turno del robot, nodi gialli per il turno dell'umano e nodo terminale in rosa*

Verrà inizialmente preso in considerazione un grafo semplice (fig.21) per rendere evidenti i risultati forniti dall'algoritmo. Successivamente verranno considerati grafi generati casualmente con un numero considerevolmente maggiore di nodi ed archi al fine di testarne scalabilità e bontà in casi più realistici.

3.3. L'algoritmo.

Per i primi due casi di studio si è scelto di utilizzare una soluzione di RL basata sull'algoritmo classico 'Q-learning': l'agente robot in ogni episodio e in ogni singolo step effettuava la sua nuova azione A sulla stima dei valori Q(S,a) utilizzando una strategia ε-greedy, ricevendo come feedback

dall'ambiente la coppia R,S' e correggendo la suddetta stima supponendo che nel futuro si sceglierà l'azione A che massimizza Q(S',a) :

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_a Q(S',a) - Q(S,A) \right]$$

Questo ragionamento non è più valido se si vuole allenare il robot a competere con un altro agente, in quanto potrà scegliere la propria azione solamente a turni alterni: quando non è il suo turno, l'azione la sceglie l'umano e il robot può solo 'osservare' cercando di imparare qualcosa. Quanto appena descritto non è altro che la definizione stessa di apprendimento off-policy; dunque, possiamo pensare di utilizzare ancora l'algoritmo Q-learning semplicemente facendo sì che la selezione dell'azione sia compito dell'umano che agirà sempre con strategia ε-greedy consultando la propria Q-table.

Rimane da capire come si comporta il robot quando è il suo turno. La cosa più naturale è di pensare di utilizzare l'algoritmo on-policy sarsa[22]. Il robot fa una previsione basandosi sui valori di Q(S,a) e sceglie la propria azione A con paradigma ε-greedy. Osserva la risposta dell'ambiente R,S' e corregge la precedente previsione confrontandola con quello che è effettivamente successo e quello che prevede succederà consultando Q(S',A'), dove A' è la prossima azione che verrà effettuata.

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$$

Del punto di vista dell'umano abbiamo una prospettiva duale: il suo obiettivo è opposto a quello del robot, e quindi i reward saranno diversi. Quando è il turno del robot, esso impara on-policy con sarsa, mentre l'umano apprende off-policy con q-learning e viceversa fino al termine di ogni episodio.

Mettendo insieme tutte queste considerazioni, la struttura complessiva dell'algoritmo risulta essere la seguente:

Inizializzazione a zero dei due dizionari $Q_{robot}(s,a) = 0$, $Q_{umano}(s,a) = 0$, $\forall s \in S, a \in A(s)$

[22] DAVID SILVER, *Lecture 5: Model-Free Control*, in «Introduction To Reinforcement Learning with David Silver», 2015.

Ripeti per ogni episodio:

Turno robot = True

S = 0

Ripeti per ogni step dell'episodio:

Se è il turno del robot:

Scegli l'azione A applicando ε-greedy a $Q_{robot}(S,a)$

Osserva il nuovo stato S'

Se S' è lo stato finale:

$R_{robot} = 5$

$R_{umano} = -5$

Se S' non è lo stato finale:

$R_{robot} = -1$

$R_{umano} = +1$

Scegli la prossima azione A' applicando ε-greedy a $Q_{umano}(S,a)$

$Q_{robot}(S,A) \leftarrow Q_{robot}(S,A) + \alpha[R_{robot} + \gamma\, Q_{robot}(S',A') - Q_{robot}(S,A)]$

$Q_{umano}(S,A) \leftarrow Q_{umano}(S,A) + \alpha[R_{umano} + \gamma\, max_a(Q_{umano}(S',a)) - Q_{umano}(S,A)]$

Turno robot = False

Se è il turno dell'umano:

Scegli l'azione A applicando ε-greedy a $Q_{umano}(S,a)$

Osserva il nuovo stato S'

Se S' è lo stato finale:

$R_{robot} = 5$

$R_{umano} = -5$

Se S' non è lo stato finale:

$R_{robot} = -1$

$R_{umano} = +1$

Scegli la prossima azione A' applicando ε-greedy a $Q_{robot}(S,a)$

$Q_{umano}(S,A) \leftarrow Q_{umano}(S,A) + \alpha[R_{umano} + \gamma\, Q_{umano}(S',A') - Q_{umano}(S,A)]$

$Q_{robot}(S,A) \leftarrow Q_{robot}(S,A) + \alpha[R_{robot} + \gamma\, max_a(Q_{robot}(S',a)) - Q_{robot}(S,A)]$

Turno robot = True

$S \leftarrow S'$

Fino a quando S non è lo stato finale

▢ : sarsa

▢ : q-learning

3.4. Risultati e considerazioni su DAG semplice.

L'algoritmo è stato messo alla prova inizialmente sul grafo semplice riportato in fig.22 per due motivi:

- verificare la correttezza dell'output in una situazione computazionalmente poco gravosa: infatti per esplorare in maniera pervasiva il grafo non è necessario un alto numero di episodi di training;
- poter avere la possibilità di dare una chiara ed intuitiva interpretazione ai risultati ottenuti dall'algoritmo con una semplice ispezione visiva ai dati e al grafico del grafo: vedremo più avanti che con DAG aventi decine o centinaia di nodi e archi la rappresentazione grafica non sarà più di nessun aiuto per discutere la correttezza del risultato.

Vengono riportate le policy apprese e i dizionari $Q_{robot}(s,a)$ e $Q_{umano}(s,a)$ in tre diverse run aventi come parametri:

1) $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$, episodi = 500 (tab.10A);
2) $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$, episodi = 50000 (tab.10B);
3) $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$, episodi = 5000000 (tab.10C);

N.B.: per ogni nodo avente due o più archi uscenti, essi rappresentano le azioni possibili in ciascuno stato e vengono numerate in ordine crescente a partire da 0 dall'alto vero il basso.

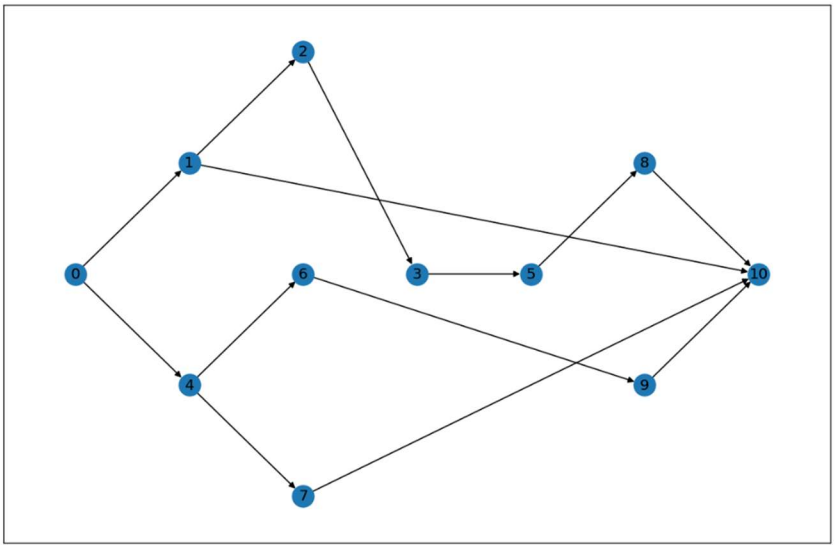


***Figura 22**: Grafo dell'MDP semplice con zero come nodo iniziale e 10 come nodo finale.*

***Tabella 10A**: Q-Table appresa con $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$ e 500 episodi di training.*

Q(S,A) ROBOT	AZIONE = 0	AZIONE = 1
STATO = 0	0.23576016357849908	2.0000000000000053
STATO = 1	0.9687563980829371	3.2
STATO = 2	1.9923517470926064	
STATO = 3	2.998644633401987	
STATO = 4	2.9999999999999982	3.999903791905307
STATO = 5	3.9998453392331172	
STATO = 6	3.9999999999999987	
STATO = 7	4.999994882548155	
STATO = 8	4.999991470913591	
STATO = 9	4.999999999999999	

Q(S,A) UMANO	AZIONE = 0	AZIONE = 1
STATO = 0	0.09405519884752289	-1.999999999999998
STATO = 1	-0.9687563980829371	-3.2
STATO = 2	-1.9923517470926064	
STATO = 3	-2.998644633401987	
STATO = 4	-2.9999999999999982	-3.999903791905307
STATO = 5	-3.9998453392331172	
STATO = 6	-3.9999999999999987	
STATO = 7	-4.999994882548155	
STATO = 8	-4.999991470913591	
STATO = 9	-4.999999999999999	

***Tabella 10B**: Q-Table appresa con $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$ e 50000 episodi di training.*

Q(S,A) ROBOT	AZIONE = 0	AZIONE = 1
STATO = 0	1.2636966846260 76e-05	2.0000696019903126
STATO = 1	0.9999999999999979	4.999999999999999
STATO = 2	1.999999999999998	
STATO = 3	2.9999999999999982	
STATO = 4	2.9999999999999982	3.9999999999999987
STATO = 5	3.9999999999999987	
STATO = 6	3.9999999999999987	
STATO = 7	4.999999999999999	
STATO = 8	4.999999999999999	
STATO = 9	4.999999999999999	

Q(S,A) UMANO	AZIONE = 0	AZIONE = 1
STATO = 0	2.1094237467877 98e-15	-1.999999999999998
STATO = 1	-0.9999999999999979	-4.999999999999999
STATO = 2	-1.999999999999998	
STATO = 3	-2.9999999999999982	
STATO = 4	-2.9999999999999982	-3.9999999999999987
STATO = 5	-3.9999999999999987	
STATO = 6	-3.9999999999999987	
STATO = 7	-4.999999999999999	
STATO = 8	-4.999999999999999	
STATO = 9	-4.999999999999999	

***Tabella 10C**: Q-Table appresa con $\varepsilon=0.1$, $\alpha = 0.4$, $\gamma = 1$ e 5000000 episodi di training.*

Q(S,A) ROBOT	AZIONE = 0	AZIONE = 1
STATO = 0	3.6005867163581787e-09	1.9999999999999987
STATO = 1	0.9999999999999979	4.999999999999999
STATO = 2	1.999999999999998	
STATO = 3	2.9999999999999982	
STATO = 4	2.9999999999999982	3.9999999999999987
STATO = 5	3.9999999999999987	
STATO = 6	3.9999999999999987	
STATO = 7	4.999999999999999	
STATO = 8	4.999999999999999	
STATO = 9	4.999999999999999	

Q(S,A) UMANO	AZIONE = 0	AZIONE = 1
STATO = 0	2.1094237467877 98e-15	-1.999999999999998
STATO = 1	-0.9999999999999979	-4.999999999999999
STATO = 2	-1.999999999999998	
STATO = 3	-2.9999999999999982	
STATO = 4	-2.9999999999999982	-3.9999999999999987
STATO = 5	-3.9999999999999987	
STATO = 6	-3.9999999999999987	
STATO = 7	-4.999999999999999	
STATO = 8	-4.999999999999999	
STATO = 9	-4.999999999999999	

Ciò che è subito evidente è che sono sufficienti soli 500 episodi per ottenere la policy ottimale e anche i valori delle mappe $Q_{robot}(s,a)$ e $Q_{umano}(s,a)$ risultano convergere velocemente ai valori reali che si possono calcolare manualmente tramite una rapida ispezione visiva del grafo:

***Tabella 11**: Q-Table ottimale*

$Q_{robot}(s,a)$	AZIONE = 0	AZIONE = 1
STATO = 0	0	2
STATO = 1	1	5
STATO = 2	2	\
STATO = 3	3	\
STATO = 4	3	4
STATO = 5	4	\
STATO = 6	4	\
STATO = 7	5	\
STATO = 8	5	\
STATO = 9	5	\

$Q_{umano}(s,a)$	AZIONE = 0	AZIONE = 1
STATO = 0	0	-2
STATO = 1	-1	-5
STATO = 2	-2	\
STATO = 3	-3	\
STATO = 4	-3	-4
STATO = 5	-4	\
STATO = 6	-4	\
STATO = 7	-5	\
STATO = 8	-5	\
STATO = 9	-5	\

I risultati ottenuti sono dunque soddisfacenti; l'esplorazione dello spazio delle soluzioni e il bilanciamento tra exploration and exploitation in questo semplice DAG risultano più che adeguati anche mantenendo ε costante.

3.5. Struttura e caratteristiche del grafo.

In precedenza si è affermato che l'addestramento con adversarial learning sarebbe stato effettuato su dei grafi con caratteristiche ben precise: i DAG (Directed Acyclic Graph). La scelta di utilizzare questo tipo di "campo di gioco" per la competizione dei due agenti è dettata dalla necessità di garantire che durante le partite non possano verificarsi situazioni di impasse: a tal proposito si consideri la situazione illustrata in fig.23, in cui nello stato 1 è il turno del robot e nello stato 2 tocca all'uomo.

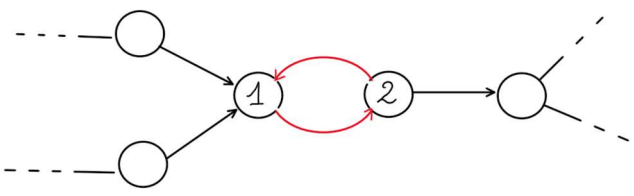


***Figura 23**: Criticità dei cicli nei grafi degli MDP.*

Risulta evidente che l'operatore umano può imporre la reiterazione dello stesso passaggio 2→1 infinite volte: la partita avrebbe durata infinita e pregiudicherebbe l'apprendimento dei due agenti che, infatti, non riuscirebbero a portare a termine l'episodio. L'esigenza è quindi di considerare solamente grafi aciclici ed impedire, durante il cammino del grafo, di poter "tornare indietro".

Ci si potrebbe chiedere se questo tipo di vincolo possa essere in qualche modo troppo stringente e rendere il DAG inadatto a modellare uno schema di assemblaggio collaborativo reale che, di fatto, dovrebbe tener conto di un possibile errore e/o della necessità di dover reiterare più di una volta alcuni passaggi. Per superare questo problema, la soluzione proposta è di considerare anche i grafi ciclici e, sotto l'ipotesi che l'errore (o la reiterazione) di un passaggio possa avvenire solo una volta, convertirli in DAG ridondando le porzioni di grafo che possono essere ripetute (fig.24).

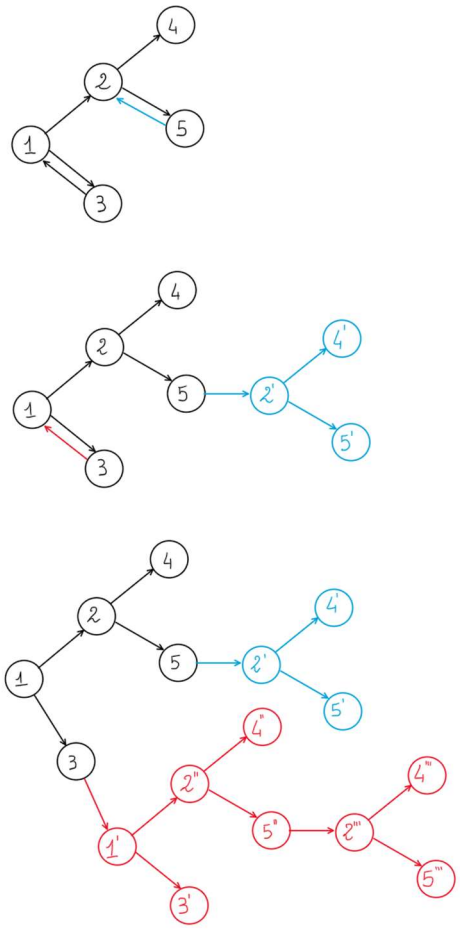


***Figura 24**: Conversione da grafo ciclico a DAG.*

Grazie a questo procedimento possiamo quindi ritenere i DAGs adatti a rappresentare un qualsiasi processo di assemblaggio.

Rimane, tuttavia, da risolvere un'ambiguità che sorge in alcuni casi e che rende fallace l'addestramento dei due agenti. Si consideri il grafo riportato in fig.25:

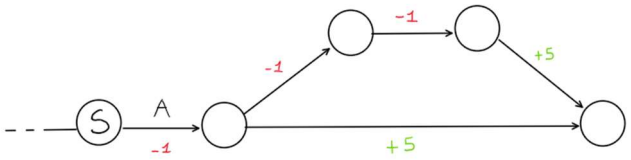


***Figura 25**: La porzione di grafo mette in evidenza la criticità che sorge nel non sapere a priori di chi sia il turno nello stato S.*

Ponendoci dal punto di vista del robot, non siamo in grado di calcolare il valore $Q_{robot}(S,A)$ se non sappiamo di chi sia il turno nello stato S. Infatti:

1) se è il turno del robot allora $Q_{robot}(S,A) = 2$;
2) se è il turno dell'uomo abbiamo $Q_{robot}(S,A) = 4$.

Questa ambiguità renderà non deterministico il valore $Q_{robot}(S,A)$ appreso dal robot e durante l'addestramento potrà convergere a 2, a 4 oppure continuare a variare in questo intervallo senza arrivare a convergenza. Per evitare questo comportamento è necessario che ciascuno stato sia di "proprietà" esclusiva o del robot o dell'umano, in modo tale che nello stesso stato sia sempre e solo il turno dello stesso agente durante tutti gli episodi di training.

Questo ulteriore vincolo non toglie nulla alla capacità espressiva del DAG in un generico caso realistico di assemblaggio.

Dal punto di vista implementativo, questa caratteristica può essere realizzata organizzando il grafo in una struttura a layers: tutti i nodi che appartengono a un determinato livello sono di proprietà del robot o dell'umano. Gli archi uscenti dai nodi di questo livello potranno raggiungere solamente i livelli appartenuti all'avversario.

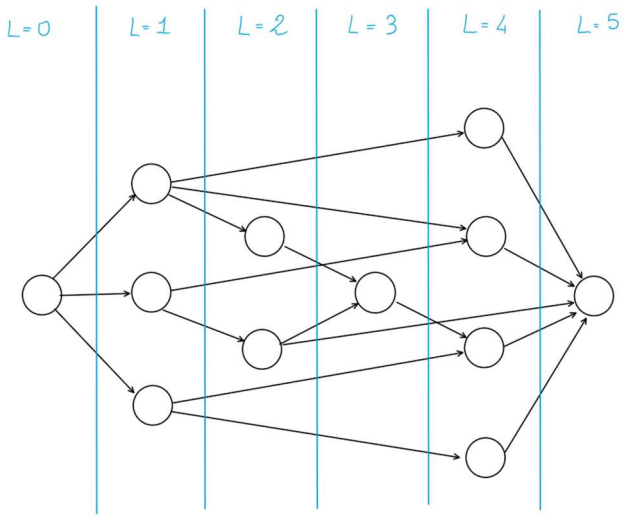


***Figura 25**: Suddivisione del grafo in livelli.*

L'esempio in fig.25 mostra che, assegnando i livelli pari (L = [0, 2, 4]) al robot e quelli dispari (L = [1, 3, 5]) all'umano, vengono tracciati solamente archi che collegano livelli pari a livelli dispari e viceversa, garantendo così la corretta turnazione tra uomo e robot e l'univocità di appartenenza di ciascun nodo.

3.6. Costruzione DAG casuale.

Finora abbiamo trattato approfonditamente le caratteristiche che deve avere il grafo per poter correttamente rappresentare un generico assemblaggio e consentire l'addestramento dei due agenti; inoltre abbiamo potuto constatare che l'algoritmo si comporta in maniera decisamente soddisfacente nel caso semplice rappresentato in fig.21.

Il prossimo obiettivo è di valutare l'efficacia delle policy apprese in situazioni ben più complesse, ovvero su DAG con un numero considerevolmente maggiore di nodi e archi. Si vorrebbe anche provare il fatto che i risultati positivi ottenuti non sono il frutto di una fortunata combinazione di eventi, ma conseguenza alla reale efficacia della metodologia proposta: è dunque necessario testare l'algoritmo su un'ampia varietà di grafi con topologia casuale (ma che rispettino le caratteristiche discusse in precedenza).

A tale scopo è stata implementata una funzione il cui prototipo è gen_graph (layers, max_actions, max_length) il cui obiettivo è la creazione di DAG casuali che verranno successivamente dati in input all'algoritmo.

I parametri in input della funzione sono:

- layers: il massimo numero di livelli che avrà il grafo finale;
- max_length: il massimo numero di nodi che ci saranno in ciascun livello;
- max_actions: il massimo numero di archi (azioni) uscenti da ciascun nodo.

Per analizzare la funzione proposta si ipotizzi di chiamarla con i seguenti parametri: layers = 6, max_actions = 3, max_length = 4. Il primo passaggio è di disporre 4 nodi in ciascuno dei 6 livelli, tranne nel livello 0 in cui ci sarà solo il nodo di partenza:

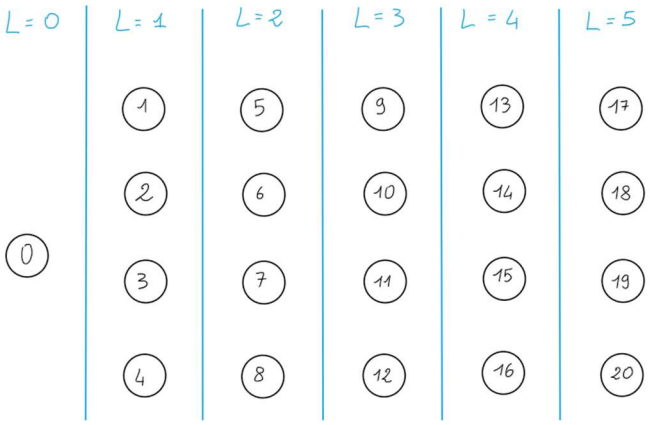


***Figura 26**: Primo passaggio per la creazione procedurale di DAGS a livelli: distribuire i nodi nei vari livelli.*

Per ciascun nodo - eccetto quelli dell'ultimo livello - bisogna decidere quanti archi uscenti generare e la loro destinazione. Siccome il grafo deve essere casuale, tali scelte saranno il risultato di un'estrazione: ciascun nodo sorteggerà un numero nell'intervallo discreto [1, max_actions] e questo rappresenterà i suoi archi uscenti. Per ciascun arco, la destinazione verrà sorteggiata tra uno dei nodi successivi compatibili. Ad esempio, il nodo 3 potrà estrarre solo i nodi [5, 6, 7, 8, 13, 14, 15, 16].

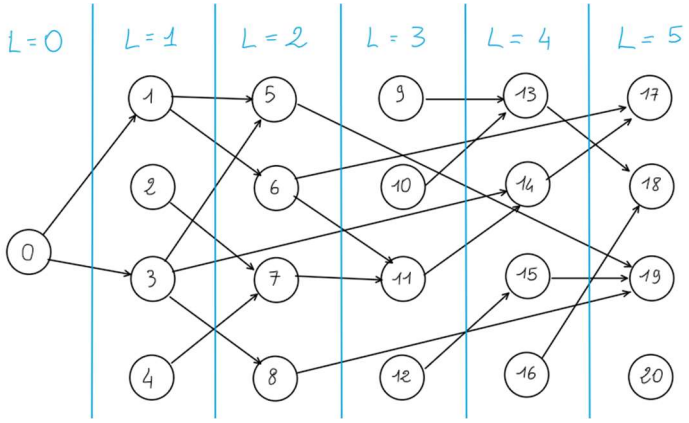


***Figura 27**: Secondo passaggio per la creazione procedurale di DAGS a livelli: creare gli archi tra i vari nodi.*

A questo punto è necessario rimuovere gli stati che non sono raggiungibili dal nodo 0.

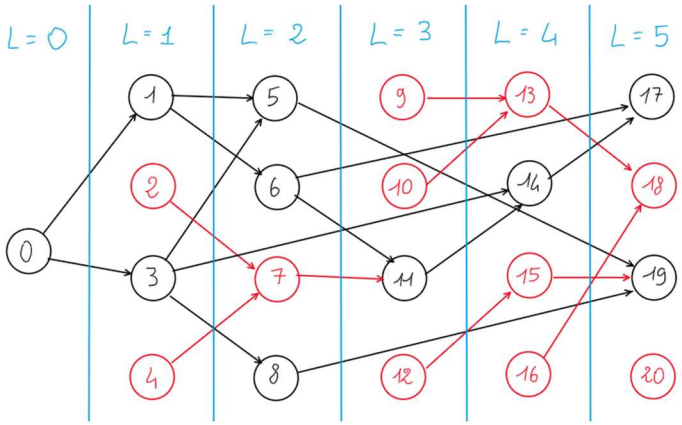


***Figura 28**: Terzo passaggio per la creazione procedurale di DAGS a livelli: individuazione dei nodi non raggiungibili dal nodo iniziale 0.*

Infine, si rinumerano i nodi superstiti.

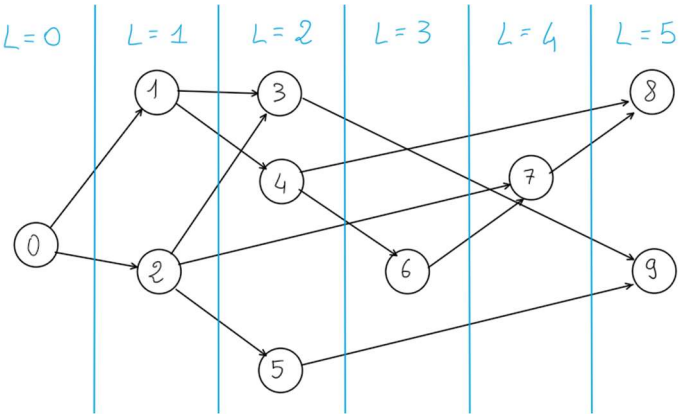


***Figura 29**: Ultimo passaggio per la creazione procedurale di DAGS a livelli: eliminazione dei nodi non raggiungibili e rinumerazione dei nodi superstiti.*

Rimane da discutere il processo di sorteggio delle azioni e delle destinazioni degli archi.

Con una banale estrazione equiprobabile, i grafici generati risultano troppo spesso sbilanciati verso i nodi finali, dove si addensa la maggior parte degli archi. Mediamente si osserva una tipica forma ad 'imbuto' orizzontale che risulta inadatta a rappresentare scenari interessanti o anche solo simili ad un assemblaggio reale. Vengono proposti i grafi ottenuti chiamando tre volte la funzione gen_graph(15, 4, 8):

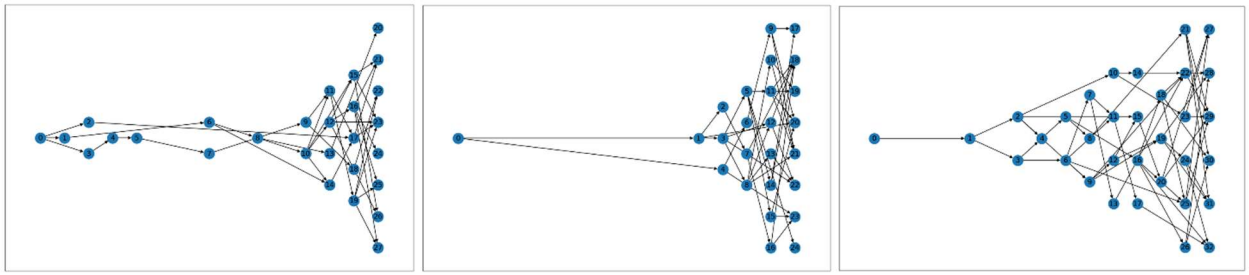


***Figura 30**: Grafi generati casualmente chiamando la funzione gen_graph(15, 4, 8).*

Per ridistribuire in maniera più uniforme su tutto il grafo gli archi e i nodi superstiti appare dunque necessario fare in modo che i primi nodi abbiano una probabilità maggiore di fare un numero superiore di azioni rispetto ai nodi finali. Ma non basta: gli archi dei nodi iniziali devono tendere a raggiungere nodi nei livelli vicini piuttosto che quelli lontani. In questo modo si penalizzano gli archi che saltano molti livelli e si riduce la probabilità che si formi l'imbuto.

Questo processo di ribilanciamento, dal punto di vista implementativo, è stato realizzato tramite l'estrazione del numero di azioni nel caso di una distribuzione di probabilità multinomiale che cambia a seconda del nodo considerato. Considerando il grafo di fig.29, vorremmo che il nodo 0 facesse con maggiore probabilità un alto numero di azioni, quindi avremo:

P(0) = probabilità di fare 1 azione = 1/6 , P(2) = 2/6, P(3) = 3/6. Se invece consideriamo uno qualunque dei nodi del livello 4 in cui vogliamo pochi archi uscenti, imporremo come distribuzione: P(1) = 3/6, P(2) = 2/6, P(3) = 1/6. Un procedimento analogo è stato effettuato per la scelta dei nodi destinazione di ciascun arco.

Nuovamente, vengono mostrati i grafi ottenuti con la funzione gen_graph(15, 4, 8) contenente i miglioramenti sopra esposti:

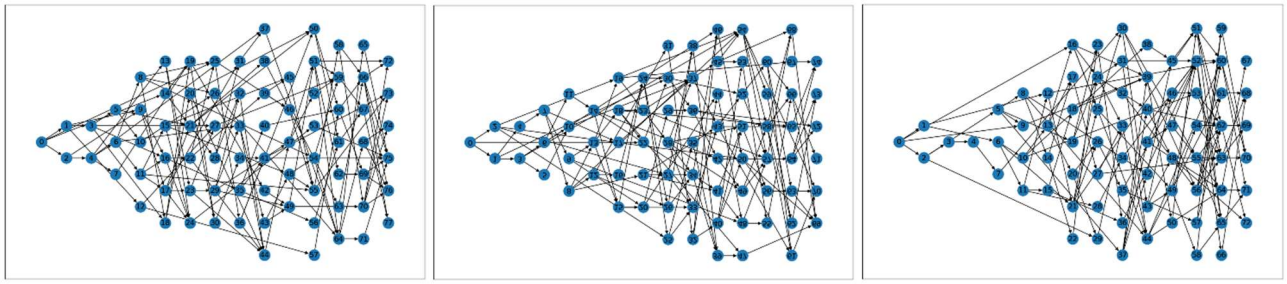


***Figura 31** : Grafi generati casualmente chiamando la funzione gen_graph(15, 4, 8) dopo la modifica della funzione.*

3.7. Metodo di analisi su DAG casuale di medio-grandi dimensioni.

Ora che abbiamo messo a punto l'algoritmo e la funzione di generazione casuale dei DAGs, possiamo finalmente mettere alla prova lo script in situazioni ben più realistiche e complesse.

Tuttavia, a differenza di come avevamo potuto procedere nel caso di fig.21, ora non sarà più possibile validare i risultati generati dall'algoritmo tramite ispezione visiva del grafico e il calcolo manuale della mappa $Q_{robot}(s,a)$. Infatti, stiamo valutando DAGs con circa 150-200 stati e milioni di percorsi di attraversamento possibili. La rappresentazione grafica non risulta essere di alcun aiuto come si evince chiaramente da fig.32

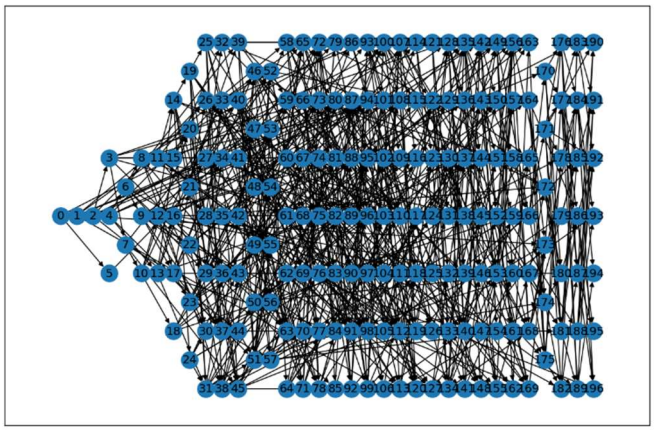


***Figura 32**: Generazione di grafo casuale.*

Quindi, una volta che lo script ha generato la policy del robot, ci si pone il problema di come verificare che sia effettivamente valida. Inoltre, viene generata una policy anche per l'umano, ma questa è importante solo nella misura in cui viene utilizzata per addestrare il robot a far fronte a situazioni complesse, quindi, non sarà oggetto di misura diretta: se riusciamo a dimostrare che la policy del robot è promettente allora vuol dire che anche quella imparata dall'umano è stata efficace.

Una volta calcolate le policy dell'umano e del robot, il processo di validazione proposto è il seguente:

1) Si considera la partita tra umano e robot dove entrambi utilizzano le policy appena apprese e si memorizza la lunghezza del path generato:

   $p_{\bar{r},\bar{h}}$: path generato con policy di uomo e robot fissate;

   $l_{\bar{r},\bar{h}}$ = length($p_{\bar{r},\bar{h}}$) : lunghezza di $p_{\bar{r},\bar{h}}$;

2) Si memorizza l'insieme di tutti i path generati $P_{\bar{r}}$ liberando l'umano dal vincolo di seguire la propria policy. Il robot continua a seguire la policy appresa;
3) Si verifica se è vera la condizione $l_{\bar{r},\bar{h}} \geq length(p)\ \forall\ p \in P_{\bar{r}}$;
4) Si collezionano tutti i possibili path del grafo e si fornisce un valore statistico di $p_{\bar{r},\bar{h}}$.

Consideriamo a titolo esemplificativo un grafo ancora relativamente piccolo che consente di avere una rappresentazione intuitiva della situazione passo-passo:

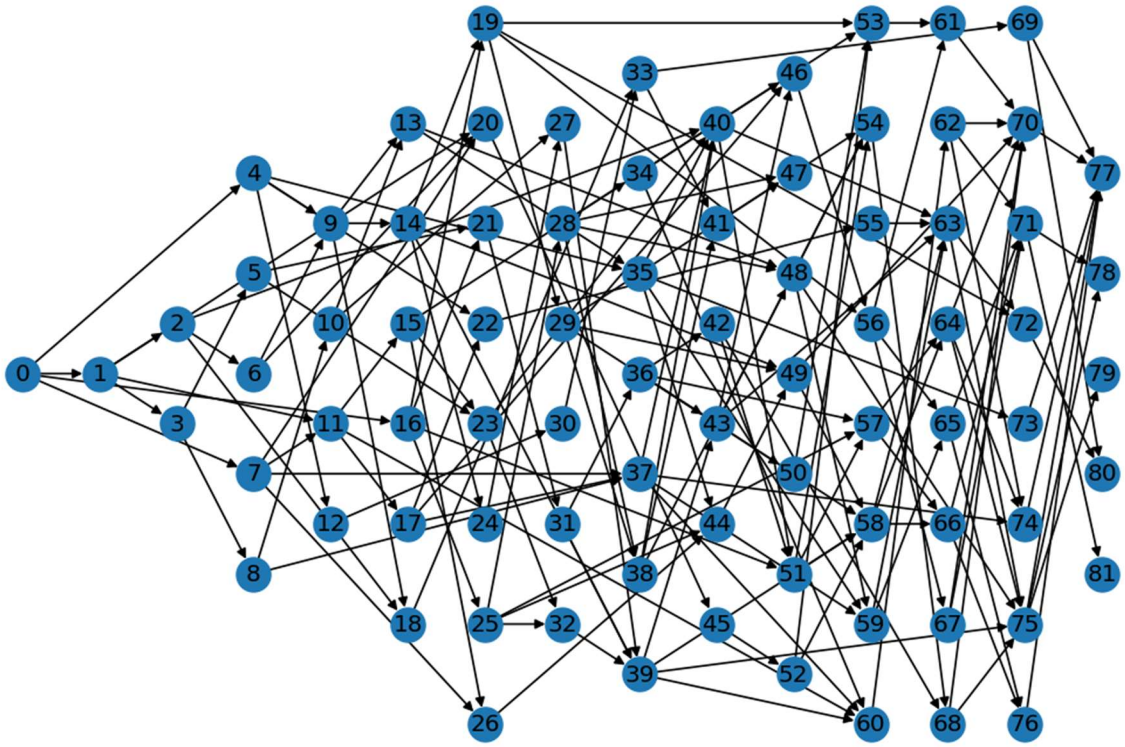


***Figura 33**: Grafo generato casualmente chiamando la funzione gen_graph(15, 4, 8)*

Si esegue lo script ottenendo le policies di robot e umano con i conseguenti valori $p_{\bar{r},\bar{h}}$ e $l_{\bar{r},\bar{h}}$:

***Tabella 12**: A sinistra la policy appresa dal robot e in rosso le azioni intraprese nella partita contro l'umano. A destra la policy appresa dall'umano e in verde le azioni scelte durante la partita contro il robot.*

STATE	NEXT
0	4
2	18
3	8
9	13
10	27
11	60
12	30
19	72
20	45
21	31
22	55
23	32
24	27
25	44
26	44
33	69
34	53
35	73
36	58
37	74
38	40
39	75
46	56
47	54
48	75
49	70
50	53
51	54
52	53
61	70
62	70
63	72
64	75
65	76
66	76
67	71
68	75

STATE	NEXT
1	3
4	12
5	21
6	20
7	20
8	10
13	35
14	24
15	34
16	21
17	46
18	33
27	39
28	38
29	38
30	33
31	36
32	39
40	51
41	47
42	51
43	46
44	49
45	52
53	61
54	68
55	63
56	67
57	64
58	63
59	65
60	62
69	77
70	77
71	78
72	80
73	77
74	77
75	79
76	77

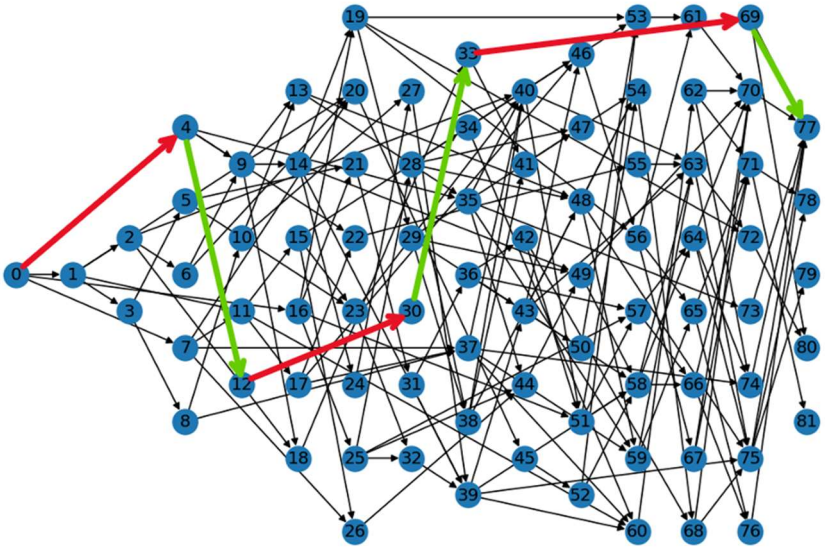


***Figura 34**: DAG con segnate le azioni durante la partita dettate dalle policy apprese: in rosso quelle del robot, in verde quelle dell'umano.*

$$p_{\bar{r},\bar{h}} = [0, 4, 12, 30, 33, 69, 77]$$

$$l_{\bar{r},\bar{h}} = lungth(p_{\bar{r},\bar{h}}) = 6$$

Vengono poi calcolati tutti i percorsi lasciando all'umano la libertà di tentare tutte le azioni possibili ottenendo l'insieme $P_{\bar{r}}$:

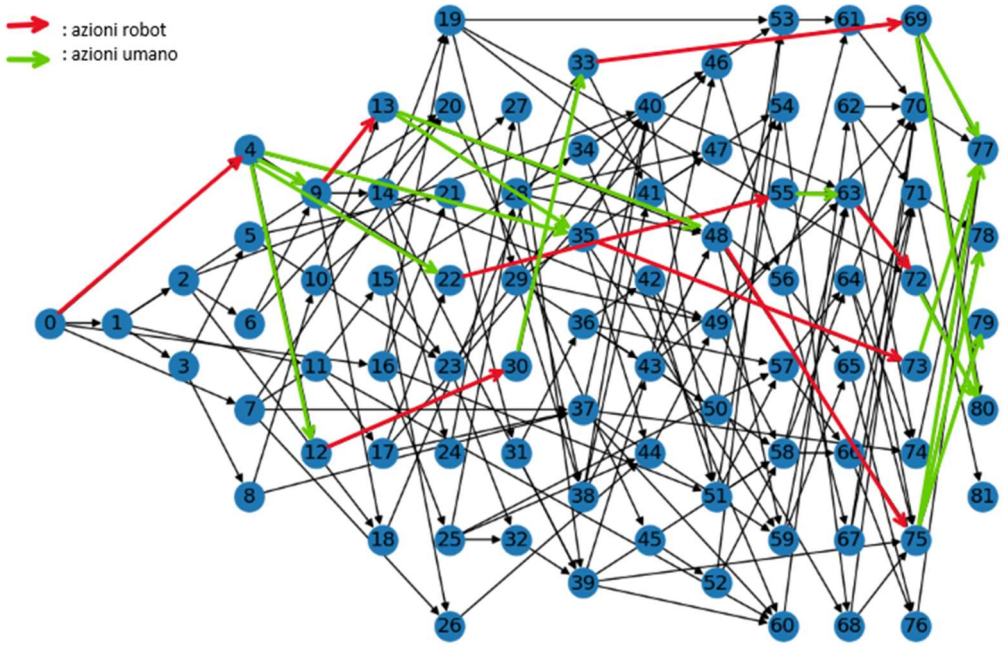


***Figura 35**: DAG con frecce rappresentanti tutte le possibili combinazioni di azioni tra robot e umano. In rosso le azioni effettuate dal robot, in verde le azioni dell'umano.*

Vediamo che l'insieme dei possibili cammini $P_{\bar{r}}$ è:

- P1: [0, 4, 12, 30, 33, 69, 77] , length(P1) = 6
- P2: [0, 4, 35, 73, 77], length(P2) = 4
- P2: [0, 4, 9, 13, 35, 73, 77], length(P3) = 6
- P3: [0, 4, 9, 13, 48, 75, 77], length(P4) = 6
- P4: [0, 4, 9, 13, 48, 75, 78], length(P5) = 6
- P5: [0, 4, 9, 13, 48, 75, 79], length(P6) = 6
- P6: [0, 4, 22, 55, 63, 72, 80], length(P7) = 6
- P7: [0, 4, 12, 30, 33, 69, 80], length(P8) = 6

Siccome $l_{\bar{r},\bar{h}} = 6,$ è soddisfatta la condizione $l_{\bar{r},\bar{h}} \geq length(p)\ \forall\ p \in P_{\bar{r}}$.

L'interpretazione che possiamo dare a questo risultato è che se il robot segue la policy appresa durante l'addestramento allora, qualsiasi sia il comportamento dell'umano, abbiamo la garanzia che l'attraversamento del grafo verrà completato al massimo in 6 step. Questo è un risultato estremamente incoraggiante poiché rende il processo di assemblaggio indipendente dall'imprevedibilità delle azioni umane.

La distribuzione della lunghezza dei percorsi è mostrata in figura 36. Su 1561 possibili cammini, 1445 hanno una lunghezza maggiore di 6; dunque $l_{\bar{r},\bar{h}}$ non solo rappresenta la garanzia di terminare il lavoro in 6 step, ma è anche un valore minore del 92,57 % del totale.

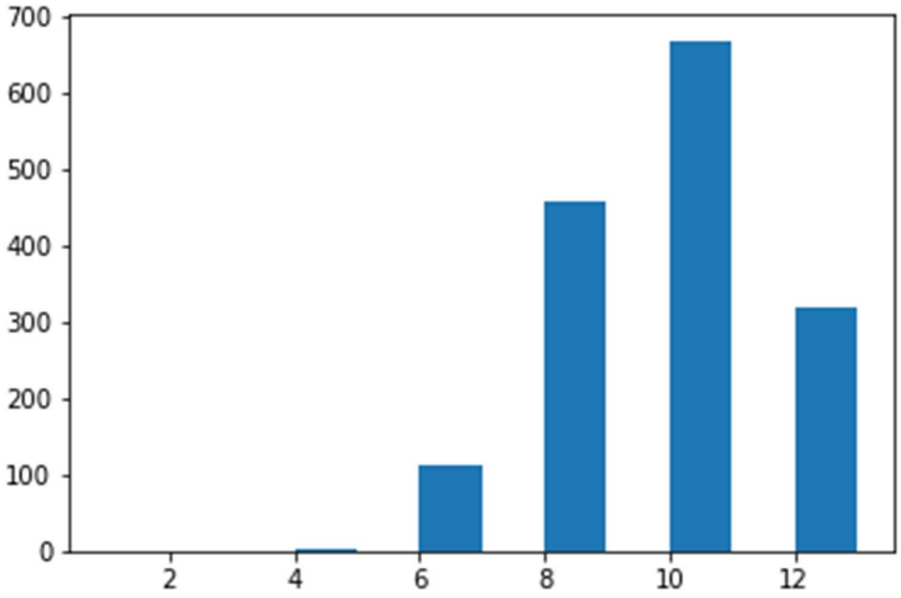


***Figura 36**: Istogramma con le frequenze della lunghezza dei percorsi per attraversare il grafo di fig.33.*

Vengono ora proposti gli esiti di altre 2 esecuzioni del programma che dovrà affrontare DAGs casuali generati chiamando la funzione gen_graph(35, 8, 7). Vengono riportati anche i grafici per fornire un'idea della caoticità e complessità dei grafi prodotti; non verranno fornite le tabelle contenenti la policy appresa.

3.7.1. Run 1.

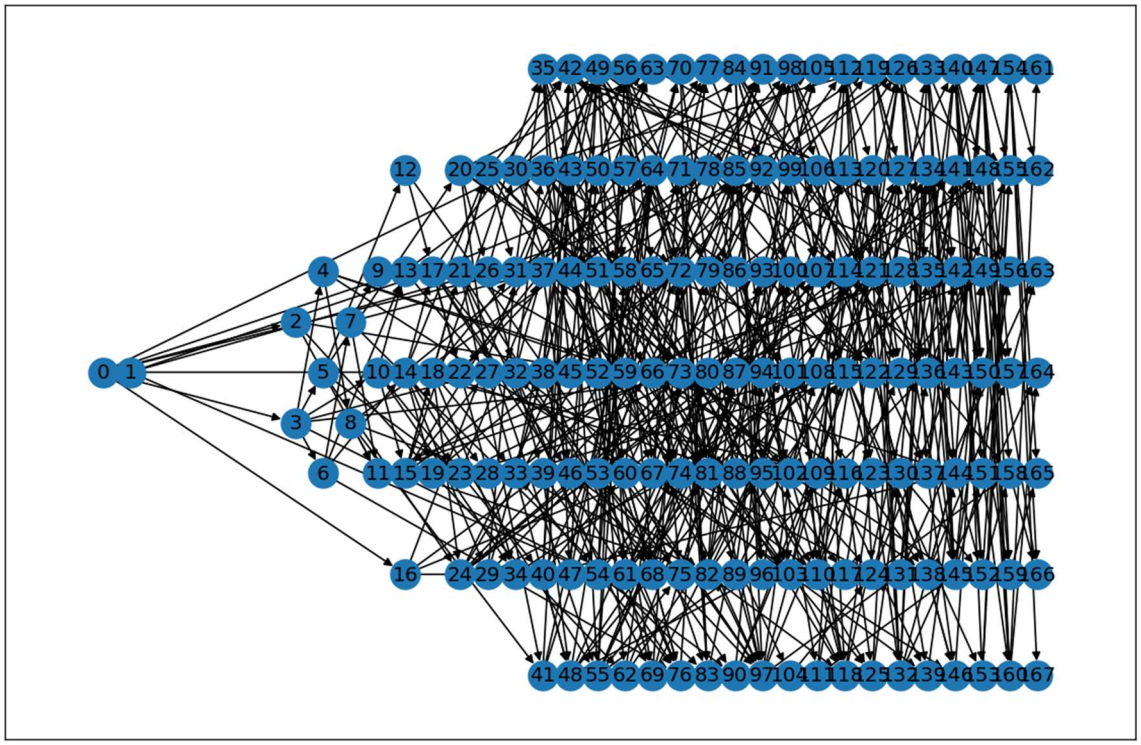


***Figura 37**: Primo DAG generato chiamando la funzione gen_graph(35, 8, 7).*

$$p_{\bar{r},\bar{h}} = [0, 31, 105, 112, 121, 132, 134, 156, 163]$$

$$l_{\bar{r},\bar{h}} = 8$$

$$count(P_{\bar{r}}) = 37$$

$$\max_{p}(length(p)) = 8 \quad con\ p \in P_{\bar{r}}$$

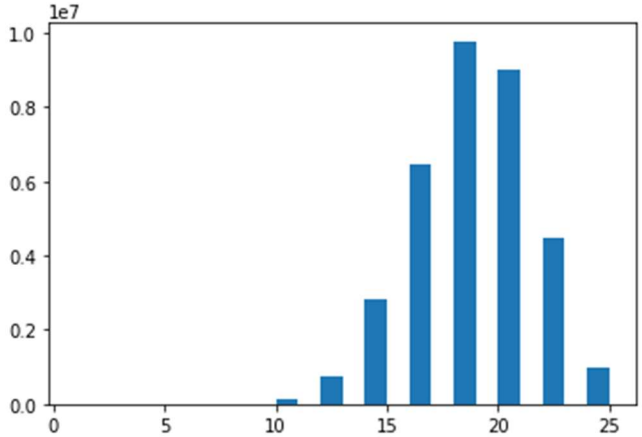


***Figura 38**: Istogramma con le frequenze della lunghezza dei percorsi per attraversare il grafo di fig.37.*

Anche se il cammino minimo ha lunghezza 4, la policy del robot riesce a garantire l'attraversamento del grafo in massimo 8 steps, che è comunque un percorso più rapido rispetto al 99.99968 % del totale.

3.7.2. Run 2.

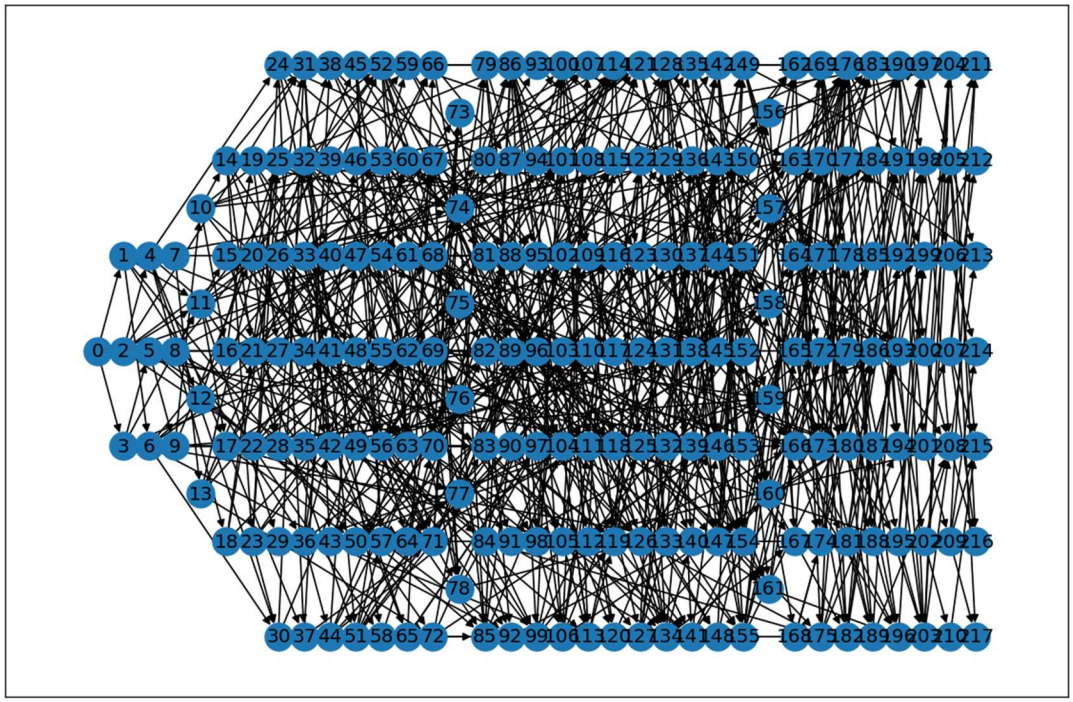


***Figura 39**: Secondo DAG generato chiamando la funzione gen_graph(35, 8, 7).*

$$p_{\bar{r},\bar{h}} = [0, 125, 132, 151, 172, 180, 189, 208, 215]$$

$$l_{\bar{r},\bar{h}} = 8$$

$$count(P_{\bar{r}}) = 60$$

$$\max_{p}(\ length(p)\ ) = \ 8 \quad con\ p \in \ P_{\bar{r}}$$

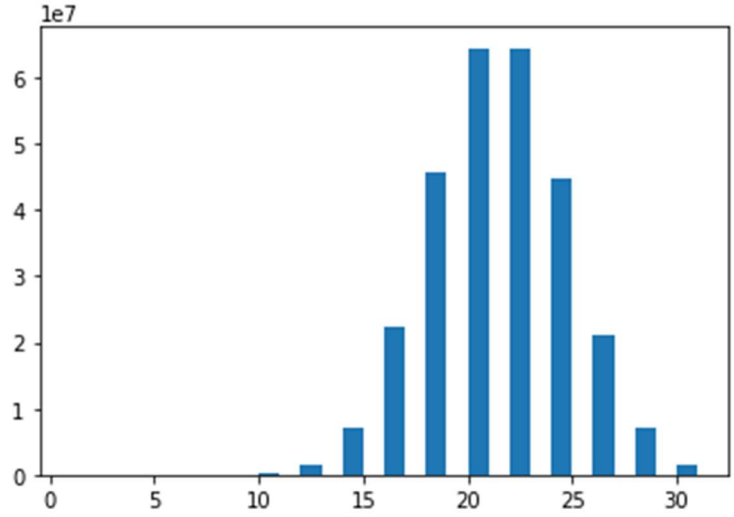


***Figura 40**: Istogramma con le frequenze della lunghezza dei percorsi per attraversare il grafo di fig.39.*

Anche in questo caso il robot garantisce l'attraversamento del grafo in massimo 8 step indipendentemente dalle azioni dell'umano. Il 99.99996 % dei cammini è più lungo di $l_{\bar{r},\bar{h}}$.

3.8. Considerazioni.

Ragionamenti: abbiamo visto che siamo riusciti a trovare delle policy del robot che consentono di garantire un massimo tempo di percorrenza del grafo. Questo numero è tipicamente maggiore della minima lunghezza di percorrenza del grafo, ma questo non è un problema: si ricordi in fig.21 che non stiamo cercando il percorso più breve (questo problema è stato già stato ampliamente dibattuto e risolto in letteratura[23]), ma stavamo cercando di limitare il più possibile l'influenza che l'imprevedibilità dell'uomo può avere sul tempo di percorrenza; ottenendo $l_{\bar{r},\bar{h}}$ ci siamo riusciti.

È importante notare che il tempo di calcolo del programma aumenta esponenzialmente al crescere del grafo; questo accade non tanto a causa dell'addestramento e della scoperta della policy ottimale (processo che comunque è adattabile con il tuning di alcuni parametri, quali numero di

[23] K. ALIEV, D. ANTONELLI, G. BRUNO, *Task-based Programming and Sequence Planning for Human-Robot Collaborative Assembly*, in «9th IFAC Conference on manufacturing modelling, management and control», MIM, Berlino, pp. 1638-1643, 2019.

episodi, epsilon, ecc.), quanto della funzione di validazione. Questa prevede l'esplorazione completa di tutti i possibili percorsi (decine/centinaia di milioni) per raccogliere le statistiche necessarie a valutare la policy del robot.

Conclusioni e lavori futuri

Nella trattazione dei primi due casi di studio è stato possibile trovare le policy del robot che gli permettessero di:

1) Conoscere il pattern più rapido per arrivare al termine del lavoro;
2) Correggere la sequenza di assemblaggio qualora l'agente umano forzi uno stato alternativo rispetto alla sequenza standard.

I risultati positivi ottenuti nei primi due casi di studio mettono in evidenza la potenzialità delle tecniche tradizionali di RL nell'ottica dell'HRC, ma si tratta pur sempre di esempi con limitato impatto concreto in quanto un reale processo di assemblaggio può essere composto da centinaia di possibili stati e abbiamo visto in [2.3] che l'addestramento inizia ad essere inefficace. Inoltre, l'intelligenza dei robot nei primi due casi si limitava a renderlo capace di trovare strade alternative quando viene depistato dall'uomo.

Il terzo caso di studio aveva un obiettivo senz'altro più ambizioso e ha anche restituito i risultati più interessanti: questa volta il robot non tenta più di percorrere a prescindere il percorso più breve. Le sue azioni sono volte a ridurre il più possibile l'impatto negativo che le scelte dell'uomo possono avere sul tempo di assemblaggio. Spesso viene sacrificato il percorso che in assoluto sarebbe il migliore, in favore di sequenze di assemblaggio più "sicure" e meno influenzabili dall'errore umano. Nel terzo caso è stato anche speso un notevole sforzo nel tentativo di creare delle situazioni realistiche e sufficientemente complesse da non renderne la soluzione un mero esercizio fine a sé stesso ma un effettivo candidato per una futura implementazione.

La fase di validazione è stata la più avida di potenza di calcolo: l'esplorazione di tutto lo spazio delle soluzioni richiede una quantità di risorse che cresce esponenzialmente in proporzione alla crescita del grafo. Nella presente ricerca si è scelto di accettare questo malus poiché era necessario avere la certezza che la policy appresa fosse efficace in tutti i casi possibili: solo in questo modo si può essere certi dell'efficacia dell'algoritmo. È necessario osservare che se si volesse testare

l'adversarial nel contesto di grafi più importanti rispetto a quelli considerati in 3.7.1 e 3.7.2 sarebbe impossibile dimostrare l'ottimalità della policy risultante in quanto sarebbe praticamente impossibile esplorare l'intero grafo.

L'algoritmo mostra risultati promettenti anche in termini di scalabilità: nei test eseguiti è sempre stata trovata la policy corretta con addestramenti di poche decine di secondi sia nei grafi più modesti (fig.21) che in quelli più complessi (fig.39). Qualora si riscontrasse un improvviso degrado nella qualità della policy in determinati scenari o a partire da una certa cardinalità dei DAGs, sarebbe possibile mantenere il core dell'algoritmo proposto apportando piccole modifiche per migliorare l'esplorazione dello spazio delle soluzioni o aumentarne la velocità di convergenza: ad esempio si può prendere in considerazione di implementare una variante di sarsa chiamato λ-sarsa[24].

Una possibile ricerca futura potrebbe porsi l'obiettivo di arricchire il modello del DAG aggiungendo livelli di difficoltà/tempo impiegato/risorse necessarie per descrivere ciascuna azione, in modo tale che il robot possa scegliere le future mosse anche in funzione di essi. Bisogna infatti considerare che ogni azione può avere un costo diverso in termini di tempo e destrezza necessaria per essere portata a termine. Lo studio presentato non tiene in considerazione di tali aspetti e si limita a considerare come parametro di misura il numero di azioni necessarie per terminare l'assemblaggio (il reward è sempre uguale per ogni azione).

In analogia a *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learning*[25], una futura implementazione di un reale assemblaggio in ambiente sperimentale consentirebbe di mettere alla prova i reali benefici dell'addestramento del robot.

[24] DAVID SILVER*, Lecture 5:*, in «Introduction To Reinforcement Learning with David Silver», 2015.
[25] DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learning*, «Science Direct», Vol. 8, 2022.

Bibliografia

K. ALIEV, D. ANTONELLI, G. BRUNO, *Task-based Programming and Sequence Planning for Human-Robot Collaborative Assembly*, in «9th IFAC Conference on manufacturing modelling, management and control», MIM, Berlino, 2019.

DARIO ANTONELLI, KHURSHID ALIEV, *Robust assembly sequence generation in a Human-Robot collaborative workcell by Reinforcement Learnin*g, «Science Direct», Vol. 8, 2022.

MARCO BUTTU, *Python, Guida completa*, Milano, LSWR edizioni, 2020.

MAURIZIO GIACCI, *Appunti Laboratorio di Algoritmi e Strutture Dati*.

ISO/TS 15066:2016 Robots and robotic devices – Collaborative Robots.

YUXI LIU, *PyThorch 1.x, Reinforcement Learning, Cookbook*, Birmingha, Packt, 2019.

ELOISE MATHESON, RICCARDO MINTO, EMANUELE G. G. ZAMPIERI, MAURIZIO FACCIO, GIULIO ROSATI, *Human–Robot Collaboration in Manufacturing Applications: A Review*, «MDPI Journal Robotics», 2019.

RAINER MÜLLER, MATTHIAS VETTE, AARON GEENEN, *Skill-based Dynamic Task Allocation in Human-Robot-Cooperation with the Example of Welding Application*, «Procedia Manufacturing», Vol. 11, 2017.

https://pytorch.org/ consultato in data 16 Ottobre 2022.

DAVID SILVER, «Introduction To Reinforcement Learning with David Silver», 2015, https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver

K. THULASIRAMAN, M. N. S. SWAMY, *Graphs: Theory and Algorithms*, John Wiley and Son, 1992.

WILLIAM T. B UTHER, MANUELA M. VELOSO, *Adversarial Reinforcement Learning, Gennaio* 2003.

ALES VYSOCKY, PETR NOVAK, *Human – robot collaboration in industry*, «MM Science Journal», giugno 2016.