Università degli studi di
Milano-Bicocca

Decision Models

Progetto Finale

Il problema del labirinto: implementazione di algoritmi tradizionali e ibridi

Autori:
Lorenzo Famiglini - 838675 - lorenzofamiglini@gmail.com
Giorgio Bini - 838674 - g.bini@campus.unimib.it

15 luglio, 2019

Abstract

Il progetto si pone l'obiettivo di affrontare il problema del labirinto attraverso l'implementazione di diversi metodi. Per alcuni algoritmi ci siamo basati sulla loro formulazione classica, come nel caso del Reinforcement Learning, mentre per altri, tra cui l'algoritmo Genetico, l'Ant Colony Optimization e l'algoritmo di Dijkstra, abbiamo implementato delle modifiche che hanno notevolmente migliorato i risultati. In particolare dimostreremo come, nonostante siano state eseguite varie modifiche volte a migliorare la performance dell'algoritmo genetico, quest'ultimo presenta molti limiti all'aumentare della dimensione di una matrice. Per questo motivo sono stati implementati algoritmi di natura diversa che hanno registrato tempi di convergenza alla risoluzione del problema decisamente migliori. Verranno presentate le tecniche adottate per migliorare le prestazioni dei vari algoritmi e infine dimostreremo quali sono quelli più adatti per risolvere un problema di questo tipo sia per quanto riguarda la ricerca del percorso ottimale, sia a livello computazionale.

1 Introduzione

Lo studio, nasce dalla volontà di voler creare degli algoritmi che permettono ad un agente di muoversi in maniera ottimale in un ambiente ricco di ostacoli da evitare e vicoli ciechi senza una via di uscita. Al giorno d'oggi, gli agenti autonomi hanno una vasta applicazione in diversi campi: da semplici strumenti di pulizia all'interno di una casa fino ad essere impiegati in territori di guerra per esaminare il suolo ed evitare le mine. Questo problema è stato affrontato in modo astratto attraverso l'utilizzo di un labirinto, essendo un ambiente in cui è molto difficile orientarsi. Infatti, in letteratura, l'essere intrappolati in un labirinto viene identificata come una situazione "critica", dove una volta entrati non si esce più. Ma grazie all'utilizzo di algoritmi ottimizzati per tale problema, il labirinto diventa una semplice via da percorrere in modo efficiente e rapido. Di seguito verrà descritto, in maniera generale, il funzionamento degli algoritmi tradizionali ai quali ci siamo ispirati, mentre nel capitolo 3 verranno discusse le modifiche adottate a tali metodi classici.

1.1 Metaeuristica: Algoritmo Genetico

Questo algoritmo si basa su un processo biologico: infatti si inizia con la creazione di una popolazione casuale di genitori, dove attraverso un processo di "accoppiamento", nascono dei figli che preservano caratteristiche comuni ad entrambi i genitori, a meno di qualche eventuale cambiamento (mutazione). All'aumentare delle generazioni, le soluzioni trovate per risolvere il problema (rappresentate da cromosomi) migliorano di volta in volta. Questo succede attraverso i seguenti step. Per prima cosa ad ogni iterazione viene associata una probabilità di selezione alle sequenze generate. Questa probabilità, a sua volta, dipende dal valore che tale soluzione assume nella fitness function. Successivamente, con il metodo roulette wheel selection, che consiste in un campionamento casuale semplice con ripetizione, vengono scelti i cromosomi che poi effettueranno la fase di crossover. Dopo tale fase, con una probabilità definita dall'utente, tali sequenze subiranno una mutazione genetica. Tale processo viene ripetuto per n-generazioni fino a creare il cromosoma ottimale. Questo algoritmo si basa sul principio dell'evoluzione darwiniana, per il quale, con il passare delle generazioni, solo i più forti sopravvivono. [4].

1.2 Algoritmo di Dijkstra

Esso è definito come l'algoritmo dei cammini minimi. Si basa sull'utilizzo di strutture a grafo e l'algoritmo si muove su vari nodi che vengono etichettati come: visitati V, di frontiera F, e da visitare (sconosciuti). Sostanzialmente, possiamo riassumere tale metodo, nel seguente modo: dai nodi di frontiera si sceglie un nodo z con distanza minima dal punto di partenza. Perciò z viene identificato come nodo visitato, $F \rightarrow V$. La ricerca prosegue fino a quando non verranno visitati tutti i nodi e ad ogni iterazione si aggiornano i pesi w e un valore dw che misura la distanza dal nodo di partenza. Il percorso ottimale sarà definito dai nodi che hanno un peso minore [3].

1.3 Metaeuristica: Ant Colony Optimization

Così come il GA, anche questa metaeuristica prende spunto dalla biologia. Essa infatti si basa sull'emulare il comportamento delle formiche quando sono alla ricerca del percorso più breve tra il formicaio e il cibo. In natura, le formiche sono praticamente cieche, quindi per seguire un determinato per-

corso rilasciano una sostanza chiamata feromone. Il percorso migliore è quello che presenta la più alta quantità di feromone. In sostanza, le formiche sono identificate come degli agenti, che devono esplorare un percorso, e sono dotate di una memoria attraverso il supporto di una matrice dove i valori di ciascuna cella (la quantità di feromone) vengono aggiornati in modo iterativo. Ad ogni iterazione inoltre tali valori della matrice decrescono a causa di un fenomeno chiamato evaporazione del feromone. Alla fine del processo, avremo le informazioni per seguire il tracciato migliore. [5].

1.4 Reinforcement Learning: Q-Learning e SARSA

Il Reinforcement Learning è un approccio non supervisionato (o quasi), che si basa su l'interazione tra un'agente e l'ambiente in cui si muove. L'agente, non ha nessuna informazione pregressa su come muoversi nell'ambiente, ma lo impara con il tempo in quanto, a seconda delle azioni che decide di intraprendere, riceve una ricompensa o una penalità. Gli elementi principali da definire in un problema di Reinforcement Learning sono: agente, ambiente, azioni, stati, policy, ricompense e penalità. La policy definisce il comportamento che deve seguire l'agente in un determinato stato. In altre parole, permette di indicare quali sono le azioni ottimali da intraprendere per ogni stato dell'ambiente. Nei problemi in cui l'ambiente può essere rappresentato attraverso una matrice, essa assume il nome di Grid World. In queste situazioni, la policy viene rappresentata quantitativamente attraverso una Q-table che ha dimensioni pari a: [numero di righe, numero colonne della Grid World, numero di azioni che può compiere l'agente]. Due algoritmi ampiamente utilizzati per risolvere questo tipo di problemi sono Q-Learning (1) e SARSA (2). Essi permettono di aggiornare ogni valore della Q-table attraverso le seguenti equazioni:

$$Q(s^t, a^t) = Q(s^t, a^t) + \alpha \times [r(s^t, a^t) + \gamma \times max_{a^{t+1}} Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t)] \quad (1)$$

$$Q(s^t, a^t) = Q(s^t, a^t) + \alpha \times [r(s^t, a^t) + \gamma \times Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t)] \quad (2)$$

Dove γ è il fattore di sconto che varia tra [0,1], α lo possiamo identificare come il learning rate, r è la ricompensa che l'agente riceve compiendo l'azione a al tempo t nello stato s.

2 L'architettura del labirinto

Il labirinto (o maze) si basa su una struttura complessa con una sequenza formata da percorsi interconnessi. L'obiettivo è quello di collegare un punto di inizio ad un punto di fine, separandoli da percorsi più brevi o più lunghi che possono portare anche in vicoli ciechi. La creazione del labirinto si basa sul metodo di Prim. Esso è un algoritmo greedy, a grafo, utilizzato per trovare i Minimum Spanning Trees (MST). All'inizio si collegano i nodi che hanno distanza minima e attraverso un processo iterativo i nodi vengono incorporati sequenzialmente nell'albero. L'algoritmo termina quando tutti i nodi sono collegati tra loro. [1].

3 Approccio metodologico

In questa sezione, andremo a descrivere i vari algoritmi che sono stati costruiti sulla base di diverse metodologie:

3.1 Maze Genetic Algorithm

• Fitness Function:
In primo luogo, è stata definita la fitness function come il rapporto tra [2]:

$$F = \frac{CC - SC}{FC - CC} \times 100$$

dove CC è la cella corrente, SC è la cella di partenza e FC è quella di arrivo. Come misura tra le coppie, è stata utilizzata la distanza euclidea, perciò l'obiettivo all'interno dell'algoritmo è quello di massimizzare tale rapporto affinché un punto, dopo una determinata azione, si trova più vicino alla cella di arrivo e molto lontano dalla cella di partenza.
• Random Population:
Partendo da una sequenza di azioni: ["N","S","E","W"], Nord, Sud, Est e Ovest, viene generata la prima popolazione di genitori con una sequenza pari al minimo della lunghezza tra la base e l'altezza del labirinto diviso 2. Per dimensioni abbastanza ridotte il numero ottimale di partenza è di 6 cromosomi.
• Moving Function:
Questa funzione è stata essenziale per poter calcolare, dato un punto di partenza, il punto di arrivo dopo una sequenza di azioni. All'interno di essa, è stata inserita una condizione che se, data una direzione, si va incontro ad un muro, la cella corrente è uguale alla cella di arrivo.
• Score Function:
In questa fase, viene calcolata la fitness di ciascun cromosoma, per poi stimare un valore che varia tra 0 e 1, in modo tale da ottenere una probabilità di selezione, data come il rapporto tra l'i-esimo valore della fitness e la somma totale dei valori della funzione per ogni possibile soluzione ottenuta.

• Crossover:
È la fase più significativa di un algoritmo genetico. Per ogni coppia di genitori da associare viene scelto a caso un punto di crossover all'interno dei geni. La strategia per la ricerca del punto di break dei cromosomi è stata quella di scegliere il valore, tra la coppia di genitori, della sequenza più corta e dividere per due (scegliendo la parte intera). In questo modo, abbiamo una divisione più "equilibrata" dei cromosomi. La scelta delle coppie si basa su un campionamento delle sequenze ottenute dalla popolazione con probabilità di selezione, calcolata grazie alla score function. Successivamente, il numero di "geni" delle sequenze viene incrementato ad ogni iterazione, ottenendo vettori di lunghezza differente al crescere delle generazioni. È stato inserito anche il concetto di elitismo, tenendo i due miglior cromosomi che massimizzano la una funzione obiettivo (prima che avvenga il crossover). I cromosomi selezionati dall'elitismo, al contrario degli altri, non subiscono alcun cambiamento nella fase del crossover. Grazie a questa strategia, risulta molto più difficile cadere in punti di minimo locale, poichè con l'aumentare delle generazioni abbiamo sequenze di lunghezza differente che producono differenti "itinerari" per raggiungere l'uscita. Esempio di offspring tra due cromosomi dopo questa fase:

N	S	E	E	W	N

S	E	E	N	N	N	E

• Mutazione:
Dopo il crossover, ogni gene con una probabilità del 10% può essere soggetto ad una mutazione. Ciò implica che alcune azioni della sequenza possono essere invertite. Viene di seguito fornito un esempio che chiarisce il funzionamento della mutazione.

N	S	E	E	W	N
N	S	E	N	E	W

Alla fine delle n-iterazioni, il GA propone una soluzione al nostro problema. Tale soluzione, tuttavia, potrebbe includere azioni che fanno scontrare l'agente contro il muro. Per questo, alla fine dell'algoritmo, è stata implementata una funzione che elimina tali azioni e in questo modo si ottengono sequenze più pulite.

3.2 Hybrid Ant Colony With Backtracking

La formulazione di questo algoritmo, si basa in buona parte sull'implementazione del Fuzzy Counter Ant Algorithm for Maze Problem [6]. Gli "agenti" che devono esplorare il labirinto sono rappresentati metaforicamente dalle formiche. Il labirinto viene rappresentato inizialmente come una matrice composta da

Nan (che identificano i muri) e valori nulli. Questo perché le celle della matrice con valore nullo rappresentano le posizioni del labirinto non ancora esplorate. Se una formica passa sopra una cella con valore pari a zero, essa rilascia del feromone e il valore di tale cella viene trasformato in 1. Differentemente dal tradizionale Ant Colony Optimization Algorithm (in cui le formiche prediligono le direzioni in cui vi è massima concentrazione di feromoni) in questa implementazione ogni volta che l'agente deve prendere una decisione, la preferenza è sempre data alla direzione che permette di raggiungere una cella che ha la quantità minore di feromone. Perciò se l'agente deve scegliere se andare verso una cella con valore pari a 1 oppure se andare in una cella con valore nullo, sceglierà di muoversi verso quest'ultima e in questo modo si garantisce una rapida esplorazione dell'ambiente[1]. Un'importante implementazione svolta per la risoluzione del problema consiste nell'aver effettuato una mappatura dei nodi all'interno del labirinto. Essi identificano una cella del labirinto in cui l'agente deve prendere una decisione "non banale". In altre parole, se una formica si trova davanti ad una scelta decisionale in cui almeno due direzioni la portano in posizioni non ancora esplorate (con valore nullo nella matrice), allora tale cella viene classificata come nodo. Se un agente passa sopra ad un nodo, il valore di tale cella viene trasformato in 999. Inoltre, quando una formica trova un nodo, allora automaticamente vengono create tante formiche quante sono le possibili direzioni che portano a valori nulli della matrice. Ad esempio, se in un nodo, ci sono tre direzioni che portano in celle non ancora esplorate, allora verranno create tre formiche che prenderanno le tre possibili direzioni in questione, per esplorare il labirinto partendo da quel nodo. Nella nostra formulazione, in un labirinto con più formiche, dopo ogni iterazione ciascuna formica avrà effettuato una singola azione. Questa implementazione ha molteplici vantaggi: in primo luogo garantisce che le formiche non si incrocino mai nei loro tragitti, e che quindi ognuna esplori l'ambiente nella porzione di labirinto assegnata da tale formulazione. Infatti, dal momento in cui i nodi vengono classificati da un valore pari a 999, le formiche non ritorneranno mai verso nodi già esplorati in quanto, ricordiamo, la loro scelta privilegia sempre la direzione che permette di arrivare nella cella adiacente con il valore minimo. In secondo luogo, questa implementazione permette di accorciare i tempi di risoluzione del problema perché ad ogni iterazione, ci sono più agenti che simultaneamente esplorano l'ambiente. Con tale formulazione, tuttavia, si potrebbe incorrere nel fenomeno della stagnazione. La condizione di stagnazione è un caso comune che può verificarsi quando tutti i percorsi che circondano l'agente, sono già stati esplorati. La concentrazione di feromoni in tutti questi percorsi sarà uguale, perciò l'agente non sa quale percorso prendere. Questa problematica viene risolta inserendo nell'algoritmo il fenomeno di evaporazione dei feromoni, grazie al quale la concentrazione di feromoni diminuisce ad

[1]Le successive implementazioni dell'algoritmo che verranno descritte si differenziano da quelle scelte dagli autori del paper citato, in quanto sono state frutto di una nostra interpretazione per la risoluzione del problema.

ogni iterazione. In altre parole, tutti i valori delle celle del labirinto vengono moltiplicati per 0.99 ad ogni step. L'importanza di questa operazione risiede soprattutto nell'offrire la possibilità di risalire al percorso ottimale per risolvere il labirinto. Infatti, il percorso ottimale sarà dettato dalla formica che per prima avrà raggiunto l'uscita. Per risalire al percorso ottimale è stata adottata una tecnica di "Back-Tracking" attraverso la quale, partendo dalla cella di uscita del labirinto, si procede a ritroso nella direzione in cui è minima la quantità di feromone. In questa fase, i nodi assumono il valore pari al valore minimo delle celle ad esso adiacenti. Così facendo, l'agente riesce a risalire alla cella di partenza, e salvando in una lista le azioni che ha effettuato, si può facilmente risalire al percorso migliore. Questa strategia permette di ottenere sempre e soltanto un'unica soluzione al problema, ovvero quella ottimale. Inoltre, è anche molto efficiente in quanto non c'è neanche bisogno di esplorare tutte le celle della matrice: per ottenere la soluzione ottimale basta che si arrivi alla cella finale.

3.3 Hybrid Dijkstra With Ant Colony Exploration

Dal momento che, nella formulazione dell'algoritmo precedentemente descritto, sono stati utilizzati dei nodi, abbiamo deciso di implementare anche un algoritmo di Dijkstra per trovare il percorso che permette di arrivare alla soluzione ottimale. L'implementazione dell'algoritmo, in una prima fase, è abbastanza simile a quella precedente, a meno di alcune differenze. Le celle della matrice vengono sempre inizializzate a zero prima dell'addestramento e la quantità di feromoni rilasciati dalle formiche è sempre pari a 1, nel caso in cui si percorrono celle che non erano ancora state esplorate, e pari a 999, nel caso in cui l'agente si posiziona in un nodo. Anche in questo caso, inoltre, la scelta dell'agente è sempre quella del percorso che porta alla cella con valore minimo. Tuttavia, in questo algoritmo, è necessario salvare la lunghezza dei percorsi che collegano i vari nodi (che ricordiamo, sono definiti come le celle in cui l'agente deve decidere tra varie opzioni che lo porterebbero in posizioni non ancora esplorate). Per far ciò, è stato tenuto conto del "numero di nodi visitati" da ogni formica. Tale informazione è stata salvata in un dizionario, insieme anche al numero di azioni che l'agente ha compiuto. Quest'ultimo valore incrementa di 1 ad ogni iterazione fino a che la formica non raggiunge il secondo nodo visitato. Non appena l'agente raggiunge tale cella, viene salvata l'informazione della distanza tra i due nodi. Inoltre, le celle di partenza e quelle di arrivo sono in questo caso classificate come nodi. Anche questo processo di esplorazione è molto efficiente perché si devono esplorare soltanto una volta tutte le celle della matrice per arrivare ad una soluzione. Alla fine del processo di esplorazione si ha una mappa completa delle distanze tra i nodi del labirinto e si può implementare l'algoritmo di Dijkstra per trovare il percorso che collega il nodo di partenza da quello di arrivo. Grazie a questa formulazione del problema, ci è possibile trovare il percorso più breve per uscire dal labirinto partendo da ciascun nodo individuato. Se volessimo essere a conoscenza del percorso più breve per uscire dal labirinto partendo da una

qualsiasi delle celle nella matrice, invece, c'è bisogno di un altro approccio. In questo caso, infatti, occorre utilizzare il Reinforcement Learning, che è stato un'ulteriore strumento di analisi in questo progetto.

3.4 Reinforcement Learing: Q-learning e SARSA

Per utilizzare le tecniche di Reinforcement Learning è stata implementata una matrice (Grid World) di dimensioni pari a quelle del labirinto, che rappresenta l'ambiente entro il quale si deve muovere l'agente. Anche in questo caso i muri vengono identificati dai Nan. L'epoca in questo problema è definita come una singola mossa dell'agente mentre l'episodio è l'insieme delle epoche che l'agente compie partendo dalla cella di partenza fino a quella di arrivo. Il numero di episodi necessari all'allenamento, ovviamente, cresce all'aumentare della dimensione del labirinto. Lo stato, invece, è rappresentato dalla posizione dell'agente. Esso può decidere di compiere quattro azioni: muoversi a nord, a sud, a est oppure a ovest.
Per quanto riguarda la reward, essa è stata fissata a 10 e la si ottiene soltanto uscendo dal labirinto. Inoltre abbiamo deciso di penalizzare l'agente ad ogni passo effettuato, così da garantire una convergenza verso la soluzione ottimale in tempi più rapidi. Inoltre, inserendo una penalità si evita che l'agente faccia delle mosse inutili prima di arrivare all'uscita del labirinto. Perciò, ad ogni passo effettuato, l'agente riceve una reward negativa pari a -0.1. Per ogni epoca il valore di un'azione, dato un determinato stato, viene aggiornato con le equazioni del Q-Learning e di SARSA. Azioni che portano l'agente verso l'uscita del labirinto verranno premiate mentre le azioni che lo portano in vicoli ciechi verranno penalizzate. Per quanto riguarda la scelta dei parametri, è stato fissato γ a 0.9, mentre ϵ e α sono stati ottimizzati attraverso una procedura iterativa di ricerca dei parametri. In questo caso abbiamo ottimizzato α e ϵ con lo scopo di trovare i valori che permettono la risoluzione del labirinto nel minor tempo possibile. In particolare, per matrici piccole (come la 20x20), abbiamo ricercato entrambi i parametri nell'intero range [0, 1], con un incremento pari a 0.1 ad ogni iterazione. Invece per matrici più grandi abbiamo ristretto lo spazio della ricerca per eseguire le iterazioni in tempi ragionevoli. In particolare, abbiamo cercato α nello spazio [0.4, 0.9] e ϵ nello spazio [0.1, 0.8].

4 Risultati

In questo capitolo verranno presentati i risultati relativi all'ottimizzazione dei parametri per il Reinforcement Learning e successivamente, verranno discusse le performance dei vari algoritmi. Vista la notevole differenza dei tempi di convergenza tra i vari algoritmi, è stato deciso di strutturare l'analisi in due diverse sezioni. Una dedicata al GA, ed una relativa agli altri. Tutti gli algoritmi sono stati processati su un Dell XPS, Intel Core^{TM} i7-8750H 2.220 GHZ, 16 GB RAM.

4.1 Ottimizzazione del Reinforcement Learning

Per quanto riguarda l'ottimizzazione nel Reinforcement Learning, i parametri ottimali nell'80% dei casi, per matrici di dimensioni 20x20, sono risultati i seguenti:

- $\epsilon = 0.1$, $\alpha = 0.8$ per quanto riguarda Q-Learning
- $\epsilon = 0.1$, $\alpha = 0.7$ per quanto riguarda SARSA

Anche per matrici più grandi, nel 75% dei casi i parametri ottimali per Q-Learning si sono confermati quelli appena elencati. Invece l'algoritmo SARSA è risultato fortemente inefficace per labirinti di medie e grandi dimensioni e per questo motivo non verranno presentate in questo capitolo le sue performance in termini di tempo di esecuzione se non per labirinti 20 x 20. I motivi di tale inefficacia verranno descritti ampiamente nel prossimo capitolo.

4.2 Maze GA

In Tabella 1 abbiamo i risultati relativi al tempo medio di convergenza di due differenti versioni dell'algoritmo genetico. I valori sono stati calcolati effettuando 10 prove. La differenza sostanziale tra i due metodi consiste nel fatto che in quello "probabilistico", il gene viene aggiunto alla fine del cromosoma con una certa probabilità (che in questo caso è stata fissata pari a 0.5). Invece in quello "migliorato" l'incremento di un gene alla fine della fase di cross-over avviene costantemente in tutte le generazioni.

GA	Maze	Tempo medio	Dev.Std.	N. iterazioni
Probabilistico	10x10	27s	0.9	80
Migliorato	10x10	3.06s	1.8	30
Probabilistico	15x15	1min, 26s	3.12	150
Migliorato	15x15	10.65s	2.21	50
Probabilistico	20x20	3min, 02s	3.82	200
Migliorato	20x20	48s	3.03	80

Tabella 1. Tempi medi di convergenza

In Figura 1 vengono proposte due soluzioni per un labirinto 20 x 20 risolto dal GA migliorato. A sinistra vi è un percorso semi-ottimale, in cui vi sono due azioni superflue (est-ovest) in corrispondenza della colonna 16. Il percorso viene di conseguenza allungato da 36 a 38. La seconda figura, invece, rappresenta una soluzione di lunghezza 37 che invece non include mosse superflue. Questo esempio è molto significativo in quanto mostra i limiti del GA, che pur trovando buone soluzioni al problema, non assicura di ottenere quella ottimale.

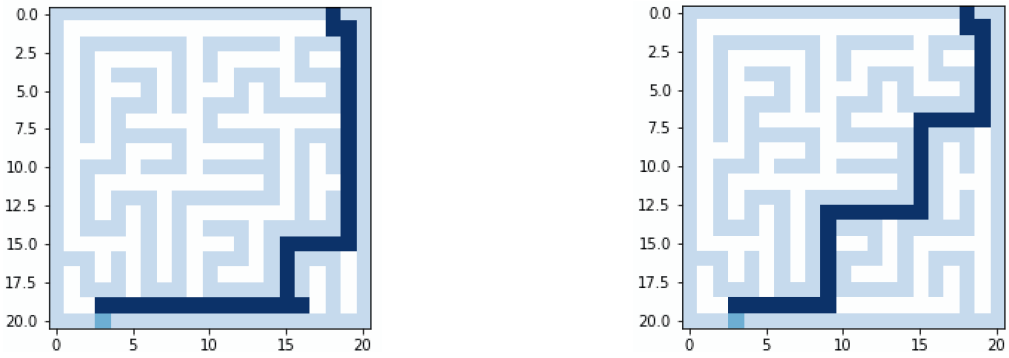

Figura 1. GA: Due soluzioni per Maze 20x20

4.3 Valutazione del tempo medio di convergenza:

Per valutare le performance dei vari algoritmi è stato scelto il tempo di convergenza medio come metrica per il confronto. Verranno confrontate le performance per labirinti di varie dimensioni: 20x20, 80x80, 200x200. Per ciascuna di queste matrici, e per ogni algoritmo, sono stati raccolti 10 tempi di risoluzione del problema. Ne è stata calcolata una media e un intervallo di confidenza al 95% per fornire anche una misura della variabilità delle performance. Vengono di seguito forniti i tempi medi di convergenza alla soluzione ottimale per i vari algoritmi. Nel caso della 20x20 è stata scelta una rappresentazione grafica, mentre per le altre grandezze abbiamo optato per una forma tabulare. Si ricorda che per labirinti di medie e grandi dimensioni non è stato registrato il tempo di SARSA e GA, che sono risultati inefficienti.

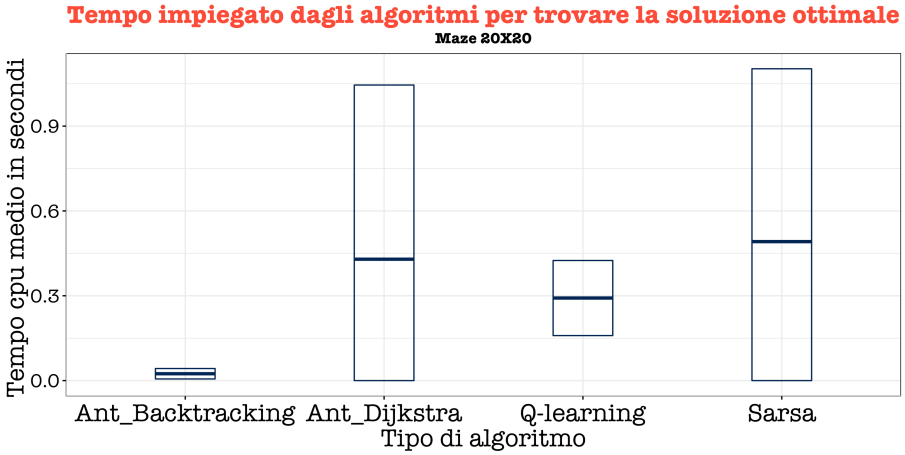

Figura 2. Intervallo di confidenza del tempo impiegato per convergere

Algoritmo	Maze	Tempo Medio	Dev. std
Ant_Backtracking	80x80	1.49s	0.43
Ant_Dijkstra	80x80	4.51s	1.68
Q-Learning	80x80	21.65s	3.64
Ant_Backtracking	200x200	33.1s	7.5
Ant_Dijkstra	200x200	35.6s	3.58
Q-Learning	200x200	13min, 1s	5.15

Tabella 2. Tempo medio per la convergenza e deviazione standard

Abbiamo inoltre deciso di risolvere un labirinto di dimensioni 500 x 500 con l'Hybrid Ant Colony With Backtracking e il risultato è stato sorprendente. ci sono voluti appena 13 minuti. Tuttavia, dal momento che, per rappresentare graficamente un problema così grande servirebbe molto più spazio, viene di seguito proposta una visualizzazione che si riferisce ad un labirinto di dimensioni pari a 200 x 200.

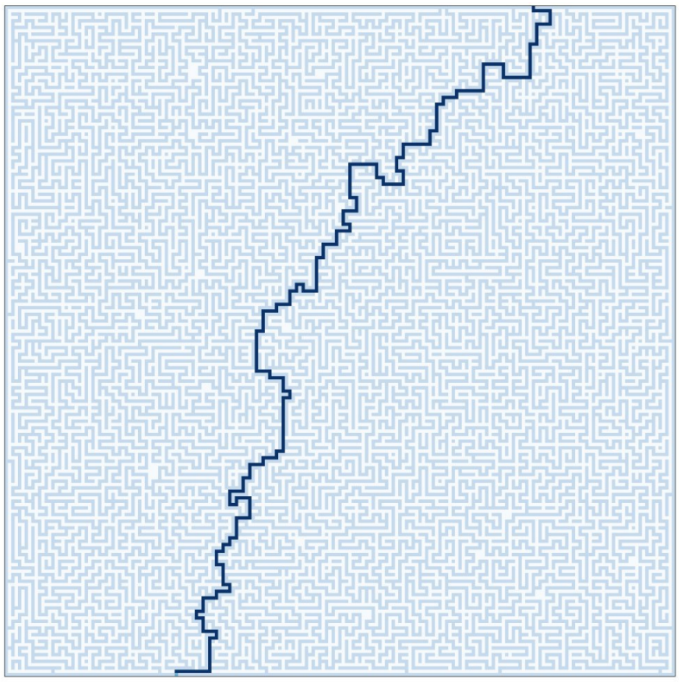

Figura 3. Hybrid Ant Colony With Backtracking: Maze 200x200

5 Discussione

5.1 Algoritmo Genetico

Per la metaeuristica GA, sono stati analizzati e costruiti diversi metodi per risolvere il problema del labirinto: una strategia iniziale è stata quella di aumentare in modo unitario le sequenze del crossover con una probabilità pari al 50%, mentre nella fase dell'elitismo, con una probabilità del 10%, i cromosomi migliori subivano anch'essi un incremento di un'azione. Tuttavia, questo ha prodotto dei risultati poco soddisfacenti, ma più efficaci rispetto al metodo preso come spunto dalla ricerca [2]. Infatti, in quel lavoro, si sceglieva semplicemente la lunghezza di un lato del labirinto moltiplicata per 2 per inizializzare la popolazione. Inoltre, durante la fase di crossover, veniva costruita una funzione definita n-module 4 che, facendo delle sottrazioni e addizioni, generava i nuovi figli. Infine, un'ulteriore differenza tra la ricerca appena citata e la nostra implementazione, consiste nell'utilizzo della tecnica dell'elitismo, che l'autore del lavoro ha deciso di non utilizzare. Le modifiche che sono state da noi apportate, hanno prodotto dei miglioramenti rispetto alla versione citata. Infatti, le conclusioni della fonte [2], affermano che in un labirinto di dimensione 20x20, per poter trovare la soluzione ottimale il numero di iterazioni necessario è pari a 500 (con una popolazione iniziale di lunghezza 100). Mentre, attraverso la nostra implementazione, si converge ad una buona soluzione con soli 6 genitori di partenza e 80 iterazioni.
Tuttavia, più la dimensione del labirinto aumenta e più il tempo del GA cresce vertiginosamente, ma soprattutto non è garantito il percorso più breve. Per tale motivo, abbiamo voluto approfondire lo studio implementando degli algoritmi ibridi molto efficienti.

5.2 L'inefficacia di SARSA

Vengono di seguito spiegate le ragioni dell'inefficacia dell'algoritmo SARSA. Essendo un metodo on-policy, l'aggiornamento del valore $Q(s, a)$ dipende dalla ϵ-greedy policy. Infatti a^{t+1}, con una certa probabilità ϵ, sarà un'azione che porterà l'agente in una cella casuale anche dopo molte iterazioni dell' algoritmo. Questo vuol dire che, se l'azione a^{t+1} corrisponde ad una azione casuale, il metodo SARSA potrebbe penalizzare il valore di $Q(s, a)$ appartenente alla sequenza ottimale, indirizzando l'agente verso percorsi che lo portano in vicoli ciechi. Invece Q-Learning sceglie a^{t+1}in modo greedy e questa tecnica fa si che i valori della Q-table convergono al loro valore effettivo in modo più rapido. Per questo motivo si dice che Q-Learning cerca di massimizzare la reward con il passare degli episodi, mentre Sarsa massimizza la reward all'interno dell'episodio stesso. Visto che siamo interessati nel massimizzare la reward (uscire dal labirinto) nel lungo periodo, allora Q-Learning risulta il metodo più efficiente. Infatti per matrici 80x80, quest'ultimo algoritmo converge alla soluzione ottimale in circa 1500 episodi mentre SARSA,

in alcuni casi, non ha trovato la soluzione ottimale neanche in 1.000.000 di iterazioni.

5.3 Gli algoritmi ibridi e il Reinforcement Learning

Per quanto riguarda il confronto delle performance, è evidente che l'algoritmo Hybrid Ant Colony With Backtracking risulta il più veloce per ciascuna dimensione del labirinto. Con gli algoritmi di Reinforcement Learning si ottengono buone prestazioni per matrici piccole, mentre all'aumentare della dimensione del problema, essi risultano meno efficienti dell'Hybrid Dijkstra Algorithm With Ant Colony Exploration. C'è una spiegazione molto logica a questi risultati. Infatti, c'è una diretta proporzione tra il tempo di convergenza e la quantità di informazioni fornite per risolvere il problema. In particolare, l'Hybrid Ant Colony With Backtracking, che è il più veloce, ci dice qual è la soluzione ottimale partendo dalla cella iniziale fino ad arrivare alla cella finale. L'Hybrid Dijkstra Algorithm With Ant Colony Exploration, invece, permette di risolvere il problema partendo da uno qualsiasi dei nodi individuati nel labirinto, perciò ci fornisce un'informazione in più. Infine abbiamo il Reinforcement Learning, che è certamente l'algoritmo più lento, ma è anche quello che fornisce il maggior contenuto informativo. Infatti, attraverso questo algoritmo, è possibile sapere come uscire dal labirinto da una qualsiasi delle celle in cui si trova l'agente, che deve semplicemente seguire la policy ottimale. C'è da dire anche che il Reinforcement Learning non ha nei suoi punti di forza, per natura, quello di convergere in breve tempo ad una soluzione in un problema del genere. Infatti, l'agente può esplorare l'ambiente all'infinito migliorando sempre di più la precisione dei valori della Q-table. Abbiamo deciso di utilizzare la metrica del tempo soltanto per poter confrontare i risultati con gli altri algoritmi. Uno dei grandi vantaggi riscontrati nell'utilizzo del Reinforcement Learning è stato la semplicità di scrittura. Non c'è bisogno, infatti, di alcuna condizione if-else tipica degli algoritmi di programmazione, perché l'agente impara da solo a muoversi nell'ambiente in modo ottimale.

6 Conclusioni

In questo progetto è stato dimostrato come il problema del labirinto possa essere risolto attraverso diversi algoritmi che, nella loro diversità, soddisfano molteplici necessità. Se l'obiettivo di un ricercatore fosse quello di trovare il percorso ottimale dall'entrata all'uscita nel minor tempo possibile, allora l'Ant Colony With Backtracking risulterebbe il più efficiente. Se invece si avesse la necessità di conoscere il miglior percorso da seguire partendo da una posizione qualsiasi, allora si potrebbe utilizzare il Reinforcement Learning che è più lento, ma molto più semplice da scrivere. L'Hybrid Dijkstra Algorithm With Ant Colony Exploration si colloca a metà tra questi due algoritmi appena citati in quanto permette di risolvere il problema partendo da uno qualsiasi dei nodi nel labirinto e registra ottimi tempi di risoluzione. Per

quanto riguarda il GA, invece, sono stati ampliamente delineati i suoi punti deboli, uno tra tutti, quello di non essere a conoscenza della posizione dei muri all'interno del labirinto. Inoltre tale algoritmo, nonostante i lunghi tempi di risoluzione, non da alcuna garanzia dal punto di vista della ricerca del percorso ottimale e spesso fornisce soluzioni ricche di azioni superflue. Si sconsiglia pertanto il suo utilizzo per questo specifico problema.

References

[1] Ramadhian Fauzan, [*Implementation of Prim's and Kruskal's Algorithms' on Maze Generation*], 2013

[2] Choubey Nitin, *International Journal of Computer Applications (0975 – 8887)* [*A-Mazer with Genetic Algorithm*]. Vol 58: 48-54, 2012.

[3] Edsger W. Dijkstra, [*A note on two problem in connexion with graphs*], Numerische Mathematik, 1:269– 271, 1959.

[4] Lingaraj, Haldurai, *International Journal of Computer Sciences and Engineering* [*A Study on Genetic Algorithm and its Applications*], Vol 4: 139-143, 2016

[5] Marco Dorigo, Scholarpedia, 2(3):1461.

[6] Ahuja, Mohit and HomChaudhuri, Baisravan and Cohen, Kelly and Kumar, Manish [*Fuzzy Counter Ant Algorithm for Maze Problem*], 2010.