UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
CORSO DI ELEMENTI DI INTELLIGENZA ARTIFICIALE

Progetto di Elementi di Intelligenza Artificiale

Autori:

Salvatore Raiola - **N46006075**

Emmanuel Russo - **N46006096**

Sommario:

1. Scopo del progetto

Il progetto mira allo sviluppo di un software dedicato alla risoluzione di problemi attraverso tecniche di ricerca.

In particolare, dato un **dataset** (la cui struttura verrà dettagliata in seguito) che rappresenta uno spazio degli stati – includendo tutti gli stati possibili e le loro connessioni – e due **stati** forniti dall'utente (uno come stato iniziale e l'altro come obiettivo), il software sarà in grado di determinare, se presente, una **soluzione**.

Quest'ultima consisterà nella sequenza di stati da percorrere per raggiungere l'obiettivo a partire dallo stato iniziale.

L'utente può selezionare l'algoritmo di ricerca da applicare scegliendo tra le cinque opzioni disponibili:

1) Breadth-First Search (**BFS**)

2) Uniform Cost Search (**UCS**) - Algoritmo di Dijkstra (**DJKS**)

3) Depth-First Search (**DFS**)

4) Depth-Limited Search (**DLS**)

5) Iterative Deepening Search (**IDS**)

Infine, si assume che eventuali elementi di casualità (**componenti aleatorie**) abbiano un impatto trascurabile sul processo di ricerca, trattandosi di un **problema deterministico**.

2. Considerazioni sull'implementazione (indicazioni utili per l'uso)

2.1. Dataset

Il dataset dovrà essere fornito sotto forma di un **file di testo**, in cui gli stati e le relative connessioni siano rappresentati secondo le seguenti regole:

1) Il file deve contenere **due colonne**: la prima indica lo stato di partenza, mentre la seconda specifica lo stato di destinazione.

Ogni riga del file corrisponde a un **collegamento diretto** tra due stati.

2) Gli stati devono essere identificati esclusivamente da un **codice numerico**, senza l'uso di simboli o altri caratteri.

Questa restrizione, seppur apparentemente rigida, facilita il rilevamento precoce di eventuali errori o manomissioni nel dataset, prevenendo possibili compromissioni nella ricerca della soluzione.

3) Le colonne devono essere separate **solo da spazi** (uno o più).

Qualsiasi altro carattere, ad eccezione dei numeri che identificano gli stati e degli spazi separatori, non è ammesso.

4) Come specificato in precedenza, la prima colonna indica lo stato di partenza e la seconda lo stato di arrivo, definendo un collegamento **direzionato**.

Se un collegamento tra due stati deve essere considerato **bidirezionale**, sarà necessario aggiungere una seconda riga con gli stessi stati riportati in ordine inverso.

5) Per inserire commenti o annotazioni che non influiscano sulla definizione dello spazio degli stati, ogni riga dedicata a tali informazioni dovrà iniziare con il simbolo '#'.

2.2. Interazioni software - utente

Riguardo il codice sorgente e le interazioni software – utente:

1) L'utente deve specificare il **percorso** del file contenente il dataset, oppure indicarne semplicemente il nome nel caso in cui si trovi nella stessa cartella dell'eseguibile.

Se il software viene eseguito su Windows, è necessario includere l'estensione **'.txt'** nel nome del file.

2) Durante l'esecuzione, il software richiede di configurare un **parametro di tolleranza**, che determina il comportamento del programma in caso di errori di formato nel file del dataset.

Le opzioni disponibili sono:

- **[IGNORE]**: eventuali righe non conformi vengono semplicemente ignorate.
- **[WARNING]**: l'utente viene avvisato dell'anomalia e deve confermare per continuare.
- **[RAISE FAULT]**: l'errore viene segnalato e l'esecuzione del programma viene interrotta.

3) Gli stati inseriti dall'utente devono essere numeri validi, corrispondenti a stati effettivamente presenti nel dataset.

Poiché il processo di ricerca può risultare computazionalmente oneroso, il software esegue un controllo preliminare sugli stati specificati.

Se uno o entrambi non sono presenti nel dataset, l'utente avrà la possibilità di reinserire i parametri correttamente.

4) Se il software individua una soluzione, fornirà in output i seguenti dati:

- Percorso della soluzione
- Lunghezza del percorso (numero di nodi esplorati)
- Numero di iterazioni effettuate (**complessità temporale**)
- Uso massimo della memoria (**complessità spaziale**)
- Tempo di esecuzione

Per un approfondimento su questi parametri, si faccia riferimento alla sezione dedicata.

Se invece non esiste alcuna soluzione, l'utente verrà notificato dell'esito negativo.

5) Nel caso in cui si desideri eseguire più ricerche in rapida successione, si consiglia di **non chiudere il programma tra un'operazione e l'altra**.

Il dataset viene caricato e analizzato solo una volta per ogni esecuzione del software, permettendo così di effettuare più ricerche in modo efficiente.

Interrompere il programma e riavviarlo comporterebbe una nuova analisi del dataset, con possibili rallentamenti, specialmente se il file è di grandi dimensioni.

3. Metodologie e tecniche adottate

Prima di procedere con il **testing** del software, è fondamentale soffermarsi sulla teoria alla base degli algoritmi di ricerca implementati.

Il compito principale di questi algoritmi è consentire ad un **agente** – un'entità in grado di percepire l'ambiente attraverso sensori e di interagire con esso tramite attuatori – di risolvere un problema, ovvero di raggiungere un obiettivo.

L'interazione tra **agente e ambiente** può essere rappresentata attraverso il concetto di **stato**, che costituisce un'astrazione di una situazione reale.

Il livello di astrazione dipende dal contesto specifico e gli stati vengono descritti tramite dati strutturati.

Risolvere un problema significa, partendo da uno stato iniziale e conoscendo lo stato obiettivo (goal) da raggiungere, individuare la sequenza di stati da attraversare, eseguendo determinate azioni, per arrivare alla soluzione.

L'insieme di tutti gli stati possibili, chiamato **spazio degli stati**, può essere efficacemente rappresentato mediante un **grafo**.

In questa struttura:

1) Gli stati corrispondono ai **nodi** del grafo, ciascuno dei quali è definito in modo univoco da una serie di parametri.

2) Le azioni che l'agente può compiere sono rappresentate dagli **archi**, che collegano due nodi.

Poiché un arco collega due stati, esso rappresenta un'azione che l'agente può eseguire per passare da uno stato all'altro.

Questo implica che **uno stato può essere l'evoluzione diretta di un altro** e che il percorso nel grafo descrive la successione di trasformazioni necessarie per raggiungere l'obiettivo.

Questa rappresentazione consente di visualizzare chiaramente le **possibili evoluzioni del sistema** e di applicare strategie di ricerca per individuare il percorso ottimale verso la soluzione.

3.1. Tipologie di problemi e impatto sulla ricerca della soluzione

La rappresentazione di un problema e la sua risoluzione dipendono strettamente dalle caratteristiche della **coppia agente-ambiente**.

Di seguito vengono analizzati i principali scenari che influenzano il processo di ricerca della soluzione:

• Problema **deterministico** e **completamente osservabile** (**single-state problem**)

In questo caso, ogni azione dell'agente ha un esito ben definito e l'ambiente è interamente percepibile. Lo spazio degli stati è completo, poiché l'agente è in grado di osservare tutte le caratteristiche che definiscono l'ambiente e di mappare tutti gli stati possibili.

Inoltre, grazie alla natura deterministica del problema, l'agente può pianificare una sequenza di azioni esatta per raggiungere l'obiettivo, garantendo così una soluzione ottimale.

• Problema **non osservabile**

Questa situazione si verifica quando l'agente ha una percezione limitata dell'ambiente, a causa dell'assenza o del malfunzionamento dei sensori.

Nel caso più estremo, l'agente non riceve alcuna informazione dall'ambiente, rendendolo **incapace di determinare il proprio stato attuale** e quindi impossibilitato a trovare una soluzione.

• Problema **non deterministico** e/o **parzialmente osservabile** (**problema di contingenza**)

Qui l'agente opera in un ambiente **dinamico e influenzato da elementi aleatori**, che rendono l'esito delle sue azioni incerto. In questi casi si utilizza un approccio flessibile, noto come **piano di contingenza**: l'agente formula ipotesi sullo stato dell'ambiente (**belief state**) e pianifica un'azione basata su queste supposizioni.

Tuttavia, a causa della natura mutevole dell'ambiente, la soluzione sarà soggetta a modifiche in tempo reale. In questo scenario, l'agente opera in modalità **online**, ossia deve prendere decisioni e agire senza possedere una conoscenza completa della situazione.

• Problema di **esplorazione** (**spazio degli stati non noto**)

Alcuni problemi, in particolare quelli affrontati da agenti online, presentano uno spazio degli stati **sconosciuto** o **potenzialmente infinito**.

In questi casi, l'obiettivo principale non è solo trovare la soluzione, ma anche **esplorare l'ambiente in modo efficiente**, costruendo progressivamente una rappresentazione dello spazio degli stati.

Poiché la modellazione del problema avviene tramite un **grafo**, trovare una soluzione implica sviluppare un algoritmo di esplorazione del grafo.

Il metodo più diretto è l'adozione di algoritmi di ricerca ad albero (**tree search algorithm**).

Questi algoritmi, di tipo **offline** (si assume che abbiano accesso all'intera struttura del grafo prima di prendere una decisione), effettuano la ricerca sfruttando una struttura ad albero.

Durante l'esplorazione, l'algoritmo ha visibilità solo sui nodi direttamente collegati (**one-hop**) al nodo corrente.

Procedendo iterativamente, raccoglie informazioni sugli altri stati fino a che:

A) **Trova il nodo corrispondente allo stato obiettivo**, determinando così il percorso dalla partenza all'arrivo.

B) **Termina l'esplorazione senza trovare il goal**, concludendo che non esiste una soluzione valida.

3.2. Metriche di valutazione degli algoritmi di ricerca

Poiché la ricerca di una soluzione a un problema può essere ricondotta **all'esplorazione di un grafo**, è fondamentale determinare il metodo più efficace per condurre tale esplorazione.

Per valutare quale algoritmo sia il più adatto a seconda del contesto, si utilizzano diverse **metriche di valutazione**, tra cui:

• Completezza

Un algoritmo è considerato **completo** se garantisce che, nel caso in cui esista almeno una soluzione, sarà in grado di trovarla.

Se il problema ammette **soluzioni multiple**, un algoritmo completo ne identificherà almeno una.

• Complessità spaziale e temporale

In un mondo ideale, con **capacità computazionali illimitate** e **tempo infinito**, sarebbe possibile trovare una soluzione per la maggior parte dei problemi.

Tuttavia, nella pratica, sia il **tempo di esecuzione** che le **risorse di memoria** sono limitati e spesso comportano costi elevati.

Per questo motivo, è necessario stimare il costo computazionale di un algoritmo in termini di:

1) **Maximum branching factor** (**b**) → massimo numero di nodi generati da un singolo nodo durante la ricerca.

2) **Maximum depth of the state space** (**m**) → profondità massima dello spazio degli stati.

3) **Depth of the least-cost solution** → profondità minima alla quale si trova la soluzione ottimale.

• Ottimalità

Un algoritmo è considerato **ottimale** se garantisce che la soluzione trovata sia la **migliore possibile** in base al criterio di costo adottato, ovvero quella con il **costo minore** rispetto alle alternative disponibili.

3.3. Valutazione degli algoritmi

Concludiamo questa breve trattazione teorica andando ad analizzare gli algoritmi implementati sulla base delle metriche appena definite:

1) **Breadth-First Search** (**BFS**):

Descrizione: Algoritmo basato sulla **ricerca in ampiezza**, in cui si esplorano prima i nodi più vicini al nodo iniziale.

Questi sono i nodi raggiungibili attraversando meno nodi intermedi.

Memorizzazione: Utilizza una coda **FIFO** (First In, First Out), dove il primo nodo inserito è anche il primo a essere esplorato.

Proprietà:

• **Non Completo** in caso di ramificazioni infinite (**b = ∞**): se la soluzione si trova a una profondità maggiore, essa non verrà mai trovata, poiché i nodi a profondità inferiore continueranno a essere esplorati all'infinito.

L'algoritmo è invece completo se b ha valore finito.

• **Complessità Temporale e Spaziale**: L'algoritmo presenta una complessità **esponenziale** in entrambi i casi.

La complessità temporale e spaziale è **O(b^d+1)**, dove **b** è la **ramificazione** e **d** è la **profondità** della soluzione.

• **Ottimalità**: Non è ottimo poiché la soluzione trovata dipende dalla posizione nell'albero di ricerca e non dal costo minimo.

In altre parole, la soluzione corrisponde al nodo finale a profondità inferiore, senza alcuna garanzia che sia la soluzione ottima.

Tuttavia, se i costi sono uniformi, la soluzione trovata è quella ottima.

2) **Dijkstra's Algorithm** (**DJKS**):

Descrizione: Algoritmo di ricerca utilizzato per trovare il cammino più breve in un grafo con pesi non negativi.

A differenza degli algoritmi di ricerca basati su esplorazione di nodi, Dijkstra **mantiene il controllo sui costi minimi** per arrivare ai nodi da una sorgente e li esplora in ordine crescente di distanza dal nodo di partenza.

Memorizzazione: Utilizza una **coda di priorità** (min-heap o struttura simile) per selezionare il nodo con il costo minimo.

Proprietà:

• **Completo**: È completo poiché esplora ogni nodo del grafo una volta e garantisce che la soluzione trovata sia quella ottima.

• **Complessità spaziale**: **O(n)**, dove n è il numero di nodi nel grafo.

• **Complessità temporale**: La complessità temporale è **O((n + m) log n)**, dove **n** è il numero di nodi ed **m** è il numero di archi del grafo, quando si utilizza una coda di priorità.

• **Ottimalità**: È ottimo per grafi con pesi non negativi, in quanto garantisce di trovare il cammino più breve dal nodo di partenza a qualsiasi altro nodo del grafo.

3) **Depth-First Search** (**DFS**):

Descrizione: Algoritmo che esplora prima i nodi più **profondi**, ossia quelli che si allontanano maggiormente dal nodo iniziale.

Memorizzazione: Utilizza una coda **LIFO** (Last In, First Out), dove l'ultimo nodo inserito è il primo a essere esplorato.

Proprietà:

• **Non Completo** in caso di profondità infinita (**m = ∞**): se la soluzione si trova lungo un ramo che non viene esplorato inizialmente, essa non verrà mai trovata.

L'algoritmo è invece completo se m ha valore finito.

• **Complessità spaziale** è **O(b*m)**, in quanto si memorizza solo il percorso corrente.

• **Complessità temporale** è **O(b^m)**, che cresce esponenzialmente con la profondità del problema.

• **Ottimalità**: Non è ottimo, poiché la soluzione dipende dalla posizione nell'albero di ricerca e non dal costo minimo.

In pratica, non c'è alcuna garanzia che la soluzione trovata sia quella ottima.

L'algoritmo tende a espandere profondamente i rami senza considerare l'efficienza complessiva.

4) **Depth-Limited Search** (**DLS**):

Descrizione: Algoritmo che limita la profondità della ricerca impostando un valore massimo di profondità (**l**).

Non esplorerà mai al di là di questo limite.

Memorizzazione: Funziona in modo simile al Depth-First Search, ma con una profondità limitata.

Proprietà:

• **Non Completo**: La ricerca potrebbe non trovare la soluzione se il nodo goal si trova a una profondità maggiore di l. Se il nodo si trova entro il limite l, invece, la soluzione verrà trovata.

• **Complessità spaziale**: **O(l)**, in quanto l'algoritmo memorizza solo i nodi fino alla

profondità l.

• **Complessità temporale**: **O(b^l)**, che cresce esponenzialmente con l'altezza del limite imposto.

• **Ottimalità**: Non garantisce una soluzione ottima poiché la soluzione trovata dipende dalla posizione del nodo finale nell'albero, non dal costo.

Se la profondità massima l non fosse scelta correttamente, la soluzione potrebbe non essere la migliore possibile.

5) **Iterative Deepening Search** (**IDS**):

Descrizione: Algoritmo che cerca di risolvere il problema del Depth-Limited Search, eseguendo ripetutamente una ricerca con limiti di profondità crescenti.

Inizia con un limite di profondità l = 0 e aumenta progressivamente fino a trovare il nodo goal.

Memorizzazione: L'algoritmo esegue una serie di ricerche in profondità con l crescenti, combinando le caratteristiche di Depth-First Search e Breadth-First Search.

Proprietà:

• **Completo**: Poiché l'algoritmo esplora iterativamente tutte le profondità fino a trovare la soluzione, è completo, anche in caso di b = ∞.

Non si corre il rischio di perdere la soluzione a causa della profondità infinita.

• **Complessità spaziale**: **O(b*d)**, dove **d** è la profondità della soluzione.

• **Complessità temporale**: **O(b^d)**, che è simile a quella di una ricerca in ampiezza.

• **Ottimalità**: Non è ottimo, in quanto, similmente agli altri algoritmi, la soluzione dipende dalla posizione del nodo finale nell'albero, non dal costo.

Tuttavia, se i costi sono uniformi, l'algoritmo è completo e trova la soluzione ottima.

4. Risultati sperimentali

Si è deciso di analizzare le prestazioni dei vari algoritmi di ricerca, considerando il tempo impiegato, l'uso di memoria, numero di iterazioni e lunghezza del percorso al fine di comprendere meglio le differenze tra gli algoritmi e osservare in quali scenari ciascuno di essi potrebbe essere preferito rispetto agli altri.

I test sono stati eseguiti con i seguenti algoritmi di ricerca, già descritti in precedenza:

1) Breadth-First Search (**BFS**)

2) Uniform Cost Search (**UCS**) - Algoritmo di Dijkstra (**DJKS**)

3) Depth-First Search (**DFS**)

4) Depth-Limited Search (**DLS**)

5) Iterative Deepening Search (**IDS**)

4.1. Algoritmi utilizzati e metodologia di valutazione

1) **Breadth-First Search** (**BFS**): L'algoritmo di ricerca in ampiezza è stato scelto per testare la sua capacità di trovare soluzioni ottime in scenari in cui la profondità non è eccessiva.

La sua semplicità e l'uso di una coda FIFO lo rendono adatto per risolvere il problema della ricerca del percorso più breve in grafi non ponderati.

2) **Uniform Cost Search** (**UCS**) - **Algoritmo di Dijkstra** (**DJKS**): L'algoritmo di Dijkstra è stato implementato per testare la sua capacità di trovare il percorso più breve, considerando una struttura di grafo con **pesi non negativi**.

Dijkstra è stato scelto come confronto con gli altri algoritmi in quanto è noto per la sua efficacia in grafi con pesi, trovando il percorso più breve in modo ottimale.

3) **Depth-First Search** (**DFS**): Sebbene non ottimo e con una maggiore probabilità di non trovare la soluzione in caso di profondità infinita, l'algoritmo di ricerca in profondità è stato utilizzato per testare la sua efficienza in scenari dove la soluzione si trova a una **profondità relativamente bassa** (0-420 e 0-1500) e per verificare il comportamento in termini di uso della memoria, che cresce linearmente con la profondità.

4) **Depth-Limited Search** (**DLS**): Questo algoritmo è stato scelto per testare il comportamento di una ricerca che **limita la profondità**.

Sebbene non completo, in quanto la soluzione potrebbe non essere trovata se la profondità massima è troppo bassa, è utile in scenari dove si vuole limitare la ricerca a una certa profondità specificata.

L'algoritmo risulta comunque efficace a parità di profondità, ma la sua qualità dipende strettamente dalla profondità scelta.

Nei test effettuati, tale algoritmo viene testato con la stessa profondità trovata da IDS, così da mostrare le differenze tra i due.

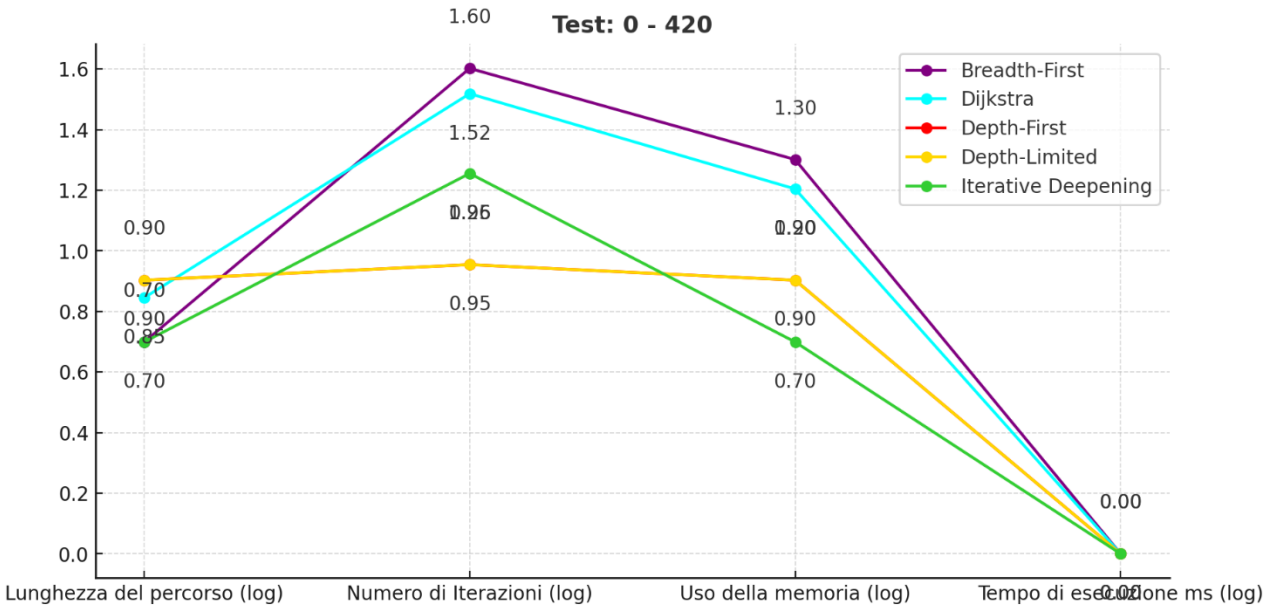
5) **Iterative Deepening Search** (**IDS**): L'algoritmo che combina le caratteristiche della ricerca in ampiezza e della ricerca in profondità è stato incluso per testare la sua capacità di trovare una soluzione in modo completo, senza rischi di esplorare all'infinito nodi a profondità sempre maggiore.

IDS risulta utile quando il grafo ha una profondità finita ma non è noto il valore esatto della stessa.

4.2. Test eseguiti

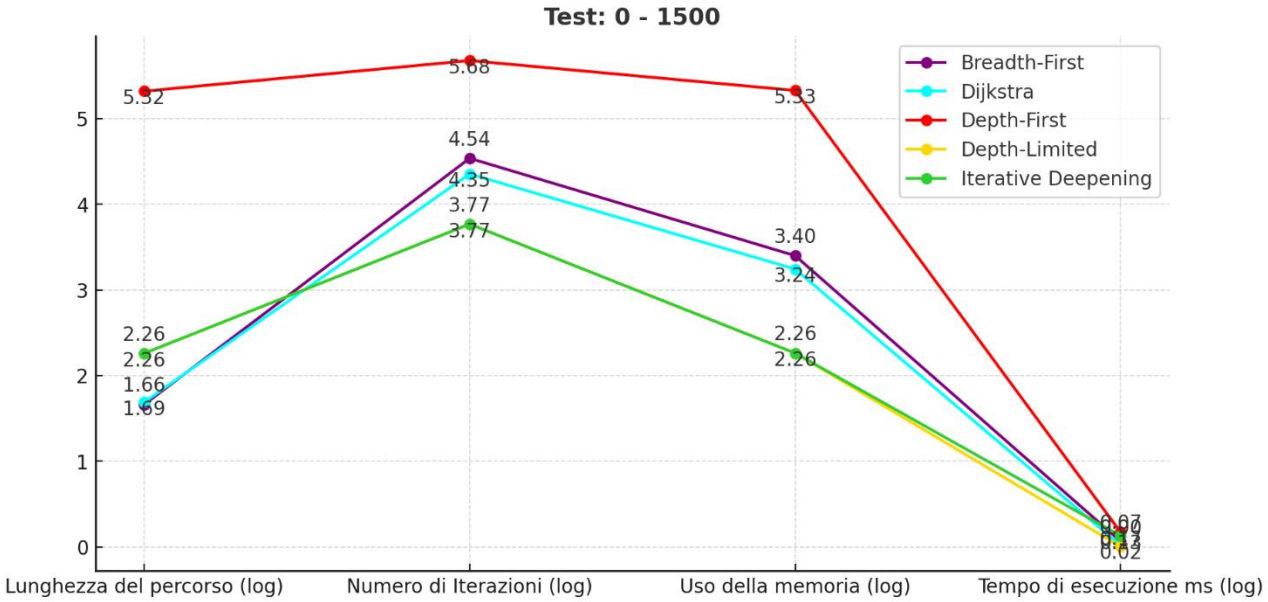
I seguenti test sono stati effettuati per analizzare le prestazioni degli algoritmi nelle varie situazioni:

- Ricerca percorso dal nodo 0 al nodo 420

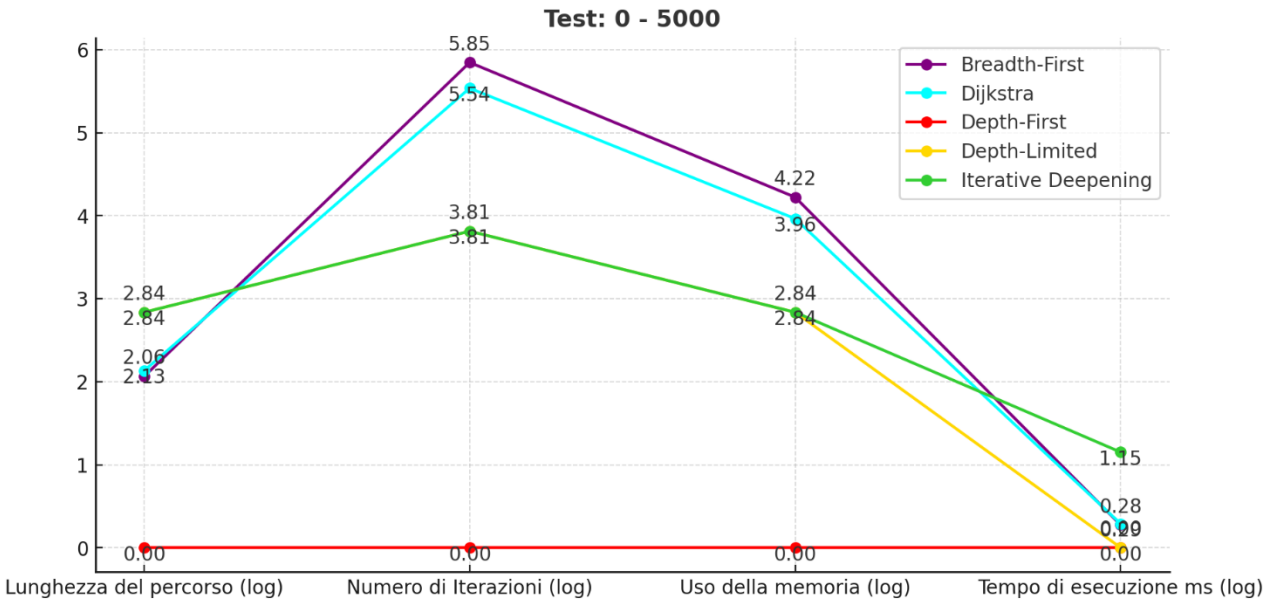


NB: I grafici di Depth-First (rosso) e Depth-Limited (giallo) sono sovrapposti.

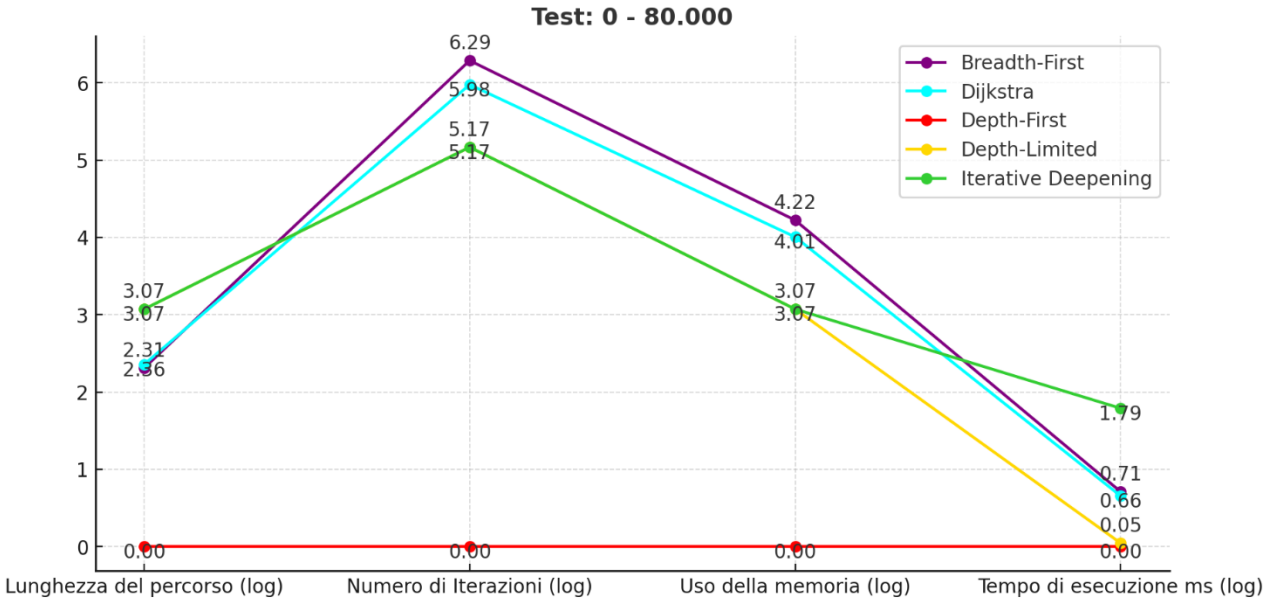
- Ricerca percorso dal nodo 0 al nodo 1500



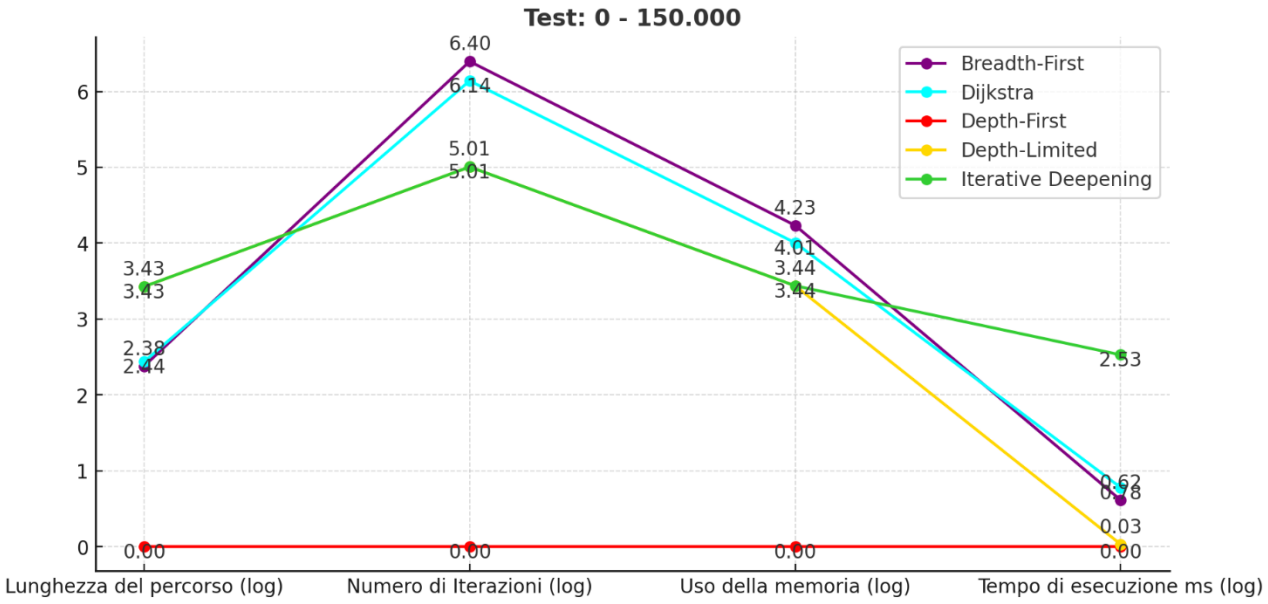
- Ricerca percorso dal nodo 0 al nodo 5000



- Ricerca percorso dal nodo 0 al nodo 80000



- Ricerca percorso dal nodo 0 al nodo 150000



Questi test sono stati scelti in modo da evidenziare le differenze tra i vari algoritmi.

In particolar modo si tenta di mettere in evidenza la differenza tra algoritmi come IDS e DLS (**tempo di esecuzione**), Dijkstra e BFS e la dipendenza di DFS della profondità del nodo.

4.3. Considerazioni sui risultati

Man mano che aumenta il valore del nodo (e quindi la complessità del grafo), la differenza nelle prestazioni tra gli algoritmi diventa sempre più evidente.

• In particolare, i tempi di esecuzione di **Iterative Deepening Search** (IDS) e **Depth-Limited Search** (DLS) tendono a differenziarsi significativamente a parità di profondità scelta.

• **Breadth-First Search** (BFS), seppur efficace nel trovare il percorso più breve, potrebbe essere penalizzato dal bisogno di memorizzare tutti i nodi a livello di profondità inferiore, ma continua a essere competitivo in grafi di dimensioni moderate.

• L' **Algoritmo di Dijkstra**, invece, ha mostrato un comportamento **ottimale** e **costante**, trovando sempre il percorso più breve.

Tuttavia, la memoria necessaria per memorizzare le informazioni sui percorsi può diventare un fattore limitante con l'aumento della dimensione del grafo.

• In generale, **Depth-First Search** (DFS) ha avuto prestazioni **meno ottimali** rispetto agli altri algoritmi, in quanto la sua natura esplorativa tende a farlo arrivare più facilmente in percorsi lunghi senza mai trovare la soluzione, soprattutto in grafi grandi.

Dai test si evince inoltre che DFS **non è completo** (non trova sempre la soluzione), in particolare nei test [0-5000], [0-80000] e [0-150000].