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:

Francesco Pio Vitiello - N46006855

Luigi Mercurio – N46006810

Mario Sorrentino – N46006517

Sommario

1. Scopo del progetto

Il progetto ha come obiettivo lo sviluppo di un software in grado di risolvere problemi di ricerca nello spazio degli stati attraverso l'applicazione di algoritmi di ricerca informata e non informata.

In particolare, il software opera su un dataset reale – rappresentato da un grafo diretto estratto dal co-purchasing network di Amazon – nel quale ciascun nodo corrisponde a un prodotto e ciascun arco rappresenta una relazione di co-acquisto tra prodotti.

Forniti due nodi (uno di partenza e uno di arrivo), il sistema è in grado di determinare, se esistente, un percorso che collega il nodo iniziale a quello obiettivo. Il risultato, quando disponibile, consiste in una sequenza ordinata di nodi che rappresentano il cammino da seguire.

L'utente può scegliere l'algoritmo di ricerca da applicare selezionandolo tra i seguenti:

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**)

Il problema è affrontato in contesto **deterministico** e **completamente osservabile**: si assume che eventuali elementi di casualità abbiano un impatto trascurabile sul processo di ricerca.

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

2.1. Dataset

Il dataset utilizzato dal software è un file di testo che rappresenta un grafo diretto, in cui ciascun nodo è identificato da un intero positivo e rappresenta un prodotto Amazon. Ogni riga del file descrive un collegamento (arco) tra due prodotti, con un peso associato che simula un costo o una distanza tra i due.

Le regole di formattazione del dataset sono le seguenti:

1) Il file deve contenere **tre colonne**:

   - la prima rappresenta il nodo di partenza,
   - la seconda il nodo di destinazione,
   - la terza il **peso** associato all'arco (numero intero positivo).

2) I nodi devono essere rappresentati da **numeri interi senza simboli**.
   Questo facilita la lettura e l'elaborazione del file da parte del parser, prevenendo errori legati al formato.
3) Il file può contenere **commenti** o intestazioni: tutte le righe che iniziano con il simbolo # vengono ignorate.
4) Il grafo è considerato **diretto**: un collegamento da A a B non implica automaticamente la presenza di un collegamento da B a A.
   Se si vuole rappresentare un arco bidirezionale, occorre inserire due righe separate.
5) Il software è progettato per caricare il dataset una sola volta, all'avvio del programma. Ciò consente di eseguire più ricerche consecutive senza rielaborare l'intero grafo, migliorando così l'efficienza nei casi di grafi di grandi dimensioni.

2.2. Interazioni software - utente

Il programma è progettato per essere utilizzato tramite **interfaccia testuale (CLI)**, ed è pensato per garantire un'interazione semplice ed efficace anche su grafi di grandi dimensioni.

1) Il dataset è già integrato nel progetto e il suo percorso è preconfigurato all'interno del codice (dataset/amazon0312-weighted.txt).
   L'utente non deve quindi inserire manualmente il percorso del file: il grafo viene caricato automaticamente all'avvio del programma.
2) Non è richiesta alcuna configurazione relativa alla tolleranza degli errori nel file, poiché il parser gestisce automaticamente:

   - la presenza di righe non valide (commenti # o formati errati),
   - eventuali problemi vengono notificati a terminale tramite messaggio di errore standard.

3) Durante l'esecuzione, l'utente è guidato attraverso una sequenza di richieste:

   - inserimento del **nodo di partenza** e del **nodo obiettivo** (solo se presenti nel dataset),
   - selezione dell'**algoritmo di ricerca** da utilizzare tra quelli disponibili,
   - eventuale inserimento della **profondità massima** nel caso di Depth-Limited Search.

Se i nodi specificati non esistono nel grafo, il programma avvisa l'utente e consente di reinserirli.

4) Dopo l'esecuzione dell'algoritmo scelto, il software restituisce a terminale:

   - il **percorso trovato** (lista dei nodi),
   - la **lunghezza del percorso** (in numero di nodi),
   - il **numero di iterazioni** eseguite,
   - la **memoria massima utilizzata** (in termini di dimensione della struttura dati),
   - il **tempo di esecuzione** in secondi,
   - il **costo totale del percorso** (solo nel caso di Dijkstra).

   In caso non venga trovata alcuna soluzione, il programma lo segnala all'utente.

5) Il dataset viene caricato **una sola volta** all'avvio del programma. Questo consente di effettuare più ricerche consecutive senza dover rielaborare l'intero grafo, migliorando le prestazioni. Al termine di ogni ricerca, l'utente ha la possibilità di effettuare una nuova ricerca o di uscire dal programma.

3. Metodologie e tecniche adottate

Prima di procedere con l'analisi sperimentale, è utile richiamare le basi teoriche su cui si fondano gli algoritmi di ricerca implementati nel software.

Questi algoritmi nascono con lo scopo di consentire a un agente – ovvero un'entità capace di percepire l'ambiente e di compiere azioni – di risolvere un problema individuando una **sequenza di passi** che lo porti da una situazione iniziale a una situazione obiettivo.

Nel nostro contesto, il problema di ricerca è rappresentato da un **grafo diretto**, in cui:

- ogni **nodo** rappresenta un prodotto (identificato da un ID numerico),
- ogni **arco** rappresenta una relazione di co-acquisto tra prodotti,
- e ogni **peso** associato all'arco indica un costo (fittizio) di attraversamento.

L'intero **spazio degli stati** è quindi rappresentato dal grafo stesso:

- gli **stati** sono i prodotti (nodi),
- le **azioni** corrispondono al passaggio da un prodotto a un altro (archi).

Poiché gli archi sono direzionati, una transizione è valida solo nella direzione specificata. Se si desidera rendere una connessione bidirezionale, è necessario che entrambe le direzioni siano esplicitamente presenti nel dataset.

Questa astrazione consente di applicare efficacemente strategie di **ricerca su grafo** per individuare il cammino (se esiste) che porta dal nodo di partenza al nodo di destinazione.

Il software sviluppato permette di simulare questo scenario, fornendo all'utente strumenti per:

- scegliere lo stato iniziale e quello obiettivo,
- applicare l'algoritmo desiderato,
- analizzare le performance e il comportamento degli algoritmi in base alle caratteristiche del grafo sottostante.

3.1. Tipologie di problemi e impatto sulla ricerca della soluzione

La natura del problema da risolvere e la strategia adottata dipendono strettamente dalle caratteristiche dell'interazione tra l'agente e l'ambiente. In letteratura, si distinguono diversi scenari che condizionano il processo di ricerca della soluzione.

Nel contesto di questo progetto, il problema affrontato è di tipo **deterministico**, **completamente osservabile** e gestito in modalità **offline**, ma per completezza si riassumono di seguito le principali tipologie.

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

  Ogni azione ha un effetto prevedibile e l'agente ha accesso completo alla struttura dell'ambiente. In questo caso, lo **spazio degli stati** è noto e interamente rappresentabile tramite un **grafo**.

L'agente può pianificare un percorso preciso dal nodo iniziale al nodo obiettivo. È il caso considerato nel nostro progetto, in cui il grafo è fornito per intero e le azioni corrispondono a transizioni tra nodi.

- Problema non osservabile

  In questo scenario, l'agente non ha informazioni sull'ambiente circostante: non può determinare in quale stato si trovi né prevedere le conseguenze delle azioni.

  Questa situazione è fuori dallo scopo del presente progetto, poiché lavoriamo con dati noti e statici.

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

  L'agente si muove in un ambiente incerto, dove le azioni possono avere esiti diversi e lo stato corrente potrebbe non essere completamente percepibile.
  Richiede la gestione di stati ipotetici (belief states) e piani di contingenza. Anche questa tipologia non è trattata nel nostro software, che si basa su informazioni complete e deterministiche.

- Problema di esplorazione (spazio degli stati non noto)

  Si presenta quando l'agente deve scoprire dinamicamente lo spazio degli stati, come in scenari online.

  Il nostro sistema, invece, lavora in modalità **offline**: l'intero grafo viene caricato all'avvio e mappato completamente.

Risoluzione mediante algoritmi di ricerca

Nel nostro caso, il problema si configura come una **ricerca su grafo noto**, dove ogni stato è un nodo, ogni azione è un arco, e la soluzione corrisponde a un cammino tra due nodi.

Il software adotta **algoritmi di ricerca ad albero (tree search)**, i quali:

- partono dal nodo iniziale,
- espandono iteravamente i nodi vicini (one-hop),
- terminano non appena trovano il nodo obiettivo o esauriscono le possibilità.

Questa strategia permette di confrontare diversi approcci di ricerca su uno stesso spazio degli stati, variando solo la logica di esplorazione adottata.

3.2. Metriche di valutazione degli algoritmi di ricerca

La ricerca di un percorso in un grafo può essere vista come un processo di esplorazione dello spazio degli stati. In questo contesto, è essenziale identificare il metodo più efficace per raggiungere l'obiettivo, scegliendo l'algoritmo più adatto in base al problema specifico.

Per confrontare le prestazioni degli algoritmi di ricerca implementati, si adottano le seguenti **metriche di valutazione**:

- **Completezza**

Un algoritmo è detto **completo** se garantisce di trovare una soluzione, qualora questa esista.
In presenza di più soluzioni possibili, un algoritmo completo è in grado di individuarne almeno una.

Questa caratteristica è particolarmente importante nei grafi molto estesi o con ramificazioni complesse.

- **Complessità spaziale e temporale**

Nella realtà, le risorse computazionali (tempo e memoria) sono limitate.
Pertanto, è utile stimare **il costo computazionale** di un algoritmo considerando:

1. **Maximum branching factor (b):** numero massimo di archi uscenti da un nodo (ampiezza del grafo).
2. **Maximum depth of the state space (m):** profondità massima del grafo (in nodi).
3. **Depth of the least-cost solution (d):** profondità della soluzione più economica (in termini di passi o costo cumulato).

La **complessità temporale** misura quante operazioni (iterazioni) servono per trovare una soluzione, mentre la **complessità spaziale** riflette la quantità di memoria necessaria per tenere traccia dei nodi visitati o da esplorare.

- **Ottimalità**

Un algoritmo è **ottimale** se è in grado di restituire sempre la **soluzione migliore** secondo un certo criterio.

Nel nostro progetto, tale criterio è il **costo totale del percorso** (es. somma dei pesi negli archi).

Un algoritmo ottimale garantisce che il percorso trovato sia il **meno costoso** tra quelli possibili.
Questo vale ad esempio per l'algoritmo di **Dijkstra**, che è ottimale per grafi con pesi non negativi.

3.3. Valutazione degli algoritmi

Con riferimento alle metriche di valutazione precedentemente descritte (completezza, complessità computazionale e ottimalità), si riportano di seguito le proprietà dei cinque algoritmi implementati nel progetto.

1) Breadth-First Search (BFS):

Descrizione: Algoritmo basato sulla ricerca in ampiezza. Esplora prima i nodi più vicini al nodo iniziale, ovvero quelli raggiungibili con il minor numero di passi. Questi sono i nodi raggiungibili attraversando meno nodi intermedi.

Memorizzazione: Utilizza una **coda FIFO** (First In, First Out).

Proprietà:

- **Completezza:** garantita se il branching factor b è finito.L'algoritmo è invece completo se b ha valore finito.
- **Complessità spaziale:** O(b^d), cresce rapidamente con la profondità.

- **Complessità temporale:** O(b^d), con d profondità della soluzione.
- **Ottimalità:** non garantita in generale. Tuttavia, se tutti i costi sono uguali (grafo non pesato), restituisce la soluzione ottima in numero di passi.

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

Descrizione: Algoritmo informato per la ricerca del cammino a costo minimo in un grafo con pesi non negativi.

Esplora i nodi in ordine crescente di costo dalla sorgente.

Proprietà:

- **Completezza:** sempre garantita.
- **Complessità spaziale:** O(n), con n numero di nodi.
- **Complessità temporale:** O((n + m) log n), con m numero di archi.
- **Ottimalità:** sempre garantita (per grafi con pesi ≥ 0).

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

Descrizione: Algoritmo che esplora in profondità prima di tornare indietro. Spinge la ricerca lungo un ramo fino al termine prima di esplorare i fratelli.

Memorizzazione: Utilizza una **pila LIFO** (Last In, First Out) o ricorsione.

Proprietà:

- **Completezza:** non garantita in presenza di cicli o profondità infinita.
- **Complessità spaziale:** O(b * m), con m profondità massima.
- **Complessità temporale:** O(b^m).
- **Ottimalità:** non garantita. L'algoritmo può trovare soluzioni molto lunghe.

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

Descrizione: Estensione di DFS in cui si impone un limite massimo di profondità (l).

Memorizzazione: Simile a DFS, ma interrotta al raggiungimento del limite (l).

Proprietà:

- **Completezza:** solo se la soluzione si trova entro il limite l.
- **Complessità spaziale:** O(l).
- **Complessità temporale:** O(b^l).
- **Ottimalità:** non garantita. Dipende dal limite impostato.

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

Descrizione: Algoritmo che esegue più DFS con profondità crescente, combinando i vantaggi di DFS e BFS.

Memorizzazione: Ripete ricerche in profondità da l = 0 fino a trovare la soluzione.

Proprietà:

- **Completezza:** sempre garantita.
- **Complessità spaziale:** O(b * d), con d profondità della soluzione.
- **Complessità temporale:** O(b^d).
- **Ottimalità:** non garantita in generale, ma ottima se i costi sono uniformi.

4. Risultati sperimentali

Per valutare in modo pratico il comportamento degli algoritmi implementati, sono stati eseguiti test su istanze reali del grafo di co-acquisto amazon0312-weighted.txt, misurando le seguenti metriche:

- **Tempo di esecuzione** (in secondi),
- **Numero di iterazioni** (ovvero espansioni di nodi),
- **Uso massimo della memoria** (es. lunghezza massima della coda/pila),
- **Lunghezza del percorso** (numero di nodi nel cammino),
- **Costo totale del percorso** (solo per Dijkstra).

L'obiettivo è confrontare le prestazioni dei diversi algoritmi in scenari concreti, osservando come variano le metriche al variare della distanza tra nodo di partenza e nodo di arrivo.

I test sono stati effettuati scegliendo coppie di nodi con distanza crescente (es. da nodo 2 a 420, 1500, 5000, ecc.) per simulare diversi livelli di complessità nel grafo.

Gli algoritmi testati sono:

1. **Breadth-First Search (BFS)**
2. **Uniform Cost Search (UCS)** – Dijkstra
3. **Depth-First Search (DFS)**
4. **Depth-Limited Search (DLS)**
5. **Iterative Deepening Search (IDS)**

I risultati ottenuti sono stati successivamente riportati in forma tabellare e visualizzati tramite grafici, con scala logaritmica, per facilitare il confronto tra gli algoritmi anche in presenza di ordini di grandezza differenti.

4.1. Algoritmi utilizzati e metodologia di valutazione

Per effettuare l'analisi comparativa, sono stati selezionati cinque algoritmi di ricerca non informata e informata, già illustrati nella parte teorica.
Ogni algoritmo è stato testato su coppie di nodi all'interno del grafo dei co-acquisti di Amazon, per valutare come si comporta in termini di efficienza e accuratezza.

1) Breadth-First Search (BFS)

Questo algoritmo esplora i nodi per livelli, garantendo la scoperta del cammino più breve in termini di numero di passi nei **grafi non pesati**.
È stato selezionato per valutare il comportamento ottimale in situazioni con bassa profondità e costi uniformi. L'utilizzo di una coda FIFO lo rende semplice ma costoso in termini di memoria su grafi molto ramificati.

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

L'algoritmo di Dijkstra è stato scelto per testare la ricerca ottimale in **grafi pesati con costi non negativi**. È stato utilizzato come riferimento per confrontare la qualità delle soluzioni trovate dagli altri algoritmi, grazie alla sua capacità di garantire sempre il **percorso di costo minimo**. L'implementazione impiega una coda di priorità per l'espansione dei nodi con costo cumulativo più basso.

3) Depth-First Search (DFS)

L'algoritmo di ricerca in profondità è stato incluso per valutarne le performance in termini di **uso di memoria ridotto** e comportamento su istanze con **soluzioni a bassa profondità**. Nonostante non sia ottimale, né completo in presenza di grafi infiniti, è utile per studiare scenari in cui l'esplorazione veloce di cammini profondi può portare rapidamente a una soluzione.

4) Depth-Limited Search (DLS)

Il DLS introduce un **vincolo esplicito sulla profondità massima** da esplorare.
È stato selezionato per verificare come la limitazione della profondità influenzi le prestazioni e la capacità di trovare soluzioni.
L'obiettivo è osservare i compromessi tra completezza e complessità computazionale in presenza di limiti imposti.

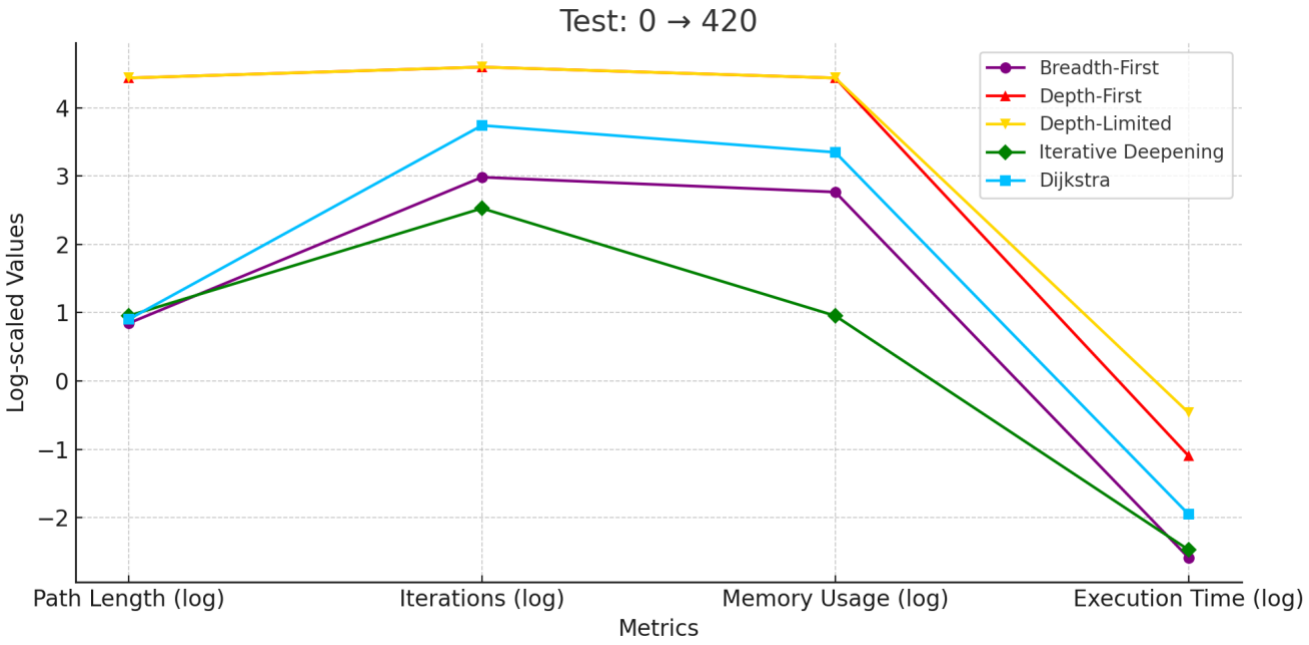
5) Iterative Deepening Search (IDS)

L'IDS combina i vantaggi della **ricerca in ampiezza** (completezza) con quelli della **ricerca in profondità** (efficienza nello spazio), eseguendo ripetutamente una Depth-Limited Search con profondità crescente.
È stato adottato per testarne l'efficacia nel trovare soluzioni senza dover definire a priori una profondità massima, e per valutarne il comportamento nei confronti dei costi computazionali derivanti dalla ripetizione dell'esplorazione.

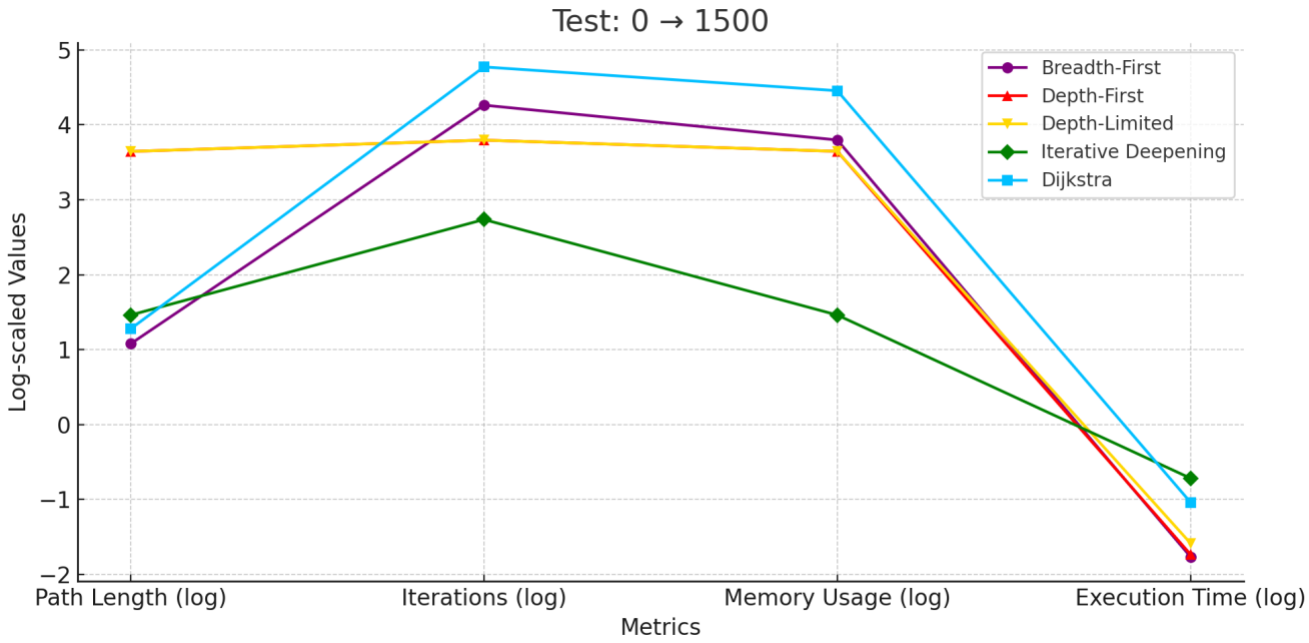
4.2. Test eseguiti

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

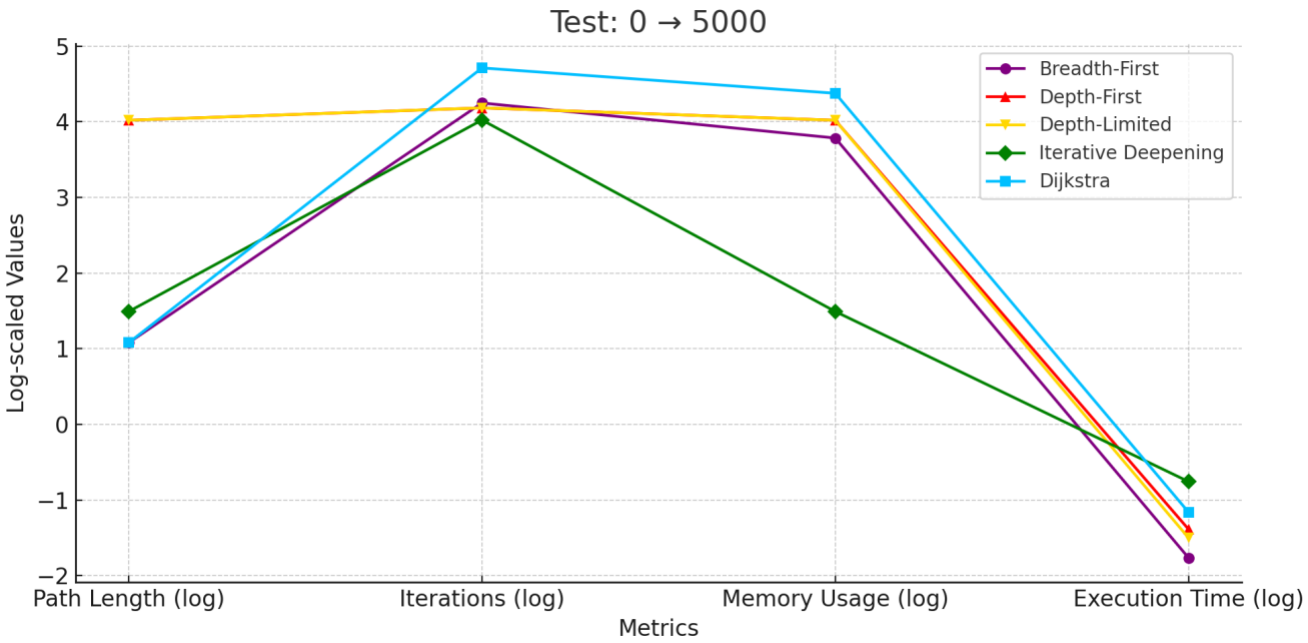
- Ricerca percorso dal nodo 0 al nodo 1500



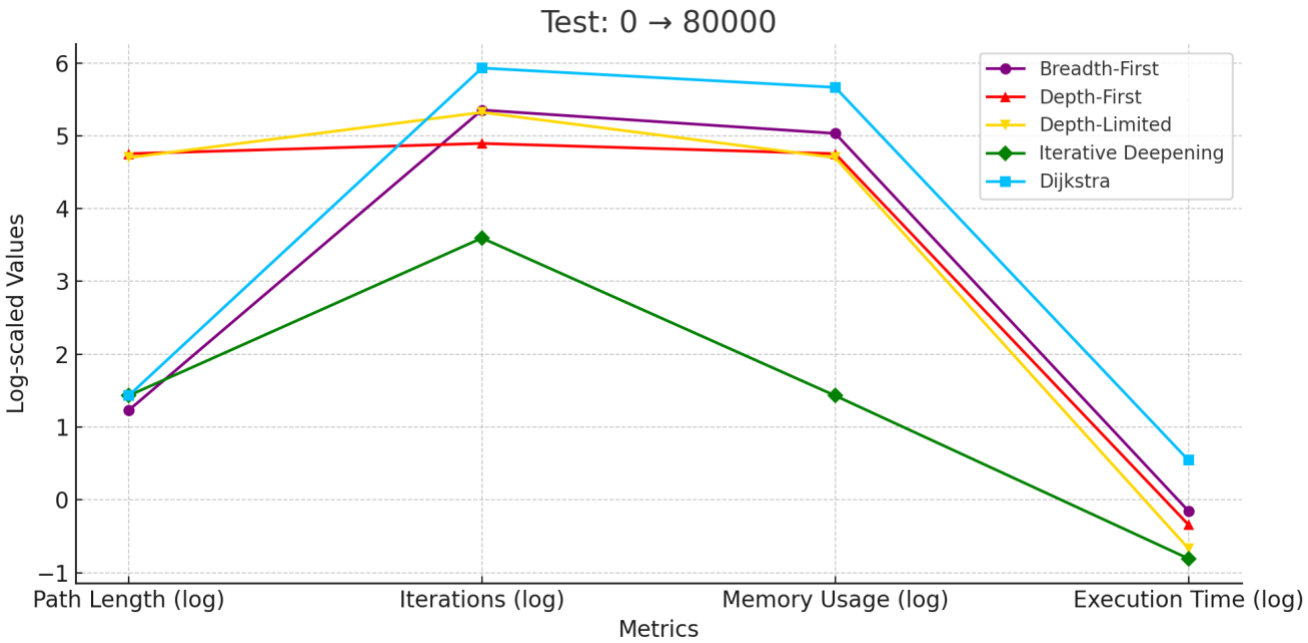
- Ricerca percorso dal nodo 0 al nodo 1500



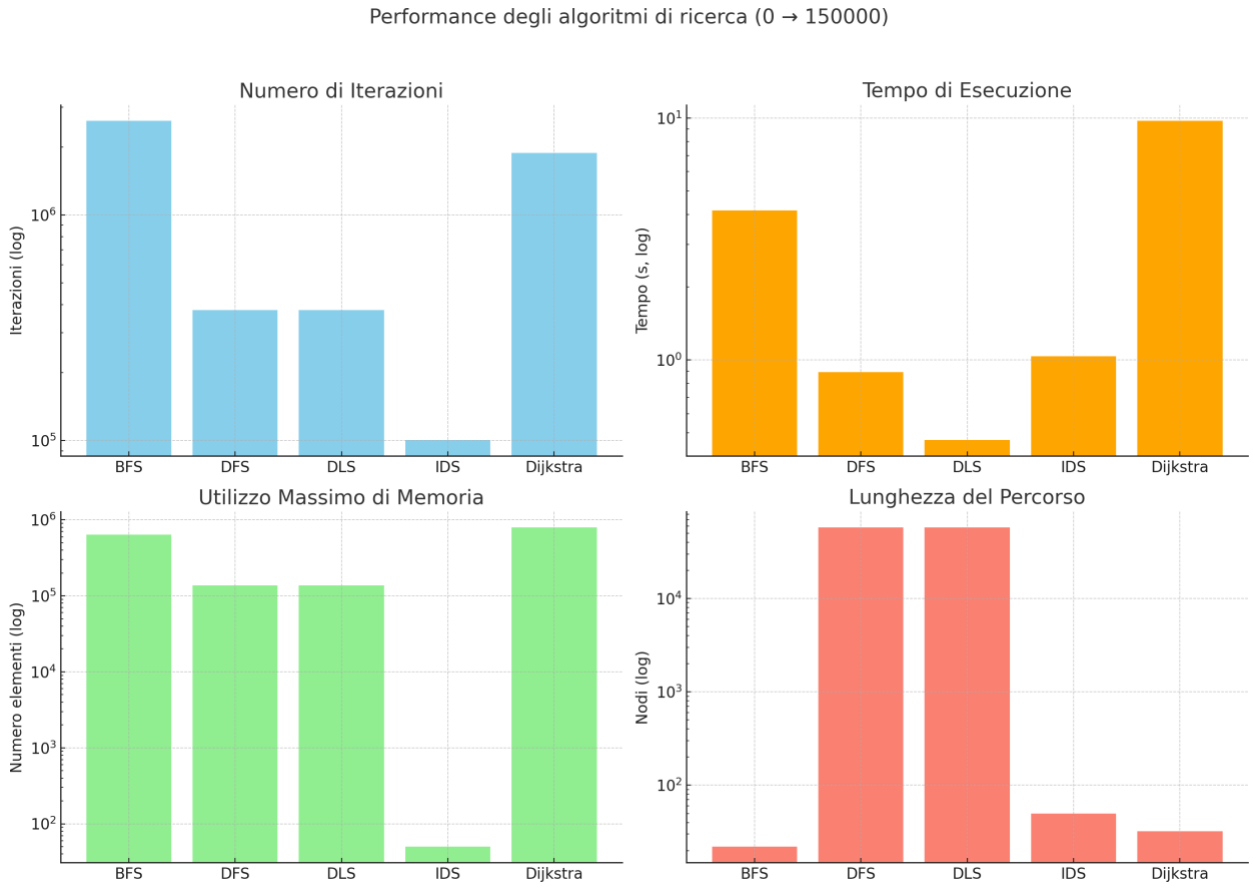
- Ricerca percorso dal nodo 0 al nodo 5000



- Ricerca percorso dal nodo 0 al nodo 80000



- Ricerca percorso dal nodo 0 al nodo 150000



Questi test sono stati progettati con l'obiettivo di mettere in evidenza le principali differenze prestazionali tra gli algoritmi implementati.

In particolare, si è voluto:

- Confrontare **IDS** e **DLS** in termini di **tempo di esecuzione**, a parità (o quasi) di profondità esplorata;
- Osservare l'efficienza di **BFS** rispetto a **Dijkstra**, evidenziando come il secondo riesca a individuare percorsi più economici in grafi pesati, a fronte però di un maggiore consumo di risorse;
- Analizzare il comportamento di **DFS** al crescere della profondità del nodo obiettivo, mettendo in luce i limiti di completezza e la sua tendenza a seguire percorsi lunghi e subottimali nei casi più complessi.

4.3. Considerazioni sui risultati

L'analisi sperimentale mostra che, al crescere della distanza tra nodo iniziale e nodo obiettivo (e quindi della complessità del problema), emergono con chiarezza le differenze prestazionali tra i vari algoritmi di ricerca.

- **Iterative Deepening Search (IDS)** e **Depth-Limited Search (DLS)**, pur condividendo la stessa logica di base (visita in profondità), evidenziano andamenti molto diversi. In particolare, IDS si distingue per la capacità di trovare sempre la soluzione (completezza), mentre DLS fallisce se la profondità impostata non è sufficiente. Tuttavia, IDS paga questa affidabilità con tempi di esecuzione sensibilmente maggiori, specialmente per destinazioni lontane.

- **Breadth-First Search (BFS)** garantisce completezza ed è in grado di trovare il percorso più breve nei grafi non pesati. Tuttavia, la sua richiesta di memoria aumenta esponenzialmente con la profondità del nodo obiettivo. In grafi estesi, questa caratteristica può rappresentare un limite significativo, ma per distanze brevi o moderate si è dimostrato estremamente competitivo.

- L'**algoritmo di Dijkstra** si è confermato ottimale su tutti i test eseguiti, trovando sempre il percorso minimo in termini di costo. Tuttavia, la sua efficienza temporale e spaziale degrada all'aumentare delle dimensioni del grafo, in particolare a causa dell'uso di una coda di priorità e della gestione dei costi parziali accumulati. Il numero di iterazioni e la memoria usata sono risultati i più alti nei test di lunga distanza.

- **Depth-First Search (DFS)** si è rivelato il meno efficace in termini di affidabilità. Nonostante consumi meno memoria rispetto a BFS e Dijkstra, ha fallito nel trovare una soluzione nei test con nodi obiettivo molto lontani (0–5000, 0–80000, 0–150000). Questo conferma che DFS non è completo: può perdersi in rami lunghi senza mai raggiungere il goal, specialmente in grafi con elevata ramificazione e profondità.

In sintesi, nessun algoritmo risulta il migliore in assoluto: ciascuno presenta vantaggi e limitazioni, che emergono con forza in base alla profondità del nodo obiettivo, alla struttura del grafo e alla presenza o meno di pesi sugli archi.