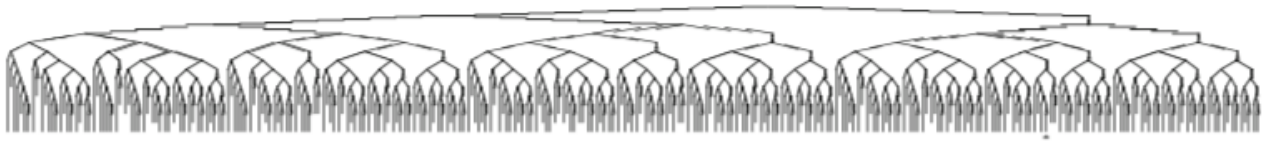
Intelligenza Artificiale – Lezione 5
8 maggio 2023

Search

La ricerca di una soluzione in un albero di ricerca estremamente grande presenta un problema per quasi tutti i sistemi di inferenza. Dallo stato iniziale ci sono molte possibilità per il primo passaggio di inferenza. Per ciascuna di queste possibilità ci sono ancora molte altre possibilità nel passaggio successivo, e così via. Anche nella dimostrazione di una formula molto semplice con tre clausole di Horn, ciascuna con al massimo tre letterali, l'albero di ricerca per la risoluzione SLD (Selection-rule-driven Linear resolution for Definite clauses) ha la seguente forma:



L'albero ha una soluzione nel nodo fogliare contrassegnato da *. È possibile rappresentarlo solo grazie al piccolo fattore di ramificazione (*branching* factor) di al massimo 2. Per problemi realistici, il fattore di ramificazione e la profondità della prima soluzione possono diventare significativamente maggiori.
Supponiamo che il fattore di ramificazione sia una costante uguale a 30 e che la prima soluzione sia a profondità 50. L'albero di ricerca ha $30^{50} \approx 7{,}2 \cdot 10^{73}$ nodi foglia. Ma il numero di passaggi di inferenza è ancora maggiore perché non solo ogni nodo foglia, ma anche ogni nodo interno dell'albero corrisponde a un passaggio di inferenza. Pertanto dobbiamo sommare i nodi su tutti i livelli e ottenere il numero totale di nodi dell'albero di ricerca:

$$\sum_{d=0}^{50} 30^d = \frac{1 - 30^{51}}{1 - 30} = 7.4 \times 10^{73}$$

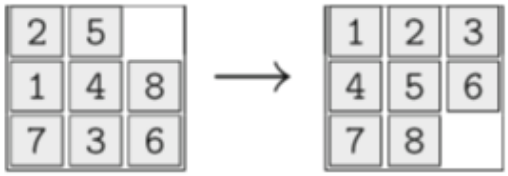
che non cambia di molto il conteggio dei nodi. Evidentemente, quasi tutti i nodi di questo albero di ricerca si trovano all'ultimo livello. Questo è un risultato generale che vale per quasi tutti gli alberi. Ma ora torniamo all'albero di ricerca con $7{,}4 \cdot 10^{73}$ nodi. Supponiamo di avere 10.000 computer che possono eseguire ciascuno un miliardo di inferenze al secondo e supponiamo di poter distribuire il lavoro su tutti i computer senza alcun costo (*overhead*). Il tempo di calcolo totale per tutte le inferenze $7{,}4 \cdot 10^{73}$ sarebbe approssimativamente uguale a

$$\frac{7.4 \times 10^{73} \text{ inferences}}{10000 \times 10^9 \text{ inferences/sec}} = 7.4 \times 10^{60} \text{ sec} \approx 2.3 \times 10^{53} \text{ years}$$

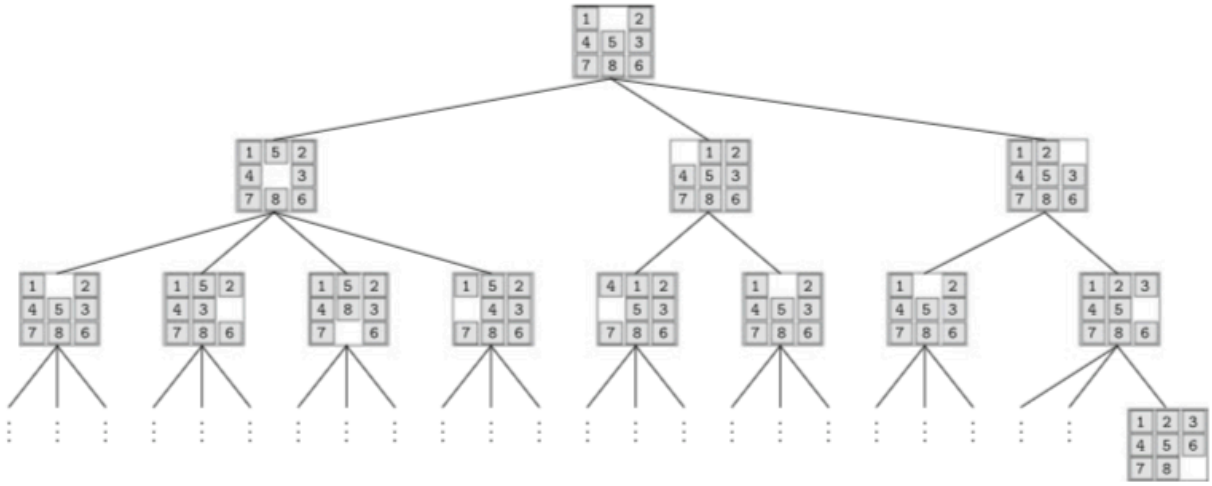
che è circa 10^{43} volte più tempo dell'età del nostro universo. Con questo semplice esercizio mentale, possiamo rapidamente riconoscere che non esiste alcuna possibilità realistica di esplorare questo tipo di spazio di ricerca completamente con i mezzi a nostra disposizione in questo mondo.

Come può essere allora che ci siano buoni giocatori di scacchi - e oggigiorno anche buoni computer di scacchi? Come può essere che i matematici trovino prove per teoremi in cui lo spazio di ricerca è ancora più grande? Evidentemente noi umani usiamo strategie intelligenti che riducono drasticamente lo spazio di ricerca. Il giocatore di scacchi esperto, proprio come il matematico esperto, con la semplice osservazione della situazione escluderà immediatamente molte azioni bollandole come prive di senso. Attraverso la sua esperienza, ha la capacità di valutare varie azioni per la loro utilità nel raggiungimento dell'obiettivo. Spesso, una persona si baserà sul suo istinto. Se uno chiede a un

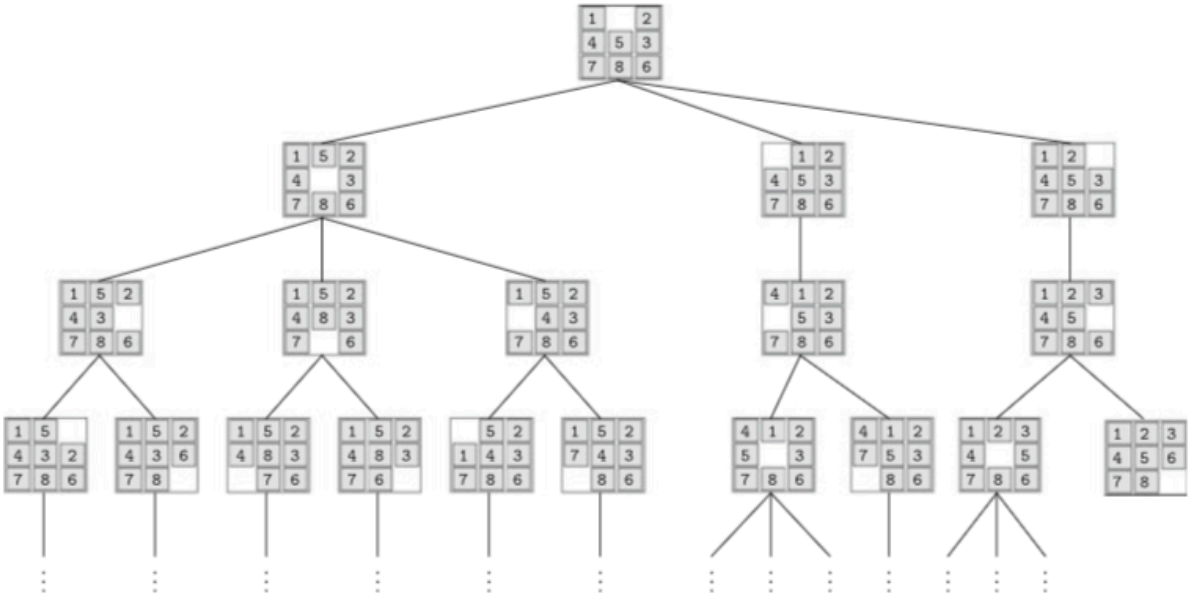
matematico come ha trovato una prova, può rispondere che l'intuizione gli è venuta in sogno. In casi difficili, molti medici trovano una diagnosi "a naso", basandosi su tutti i sintomi noti. Soprattutto in situazioni difficili, spesso non esiste una teoria formale per la ricerca di una soluzione che garantisca una soluzione ottimale.
Esistono processi con i quali i computer possono, analogamente agli esseri umani, migliorare le loro strategie di ricerca mediante conoscenze pregresse programmate o anche l'apprendimento.
Per prima cosa, però, dobbiamo capire come funziona la ricerca senza informazione (*uninformed search*), basata sul provare ciecamente tutte le possibilità. Cominciamo con un esempio.
In un 8-puzzle, quadrati con i numeri da 1 a 8 sono distribuiti in una matrice 3 x 3 come quella in figura. L'obiettivo è quello di raggiungere un certo ordine dei quadrati, ad esempio in ordine crescente per righe. In ogni passaggio un quadrato può essere spostato a sinistra, a destra, in alto o in basso nello spazio vuoto. Lo spazio vuoto si muove quindi nella corrispondente direzione opposta. Per l'analisi dello spazio di ricerca, è conveniente guardare sempre ai possibili movimenti del campo vuoto.



L'albero di ricerca per uno stato iniziale è rappresentato in figura (il goal è il nodo in fondo a destra):



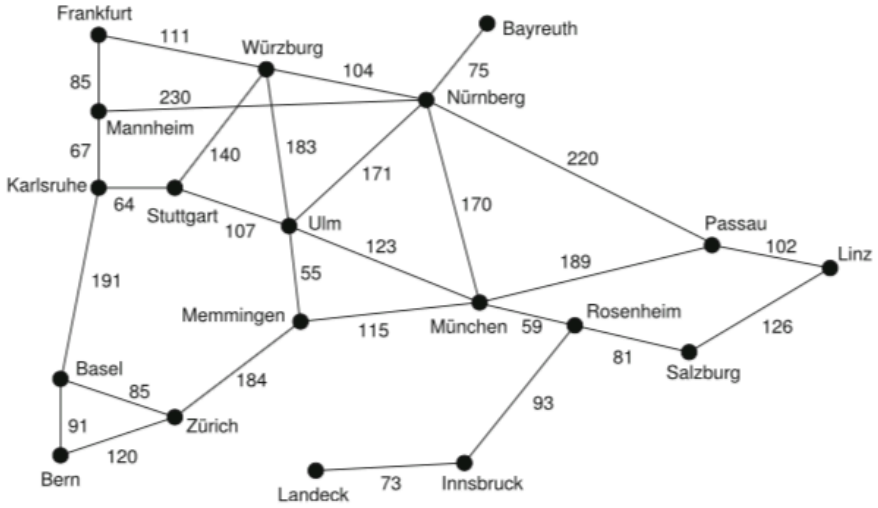
Notiamo che lo stato iniziale viene ripetuto più volte su due livelli più in basso di profondità perché in una semplice uninformed search, ogni azione può essere invertita nel passaggio successivo.
Se non permettiamo cicli di lunghezza 2 (in cui si ritorna a uno stato precedente annullando gli effetti della mossa appena fatta), allora per lo stesso stato iniziale otteniamo un albero di ricerca fatto come segue:



Prima di iniziare a descrivere gli algoritmi di ricerca, sono necessari alcuni nuovi termini. Abbiamo a che fare con problemi di ricerca discreti. Essendo nello stato *s*, un'azione a_1 porta a un nuovo stato *s'*. Quindi $s' = a_1(s)$. Un'azione diversa può portare allo stato *s''*, in altre parole: $s'' = a_2(s)$. L'applicazione ricorsiva di tutte le azioni possibili a tutti gli stati, a partire dallo stato iniziale, produce l'albero di ricerca.

Definizioni: Un *problema di ricerca* è definito dai seguenti valori:
Stato: descrizione dello stato del mondo in cui si trova l'agente di ricerca.
Stato iniziale: lo stato iniziale in cui viene avviato l'agente di ricerca.
Stato obiettivo: se l'agente raggiunge uno stato obiettivo, termina e genera una soluzione (se lo si desidera).
Azioni: il complesso delle mosse che l'agente può compiere.
Soluzione: il percorso nella struttura di ricerca dallo stato iniziale allo stato obiettivo.
Funzione costo: assegna un valore di costo a ogni azione. Tale funzione è necessaria se si vuole trovare una soluzione ottimale in termini di costi.
Spazio degli stati: insieme di tutti gli stati.
Fattore di ramificazione (*branching factor*): il numero di stati successori di uno stato *s*, indicato $b(s)$ o *b* se il fattore di ramificazione è costante.
Fattore di ramificazione effettivo: dato un albero di profondità *d* con *n* nodi totali, il suo fattore di ramificazione effettivo è definito come il fattore di ramificazione che avrebbe un albero con fattore di ramificazione costante, uguale profondità e uguale numero di nodi.
Completezza: un algoritmo di ricerca viene definito completo se trova una soluzione per ogni problema risolvibile. Se un algoritmo di ricerca completo termina senza trovare una soluzione, allora il problema è irrisolvibile.

Altro esempio:



Ci viene fornita una mappa, come quella rappresentata in figura, come un grafo con le città come nodi e le connessioni autostradali tra le città come lati ponderati con distanze. Stiamo cercando un percorso ottimale dalla città A alla città B. Formalizzando il problema come un problema di ricerca abbiamo:
Stato: una città come posizione corrente del viaggiatore.
Stato di partenza: una città arbitraria.
Stato obiettivo: una città arbitraria.
Azioni: viaggiare dalla città corrente a una città vicina.
Funzione di costo: la distanza tra le città.
Spazio degli stati: tutte le città, ovvero i nodi del grafo.

Ogni azione corrisponde a un lato nel grafo con la distanza come costo.
Per trovare il percorso di lunghezza minima bisogna tenere conto dei costi perché non sono costanti come lo erano nell'8-puzzle.

Definizione: Un algoritmo di ricerca è detto *ottimale* se, se esiste una soluzione, trova sempre la soluzione con il costo più basso.

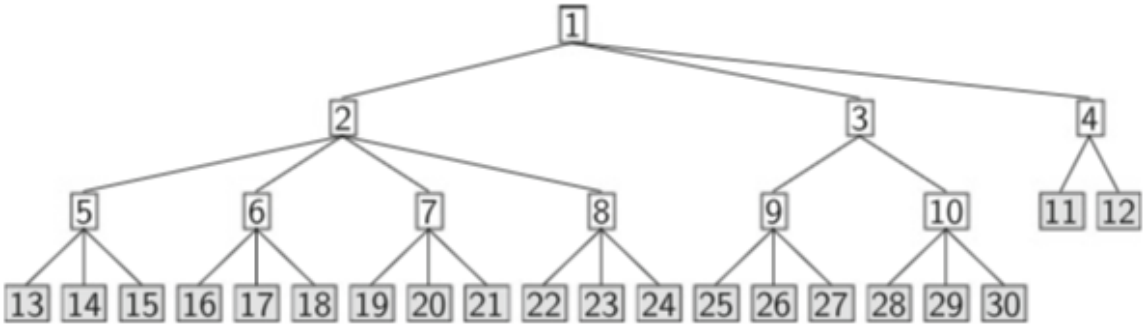
Il problema dell'8-puzzle è *deterministico*, il che significa che ogni azione porta da uno stato a uno stato successore unico. È inoltre *osservabile*, cioè l'agente sa sempre in quale stato si trova.
Nella pianificazione del percorso in applicazioni reali non sempre vengono fornite entrambe le caratteristiche. L'azione "guidare da Monaco a Ulm" può, ad esempio a causa di un incidente, portare allo stato successore "Monaco". Può anche accadere che il viaggiatore non sappia più dove si trova perché si è perso. All'inizio ignoreremo questo tipo di complicazioni. Pertanto in questa lezione esamineremo solo problemi che sono deterministici e osservabili.
Problemi deterministici e osservabili rendono la pianificazione delle azioni relativamente semplice perché, grazie a un modello astratto, è possibile trovare sequenze di azioni per la soluzione del problema senza eseguire effettivamente le azioni nel mondo reale . Nel caso dell'8-puzzle, non è necessario spostare effettivamente i quadrati nel mondo reale per trovare la soluzione. Possiamo trovare soluzioni ottimali con i cosiddetti algoritmi *offline*. Si affrontano sfide molto diverse quando, ad esempio, si costruiscono robot che dovrebbero giocare a calcio. Qui non ci sarà mai un modello astratto esatto delle azioni. Ad esempio, un robot che calcia la palla in una direzione specifica non può prevedere con certezza dove si muoverà la palla perché, tra le altre cose, non sa se un avversario prenderà o devierà la palla. Qui sono quindi necessari algoritmi *online*, che prendono decisioni in base ai segnali dei sensori in ogni situazione.

Uninformed Search

Breadth-first search
Nella "ricerca in ampiezza", l'albero di ricerca viene esplorato dall'alto verso il basso secondo l'algoritmo fornito in figura.

```
BREADTHFIRSTSEARCH(NodeList, Goal)
NewNodes = ∅
For all Node ∈ NodeList
    If GoalReached(Node, Goal)
        Return("Solution found", Node)
    NewNodes = Append(NewNodes, Successors(Node))
If NewNodes ≠ ∅
    Return(BREADTH-FIRST-SEARCH(NewNodes, Goal))
Else
    Return("No solution")
```

Prima ogni nodo nell'elenco dei nodi (NodeList) viene testato per verificare se si tratta di un nodo obiettivo e, in caso di successo, il programma viene interrotto. Altrimenti vengono generati tutti i successori del nodo. La ricerca viene quindi proseguita in modo ricorsivo nell'elenco di tutti i nodi appena generati. Il tutto si ripete fino a quando non vengono generati più successori.
Questo algoritmo è generico. Vale a dire, funziona per applicazioni arbitrarie se vengono fornite le due funzioni "GoalReached" e "Successors" specifiche all'applicazione. "GoalReached" calcola se l'argomento è un nodo obiettivo e "Successors" calcola l'elenco di tutti i nodi successori del suo argomento. La seguente figura mostra un'istantanea della ricerca in ampiezza.



Poiché la ricerca in ampiezza fa una ricerca completa di ogni profondità e raggiunge ogni profondità in un tempo finito, è completa se il fattore di ramificazione b è finito. La soluzione ottimale (cioè la più breve) si trova se i costi di tutte le azioni sono gli stessi. Il tempo di calcolo e lo spazio di memoria necessari crescono in modo esponenziale con la profondità dell'albero. Per un albero con fattore di ramificazione costante b e profondità d, il tempo di calcolo totale è dato da

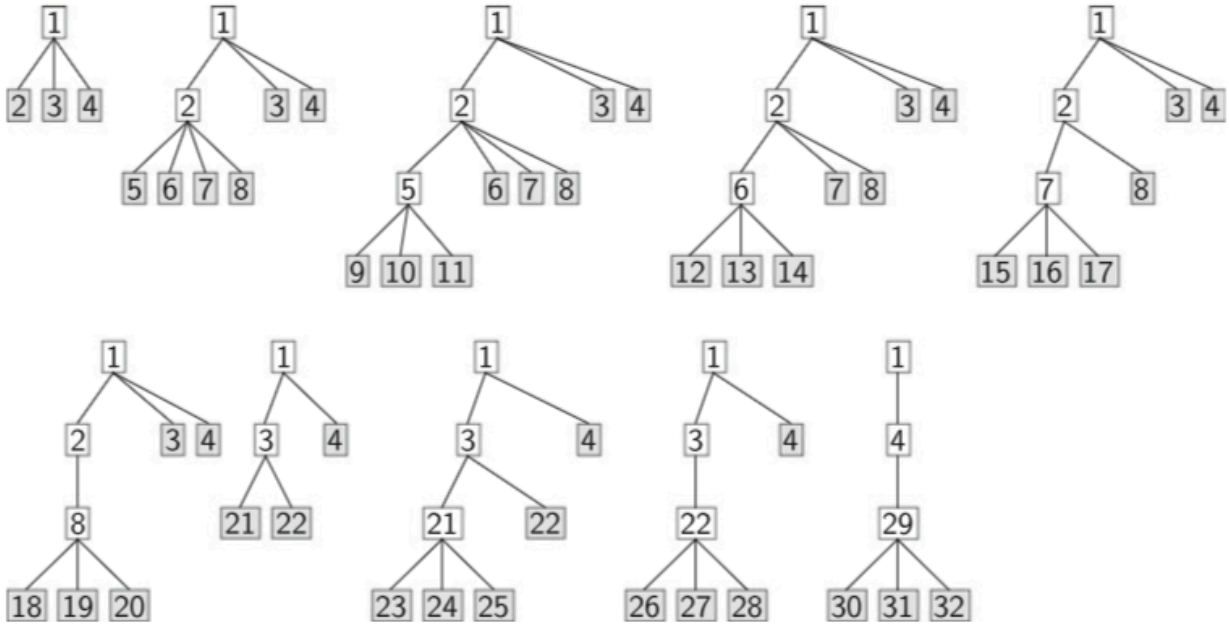
$$c \cdot \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d).$$

Sebbene solo l'ultimo livello venga salvato in memoria, anche lo spazio di memoria richiesto è $O(b^d)$.

Con la velocità dei computer odierni, che possono generare miliardi di nodi in pochi minuti, la memoria principale si riempie rapidamente e la ricerca termina. Il problema della memoria che si riempie non è stato ancora risolto. Una soluzione a questo problema è fornita dalla ricerca in profondità. Il problema della soluzione più breve non sempre trovata può essere risolto con la cosiddetta Uniform Cost Search, in cui viene sempre espanso il nodo con il costo più basso dalla lista dei nodi (che viene ordinata in ordine crescente per costo), e i nuovi nodi vengono ordinati pe costo. Così troviamo la soluzione ottimale.

Depth-first search
Nella ricerca in profondità vengono salvati in memoria solo pochi nodi alla volta. Dopo l'espansione di un nodo vengono salvati solo i suoi successori e il primo nodo successore viene immediatamente espanso. Così la ricerca diventa rapidamente molto profonda. Solo quando un nodo non ha successori e la ricerca non riesce a quella profondità, il successivo nodo aperto viene espanso tramite il backtracking fino all'ultimo ramo e così via. Il corrispondente algoritmo ricorsivo e l'albero di ricerca sono mostrati nelle figure sotto.

```
DEPTHFIRSTSEARCH(Node, Goal)
If GoalReached(Node, Goal) Return("Solution found")
NewNodes = Successors(Node)
While NewNodes ≠ Ø
    Result = DEPTH-FIRST-SEARCH(First(NewNodes), Goal)
    If Result = "Solution found" Return("Solution found")
    NewNodes = Rest(NewNodes)
Return("No solution")
```



In questa esecuzione della ricerca in profondità tutti i nodi a profondità 3 non sono obiettivo e causano il backtracking. I nodi sono numerati nell'ordine in cui sono stati generati.

La ricerca in profondità richiede molta meno memoria rispetto alla ricerca in ampiezza perché al massimo *b* nodi vengono salvati a ciascuna profondità. Quindi abbiamo bisogno al massimo di *bd* celle di memoria.
Tuttavia, la ricerca in profondità non è completa per alberi infinitamente profondi perché la ricerca in profondità dà luogo a un ciclo infinito quando non c'è soluzione nel ramo più a sinistra. Pertanto la questione di trovare la soluzione ottimale è obsoleta. A causa del ciclo infinito, non è possibile fornire alcun limite al tempo di calcolo. Nel caso di un albero di ricerca finitamente profondo con profondità *d*, vengono generati un totale di circa *bd* nodi. Così il tempo di calcolo cresce, proprio come nella ricerca in ampiezza, esponenzialmente con la profondità.
Possiamo rendere finito l'albero di ricerca impostando un limite di profondità. Ora, se non viene trovata alcuna soluzione nell'albero di ricerca ridotto, possono comunque esserci soluzioni al di là del limite impostato. Così la ricerca diventa incompleta. Ci sono tuttavia strategie per dare completezza alla ricerca.

Approfondimento iterativo (iterative deepening)
Iniziamo la ricerca in profondità con un limite di profondità di 1. Se non viene trovata alcuna soluzione, aumentiamo il limite di 1 e iniziamo la ricerca dall'inizio, e così via.
Dobbiamo arricchire il programma di ricerca in profondità fornito con due parametri aggiuntivi "Depth (Profondità)" e "Limit (Limite)". "Depth" viene aumentato di 1 al momento della chiamata ricorsiva, e la linea principale del ciclo while è sostituita da "While NewNodes≠∅ AND Depth < Limit". L'algoritmo modificato è rappresentato in figura.

```
ITERATIVEDEEPENING(Node, Goal)
DepthLimit = 0
Repeat
    Result = DEPTHFIRSTSEARCH-B(Node, Goal, 0, DepthLimit)
    DepthLimit = DepthLimit + 1
Until Result = "Solution found"
```

```
DEPTHFIRSTSEARCH-B(Node, Goal, Depth, Limit)
If GoalReached(Node, Goal) Return("Solution found")
NewNodes = Successors(Node)
While NewNodes ≠ ∅ And Depth < Limit
    Result =
        DEPTHFIRSTSEARCH-B(First(NewNodes), Goal, Depth + 1, Limit)
    If Result = "Solution found" Return("Solution found")
    NewNodes = Rest(NewNodes)
Return("No solution")
```

Il requisito di memoria è lo stesso della ricerca in profondità. Si potrebbe sostenere che il riavvio ripetuto della ricerca in profondità alla profondità zero causa molto lavoro ridondante. Per grandi fattori di ramificazione questo non è il caso. Si può dimostrare che la somma del numero di nodi di tutte le profondità fino a quella prima dell'ultima ($d_{max} - 1$) in tutti i sottoalberi precedentemente esplorati è molto inferiore al numero di nodi nell'ultimo albero esplorato. Il tempo di calcolo per tutte le iterazioni oltre all'ultima, quindi, può essere ignorato.
Proprio come la ricerca in ampiezza, questo metodo è completo e, dato un costo costante per tutte le azioni, trova la soluzione più breve.

La tabella sotto paragona i diversi algoritmi di ricerca uninformed:

	Breadth-first search	Uniform cost search	Depth-first search	Iterative deepening
Completeness	Yes	Yes	No	Yes
Optimal solution	Yes (*)	Yes	No	Yes (*)
Computation time	b^d	b^d	∞ or b^d	b^d
Memory use	$\boldsymbol{b^d}$	$\boldsymbol{b^d}$	$\boldsymbol{bd}$	$\boldsymbol{bd}$

(*) significa che l'affermazione è vera solo dato un costo di azione costante. d è la profondità massima per un albero di ricerca finito.

Controllo di cicli (cycle check)
Come mostrato nell'esempio dell'8-puzzle, alcuni nodi possono essere visitati ripetutamente durante una ricerca. Nell'8-puzzle ogni mossa può essere annullata immediatamente, il che porta a cicli non necessari di lunghezza 2. Tali cicli possono essere prevenuti registrando all'interno di ogni nodo tutti i suoi predecessori e, quando si espande un nodo, confrontando i nodi successori appena creati con i nodi predecessori. Tutti i duplicati trovati possono essere rimossi dall'elenco dei nodi successori. Questo semplice controllo costa solo un piccolo fattore costante di spazio di memoria aggiuntivo e aumenta il tempo di calcolo costante c di un'ulteriore costante δ per il controllo stesso per un totale di $c + \delta$. Questo sovraccarico per il controllo del ciclo è (si spera) compensato da una riduzione del costo della ricerca. La riduzione dipende, ovviamente, dalla particolare applicazione e quindi non può essere data in termini generali.