Uno stato contiene tutte le informazioni necessarie per prevedere gli effetti di un'azione e per determinare se uno stato soddisfa l'obiettivo. La ricerca nello spazio degli stati presuppone:

Che una soluzione sia una sequenza di azioni che porterà l'agente dal suo stato attuale a uno stato che soddisfa l'obiettivo.

Un problema nello spazio degli stati è costituito da

• un insieme di stati

• uno stato distinto chiamato lo stato iniziale

• per ogni stato, un insieme di azioni disponibili per l'agente in quello stato

• una funzione di azione che, dati uno stato e un'azione, restituisce un nuovo stato

• un obiettivo specificato come una funzione booleana, goal(s), questo è vero quando lo stato S soddisfa l'obiettivo ovvero è un goal.

• un criterio che specifica la qualità di una soluzione accettabile.

il problema di trovare una sequenza di azioni per raggiungere un obiettivo viene astratto come una ricerca di percorsi in grafi orientati DAG = direct acyclic graph.

Per risolvere un problema si necessita definire prima lo spazio di ricerca sottostante e in seguito applicare un algoritmo di ricerca a tale spazio. Molte attività di risoluzione dei problemi possono essere trasformate nel problema di trovare un percorso in un grafico.

La ricerca nei grafici fornisce un modello astratto appropriato di problem solving indipendente da un particolare dominio.

L'astrazione è necessaria perché potrebbe esserci più di un modo per rappresentare un problema come un grafo.

Un grafo orientato è costituito da

• un set N di nodi

• un set A di archi, dove un arco è una coppia ordinata di nodi.

In questa definizione, un nodo potrebbe essere qualsiasi cosa. Potrebbero esserci infiniti nodi e archi. Non assumiamo che un grafo sia rappresentato esplicitamente; abbiamo bisogno solo di una procedura per generare nodi e archi secondo necessità.

L'arco ⟨n1 , n2⟩ è un arco uscente da n1 e un arco in arrivo “o entrante” in n2.

Un nodo n2 è un vicino di n1 se c'è un arco da n1 a n2 ; cioè se ⟨n1 , n2⟩ ∈ A. Nota che essere un vicino non implica simmetria; solo perché n2 è un vicino di n1 non significa che n1 è necessariamente un vicino di n2.

Gli archi possono essere etichettati , ad esempio, con l'azione che porterà l'agente da un nodo all'altro o con il costo di un'azione o entrambi.

Un percorso dal nodo s al nodo g è una sequenza di nodi ⟨ n0 , n1 , … , nk ⟩ tale che s = n0, g = nk, e l’arco ⟨ ni-1 ,ni ⟩ ∈ A; cioè, c'è un arco da ni-1 a ni per ciascuno i tale che 0 < i <= k.

A volte è utile visualizzare un percorso come sequenza di archi, ⟨ n0 , n1 ⟩, ⟨ n1 , n2 ⟩ , … ,… ,⟨nk-1 , nk⟩ , o una sequenza di etichette di questi archi.

Sentiero ⟨ n0 ,n1 ,… ,ni⟩ è una parte iniziale di ⟨n0, n1, …, nk ⟩, Quando i ≤ K.

Un obiettivo è una funzione booleana sui nodi. Se goal(n) è vero, diciamo che il nodo n soddisfa l'obiettivo, ed n è un nodo obiettivo .

Per codificare i problemi come grafici, un nodo viene identificato come nodo iniziale . Una soluzione è un percorso dal nodo iniziale a un nodo che soddisfa l'obiettivo.

A volte c'è un costo - un numero non negativo - associato agli archi. Scriviamo il costo dell'arco ⟨ ni, nj ⟩ come cost(⟨ ni ,nj ⟩). I costi degli archi inducono un costo dei cammini.

Dato un percorso P = ⟨n0 , n1 , … ,nk⟩, il costo del percorso P è la somma dei costi degli archi nel percorso:

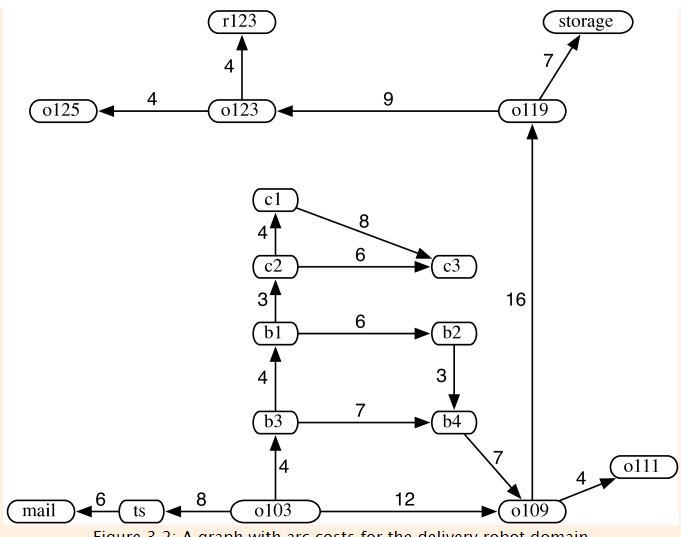
costo ( P ) = (k∑i=1)costo(⟨ni-1,ni ⟩) = costo ( ⟨n0 ,n1 ⟩ ) + … + costo ( ⟨nk-1, nk ⟩ ).

Una soluzione ottimale è una delle soluzioni che ha il costo più basso. Cioè, una soluzione ottimale è un percorso P dal nodo iniziale a un nodo obiettivo in modo tale che non vi sia alcun percorso P′ dal nodo iniziale a un nodo obiettivo dove costo ( P ′ ) < costo ( P ) cost(p′)<cost(p).

Un ciclo è un percorso non vuoto in cui il nodo finale è uguale al nodo iniziale, ovvero ⟨ n0 , n1 , … , nk ⟩ tale che n0 = nk . Un grafo diretto senza cicli è chiamato grafo aciclico diretto ( DAG ). Nota che questo dovrebbe essere chiamato un grafico diretto aciclico , perché è un grafico diretto che sembra essere aciclico, non un grafico aciclico che sembra essere diretto, ma DAG suona meglio di ADG!

Un albero è un DAG in cui c'è un nodo senza archi in entrata e ogni altro nodo ha esattamente un arco in entrata. Il nodo senza archi entranti è chiamato radice dell'albero. Un nodo senza archi uscenti è chiamato foglia .

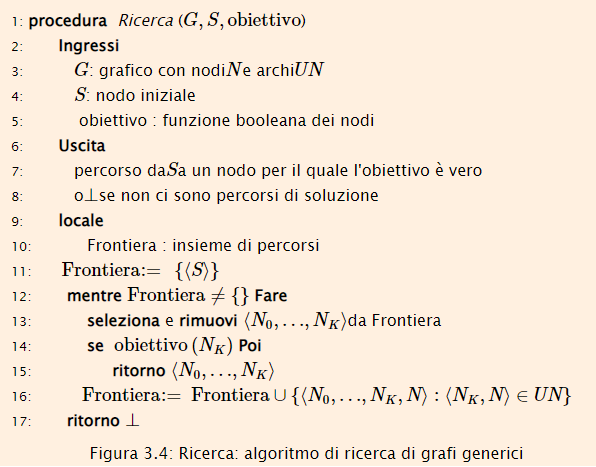
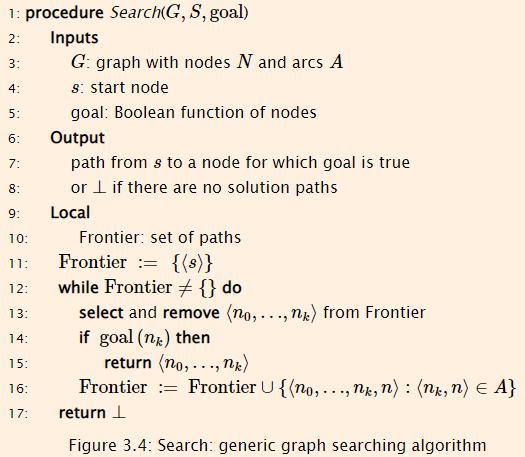
Il fattore di ramificazione in avanti di un nodo è il numero di archi in uscita dal nodo. Il fattore di ramificazione all'indietro di un nodo è il numero di archi in arrivo al nodo. Questi fattori forniscono misure per la complessità degli algoritmi su grafi.



Chart

Description automatically generated

L'idea intuitiva alla base dell'algoritmo di ricerca generico, dato un grafo, un nodo iniziale e un predicato obiettivo, è quella di esplorare i percorsi in modo incrementale dal nodo iniziale. Questo viene fatto mantenendo una **frontiera** (o **frangia** ) di percorsi dal nodo iniziale. La frontiera contiene tutti i percorsi che potrebbero formare segmenti iniziali di percorsi dal nodo iniziale a un nodo finale(ovvero parti della soluzione finale). Utilizzando diverse gestioni della frontiera in particolare implementandola attraverso diverse strutture dati è possibile avere differenti strategie di ricerca. Partendo dall’algoritmo base di ricerca:

Se il nodo alla fine del percorso selezionato non è un nodo obiettivo e non ha vicini, allora significa che il percorso sarà rimosso dalla frontiera. Questo risultato è ragionevole perché questo percorso non può far parte di un percorso dal nodo iniziale a un nodo obiettivo.

In una **procedura non deterministica**, fingiamo che un oracolo faccia ogni volta una scelta appropriata. Pertanto, un'affermazione **di scelta** si tradurrà in una scelta che porterà al successo o fallirà **se** non ci sono tali scelte. Una procedura non deterministica può avere più risposte, dove ci sono più scelte che hanno successo, e fallirà se non ci sono scelte applicabili.

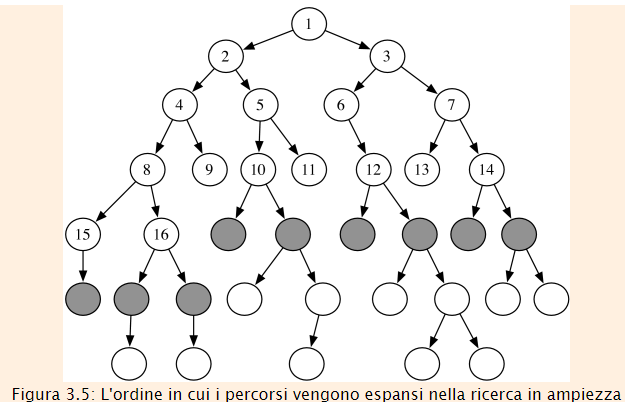
**STRATEGIE DI RICERCA NON INFORMATE**

Un problema va a determinare il grafo, il nodo iniziale e l'obiettivo, ma non quale percorso selezionare dalla frontiera. Questo è il compito di una **strategia di ricerca** . Una strategia di ricerca definisce l'ordine in cui i percorsi vengono selezionati dalla frontiera.

Le **strategie di ricerca non informate** non tengono conto della posizione dell'obiettivo. Intuitivamente, questi algoritmi ignorano dove stanno andando finché non trovano un obiettivo e ne segnalano il successo.

**BFS 🡪 Breadth first search (ricerca in ampiezza)**

Nella **ricerca in ampiezza** la frontiera è implementata come una **coda FIFO** (first-in, first-out). Pertanto, il percorso selezionato dalla frontiera è quello che è stato aggiunto per primo (supponiamo che questa aggiunta sia fatta in ordine dei figli di un nodo da sx a dx, ma cambiando tale criterio si possono ottenere differenze nel comportamento dell’algoritmo che rimane un BFS ma con diverso criterio di precedenza per l’espansione).



*Si noti come nella ricerca in ampiezza ogni percorso sulla frontiera abbia lo stesso numero di archi o un arco in più rispetto all'elemento successivo della frontiera che verrà selezionato.*

sia le complessità in spazio che in tempo sono esponenziali rispetto al numero di archi del percorso verso un obiettivo con il minor numero di archi. Questo metodo garantisce, tuttavia, di trovare una soluzione se ne esiste una e troverà una soluzione con il minor numero di archi.

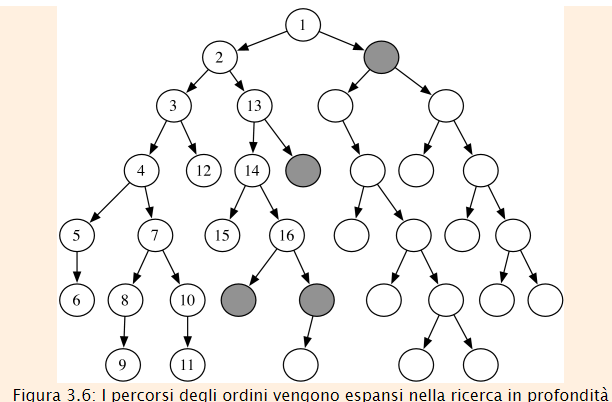
**Utilità della ricerca in ampiezza BFS:**

È utile quando il problema è di dimensioni ridotte per cui non comporta una eccessiva complessità spaziale dell’algoritmo nel dover generare il grafo di ricerca. Oppure si vuole una soluzione che in tal caso sia ottima rispetto al # di archi (quindi sia il percorso con meno archi possibili) e questo è garantito dall’algoritmo che troverà una tale soluzione se esiste ed è per questo che risulta **completo**.

È un metodo scadente quando tutte le soluzioni hanno molti archi(percorsi lunghi per arrivare al nodo goal) o sono disponibili alcune conoscenze euristiche. Non viene utilizzato molto spesso per problemi di grandi dimensioni in cui il grafico viene generato dinamicamente a causa della sua complessità spaziale esponenziale.

**DFS 🡪 Depth First Search (ricerca in profondità)**

Nella **ricerca in profondità**, la frontiera si comporta come una **pila** di percorsi **LIFO** (last-in, first-out). Usare una pila significa che il percorso selezionato e rimosso dalla frontiera in qualsiasi momento è l'ultimo percorso che è stato aggiunto al passo precedente. Questo permette di effettuare una visita lungo un ramo (o diramazione del grafo) e scendere lungo essa fino dove sarà possibile e il più velocemente possibile.



L'implementazione della frontiera come una pila si traduce in percorsi perseguiti in modo approfondito, cercando e testando un percorso fino al suo completamento prima di provare un percorso alternativo. Si dice che questo metodo implichi **il backtracking** : l'algoritmo seleziona una prima alternativa in ogni nodo e torna *all'alternativa* successiva quando ha seguito tutti i percorsi dalla prima selezione.

**Frontiera implementata come stack**

O103

P<ts>

ts B3 O109 p<b3>

p<o109>

dopo la prima iterata del DFS l’albero di ricerca avrà questa forma e la frontiera conterrà i percorsi che partono da o103 sino ai suoi vicini, adesso poiché la frontiera è gestita come uno stack sarà effettuato un pop(stack) il quale ci darà il percorso <o103, ts> che sarà espanso dall’algoritmo di ricerca nel seguente modo:

ts p<mail>

p<b3>

mail p<o109>

guardando all’albero di ricerca generato sul nostro grafo di ricerca avremo che l’algoritmo starà procedendo in questo modo:

o103

<o103,ts,mail>

ts b3 o109 <o103,b3>

<o103,o109>

mail

in questo una volta espanso il nodo mail esso sarà impilato nella frontiera e il successivo passo del DFS preleverà dalla frontiera con un pop proprio il percorso <o103,ts,mail> verificherà se mail è l’obiettivo e poiché non lo è andrà ad espanderlo, ma non avendo nodi figli non potrà essere espanso e tale percorso non sarà più inserito nella frontiera si dice allora che l’algoritmo esegue backtracking poiché la pila degli stati di ricerca lo farà tornare indietro affinché prenda un altro percorso percorribile ovvero quello in cima allo stack che risulta proprio <o103,b3> e continuerà ad espandere percorsi nello stesso modo fino a trovare una soluzione, oppure rimane incastrato in un loop ciclando indefinitivamente.

Se il fattore di ramificazione è B e il percorso selezionato sulla frontiera ha K archi, ci possono essere al massimo K\*(B−1) altri sentieri di frontiera. Questi sono (B−1)percorsi alternativi da ciascun nodo. Pertanto, per la ricerca in profondità, lo spazio utilizzato in ogni fase è lineare nel numero di archi dall'inizio al nodo corrente.

Se c'è una soluzione sul primo ramo cercato, la complessità temporale è lineare nel numero di archi nel percorso. Nel peggiore dei casi, la ricerca in profondità può rimanere intrappolata su rami infiniti e non trovare mai una soluzione, anche se ne esiste una, per grafi infiniti o per grafi con cicli. Se il grafo è un albero finito, con fattore di ramificazione in avanti minore o uguale a B e con tutti i percorsi fin dall'inizio avendo K o meno archi, la complessità temporale nel caso peggiore è O(Bk).

*La ricerca in profondità può essere migliorata potando*[*i percorsi con i cicli*](https://artint.info/2e/html/ArtInt2e.Ch3.S7.SS1.html)*.*

La ricerca in profondità è appropriata quando

* lo spazio è ristretto (ovvero abbiamo poca memoria)
* esistono molte soluzioni, forse con lunghi percorsi, in particolare per il caso in cui quasi tutti i percorsi portano a una soluzione o
* l'ordine in cui i vicini di un nodo vengono aggiunti allo stack può essere regolato in modo che le soluzioni vengano trovate al primo tentativo.

Mentre risulta scarsa quando il grafo presenta cicli, nel caso in cui esistano soluzione con percorsi brevi, questo perché l’algoritmo potrebbe addentrarsi per un percorso lungo prima di trovarla e quando si hanno rappresentazione con forte sbilanciamento del grafo (tanti nodi a sx dello start e pochi a dx).

**ITERATIVE DEEPENING 🡪 Algoritmo ad approfondimento iterativo**

Un modo per combinare l'efficienza spaziale della ricerca in profondità con l'ottimalità della ricerca in ampiezza consiste nell'utilizzare l' **approfondimento iterativo** . L'idea è di ricalcolare gli elementi della frontiera in ampiezza piuttosto che memorizzarli. L'approfondimento iterativo chiama ripetutamente un **cercatore limitato in profondità** , un cercatore in profondità che accetta un **limite di profondità** intero e non esplora mai percorsi con più archi di questo limite di profondità.

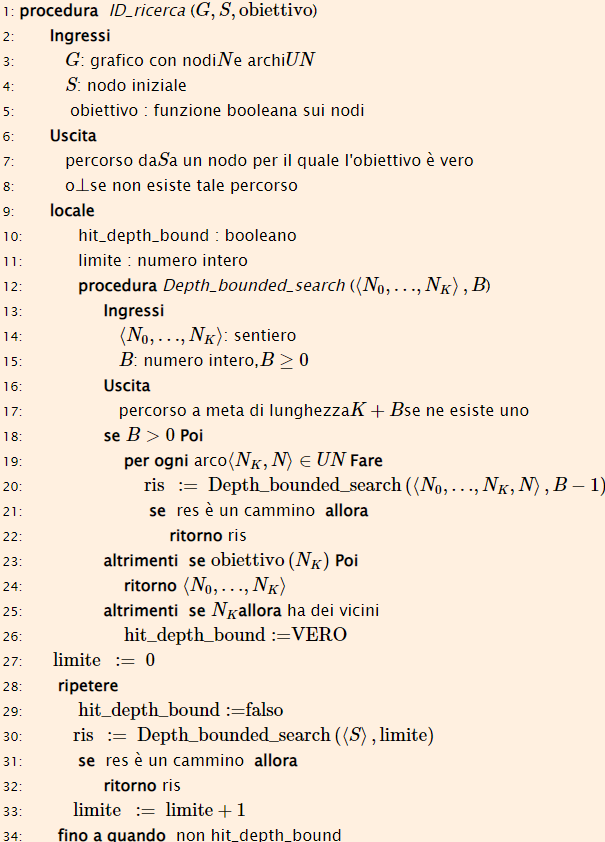
Quando una ricerca con profondità vincolata N non riesce a trovare una soluzione, può eliminare tutti i calcoli precedenti e ricominciare con un limite di profondità di N+1. Alla fine, troverà una soluzione se ne esiste una e, poiché sta enumerando i percorsi in ordine di numero di archi, verrà sempre trovato per primo un percorso con il minor numero di archi.

Per garantire l’arresto dell’algoritmo in grafi finiti, la ricerca ad approfondimento iterativo deve distinguere tra:

* fallimento perché è stato raggiunto il limite di profondità (**fallimento innaturale o temporaneo)**
* fallimento dovuto all'esaurimento dello spazio di ricerca**(fallimento naturale non esiste una soluzione).**

**Iterative deepening algoritmo**:

A picture containing text

Description automatically generated 

ID\_Search implementa una ricerca in profondità limitata alla profondità data come limite o bound(utilizzando la ricorsione per mantenere lo stack) che pone un limite alla lunghezza dei percorsi che sta cercando. Restituisce un percorso o raggiunge la fine del suo codice e ritorna senza percorso. Questo programma trova i percorsi ai nodi obiettivo nello stesso ordine della ricerca in ampiezza. Si noti che deve solo verificare la presenza di un obiettivo quando B=0 ovvero il bound è 0, perché sa che non ci sono soluzioni per limiti inferiori.

Questo algoritmo è O(Bk) e non può esserci una strategia di ricerca non informata asintoticamente migliore.

IDS TROVA I NODI OBBIETTIVO NELLO STESSO ORDINE IN CUI LI TROVEREBBE IL BFS QUESTO SIGNIFICA CHE L’ALGORITMO è COMPLETO E OTTIMALE

**LCFS 🡪 LOWEST COST FIRST SEARCH:**

Gli algoritmi di ricerca finora considerati non garantiscono di trovare i percorsi di costo minimo; infatti, non hanno utilizzato affatto le informazioni sui costi dell'arco. La ricerca in ampiezza trova prima una soluzione con il minor numero di archi, ma la distribuzione dei costi dell'arco può essere tale che un percorso con il minor numero di archi non sia quello di costo minimo.

Se vogliamo un algoritmo capace di garantire la soluzione a costo più basso utilizziamo Il LCFS ove invece di espandere un percorso con il minor numero di archi come avviene nel BFS, si seleziona un percorso con il minor costo totale dato dagli archi che lo costituiscono. Ciò viene implementato trattando la frontiera come una coda prioritaria ordinata dalla funzione di costo.

Poiché, il primo percorso verso un obiettivo che si espande è un percorso con il minor costo, il LCFS è capace di trovare soluzioni ottimali, perché l'algoritmo espande i percorsi dal nodo iniziale in ordine di costo del percorso. Se esistesse un percorso migliore verso un obiettivo rispetto alla prima soluzione trovata, sarebbe stato ampliato dalla frontiera per primo.

I metodi di ricerca visti precedentemente sono **disinformati** (o ciechi) in quanto non prendono in considerazione l'obiettivo fino a quando non espandono un percorso. Per migliorali si introduce la nozione di euristica come informazione aggiuntiva da utilizzare nella ricerca.

Una **funzione euristica** H(n), prende un nodo n e restituisce un numero reale non negativo che è una stima del costo del percorso meno costoso dal nodo n a un nodo obiettivo. La funzione H(n) è una funzione **ammissibile** se H(n) è sempre minore o al più uguale al costo effettivo di un percorso a basso costo dal nodo n ad un obiettivo.

La funzione euristica H() può essere estesa per essere applicata ai percorsi rendendo il valore euristico di un percorso uguale al valore euristico del nodo alla fine del percorso come si può vedere dall’equazione:

|  |  |
| --- | --- |
|  | H(⟨No,…,NK⟩)=H(NK) |

**RICERCA EURISTICA IN PROFONDITA’ 🡪 Heuristic DFS [stack ordinato secondo euristica]**

Un semplice utilizzo di una funzione euristica nella ricerca in profondità consiste nell'ordinare i vicini che vengono aggiunti allo stack che rappresentano la frontiera. I vicini possono essere aggiunti alla frontiera in modo che venga selezionato per primo il miglior vicino ove il miglior vicino è dato dalla funzione euristica che ne stimerà il costo. Questo è noto come **ricerca euristica in profondità** . Questa ricerca seleziona il percorso migliore a livello locale(tra i vicini il + promettente secondo l’euristica), ma esplora tutti i percorsi dal percorso selezionato prima di selezionare un altro percorso (poiché è implementata con uno stack a pila).

**GREEDY BFS 🡪 Greedy Best-First Search [coda prioritaria ordinata dalla funzione euristica]**

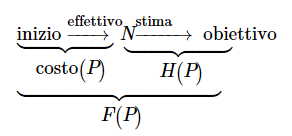
Un altro modo per utilizzare una funzione euristica consiste nel selezionare sempre un percorso sulla frontiera con il valore euristico più basso (in queste condizioni necessiteremo di una frontiera implementata come coda prioritaria). Questa si chiama **greedy best-first search** .

In entrambi questi ultimi 2 casi abbiamo problemi con gli algoritmi appena enunciati, infatti l’euristica applicata al DFS lo migliora quando lo spazio è sbilanciato, ma non risolve il problema della **non convergenza** alla soluzione a causa dei cicli infiniti. Il greedy best-first search invece nonostante in alcuni casi sia un ottima strategia da adottare, in altre situazioni potrebbe essere disastroso e prendere percorsi divergenti dalla soluzione.

**A\* Search:**

Per sfruttare l’euristica al meglio migliorando la ricerca l’A\* search utilizza una funzione per il calcolo della priorità che sarà assegnata ai percorsi presenti nella sua frontiera gestita come una coda prioritaria, per farlo utilizza il calcolo del costo degli archi(o della azione fatta) come avviene nel LCFS e in più utilizza la strategia greedy nella selezione del prossimo percorso da espandere.

Per qualsiasi percorso P sulla frontiera, A\* search definisce F(P) = costo(P) + H(P) .  Questa è una sotto-stima del costo totale del percorso P per raggiungere il nodo obiettivo. Se N è il nodo alla fine del percorso P, questo può essere rappresentato come:

****

Un algoritmo di ricerca è **ammissibile** se, ogni volta che esiste una soluzione, restituisce una soluzione ottima. Per garantire l'ammissibilità, devono valere alcune condizioni sul grafo e sull'euristica. Il seguente teorema fornisce condizioni sufficienti per A\* affinché sia ammissibile.

*A\* è ammissibile se quando c'è una soluzione,* restituisce quella ottimale *utilizzando la funzione euristica* H() e questo avviene se:

* *il fattore di ramificazione è finito*
* *tutti i costi degli archi sono maggiori di un certo* ϵ > 0
* H *è*[***un'euristica ammissibile***](https://artint.info/2e/html/ArtInt2e.Ch3.S6.html)*, il che significa che* H(N)*è minore o uguale al costo effettivo del percorso più economico dal nodo* N *a un nodo obiettivo.*

 L'ammissibilità garantisce che la prima soluzione trovata sarà ottimale anche in grafi con cicli. Non garantisce che l'algoritmo non cambi idea su quale percorso parziale sia il migliore durante la ricerca.

**IDA\* 🡪 Iterative Deepening A\* search**

L'approfondimento iterativo può essere applicato anche alla ricerca A\*. **Approfondimento Iterativo**A\* esegue ripetute ricerche DFS con limiti di profondità ove il limite anziché essere come nel classico Iterative Deepening un limite sul numero di archi nel percorso, è un limite sul valore della funzione greedy usata dall’ A\* ovvero un limite su F(n). La soglia è inizialmente il valore di F(s) dove s è il nodo iniziale. IDA\* quindi esegue una ricerca in profondità limitata dal valore di F(n) quindi non espande mai un percorso con un più alto valore rispetto al limite corrente impostato secondo la funzione greedy. Se la ricerca limitata alla profondità fallisce e il limite è stato raggiunto, il limite successivo quindi sarà impostato con il minimo valore di F(k) che ha superato il limite precedente posto.

Un aspetto importante riguarda come trovare un buona euristica, Il modo standard per costruirne una è trovare una soluzione a un problema più semplice, che è un problema con meno vincoli.

Un grafo che rappresenta uno spazio di ricerca può includere cicli, questo risulta essere un problema difficile da affrontare, infatti abbiamo visto come precedenti algoritmi vadano in loop infiniti ed altri invece perdano tempo per computare percorsi che ciclano per poi abbandonarli e trovare la soluzione quando esiste.

**CP 🡪 Cycle Pruning**

Un metodo semplice di potatura dei cicli, che garantisce che una soluzione venga trovata in un grafo finito, consiste nel non considerare i vicini che sono già sul percorso che si sta espandendo. **L'eliminazione del ciclo CP=cycle pruning** o **l'eliminazione del ciclo** controlla se l'ultimo nodo sul percorso espanso ovvero l’ultimo nodo aggiunto appare già tra gli altri nodi che formavano il percorso prima di essere espanso se così risulta tale percorso non viene aggiunto alla frontiera.  Percorsi ⟨n0,…,nk, n⟩, Dove n ∈ {n0,…,nk} non vengono aggiunti alla frontiera.

Per i metodi depth-first, l'overhead può essere basso come un fattore costante e questo è reso possibile memorizzando gli elementi del percorso corrente come un set (insieme dei visitati).  Per le strategie di ricerca che mantengono percorsi multipli, vale a dire tutti quelli con spazio esponenziale, il ciclo di potatura richiede un tempo lineare rispetto alla lunghezza del percorso cercato.

**MPP 🡪 Multiple-Path Pruning**

Spesso c'è più di un percorso per un nodo, un algoritmo di ricerca può eliminare dalla frontiera qualsiasi percorso che conduca a un nodo verso il quale ha già trovato un percorso in precedenza.

Il **MPP**  viene implementato mantenendo un insieme **set esplorato** (tradizionalmente chiamato **elenco chiuso =”closed list”** ) di nodi che si trovano alla fine dei percorsi che sono stati espansi. L'insieme esplorato è inizialmente vuoto. Quando un percorso⟨N0,…,NK⟩ è selezionato, se NK è già nell'insieme esplorato, il percorso può essere direttamente scartato. Altrimenti, NK viene aggiunto all'insieme esplorato(set explored o closed list) e l'algoritmo procede come prima.

Ora per implementare il **MPP** senza rischiare di non convergere alla soluzione ottimale dobbiamo effettuare alcuni accorgimenti:

* Assicurarsi che il primo percorso trovato per qualsiasi nodo sia un percorso a basso costo per quel nodo; quindi, elimina tutti i percorsi successivi trovati per quel nodo, avendo la garanzia di avere già salvato il percorso a costo minimo.

**Eliminazione dei percorsi verso un nodo a costo maggiore**

* Se l'algoritmo di ricerca trova un percorso verso un nodo a costo inferiore rispetto a uno già trovato, va a rimuovere tutti i percorsi che utilizzavano il percorso a costo più elevato verso il nodo considerato (poiché questi non possono trovarsi su una soluzione ottimale). Cioè, se c'è un percorso P nella frontiera ⟨S,…,N,…,M⟩, e un percorso P′ che conduca verso N (partendo da S start) e si riscontra che P′ ha un costo inferiore rispetto P considerato nel raggiungere N(ovvero considerando solo il percorso da S a N), allora P poiché sicuramente non sarà ottimale sarà eliminato dalla frontiera. Questo perché considerando che P fosse una soluzione parziale che conduca ad uno stato goal, esso non sarebbe mai il percorso ottimale poiché è possibile raggiungere il nodo N partendo dallo start S con un costo inferiore a quello speso dal percorso P e ciò significa che se continuiamo ad espandere P′ dopo N nello stesso modo in cui abbiamo fatto (o avremmo fatto per P) allora arriveremo sempre al goal, ma con una soluzione a costo minore e quindi migliore perciò P risulta inutile e viene eliminato.

**Sostituzione di parte dei percorsi in frontiera con sub-percorsi a costo inferiore**

* Ogni volta che la ricerca trova un percorso verso un nodo a basso costo rispetto a un percorso verso questo nodo già trovato, potrebbe incorporare una nuova sezione iniziale sui percorsi che hanno esteso il percorso iniziale. Quindi, se c'è un percorso P = ⟨S,…,N,…,M⟩ sulla frontiera, e un sentiero P′ a N si trova con un costo inferiore alla porzione di P da S a N, Poi P′ può sostituire la parte iniziale di P a N.

**LCFS permette associato a MPP di non perdere l’ottimalità nel convergere alla soluzione** Se prendiamo in considerazione la LCFS, il primo percorso trovato verso un nodo (ovvero, quando il nodo viene selezionato dalla frontiera) è il percorso con il costo più basso verso il nodo, per cui l'eliminazione dei percorsi successivi a quel nodo non può rimuovere un percorso a basso costo verso quel nodo, e quindi l'eliminazione dei percorsi successivi a ciascun nodo consente comunque di trovare una soluzione ottimale.

L’A\* non garantisce che la sua funzione greedy scelga per ogni nodo lungo il percorso il costo minimo questo è garantito solo dal teorema dell’ammisibilità di A\* per la soluzione ottimale che risulterà la meno costosa, né consegue che se vogliamo applicare il MPP ad A\* ci serve qualcosa che ci garantisca di non rimuovere sub percorsi ottimali per il goal. Per farlo si utilizza la proprietà di **Coerenza**  della funzione euristica, la quale viene garantita dalla restrizione monotona, ovvero:

H(N) ≤ costo(N,N′) + H(N′)

**Diagram

Description automatically generated**

Con la restrizione monotona, i valori di F(n) (funzione usata dal A\* data da costo + stima) dei percorsi selezionati dalla frontiera sono monotonicamente non decrescenti(ovvero il successivo può essere solo maggiore o al più uguale al precedente valore di F(n)). Cioè, quando la frontiera si allarga, i valori di F(n) non diminuiscono. Per cui *con un'euristica coerente, la potatura a più percorsi (MPP) non può mai impedire*A\* *la ricerca di una soluzione ottimale.*

***Utilizzo del MPP e CP con algoritmi di ricerca BFS e DFS:***

L'eliminazione a percorsi multipli (MPP) include l'eliminazione del ciclo, poiché un ciclo è un altro percorso verso un nodo e pertanto viene eliminato. L'eliminazione di percorsi multipli può essere eseguita in tempo costante, impostando un bit su ciascun nodo a cui è stato trovato un percorso se il grafo è memorizzato in modo esplicito o utilizzando una funzione hash.

 La potatura a percorsi multipli(MPP) è preferita alla potatura ciclica(CP) per i metodi in ampiezza in cui praticamente tutti i nodi considerati devono essere comunque archiviati. La ricerca in profondità non deve memorizzare tutti i nodi alla fine dei percorsi già espansi per cui memorizzarli per implementare la potatura a percorsi multipli rende esponenziale la ricerca in profondità nello spazio. Per questo motivo, la potatura ciclica è preferita alla potatura a percorsi multipli per la ricerca in profondità.

Per IDA\* la MPP non è consigliata poiché la complessità spaziale nel mantenere la closed list (l’insieme dei nodi già esplorato) farebbe aumentare così tanto la sua complessità da renderlo poco efficace.

**TABELLA DI RIEPILOGO E CONFRONTO ALGORITMI DI RICERCA:**

**Table

Description automatically generated**

 Gli algoritmi che garantiscono di trovare un percorso con il minor numero di archi o il minor costo sono completi. Un algoritmo di ricerca è **completo** se è garantito che trovi una soluzione se ce n'è una.

**Depth-First Branch-and-Bound Search:** è un modo per combinare il risparmio di spazio della ricerca depth-first con informazioni euristiche per trovare percorsi ottimali. È particolarmente applicabile quando ci sono molti percorsi verso un obiettivo.La ricerca branch-and-bound genera una sequenza di soluzioni in continuo miglioramento. La soluzione finale trovata è la soluzione ottimale. Può essere implementato in modo simile alla ricerca con limiti di profondità, dove il limite qui è in termini di costo del percorso e si riduce man mano che vengono trovati percorsi più brevi.

Text

Description automatically generated

Text

Description automatically generated

 La procedura interna cbsearch , per la ricerca cost-bounded, utilizza le variabili globali per fornire informazioni alla procedura principale. Inizialmente, bound può essere impostato su infinito, ma spesso è utile impostarlo su una sovrastima, “limite0”, del costo del cammino di una soluzione ottima. Questo algoritmo restituirà una soluzione ottimale. Se il limite iniziale è leggermente al di sopra del costo di un percorso a minor costo, questo algoritmo può trovare un percorso ottimale espandendo non più archi di A\* senza potatura a più percorsi.

**Shape, arrow

Description automatically generated**

Questo è il grafo di ricerca generato dal DF-B&B search i nodi ombreggiati sono tutti nodi obiettivi il costo per ogni arco è unitario ed uguale per tutti i nodi. *ll sottoalbero sotto il nodo numerato "5" non ha un obiettivo e viene esplorato completamente (o fino in profondità* Bound0 *se avesse un valore finito). Il nodo “9” verificato è un nodo obiettivo. Ha un costo di percorso pari a 5(# di archi), quindi il limite è impostato a 5(new Bound1=5). Da quel momento in poi, solo i percorsi con un costo inferiore a 5 vengono verificati per essere una soluzione. Anche il quindicesimo nodo controllato è un obiettivo. Ha un costo di percorso pari a 3, quindi il limite è ridotto a 3(new Bound2=3). Non sono stati trovati altri nodi obiettivo, quindi viene restituito il percorso al nodo etichettato 15 che è un percorso ottimale. C'è un altro percorso ottimale che viene potato; l'algoritmo non controlla mai i figli del nodo etichettato con 18, ma questo perché esso avrebbe costo 3 uguale al precedente e l’algoritmo controlla solo i nodi se sono obiettivo o meno per costi inferiori quindi a costo 2 e poiché 18 non è obiettivo l’algoritmo non continua.*

Quindi possiamo dire che il DF\_B&B search si basa sui seguenti 2 meccanismi:

1. **BRANCH:**  un meccanismo (o processo) per la selezione dei percorsi più promettenti, dato da una funzione euristica che andrà ad ordinare la frontiera di modo che sul ramo più promettente si vada in modalità esplorativa DFS.
2. **BOUND:** un meccanismo che permetta di valutare un percorso in espansione e se esso supera il limite fissato venga eliminato dalla frontiera di ricerca in modo da non essere più esplorato.

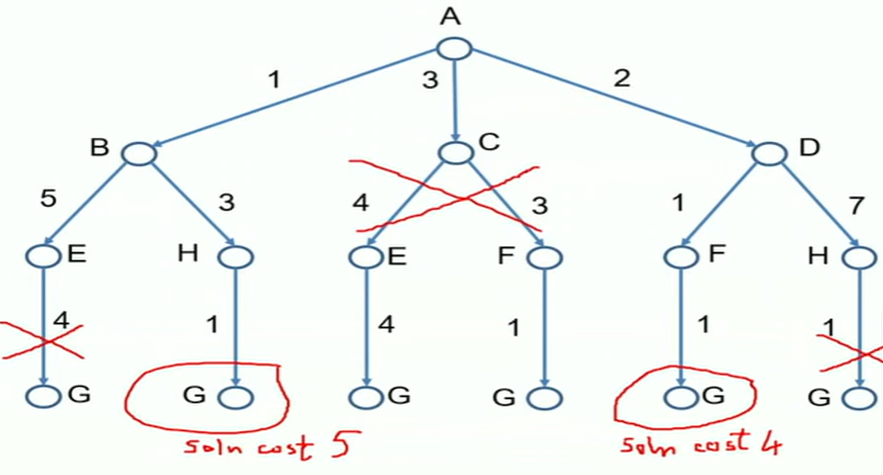
Chart

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Come euristica per il branch useremo il costo più basso verso un nodo (quindi il costo più basso del percorso verso il nodo verso dove vogliamo espandere il percorso), mentre per il bound adottiamo il costo totale del percorso trovato che inizialmente è stato posto a infinito.



Quando l’algoritmo inizierà ad espandere la sua frontiera, ricordiamo che il DF\_B&B in quanto lavora come DFS adotterà una gestione della frontiera a stack ovvero Last in first out, per cui l’inserimento dei nuovi percorsi espansi dovrà fare in modo che in cima allo stack ci sia sempre il percorso a costo minore tra quelli esplorati(serve una struttura ausiliare per ordinare tali percorsi e poi impilarli nello stack). Quindi quando inizia l’algoritmo metterà nella pila [<a,c>3, <a,d>2, <a,b>1] a questo punto sulla cima dello stack c’è <a,b> poiché il costo(<a,b>) = 1 è il minore e sarà espanso aumentando i percorsi nella pila: [<a,c>, <a,d>, <a,b,e>, <a,b,h>] quindi <b,h> in cima allo stack poiché costo(<b,h>) = 4 è il minore !!! Attenzione non tra tutti i percorsi impilati ma tra quelli espansi lungo il ramo verso il cui stiamo andando in profondità, per cui è il min(costo(<a,b,e>), costo( <a,b,h>))!!!. Estratto <a,b,h> andrà ad espanderlo con l’unico possibile nodo g ottenendo il percorso <a,b,h,g> che risulta essere la soluzione poiché h è goal in questo momento l’algoritmo non si arresta, ma aggiorna il suo upper bound da infinito a 5 = costo(<a,b,h,g>). In questo momento tale percorso trovato come soluzione è rimosso dalla frontiera e ricordiamo che la frontiera è uno stack per cui è come se stesse effettuando backtracking lungo il ramo esplorato per vedere se c’è qualcosa di meglio in termini di costi che chiaramente non ci sarà poiché il costo(<a,b,3>) = 6 > 5 e anche questo sarà eliminato dalla frontiera il prossimo in cima allo stack sarà <a,d> con costo(<a,d>) = 2 < 5 per cui il DF\_B&B continua ad espandere con i due possibili nodi che portano ai percorsi <a,d,h> scartato poiché il suo costo > 5 e <a,d,f> che sarà impilato nello stack poiché il suo costo è 3 e continuerà fino a trovare il goal h con percorso <a,d,f,g> con costo totale = 4 < 5 in questo momento aggiornerà l’upper bound e continuerà la sua ricerca effettuando backtracking e scarterà i restanti percorsi poiché superano il nuovo bound quando la frontiera sarà vuota sarà restituita quella trovata con costo = 4 che risulta essere ottimale.

L’idea dietro il DF\_B&B è quella di schiacciare la soluzione tra l’upper bound che viene iterativamente ad essere diminuito e il lower bound dato dalla costruzione del percorso per la soluzione diminuendo il primo e aumentando il seconda la soluzione ottimale che si troverà tra di essi sarà trovata nel momento in cui i due bound sono allineati

**Graphical user interface, application

Description automatically generated with medium confidence**

Icon

Description automatically generated

* Entrambi trovano la soluzione ottima
* Entrambi hanno complessità lineare grazie al limite posto dal bound
* IDA\* non andrà mai ad espandere un percorso con un nodo per il quale la funzione di scelta sia > dell’ottimo
* DF\_B&B non espanderà mai un nodo 2 volte, ma nel processo di abbassamento del upper bound può espandere percorsi sub ottimali che portano a soluzioni sub ottimali. Questo algoritmo fallisce su un grafo infinito!!
* IDA\* migliore di DF\_B&B se il grafo è infinito, ma se ho un idea buona dell’upper bound in tal caso posso settare questo al posto di infinito e in tal caso DF\_B&B sarebbe migliore rispetto a IDA\*

**DIREZIONE DELLA RICERCA**

Se valgono le seguenti condizioni:

* l'insieme dei nodi obiettivo è noto{tutti gli N : goal(N) ovvero è una soluzione}
* Per qualsiasi nodo n { N′: ⟨N′,N⟩ ∈ A appartiene all’insieme degli archi}

l'algoritmo di ricerca del grafo può iniziare con il nodo iniziale e cercare in avanti un nodo obiettivo, oppure iniziare con un nodo obiettivo e cercare all'indietro il nodo iniziale.

Nella **ricerca all'indietro** , la frontiera inizia con un nodo etichettato obiettivo. I vicini dell'obiettivo sono i nodi dell'obiettivo, { N : goal(N)}

**La ricerca in avanti** cerca dal nodo iniziale ai nodi obiettivo nel grafo originale.

 La dimensione dello spazio di ricerca è tipicamente esponenziale nel fattore di ramificazione. Accade spesso che le ricerche in avanti e all'indietro abbiano diversi fattori di ramificazione. Un principio generale è cercare nella direzione che ha il fattore di ramificazione minore.

**RICERCA BIDIREZIONALE:**

L'idea della **ricerca bidirezionale** consiste nel cercare simultaneamente in avanti dall'inizio e all'indietro rispetto alla meta. Quando le due frontiere di ricerca si intersecano, l'algoritmo deve costruire un singolo percorso che si estenda dal nodo iniziale attraverso l'intersezione della frontiera fino a un nodo finale.

Un nuovo problema sorge durante una ricerca bidirezionale, vale a dire garantire che le due frontiere di ricerca si incontrino effettivamente. Infatti, avremo poche chance se da entrambi i lati lavoriamo con un DFS mentre potremmo aver buone probabilità con un BFS.

**RICERCA GUIDATA DA ISOLE (le isole sono interpretate come conoscenza aggiuntiva)**

Uno dei modi in cui la ricerca può essere resa più efficiente consiste nell'identificare un numero limitato di punti in cui la ricerca in avanti e la ricerca all'indietro potrebbero incontrarsi.  Questi nodi prendono il nome di ISOLE e permettono di scomporre il problema in diversi sotto problemi in modo da mantenere a bada la complessità esponenziale andando ad affrontare problemi di complessità minore.

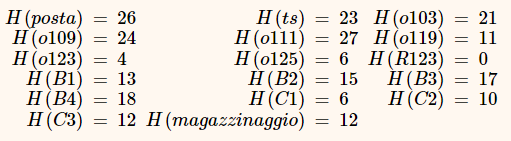
 La ricerca dell'isola sacrifica l'ottimalità a meno che non si sia in grado di garantire che le isole si trovino su un percorso ottimale.

**RICERCA IN UNA GERARCHIA DI ASTRAZIONI:**

L'efficacia della ricerca in una gerarchia di astrazioni dipende da come si scompone e si astrae il problema da risolvere. Una volta che i problemi sono astratti e scomposti, qualsiasi metodo di ricerca può essere utilizzato per risolverli. Non è facile, tuttavia, riconoscere utili astrazioni e scomposizioni di problemi

**PROGRAMMAZIONE DINAMICA:**

**La programmazione dinamica** è un metodo generale per l'ottimizzazione che implica il calcolo e la memorizzazione di soluzioni parziali ai problemi. Le soluzioni che sono già state trovate possono essere recuperate invece di essere ricalcolate.

****

**Diagram

Description automatically generated**

**Diagram

Description automatically generated**