2. Ricerca di Soluzioni in Spazi di Stati

Dispensa ICon

versione: 19/10/2024, 12:27

Risoluzione di Problemi mediante Ricerca · Spazi di Stati · Ricerca su Grafo · Algoritmo di Ricerca Generico · Strategie di Ricerca Non Informate · Ricerca Informata (Euristica) · Potatura dello Spazio di Ricerca · Strategie di Ricerca più Sofisticate

1 Risoluzione di Problemi mediante Ricerca

La risoluzione di diversi problemi spesso consiste nella *ricerca* in uno specifico modello del mondo.

Si consideri un caso semplice, quello di un sistema che, dato un *obiettivo* da raggiungere, ragiona su un *modello* del mondo fatto di *stati*, in *assenza di incertezza*. A tal fine, si supponga di adottare una rappresentazione *piatta*, i.e. non gerarchica, ovvero relativa a un singolo livello in una gerarchia.

Astrazione del problema: in un *grafo orientato*, ricerca di un *percorso* che vada da un nodo di partenza a uno dei nodi-obiettivo (*goal*). Molti problemi possono essere mappati su quest'astrazione per la quale sono disponibili diversi algoritmi di ricerca.

Esempio — *Navigatore*

L'obiettivo è la ricerca del *miglior* percorso da un luogo a un altro:

- il più *corto* / minima distanza;
- quello di minimo *costo* (ad esempio in termini di carburante, tempo, ecc.);
- il più *veloce*;
- il più *attrattivo* (ad esempio con più POI, *point-of-interest*);

Ogni *stato* include informazioni su:

- la localizzazione,
- la direzione,
- la velocità,
- il mezzo di trasporto utilizzato, ...

Ricerca

Una strategia comune a molti problemi in AI è la seguente: il sistema lavora (computazione) su una sua *rappresentazione interna* (diversamente dalla ricerca operata da robot che *agisce* direttamente sul mondo fisico e dalla ricerca sul web che mira al ritrovamento di informazioni). Il sistema riceve solo una descrizione di *COSA* rappresenti una soluzione; *COME* ottenerla sarà il compito di un algoritmo di ricerca. Si osservi che molto spesso tali problemi e i relativi algoritmi risolutivi sono NP-completi, anche nei casi in cui sia disponibile un modo *efficiente* per riconoscere le soluzioni.

Tali problemi risultano *difficili* anche per agenti umani. Spesso non esistono soluzioni ottimali generali per cui ci si accontenta di soluzioni *soddisfacenti* in mancanza d'una

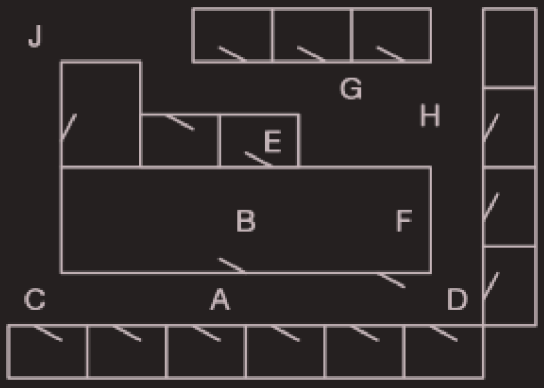
struttura mappabile su caratteristiche del mondo fisico. Ad esempio nella *crittografia* si richiede che i problemi risultino estremamente difficili anche per macchine, date le comuni limitazioni di tempo e spazio. Come si vedrà più avanti, converrà sfruttare conoscenza aggiuntiva (*euristica*), se disponibile.

2 Spazi di Stati

Formuliamo il problema in termini di uno **spazio di stati**. Uno **stato** deve contenere tutta l'informazione necessaria a predire gli effetti di un'azione (ad esempio una mossa in un gioco) e a verificare se esso soddisfi l'obiettivo (*stato goal*). Una **soluzione** sarà una sequenza di azioni che portano dallo stato corrente a un goal (fine-percorso → soluzione).

Le *assunzioni* comuni che si fanno sono la conoscenza *perfetta* dello spazio, ossia che lo stato corrente sia noto e il mondo sia *completamente osservabile*. Essenziale è anche l'esistenza di stati-obiettivo, ossia *goal* da raggiungere (facilmente) *riconoscibili*. Si assume, inoltre, che a disposizione del sistema/agente ci sia un insieme di *azioni* dagli effetti *deterministici* noti.

Esempio — Un robot che effettua consegne in un edificio dovrà essere in grado risolvere *problemi di ricerca* di un percorso da un luogo ad un altro:

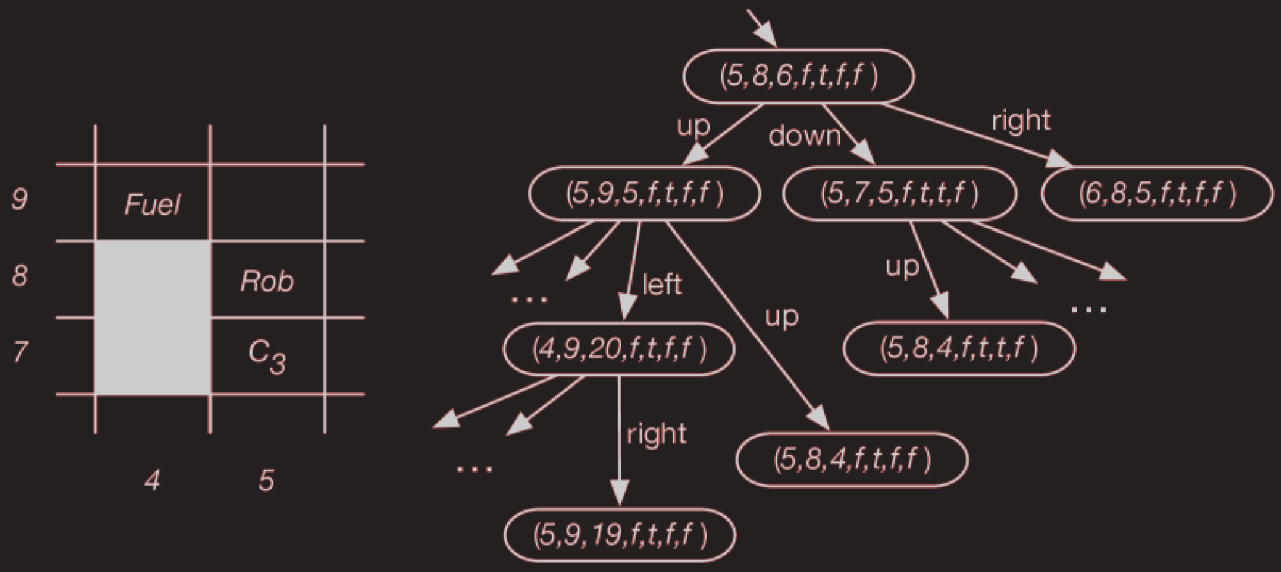


occorrerà definire:

- *stati*: posizioni distinte (stanze/esterni-stanza);
- *azioni*: spostamenti da un luogo a un altro nelle vicinanze;
- *problema*: portarsi da A a G (unico stato-obiettivo);
- *soluzione*: sequenza di spostamenti;

Esempio — Videogame (cfr. figura)

- *modello*: griglia dove ci si può muovere di una casella nelle 4 direzioni;
  - se non si è bloccati da un muro (caselle scure);
- *obiettivo*: raccogliere 4 monete $C_1, \ldots, C_4$, ognuna in una posizione iniziale nota
  - ad esempio C_3 in $(5, 7)$
- *costo*: un'unità di carburante/fuel per ogni passo:
  - nessuna mossa è possibile senza carburante;
  - rifornimento presso una casella *stazione-carburante* (+20 unità), ad esempio in $(4, 9)$;
- *stato*: dato da una posizione x, y, dalle unità di carburante u possedute e da flag c_i che indicano il possesso della relativa moneta
  - esempio $(x, y, u, c_1, c_2, c_3, c_4)$
  - possibile goal $(1, 1, ?, t, t, t, t)$ ($?$ = il quantitativo di carburante non conta)



Esempio — Sistema di *tutoring* (lettura da [PM23])

Un **problema di ricerca** in uno *spazio di stati* comprende:

- lo *spazio di stati*: insieme di *stati*
  - uno *stato* (o un sotto-insieme di stati) distinto **di partenza**;
  - un insieme di **azioni** possibili per ciascuno stato;
- la **funzione d'azione**: data una coppia (stato, azione) calcola il nuovo stato in cui si transita per effetto dell'azione;
- un *obiettivo*, funzione booleana $goal(s)$ che ha valore `vero` sse s è uno *stato obiettivo*
  - in alternativa si possono più semlicemente elencare gli stati-obiettivo, se in numero finito;
- un **criterio** di *qualità* per soluzioni *accettabili*: una **soluzione ottimale** massimizza tale criterio
  - ad esempio si ammette *qualsiasi* sequenza di azioni che porti a uno stato-obiettivo
  - ad esempio associando *costi* alle diverse azioni da cui discende un criterio del *minimo costo totale* delle azioni; si noti che, a volte, sono soddisfacenti anche soluzioni sub-ottimali, ad esempio quelle con un costo aggiuntivo del +10% rispetto a quello di una soluzione ottimale.

Possibili *estensioni* del problema sono le seguenti:

- possibilità di sfruttamento di una *struttura interna* agli stati:
  - ad esempio nei videogame;
- caso di stati *non* completamente *osservabili*:
  - ad esempio il robot-consegne non conosce la posizione iniziale delle consegne
  - ad esempio il sistema di tutoring può non conoscere bene le attitudini di uno studente
- problemi che ammettono azioni *stocastiche*:
  - ad esempio possibilità di commettere errori (robot-consegne);
  - ad esempio mancato apprendimento da parte dello studente di un argomento;
- in vece degli stati finali si possono definire *preferenze aggiuntive* complesse, in termini di *ricompense* o *punizioni*.

3 Ricerca su Grafo

Per risolvere il problema, si definisce lo spazio di ricerca e si applica un algoritmo di ricerca. Come anticipato, molti compiti possono essere ricondotti alla *ricerca* di percorsi in un grafo.

Un modello *astratto* della soluzione sarà *indipendente* dal particolare dominio applicativo di interesse. Dato un *grafo* (*orientato*) fatto di nodi connessi da archi, si dovrà trovare un *percorso* (di più archi) tra un nodo di partenza e uno obiettivo.

> ⚠ Ci possono essere più maniere per rappresentare il problema.

Un **grafo orientato** *esplicito* o anche *implicito*, se è disponibile una procedura per generare nodi/archi, consiste di:

- un insieme di *nodi* N, anche non finito;
- un insieme di *archi* A, ossia di coppie *ordinate* nodi, anche non infinito;
  - l'arco $\langle n_1, n_2 \rangle$ si dirà **uscente** da n_1 ed **entrante** in n_2;
    esso può anche essere etichettato con l'*azione* che porta da n_1 a n_2;
  - si dirà che il nodo n_2 è un **vicino** del nodo n_1 sse $\exists \langle n_1, n_2 \rangle \in A$
    (si noti che la relazione *non* è necessariamente *simmetrica*);
- un **percorso** (o *cammino*) dal nodo s al nodo g denotato da $\langle n_0, n_1, \ldots, n_k \rangle$ è una *sequenza* di nodi tale che: $s = n_0$, $g = n_k$ e $\forall i = 1, \ldots, k$: $\langle n_{i-1}, n_i \rangle \in A$;
  in alternativa, si può indicare una *sequenza di archi* $\langle n_0, n_1 \rangle, \langle n_1, n_2 \rangle, \ldots, \langle n_{k-1}, n_k \rangle$ o anche *sequenza di etichette* su tali archi: $\langle n_0, \ldots, n_i \rangle$ *parte iniziale* di $\langle n_0, n_1, \ldots, n_k \rangle$, con $i \leq k$.

Nel grafo esiste un insieme dei *nodi-obiettivo* (*goal*) che si possono identificare anche tramite il predicato $goal(\cdot)$, funzione booleana definita sui nodi. Una **soluzione** è un percorso dal nodo di partenza a uno obiettivo.

In alcuni casi è associato un **costo** a ogni arco: dato $\langle n_i, n_j \rangle$

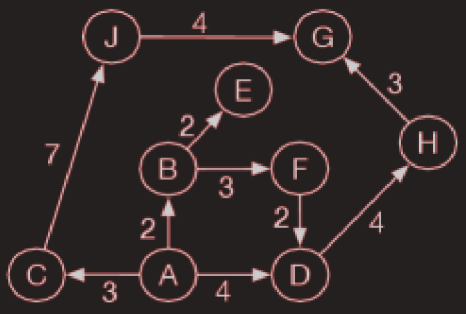
$$cost(\langle n_i, n_j \rangle) \in [0, +\infty[$$

estendibile anche ai percorsi: dato $p = \langle n_0, n_1, \ldots, n_k \rangle$

$$cost(p) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle) = cost(\langle n_0, n_1 \rangle) + \cdots + cost(\langle n_{k-1}, n_k \rangle)$$

Una soluzione **ottimale** p avrà costo minimo: $\nexists p'$ soluzione con $cost(p') < cost(p)$.

Esempio — Robot consegne (cont.): ricerca di un percorso da A a G nel mondo rappresentato nella figura precedente



- $N = \{A, B, C, D, E, F, G, H, J\}$

- $A = \{\langle A, B\rangle, \langle A, C\rangle, \langle A, D\rangle, \langle B, E\rangle, \langle B, F\rangle, \langle C, J\rangle \ldots\}$ dove:
  - E non ha vicini;
  - C ha come vicino solo J;
  - A ha i vicini B, C e D;
  - vi sono 3 percorsi da A a G: $\langle A, D, H, G\rangle$, $\langle A, C, J, G\rangle$, $\langle A, B, F, D, H, G\rangle$; se A fosse nodo di partenza e G goal, ognuno di tali percorsi sarebbe una soluzione.

Esercizio — Trovare una soluzione *ottimale* per il problema precedente.

Un **ciclo** è un percorso non vuoto in cui primo e ultimo nodo coincidono:
$\langle n_0, n_1, \ldots, n_k\rangle$ tale che $k > 0$ e $n_0 = n_k$.

Un **grafo aciclico orientato** (*directed acyclic graph*, DAG) è un grafo orientato senza cicli (come quello nella figura precedente).

Un **albero** è un DAG con un solo nodo, la **radice**, senza archi entranti, mentre tutti gli altri ne hanno esattamente uno. I nodi senza archi uscenti sono le sue **foglie**. Si definiscono relazioni fra nodi, come ad esempio genitore-figlio, fratello-sorella, antenato, ecc..., prese in prestito dalla metafora dell'*albero-genealogico*.

La complessità dei grafi si misura spesso in termini dei **fattori di ramificazione** per i nodi quello **uscente** (*forward*), ovvero il numero di archi uscenti e quello **entrante** (*backward*) ossia numero degli archi entranti. Sono utili a discutere la *complessità* degli algoritmi. Nel seguito si assumerà che siano *limitati superiormente* da una costante. Essi determinano le *dimensioni* del grafo, ad esempio un albero con fattore uscente b per ogni nodo, avrà b^n nodi a distanza di n archi per ciascuno.

Esempio — robot consegne (cont.) grafo in figura

- per A fattori di ramificazione uscente ed entrante 3 e 0, rispettivamente;
- per B, 2 e 1;
- per D, 1 e 2.

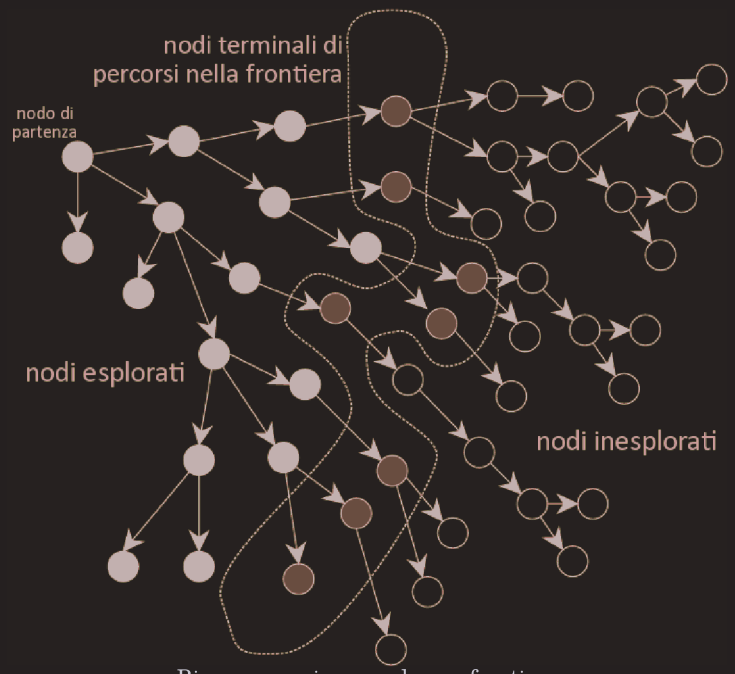
---

4 Algoritmo di Ricerca Generico

Si tratta di un algoritmo indipendente da strategia di ricerca e/o dal grafo. Dato un grafo, si esplorano *incrementalmente* percorsi dai nodi di partenza verso nodi-obiettivo. Si usa una struttura dati per rappresentare la **frontiera** (*fringe*) dei percorsi già esplorati, i.e. segmenti iniziali di percorsi da completare fino a un goal.

Inizialmente la frontiera contiene percorsi costituiti dai soli *nodi di partenza*. Successivamente si effettua l'*espansione* dei percorsi nella frontiera verso nodi inesplorati, fino a incontrare un goal:

- si seleziona un percorso (rimuovendolo dalla frontiera);
- si estende il percorso con ogni arco uscente dall'ultimo nodo;
- si aggiungono alla frontiera i percorsi ottenuti.



Ricerca generica usando una frontiera

```
procedure Search(G, s, goal)
    Input
        G: grafo con insiemi di nodi N e di archi A
        s: nodo di partenza
        goal: funzione booleana sui nodi
    Output
        percorso da s a un nodo per il quale goal sia vera
        oppure ⊥ se non ci sono percorsi/soluzioni
    Local
        frontier: insieme di percorsi

    frontier ← {⟨s⟩}
    while frontier ≠ ∅ do
        selezionare ⟨n_0, ..., n_k⟩, rimuovendolo, da frontier
        if goal(n_k) then
            return ⟨n_0, ..., n_k⟩
        frontier ← frontier ∪ {⟨n_0, ..., n_k, n⟩ : ⟨n_k, n⟩ ∈ A}
    return ⊥
```

Osservazioni

- la selezione *non è deterministica* ([1]), ha solo impatto sull'efficienza quindi una data strategia di selezione determina il percorso da scegliere;
- `return` annidato interpretabile come *temporaneo*, continuando si trovano strade alternative;
- $\bot$ indica che non vi sono (altre) soluzioni;
- $goal(s_k)$ viene testato *dopo* la selezione dalla frontiera, non all'aggiunta del nuovo nodo:
    1. a volte esiste un *arco* verso un goal ma *di costo elevato*: non sempre conviene terminare restituendo subito il percorso trovato potrebbe esserci un percorso di costo inferiore; ciò può essere importante per la minimizzazione di tale costo;
    2. va tenuto conto del fatto che il test stesso può essere *costoso*;
- un percorso che termini con un nodo non-goal senza vicini va rimosso.

5 Strategie di Ricerca Non Informate

Il problema determina il grafo e l'obiettivo mentre la **strategia di ricerca** specifica il percorso da selezionare dalla frontiera.

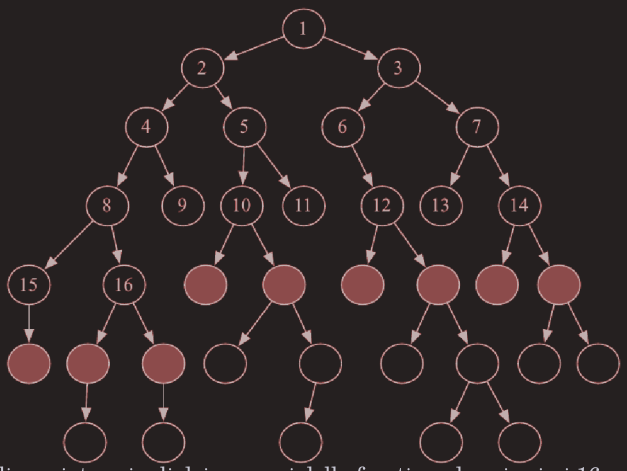
Le *strategie* **non informate** non prendono in considerazione la posizione dell'obiettivo.

Se si considera un costo unitario per ogni arco, possibili strategie sono la ricerca *in ampiezza*, quella *in profondità* o l'*iterative deepening*. Con una funzione di costo: strategie dai *costi minimi*.

5.1 Ricerca in Ampiezza

Nella **ricerca in ampiezza** (Breadth-First Search, BFS) la frontiera è implementata con una *coda*, ossia una struttura FIFO (first-in, first-out). Si seleziona il *primo* percorso aggiunto. I percorsi sono generati nell'ordine del numero di archi contenuti. A ogni passo, si seleziona uno dei percorsi più corti.

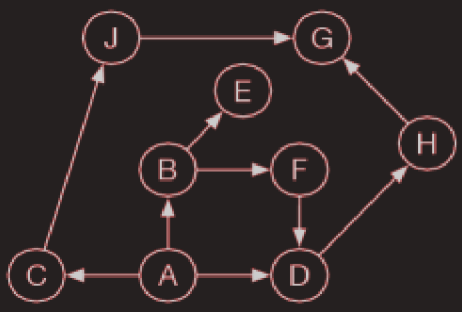
Un esempio albero di ricerca su un dato grafo ottenuto seguendo l'ordine di visita della BFS è illustrato in figura:



nodi scuri: terminali dei percorsi della frontiera dopo i primi 16 passi

---

Esempio — Considerando il grafo precedente (ignorando i costi)



- A nodo di *partenza* e G unico nodo *obiettivo*
  - frontiera: $[\langle A \rangle]$
- Estendendo $\langle A \rangle$ con i suoi vicini:
  - $[\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle]$
  - nodi a un arco di distanza da A

- Espandendo questi percorsi, nell'ordine:
  - $[\langle A, B, E\rangle, \langle A, B, F\rangle, \langle A, C, J\rangle, \langle A, D, H\rangle]$
    - percorsi da A di lunghezza 2
- Dopo l'espansione dei percorsi del passo precedente:
  - $[\langle A, B, F, D\rangle, \langle A, C, J, G\rangle, \langle A, D, H, G\rangle]$
- Selezionando, ad esempio, $\langle A, C, J, G\rangle$ questo viene restituito

NB A ogni passo, nella frontiera vi sono percorsi con approssimativamente lo stesso numero di archi (differenza di un arco).

Cenni sulla Complessità

Sia b il fattore di ramificazione. Se il primo percorso della frontiera ha n archi, ci sono almeno b^{n-1} elementi nella frontiera, con n o $n+1$ archi. Quindi le complessità in spazio e tempo sono *esponenziali* nel numero degli archi del percorso di soluzione di lunghezza minima.

C'è la *garanzia* di ritrovamento della soluzione quando esiste: quella con il minimo numero di archi.

La BFS risulta *utile* quando:

- non si hanno problemi di spazio;
- si cerca una soluzione con numero di archi *minimale*.

La BFS è *poco utile* quando:

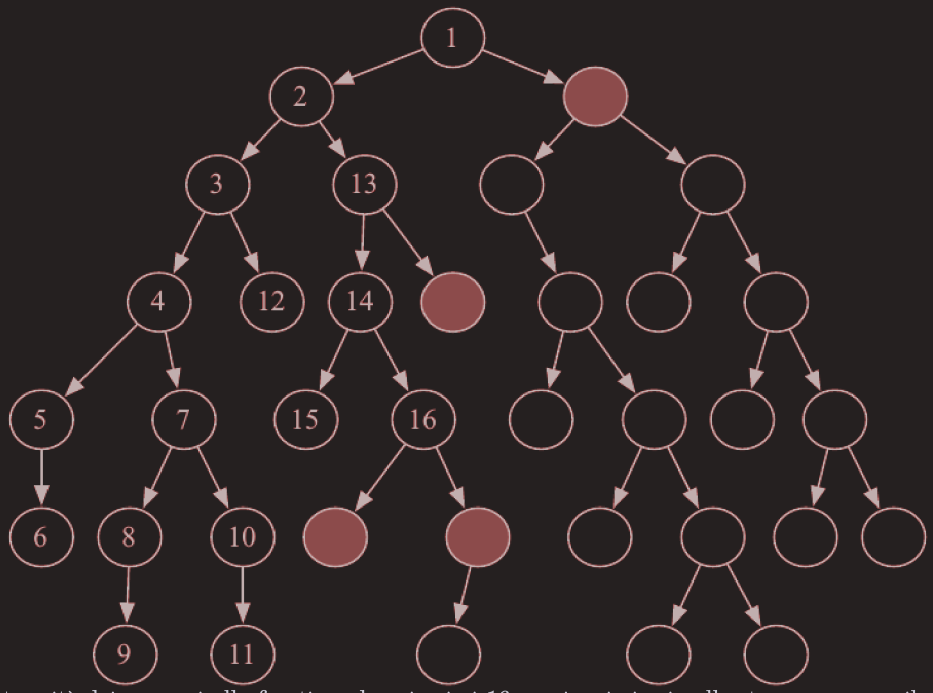
- tutte le soluzioni sono associate a percorsi lunghi;
- è disponibile conoscenza euristica;
- il grafo viene generato dinamicamente (a causa della complessità in spazio).

5.2 Ricerca in Profondità

Nella **ricerca in profondità** (Depth-First Search, DFS) la frontiera è implementata con una *pila* (struttura LIFO): gli elementi vengono aggiunti uno alla volta e quello selezionato e prelevato sarà l'ultimo aggiunto.

Partendo dalla radice, i nodi sono considerati come ordinati da sinistra a destra: il vicino più a sinistra sarà aggiunto in cima *per ultimo*. L'ordine di espansione non dipende dalla posizione dei nodi-obiettivo.

Come in precedenza si veda l'albero di ricerca ottenuto seguendo l'ordine della DFS:



Nodi scuri: estremità dei percorsi alla frontiera dopo i primi 16 passi: primi sei nello stesso percorso; il nodo 6 non ha vicini quindi il successivo sarà figlio dell'antenato più in basso con figli da espandere

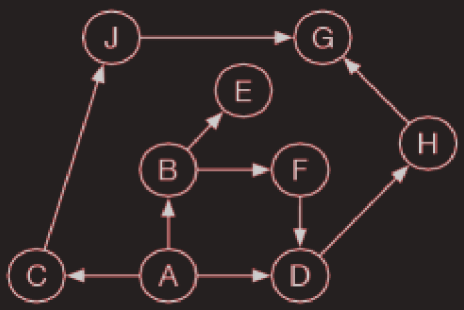
Backtracking ed Efficienza

Con una pila, la *ricerca* procede *in profondità*:

- *completamento* di un *singolo percorso* prima di provare alternative
- questo comporta il **backtracking**: si seleziona una prima alternativa per ogni nodo, *tornando indietro* alla successiva solo dopo aver tentato tutti i completamenti
- *ordine di aggiunta* dei vicini alla frontiera/pila non specificato:
  questo impatta sull'efficienza e può essere *statico*, ossia prefissato, ovvero *dinamico*, dipendendo dall'obiettivo;
- per una `DF_search` alternativa (ricorsiva) cfr. Figura 3.9 di [PM23].

> ⚠ Un grafo con possibili percorsi *infiniti* può portare l'algoritmo a *divergere*; ad esempio se contiene cicli oppure se $|N| = \infty$ (con la generazione dinamica dei nodi).

Esempio — DFS nel caso precedente:



- pila (frontiera) come lista con il primo percorso in *cima*;
- inizialmente: $[\langle A \rangle]$

- le frontiere successive sono:
  - $[\langle A, B\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, E\rangle, \langle A, B, F\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, D\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, D, H\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, D, H, G\rangle, \langle A, C\rangle, \langle A, D\rangle]$
    Il percorso in cima viene restituito come *soluzione*.

Cenni sulla Complessità

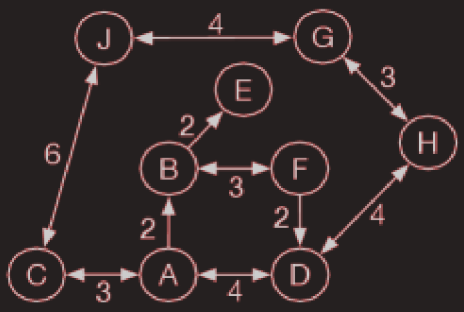
Proprietà — Se $\langle n_0, \ldots, n_k\rangle$ è il percorso scelto dalla frontiera, ogni altro percorso in essa contenuto ha la forma:

$$\langle n_0, \ldots, n_i, m\rangle$$

per qualche $i < k$ e nodo m vicino di n_i, ossia segue il percorso selezionato per un certo numero di archi e poi ha esattamente un altro nodo.

- Se b è il fattore di ramificazione e k la lunghezza del primo percorso della lista, ci sono al più *altri* $k(b-1)$ percorsi: da ogni nodo, fino a $b-1$ percorsi alternativi quindi spazio *lineare* rispetto alla lunghezza del percorso;
- Caso *ottimo*: se c'è una soluzione già sul primo ramo, complessità lineare nella lunghezza del percorso, quindi solo gli elementi nel percorso e loro fratelli
- Caso *pessimo*: diverge con grafi infiniti o contenenti cicli; può rimanere intrappolato in infinite ramificazioni senza trovare una soluzione, anche quando esiste; se il grafo è un *albero finito*, con fattore di ramificazione limitato da b e profondità k, nel caso pessimo esponenziale si attesta su $O(b^k)$.

Esempio — Dato il nuovo grafo:



- sono possibili diversi percorso infiniti quindi la DFS può divergere trascurando sistematicamente le possibili alternative;
- frontiere per le prime iterazioni:
  - $[\langle A\rangle]$
  - $[\langle A, B\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, E\rangle, \langle A, B, F\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, B\rangle, \langle A, B, F, D\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, B, E\rangle, \langle A, B, F, B, F\rangle, \langle A, B, F, D\rangle, \langle A, C\rangle, \langle A, D\rangle]$
  - $[\langle A, B, F, B, F\rangle, \langle A, B, F, D\rangle, \langle A, C\rangle, \langle A, D\rangle]$ ... DFS non termina

Si può migliorare la ricerca *evitando* i percorsi con *cicli* ($\rightarrow$ potatura).

Utilità

- DFS è *appropriata* in caso di:
  - limitazioni di spazio;
  - presenza di molteplici soluzioni, anche se costituite da percorsi lunghi
    - ideale quando *tutti* portano a una soluzione;
    - oppure se l'ordine di aggiunta dei vicini può essere variato in modo da trovare una soluzione al *primo tentativo* (senza backtracking);

- DFS è *inefficiente* quando:
  - sono possibili percorsi infiniti in caso di grafo infinito o contenente cicli;
  - pur esistendo soluzioni alternative poco profonde (più a destra), la ricerca si attarda su percorsi più lunghi.

5.3 Iterative Deepening

L'obiettivo di tale algoritmo è combinare l'efficienza in spazio di DFS con l'ottimalità di BFS. L'idea è quella di non memorizzare ma ricalcolare gli elementi della frontiera di BFS usando DFS che usa meno spazio.

DFS effettua una ricerca limitata fino a una data *profondità* come per le iterazioni di BFS. In caso di fallimento si eliminano i percorsi via calcolati e ripartendo, se necessario, estendendo il limite. Si parte cercando fino a profondità 1, poi i cammini di lunghezza 2, quindi quelli di lunghezza 3, ecc... se esiste, una soluzione verrà trovata, esplorando percorsi in tale ordine: uno con il minimo numero di archi sarà individuato per primo.

Il *fallimento* della ricerca in profondità limitata può essere:

- *innaturale*: per raggiungimento del limite di profondità e in tal caso la ricerca riparte, dopo aver incrementato il limite;

- *naturale*: quando l'intero spazio di ricerca è esaurito non esiste soluzione a nessun livello di profondità.

---

procedure ID_search($G, s, goal$)

Input

G: grafo con insiemi di nodi N e di archi A
s: nodo di partenza
$goal$: funzione Booleana sugli stati

Output

cammino da s a un nodo per il quale $goal$ sia vero ovvero $\perp$ altrimenti

Locali

hit_depth_bound: Boolean
$bound$: integer

procedure Depth_bounded_search($\langle n_0, \ldots, n_k \rangle$, b)

Input

$\langle n_0, \ldots, n_k \rangle$: percorso
b: integer, $b \geq 0$

Output

percorso fino a nodo-obiettivo di lunghezza $k + b$

if $b > 0$ **then**

for each $\langle n_k, n \rangle \in A$ **do**

$res \leftarrow$ Depth_bounded_search($\langle n_0, \ldots, n_k, n \rangle, b - 1$)
if $res \neq \perp$ **then**
return res
else if $goal(n_k)$ **then**
return $\langle n_0, \ldots, n_k \rangle$
else if n_k ha vicini **then** // fallimento innaturale
$hit\_depth\_bound \leftarrow true$
return $\perp$

$bound \leftarrow 0$

repeat

$hit\_depth\_bound \leftarrow false$
$res \leftarrow$ Depth_bounded_search($\langle s \rangle$, $bound$)
if $res \neq \perp$ **then**
return res
$bound \leftarrow bound + 1$

until not hit_depth_bound

return $\perp$ // fallimento naturale

Osservazioni

- La Depth_bounded_search implementa una DFS (ricorsiva) a profondità limitata: trova percorsi di lunghezza $k+b$, dove k lunghezza del percorso e $b \geq 0$; viene chiamato a profondità crescenti;

- I percorsi vengono trovati nello stesso ordine della BFS: si controlla di aver trovato un obiettivo solo se $b = 0$ (soluzioni per limiti inferiori non sono state trovate in precedenza)
- Per assicurare di fallire quando anche la BFS fallirebbe, tiene traccia dei casi in cui un limite maggiore potrebbe aiutare a trovare una soluzione.
  - fallisce *naturalmente* se ha esaurito tutto lo spazio di ricerca e non sono stati tagliati percorsi per limite raggiunto (in tal caso ci si può fermare restituendo $\perp$);
  - se *hit_depth_bound*, falsa alla chiamata, diventa vera alla fine, il limite può essere incrementato per la successiva iterata.

Cenni sulla Complessità

Problema: lo spreco di computazione a ogni passo

- non così grave se il fattore di ramificazione b è grande (cfr. seguito)
- *tempo*: dato $b > 1$ costante, in una ricerca con limite k
  - profondità k: b^k nodi, generati una sola volta;
    profondità $k-1$: b^{k-1} nodi generati 2 volte;
    profondità $k-2$: b^{k-2} nodi generati 3 volte;
    ...
    profondità 1: b^1 nodi generati k volte
  - totale nodi generati:

    $$b^k + 2b^{k-1} + 3b^{k-2} + \cdots + kb = b^k(1 + 2b^{-1} + 3b^{-2} + \cdots + kb^{1-k}) \leq b^k \sum_{i=1}^{\infty} ib^{(1-i)} = b^k \left(\frac{b}{b-1}\right)^2$$

    la BFS espande $b^k(b/b-1) - 1/(b-1)$ nodi quindi ID ha overhead asintotico di $b/(b-1)$ volte il costo dell'espansione di BFS: per $b = 2$ l'overhead è pari a 2; per $b = 3$ l'overhead è di 1.5;

  - risulta un algoritmo $O(b^k)$ e non ci può essere una migliore ricerca non informata
  - se b è vicino a 1, analisi problematica: denominatore vicino a 0

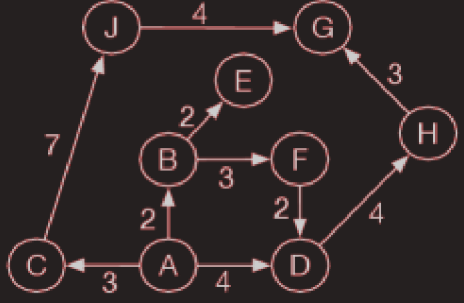
5.4 Ricerca a Costo Minimo

In caso di archi con *costi* non unitari si cerca la soluzione di *minimo costo totale*. Ad esempio un robot per le consegne potrebbe considerare un costo dell'arco in base a distanze e/o risorse necessarie all'azione rappresentata dall'arco.

Gli algoritmi precedenti non garantiscono soluzioni di costo minimo: il costo non viene preso in considerazione. BFS minimizza solo il *numero* degli archi: potrebbe esistere una soluzione alternativa ma con un percorso *più lungo* di *costo inferiore*.

La **Lowest cost-First Search** (LcFS) si comporta come BFS ma con la selezione dei percorsi di costo minimo. Viene implementato usando una *coda con priorità* come frontiera ordinata da *cost*.

Esempio — Considerando il grafo:



- Si espande sempre il percorso più a sinistra (costo indicato come pedice):
  - $[\langle A\rangle_0]$ frontiera iniziale
  - $[\langle A,B\rangle_2, \langle A,C\rangle_3, \langle A,D\rangle_4]$
  - $[\langle A,C\rangle_3, \langle A,B,E\rangle_4, \langle A,D\rangle_4, \langle A,B,F\rangle_5]$
  - $[\langle A,B,E\rangle_4, \langle A,D\rangle_4, \langle A,B,F\rangle_5, \langle A,C,J\rangle_{10}]$
  - $[\langle A,D\rangle_4, \langle A,B,F\rangle_5, \langle A,C,J\rangle_{10}]$
  - $[\langle A,B,F\rangle_5, \langle A,D,H\rangle_8, \langle A,C,J\rangle_{10}]$
  - $[\langle A,B,F,D\rangle_7, \langle A,D,H\rangle_8, \langle A,C,J\rangle_{10}]$
  - $[\langle A,D,H\rangle_8, \langle A,C,J\rangle_{10}, \langle A,B,F,D,H\rangle_{11}]$
  - $[\langle A,C,J\rangle_{10}, \langle A,D,H,G\rangle_{11}, \langle A,B,F,D,H\rangle_{11}]$
  - dopo l'espansione di $\langle A,C,J\rangle$ si selezionerà $\langle A,D,H,G\rangle$ percorso minimo fra A e G

Cenni su Calcolabilità, Ottimalità e Complessità

- Il fattore di ramificazione è *finito* se i *costi* sono *limitati* inferiormente da una costante positiva: ciò garantisce la soluzione ottimale, quando esiste. Il primo percorso trovato termina in un nodo-obiettivo ed è ottimale perché si procede in ordine di costo (se ne esistesse una migliore sarebbe stata già trovata);
- Se manca un limite inferiore allora sono possibili percorsi *infiniti*, ad esempio con nodi $n_0, n_1, n_2, \ldots$ con $\forall i > 0$: $\langle n_{i-1}, n_i\rangle$ di costo $1/2^i$ esistono infiniti percorsi $\langle n_0, n_1, n_2, \ldots, n_k\rangle$, con costo < 1; se $\exists\langle n_0, g\rangle$, $goal(g)$, con costo ≥ 1, l'arco non verrebbe mai selezionato (cfr. paradosso di Zenone: *Achille e la tartaruga*)
- La complessità è *esponenziale* in spazio e tempo (come per la BFS): genera tutti i percorsi con costo inferiore a quello della soluzione.

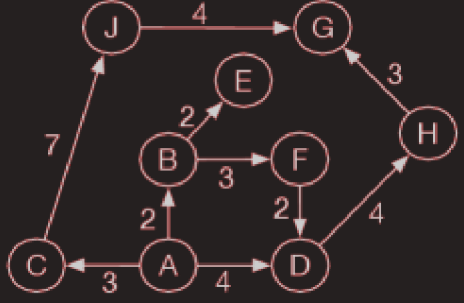
6 Ricerca Informata (Euristica)

6.1 Euristica

Tale tipo di ricerca prende in considerazione *informazioni sull'obiettivo*, nella selezione dei nodi da esplorare, attraverso l'uso di una **funzione euristica** h che associa a ogni nodo n un numero reale *non-negativo*, una stima del costo minimale di un percorso da n fino a un nodo-goal.

L'euristica h si dice **ammissibile** se *sottostima* il costo reale: $h(n)$ minore o uguale rispetto al costo minimale effettivo del percorso. Tale informazione approssimata è spesso immediatamente disponibile con un *compromesso* tra efficienza del suo calcolo e accuratezza della stima. Tipicamente, risolvendo una forma semplificata del problema, si possono poi usare i costi effettivi del problema semplificato come euristica nella soluzione del problema originario.

Esempio — Problema precedente:



Consederando come *euristica* la distanza in linea retta tra nodo e obiettivo più vicino, quindi supponendo che tali distanze siano ad esempio:

$$\begin{array}{l|l|l} h(A) = 7 & h(B) = 5 & h(C) = 9 \\ h(D) = 6 & h(E) = 3 & h(F) = 5 \\ h(G) = 0 & h(H) = 3 & h(J) = 4 \end{array}$$

h risulta ammissibile e in particolare

- essa è *esatta* per il nodo H;
- essa è molto *sottostimata* per il nodo B: sembra vicino al goal (costo 5), ma raggiungere l'obiettivo costa molto di più;
- essa è *ingannevole* per il nodo E: sembra vicino al goal (costo 3) ma non permette di raggiungere l'obiettivo.

6.2 DFS Euristica e Greedy best–First Search

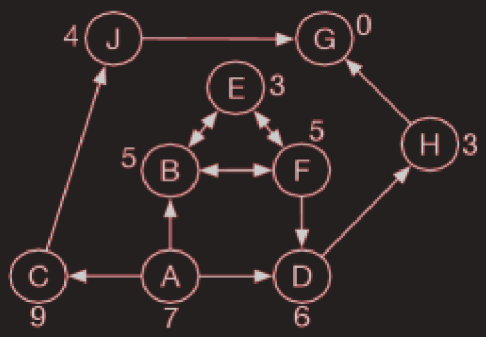
L'euristica h può essere *estesa* al caso dei percorsi (non vuoti):

$$h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$$

La DFS **Euristica** usa h per ordinare i vicini aggiunti alla pila/frontiera della DFS standard. I vicini vengono aggiunti in modo che il *migliore* vada *in cima*. Tale scelta è comunque *locale*: si esplorano i percorsi che estendono quello selezionato prima di tentarne altri. Si presentano problemi analoghi al caso della DFS: non c'è garanzia di terminazione e ritrovamento di una soluzione ottimale.

Nella *Greedy* **best-First Search**, GbFS, si sceglie il percorso della frontiera con valore di h minimo. Purtroppo potrebbe seguire cammini apparentemente promettenti (vicini all'obiettivo secondo h) che, però, potrebbero continuare a estendersi indefinitamente.

Esempio — Si consideri il grafo



- *scopo*: percorso minimo da A a G
- *costo*: ignorato → valori di h sui nodi
- h: distanza Euclidea da G
  - DFS euristica e GbFS iterano indefinitamente su $B \to E \to F \to B \to \ldots$
  - anche potendo identificare i cicli, un grafo simile, con infiniti nodi collegati con valore dell'euristica inferiore a 6, risulterebbe problematico per entrambi

---

6.3 Ricerca A*

A* combina idee da LcFS e GbFS. Nella selezione del percorso da espandere si considerano: il costo effettivo del percorso parziale fino al nodo corrente e l'euristica, ossia una stima del costo fino a un obiettivo.

Per ogni percorso $p = \langle s, \dots, n \rangle$ della frontiera, si stima il costo di un percorso che passi per p e prosegua fino a un goal g:

$$f(p) = \overset{s \xrightarrow{reale} n \xrightarrow{stima} g}{cost(p) + h(p)}$$

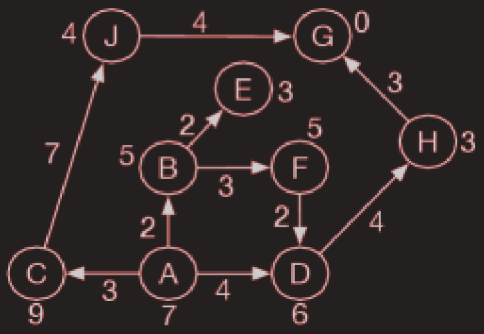
dove $cost(p)$ è il costo effettivo di p e $h(p)$ è una stima del costo del cammino da n a g.

Implementazione di A*

Si segue lo schema dell'algoritmo di ricerca generico e si rappresenta la *frontiera* con una coda con priorità, ordinata da $f(p)$. Se l'euristica $h(n)$ *ammissibile* allora $f(p)$ non sovrastima il costo d'un percorso completo che includa p.

A* può seguire molti percorsi ma *alla fine* ne trova uno di costo minimo. Altri possono sembrare (provvisoriamente) di costo inferiore. A* migliora le prestazioni degli algoritmi da cui origina (LcFS e GbFS).

Esempio — A* su grafo ed euristica precedenti

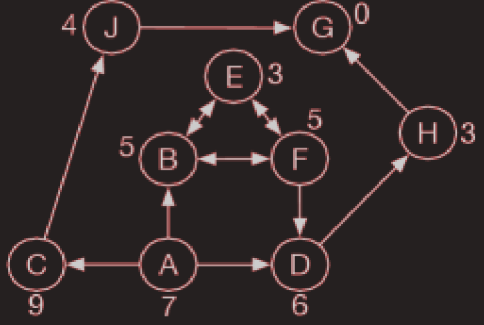


Sequenza delle frontiere (pedice = $f(p)$)

- $[\langle A \rangle_7]$ iniziale
  - essendo $h(A) = 7$ e $cost(\langle\rangle) = 0$
- $[\langle A, B \rangle_7, \langle A, D \rangle_{10}, \langle A, C \rangle_{12}]$
- $[\langle A, B, E \rangle_7, \langle A, B, F \rangle_{10}, \langle A, D \rangle_{10}, \langle A, C \rangle_{12}]$
  - $\langle A, B, E \rangle$ da rimuovere (E senza vicini)
- $[\langle A, B, F \rangle_{10}, \langle A, D \rangle_{10}, \langle A, C \rangle_{12}]$
  - *tie* fra i primi due percorsi ma $\langle A, B, F \rangle_{10}$ preferito perché $h(F) = 5 < 6 = h(D)$
- $[\langle A, D \rangle_{10}, \langle A, C \rangle_{12}, \langle A, B, F, D \rangle_{13}]$
- $[\langle A, D, H \rangle_{11}, \langle A, C \rangle_{12}, \langle A, B, F, D \rangle_{13}]$
- $[\langle A, D, H, G \rangle_{11}, \langle A, C \rangle_{12}, \langle A, B, F, D \rangle_{13}]$

Soluzione: $\langle A, D, H, G \rangle$ di costo 11.

Esempio — Grafo precedente con cicli:



- problematico per LcFS e GbFS
- sebbene A* si indirizzi inizialmente verso $B \to E \to F$ per via di h, il percorso diventa via via tanto costoso da far preferire presto o tardi un percorso ottimale attraverso D.

Ammissibilità degli Algoritmi

Un algoritmo è **ammissibile** se, quando esistono soluzioni, ne trova sempre una ottimale *(anche in caso di uno spazio degli stati infinito)*.

Proposizione — A* ammissibile se:

- il fattore di ramificazione è limitato ($\leq b$)
- i costi degli archi maggiori di un certo $\epsilon > 0$
- h è ammissibile, i.e. non supera il costo minimale di percorsi da ciascun nodo a un obiettivo.

Osservazioni

- A* garantisce che la prima soluzione sarà ottimale, anche in presenza di *cicli*;
- ma non assicura che ciascun nodo *intermedio* selezionato dalla frontiera sia su un cammino ottimo.

Importanza dell'Euristica in A*: il miglioramento dell'*efficienza* in A* è dato dall'uso di h. Sia c il costo del percorso di costo minimo. Se h è ammissibile, A* espande ogni percorso dal nodo di partenza nell'insieme

$$\{p \mid cost(p) + h(p) < c\}$$

più alcuni percorsi nell'insieme

$$\{p \mid cost(p) + h(p) = c\}$$

Per l'efficienza di A*, h migliore se riduce la cardinalità del primo insieme.

Cenni su IDA*

IDA* applica Iterative Deepening ad A*: effettua ripetute DFS limitate con *limite* sul valore di $f(n)$ (invece della profondità / numero di archi): Si fissa un limite iniziale per $f(s)$ (con s nodo di partenza) con il minimo valore di h, se ce ne sono diversi.

Osservazioni

- IDA* si comporta come DFS limitata
  ma non espande un percorso con costo-f maggiore del limite corrente: se la ricerca fallisce in modo *innaturale* il nuovo limite viene fissato al minimo valore di f che abbia superato il precedente;

- IDA* lavora sugli stessi nodi di A*
  ma li ricalcola con la DFS invece di memorizzarli. Per puntare a soluzioni sub-ottimali, si può usare il limite: $\delta + \min f()$.

6.4 Branch and Bound

L'idea di questo algoritmo è combinare l'efficienza in spazio delle strategie in profondità con informazione euristica. Risulta efficace quando esistono molti cammini verso un goal, assume che $h(n)$ sia ammissibile.

Branch-and-bound memorizza il percorso di costo minimo verso un goal trovato e il suo costo che diventa un *limite*, $bound$:

- un percorso p tale che $cost(p) + h(p) \geq bound$ può essere scartato;
- si memorizza p completo se $cost(p) < bound$ aggiornando $bound$;
- si continua poi a cercare un'eventuale soluzione migliore.

Si genera così una sequenza di soluzioni via via migliori restituendo quella finale: l'algoritmo viene tipicamente implementato usando la DFS per la ricerca in profondità limitata dal valore di $bound$.

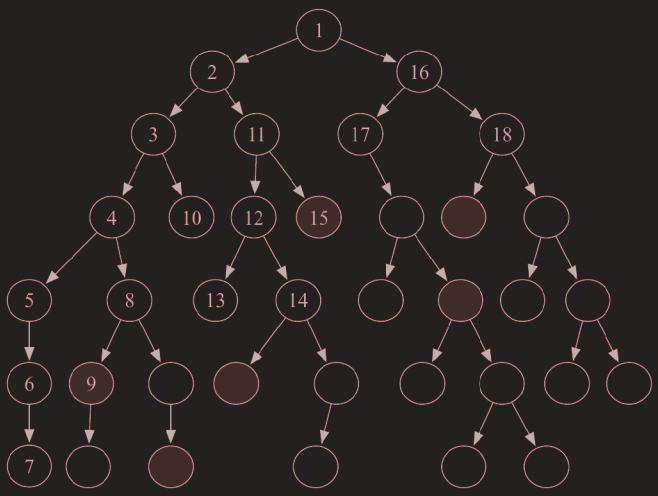
```
procedure DFBranchAndBound(G, s, goal, h, bound_0)

Input

    G: grafo con nodi N e archi A
    s: nodo di partenza
    goal: funzione di test dei nodi
    h: euristica sui nodi
    bound_0: limite iniziale (∞ se non specificato)

Output

    percorso di costo minimo da s a un nodo obiettivo
        se esiste una soluzione di costo minore di bound_0,
    oppure ⊥

Local

    best_path: percorso o ⊥
    bound: numero reale non negativo

    procedure cbsearch(⟨n_0, ..., n_k⟩)

        if cost(⟨n_0, ... n_k⟩) + h(n_k) < bound then
            if goal(n_k) then
                best_path ← ⟨n_0, ..., n_k⟩
                bound ← cost(⟨n_0, ..., n_k⟩)
            else
                for each ⟨n_k, n⟩ ∈ A do
                    cbsearch(⟨n_0, ..., n_k, n⟩)

    // main
    best_path ← ⊥
    bound ← bound_0
    cbsearch(⟨s⟩)
    return best_path
```

Osservazioni

- `cbsearch`, *cost-bounded search*, comunica attraverso variabili globali; inizialmente, $bound$ è impostata a $bound_0$, stima per eccesso del costo d'una soluzione ottimale;

- DFBranchAndBound restituirà, se esiste, una soluzione ottimale con costo inferiore a $bound_0$. Se $bound_0$ supera di poco il costo minimo, non espanderà più archi di A*. $bound_0$ è tale da far eliminare percorsi di costo maggiore: trovato un percorso completo, si esplorano solo percorsi con valore di f inferiore a quello del percorso trovato, ossia esattamente i percorsi esplorati da A* quando trova una soluzione. Quando restituisce $\perp$, se $bound_0 = \infty$ allora non ci sono soluzioni, se invece $bound_0$ è finito allora non ci sono soluzioni di costo inferiore a $bound_0$.
- Combinato con *ID* l'algoritmo può incrementare via via il limite fino a trovare una soluzione ovvero può mostrare che non c'è soluzione.

Esempio — Considerato il grafo: (nodi scuri = obiettivi)



- Si supponga che:
  - ogni arco abbia costo 1 e non ci sia informazione euristica ($\forall n : h(n) = 0$)
  - $bound_0 = \infty$
  - si scelga sempre il nodo più a sinistra

- Si noti l'ordine di test dei nodi (via $goal()$): nodi non numerati non testati
- Il sotto-albero del nodo 5 non ha goal e viene esplorato tutto
  - o fino a profondità $bound_0$ se finito
- Il nodo 9 è un goal, con percorso di costo 5 → nuovo limite
- Anche il 15 è un goal, con percorso di costo 3 → nuovo limite
- Non ci sono altri nodi obiettivo → restituisce il percorso fino al nodo 15
  - ottimale
  - l'altro, di pari costo, viene scartato: non si controllano i figli del nodo 18
- Si potrebbe usare un'euristica per tagliare parti dello spazio come in A*

6.5 Progettazione della Funzione Euristica

Un'**euristica ammissibile** è una funzione non negativa h definita sui nodi, tale che $h(n)$ mai superiore al costo reale del percorso ottimale da qualunque nodo n a un nodo-goal.

Procedura standard per definire un'euristica:

- si trovano soluzioni di un problema più semplice, ossia con *meno vincoli*, quindi spesso molto *più facile* da risolvere; una sua soluzione ottimale avrà un costo pari o inferiore a una soluzione del problema originario (ognuna lo sarà anche per la versione semplificata).

Ad esempio nei problemi relativi a spazi rappresentati da grafi di luoghi, in cui si è *vincolati* a muoversi lungo i dati archi (i.e. le strade esistenti); supponendo che il loro costo

sia proporzionale alla lunghezza dei collegamenti stradali, un'euristica ammissibile è la *distanza euclidea* tra nodi (problema semplificato, i.e. privo dei suddetti vincoli).

Esempio — robot per le consegne

- *stati* comprendenti le consegne da fare
- *costo* = distanza totale per tutte le consegne
- *euristica* possibile = massimo tra:
    1. la distanza massima per consegne ancora da fare non caricate: per ognuna, la distanza dalla sua posizione e, da qui, fino alla destinazione;
    2. la distanza dalla destinazione più lontana per le consegne trasportate
- tale massimo non rappresenta una sovrastima: è la soluzione per un problema semplificato non considerando muri e tutte le altre consegne tranne la più difficile; è appropriato: vanno consegnate quelle trasportate, passate a prendere quelle ancora non caricate e portate a destinazione;
- se il robot può portare una sola consegna, un'euristica dovrebbe sommare le distanze per il trasporto di ognuna più la distanza dalla più vicina: non è detto che sarà effettuata per prima (ma serve a garantire l'ammissibilità).

Esempio — navigatore

Minimizzazione del *tempo*

- *euristica*: distanza in linea retta dalla locazione corrente al goal divisa per la velocità massima, assumendo che ci si possa dirigere a destinazione alla massima velocità;
- *euristica* più sofisticata: date velocità massime diverse per autostrade e strade locali, si sceglie il massimo tra:
    1. il minimo tempo stimato per andare a destinazione su strade locali lente e
    2. il minimo tempo utile a raggiungere un'autostrada da una strada locale, per poi raggiungere un luogo vicino alla destinazione e arrivarci da strada locale.

Euristiche tramite ricerca:

- Un problema in forma semplificata può essere risolto attraverso una ricerca più semplice rispetto al problema originario: ricerca ripetuta anche più volte e per tutti i nodi;
- Conviene memorizzare (in una memoria *cache*) tali risultati in un **DB di pattern** che associ ai nodi del problema semplificato il valore dell'euristica;
- Nel problema semplificato, spesso si hanno meno nodi: più nodi originari sono mappati su uno *stesso* nodo di quello semplificato come fosse una *classe d'equivalenza*.

7 Potatura dello Spazio di Ricerca

Si possono migliorare gli algoritmi considerando i diversi percorsi che possono attraversare un nodo. Non tutti sono utili per cui possono essere eliminati.

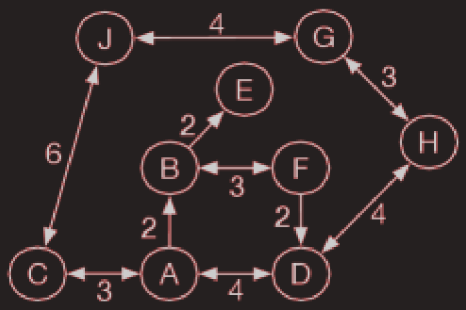
Essenzialmente vi sono due tipi di **pruning** (*potatura*) con relative strategie:

- *potare i cicli* per trovare percorsi dal costo minimo;
- *potare cammini molteplici*: per ogni nodo si considererà un solo percorso che lo attraversi, eliminando tutti gli altri.

7.1 Potatura dei Cicli

Alcuni dei metodi di ricerca visti in precedenza potrebbero rimanere intrappolati in cicli senza possibilità di uscita anche nel caso di grafi finiti.

Un esempio di grafo contenete cicli è il seguente:



- un ciclo contenuto in più cammini è il seguente: $B \to F \to D \to A \to B \ldots$

Per avere la garanzia di trovare una soluzione in caso di *grafo finito* non andranno presi in considerazione vicini già presenti nel percorso corrente.

Nella **potatura dei cicli** (*cycle/loop pruning*, CP), si prevede un controllo addizionale preventivo: dato un nodo da aggiungere, va testata l'occorrenza nel percorso.

Nello schema di algoritmo di ricerca generico si aggiungono alla frontiera solo percorsi $\langle n_0, \ldots, n_k, n \rangle$, tali che $n \notin \{n_0, \ldots, n_k\}$, scartando il percorso selezionato; in alternativa si può effettuare il controllo *dopo* la selezione del nodo.

La complessità dei controlli aggiuntivi per la verifica della presenza di un nodo nel percorso corrente dipende dal metodo di ricerca adottato:

- *costante* per metodi (DF) che memorizzano un unico percorso (in una lista o insieme); si può usare una funzione *hash* oppure si può associare un *bit* a ogni nodo (flag booleano), che sarà *acceso* quando viene aggiunto a un percorso e viene *spento* in caso di backtracking; per cui basterà non espandere nodi con il bit acceso; tale metodo funziona perché si lavora su un solo percorso;
- *lineare* nella lunghezza del percorso corrente per metodi che gestiscono più percorsi (esponenziali in spazio): serve una ricerca per evitare di aggiungere al percorso parziale un nodo già presente.

7.2 Potatura di Percorsi Multipli

Spesso più di un percorso porta a uno stesso nodo. Occorre quindi *eliminare dalla frontiera ogni percorso che porti a un nodo per il quale ne esista già un altro* (di costo inferiore).

Una strategia di **multiple-path pruning** (MPP) può essere implementata gestendo una lista di nodi terminali di percorsi già esplorati, detta **closed list** o *explored set*: inizialmente essa è vuota poi, selezionato un percorso $\langle n_0, \ldots, n_k \rangle$ della frontiera, se n_k è già nella lista esso può essere scartato, altrimenti si aggiunge n_k alla lista e si prosegue. Si veda l'algoritmo di Figura 3.16 in [PM23].

questa strategia non garantisce che non venga eliminato il percorso di costo minimo

Garanzia di Soluzioni Ottimali

Per garantire di preservare le soluzioni ottimali vi sono alcune alternative:

- assicurando che il *primo* percorso trovato per un dato nodo sia ottimale quindi gli altri si possono *eliminare*;
- se nella frontiera si dovesse inserire un percorso $p = \langle s, \ldots, n, \ldots, m \rangle$, ma fosse già presente $p' = \langle s, \ldots, n \rangle$ meno costoso della parte fino a n in p, si può: *eliminare* p oppure *oppure sostituire* in p tale parte con p'.

Algoritmi di Ricerca e MPP

In LcFS, il primo percorso verso un dato nodo (al momento della selezione dalla frontiera) è quello di costo minimo. Pertanto potare altri percorsi non elimina un cammino a costo minimo per il nodo. Ciò assicura di poter trovare una soluzione ottimale.

A^* non garantisce che quando si seleziona un percorso verso un nodo per la prima volta, questo sia quello di costo minimo: il teorema sull'ammissibilità lo garantisce SOLO per i percorsi verso nodi-goal; per quelli verso altri nodi dipende dalle *proprietà dell'euristica*.

Consistenza e Monotonicità

Un'**euristica consistente** h (non negativa) soddisfa il vincolo:

$$h(n) \leq cost(n, n') + h(n')$$

per ogni coppia di nodi n e n'.

Se $h(g) = 0$ per ogni goal g, allora h sarà consistente e non sovrastimerà mai il costo dei percorsi da un nodo verso un obiettivo.

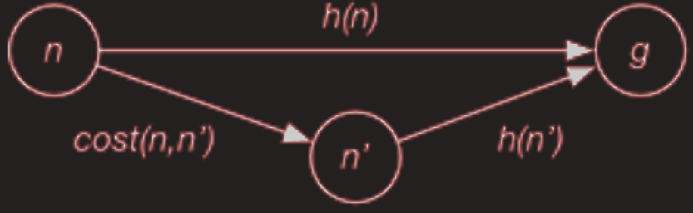
La consistenza è garantita se h soddisfa la **restrizione di monotonicità**:

$$h(n) \leq cost(n, n') + h(n')$$

per ogni arco $\langle n, n' \rangle$. Ciò è più facile da testare: dipende solo dagli archi, non dalle coppie di nodi considerate.

Consistenza e Disuguaglianza Triangolare

Considerato un grafo in cui:



per la *consistenza*, il costo stimato del percorso da n al goal g non dev'essere maggiore della stima di quello che obbliga al passaggio da n'. Ad esempio ciò vale con la distanza euclidea e con euristiche indotte dalle soluzioni di problemi semplificati.

La *restrizione monotona* richiede che i valori di f dei percorsi selezionati dalla frontiera siano monotonicamente non decrescenti.

Proposizione — Data un'euristica consistente, MPP non impedisce ad A* di trovare una soluzione ottimale.

Quindi, nelle condizioni della proposizione sull'ammissibilità di A* che garantiscono il ritrovamento della soluzione ottima, se l'euristica è consistente, anche A* con MPP la potrà trovare.

Nella pratica, A* include MPP per default. Andrebbe evidenziato esplicitamente che si usa A* senza potatura; compito del progettista è quello di definire un'euristica consistente in modo da permettere il ritrovamento di percorsi ottimali.

Confronto MPP vs CP

- MPP è *più generale* di CP: un ciclo può essere considerato come un altro percorso verso un dato nodo, destinato a essere potato;
- MPP implementabile per richiedere un tempo costante;
- MPP preferibile con metodi *in ampiezza*: virtualmente tutti i nodi considerati vanno memorizzati;
- CP preferibile con le strategie *in profondità*, ad esempio DF BRANCH-AND-BOUND; con MPP+DFS c'è il rischio di crescita esponenziale per la lista chiusa.

7.3 Consuntivo sulle Strategie di Ricerca

Strategia	Selezione da Frontiera	Soluzione Garantita	Complessità
BFS	primo nodo aggiunto	con meno archi	$O(b^d)$
DFS	ultimo nodo aggiunto	no	$O(bd)$
ID	n/a	con meno archi	$O(bd)$
GBFS	$h(p)$ minimale	no	$O(b^d)$
LCFS	$cost(p)$ minimale	costo minimo	$O(b^d)$
A*	$cost(p) + h(p)$ minimale	costo minimo	$O(b^d)$
DF B&B	n/a	costo minimo	$O(bd)$
IDA*	n/a	costo minimo	$O(bd)$

- Complessità in spazio, con d *profondità* e b limite superiore per il *fattore di ramificazione*.

Un **algoritmo** di ricerca **completo** garantisce la soluzione quando esiste.

Le strategie che trovano percorsi con #archi / costo minimi sono complete: nel *caso pessimo* serve un tempo esponenziale nel #archi dei percorsi esplorati. L'esistenza di algoritmi

completi con minore complessità dipende dalla risposta alla domanda

$$P \neq NP\ ?$$

Per algoritmi che non garantiscono la terminazione la complessità risulta non limitata.

8 Strategie di Ricerca più Sofisticate

Raffinamenti possibili delle strategie presentate sono i seguenti:

- decidere la direzione della ricerca impatta sull'efficienza: si può pensare a una ricerca retrograda per soluzioni ottimali da qualunque nodo;
- nella ricerca euristica si può usare la *programmazione dinamica*: metodi basati sulla *riduzione* del problema in un numero di problemi equivalenti ma di minore complessità.

8.1 Direzione della Ricerca

Nella ricerca in **avanti** (*forward search*) si comincia da un nodo di partenza e si prosegue fino a raggiungere i goal.

Nella ricerca all'**indietro** (*backward search*): si parte da un goal e si usa il *grafo inverso* alla ricerca del nodo di partenza; i vicini del goal considerati per inizializzare la frontiera sono tutti gli altri goal: $\{n : goal(n)\}$.

La direzione di ricerca è *indifferente* quando:

1. il numero di goal *finito*;
2. per ogni nodo n si possono generare i *vicini* nel **grafo inverso** $\{n' : \langle n', n\rangle \in A\}$.

Cenni sulla Scelta della Direzione

Le dimensioni dello spazio di ricerca sono date da b^d, dove b è il fattore di ramificazione e d dalla lunghezza del cammino. Riducendo questi parametri aumenta l'*efficienza*.

È possibile che vi sia una differenza tra i fattori di ramificazione uscente ed entrante; in tal caso un *principio generale* per la scelta della direzione della ricerca è quello di considerare b inferiore fra i due.

8.2 Ricerca Bidirezionale

Nella **ricerca bidirezionale** si riduce il tempo di ricerca procedendo in entrambe le direzioni: quando le due frontiere *si intersecano*, ricostruire un singolo cammino dal nodo di partenza al goal.

Problema: come assicurare che le due frontiere s'incontrino?

- caso di DFS: è difficile date le ridotte dimensioni delle frontiere;
- caso di BFS: è garantito;

Combinando BFS e DFS in direzioni opposte, si garantirebbe l'intersezione, ma quale direzione per ciascun algoritmo? Il *fattore* per tale scelta è per la BFS il costo del mantenimento *della* frontiera (tempo proporzionale a b^k) mentre per la DFS è costo della ricerca *nella* frontiera per trovare un'intersezione; se si tratta di una ricerca simmetrica

bidirezionale si è nell'ordine di $2b^{k/2}$, con risparmi (esponenziali) in tempo, ma la complessità rimane esponenziale.

8.3 Ricerca in una Gerarchia di Astrazioni

L'idea è quella di astrarsi dal problema, rimuovendo dettagli: una soluzione parziale del problema potrà essere ottenuta da una soluzione per quello in forma più astratta. Ad esempio passare da una stanza a un'altra richiede più modi di girare ma questi potrebbero essere omessi nella forma astratta. Con l'astrazione si punta a risolvere il problema a grana grossa, lasciando da risolvere problemi più specifici e semplici.

Diverse sono le modalità di *decomposizione/astrazione* di problemi. Tutte le strategie di ricerca presentate sono utilizzabili ma le specifiche astrazioni / decomposizioni utili non risultano facili da individuare.

Esempio — *ricerca a isole*, come ad esempio i piani di un edificio.

Trovata la soluzione a livello di isola si passa a risolvere ricorsivamente i sotto-problemi in modo analogo;

- informazioni sulle soluzioni trovate a livelli più bassi possono servire a quelle per livelli più alti;
- ai livelli più alti si può usare tale informazione per riformulare un piano;
- il processo tipicamente non garantisce soluzioni ottimali perché considera solo alcune delle decomposizioni possibili.

8.4 Programmazione Dinamica

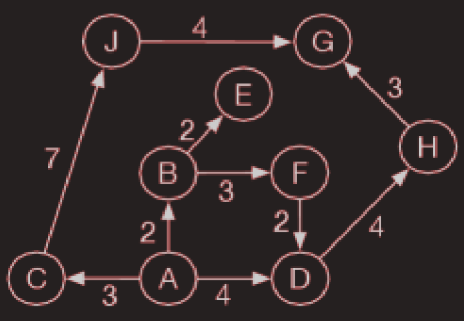
La **programmazione dinamica** (PD) fornisce un metodo generale di ottimizzazione basato sulla conservazione di *soluzioni parziali*: una soluzione per un problema più semplice già prodotta (e memorizzata) dev'essere solo ritrovata e non ricalcolata.

Si può ricorrere alla PD per la costruzione offline di un'*euristica perfetta* $cost\_to\_goal()$, costo *esatto* di un cammino completo di costo minimo su *grafi finiti*:

$$cost\_to\_goal(n) = \begin{cases} 0 & \text{se } goal(n) \\ \min\limits_{\langle n,m\rangle \in A} [cost(\langle n,m\rangle) + cost\_to\_goal(m)] & \text{altrimenti} \end{cases}$$

calcolato usando LcFS + MPP sul *grafo inverso* a partire dai nodi-obiettivo: si cerca il cammino di costo minimo da ogni nodo verso i goal e si conserva il valore di $cost\_to\_goal$ per ogni nodo.

Esempio — Dato il grafo visto in precedenza:

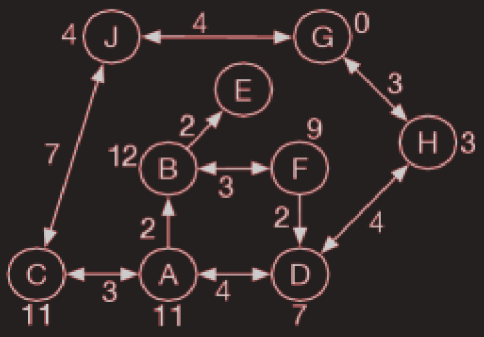


risulta:

- $cost\_to\_goal(G) = 0$, essendo il goal;
- $cost\_to\_goal(H) = 3$;

- $cost\_to\_goal(J) = 4$;
- $cost\_to\_goal(D) = 7$;
- ...

Si ottengono i costi indicati in figura:



- *nota*: nessun valore per E.

Una **policy** è una specifica dell'arco da considerare per ogni nodo; essa si definisce **ottimale** se i suoi costi non sono mai superiori a costi calcolati con altre policy;

- $cost\_to\_goal$ viene calcolata *offline* e usata nella costruzione di una policy: da n si dovrebbe andare al vicino m che minimizza $cost(\langle n, m\rangle) + cost\_to\_goal(m)$; questa policy porta sempre a un obiettivo con un camminino di costo minimo partendo da qualsiasi nodo;
- alternative: 1) memorizzare il vicino per tutti i nodi *offline*, sfruttando le associazioni tra nodi nelle decisioni online sulle azioni oppure 2) fornire $cost\_to\_goal$ pre-calcolata e calcolare i vicini online.

Cenni sulla Complessità

- PD *lineare* in tempo e spazio rispetto alle dimensioni del grafo nella costruzione di $cost\_to\_goal$, tipicamente *esponenziali* nella lunghezza del percorso;
- Data $cost\_to\_goal$, determinare l'arco migliore richiede tempo *costante* rispetto alla grandezza del grafo, essendoci un numero limitato di vicini per nodo.

Utilità

- La PD può servire a costruire euristiche per A* e Branch-and-Bound: semplificando il problema fino a ottenere spazi di ricerca ridotti e trovando in tali spazi soluzioni di lunghezza ottimale; ciò definisce un *DB di pattern* usato nell'euristica per il problema originale;
- La PD è utile quando i nodi obiettivo sono elencabili *esplicitamente* (senza usare $goal()$), la soluzione cercata è un cammino di *costo minimo*, il grafo ridotto è *finito*, con memoria sufficiente per contenere la tabella, l'obiettivo non cambia (frequentemente). La policy può essere riusata per i diversi goal: si ammortizza il costo per produrre la tabella su diverse istanze dello stesso problema; tuttavia essa va calcolata per ogni diverso goal.

Riferimenti Bibliografici

[**PM23**] D. Poole, A. Mackworth: *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press. 3a ed. 2023 (Ch.3)

[**PMG86**] D. Poole, A. Mackworth, R. Goebel: *Computational Intelligence: A Logical Approach*. Oxford University Press. 1986

[**RN20**] S.J. Russell, P. Norvig: *Artificial Intelligence*. Pearson. 4th Ed. 2020 (Ch.3-4)

Link

[**graph_search**] http://aispace.org/search/
[**State_space_search**] https://en.wikipedia.org/wiki/State_space_search
[**A***] https://en.wikipedia.org/wiki/A*_search_algorithm
[**IDA***] https://en.wikipedia.org/wiki/Iterative_deepening_A*

Note

[1] cfr. specchietto Non-deterministic Choice in [PM23]