Agenti risolutori di problemi

Risolvere i problemi mediante ricerca

Alessio Micheli

a.a. 2022/2023

Credits: Maria Simi

Russell-Norvig

Agenti *risolutori di problemi*

- Adottano il paradigma della risoluzione di problemi come **ricerca** in uno **spazio di stati** (***problem solving***).
- Sono agenti con modello (storia percezioni e stati) che adottano una rappresentazione atomica dello stato
- Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire

- Prerequisiti: complessità asintotica O*()*
  (vedi appendice AIMA)

Il processo di risoluzione

- Passi da seguire:
  1. Determinazione obiettivo (un insieme di stati in cui obiettivo è soddisfatto)
  2. Formulazione del problema (vedi dopo) *Design (qui ancora «umano»)*
     - rappresentazione degli stati
     - rappresentazione delle azioni
  3. Determinazione della soluzione mediante ricerca (un piano)
  4. Esecuzione del piano *Qui soluzione algoritmica*

e.g. Viaggio con mappa: 1. Raggiungere Bucarest

2. Azioni=Guidare da una città all'altra. Stato = città su mappa

Che tipo di assunzioni?

- L'ambiente è statico
- Osservabile
  - so dove sono (e.g. viaggio con mappa)
- Discreto
  - un insieme finito di azioni possibili
- Deterministico (1 azione → 1 risultato)
  - si assume che l'agente possa eseguire il piano "ad occhi chiusi". Niente "può andare storto".

Formulazione del problema

Un problema può essere definito formalmente mediante cinque componenti:

1. Stato iniziale
2. Azioni possibili in *s*: Azioni(*s*)
3. Modello di transizione:

   Risultato: *stato* x *azione* → *stato*

   Risultato(s, a) = s', uno stato successore

1, 2 e 3 definiscono *implicitamente* lo *spazio degli stati*

(definirlo esplicitamente può essere molto oneroso, come in quasi tutti i problemi di IA, questo sarà rilevante nel seguito, come vedremo nelle prossime lezioni)

Formulazione del problema (cnt.)

4. Test obiettivo:
   - Un insieme di stati obiettivo
   - Goal-Test: stato $\rightarrow$ {*true*, *false*}
5. Costo del cammino
   - somma dei costi delle azioni (costo dei passi)
   - costo di passo: c(*s*, *a*, *s*')
   - Il costo di un'azione/passo non è mai negativo

Algoritmi di ricerca

*«Il processo che cerca una sequenza di azioni che raggiunge l'obiettivo è detto **ricerca**»*

Gli algoritmi di ricerca prendono in input un problema e restituiscono un cammino soluzione, i.e. un cammino che porta dallo stato iniziale a uno stato goal

- *Misura delle prestazioni*

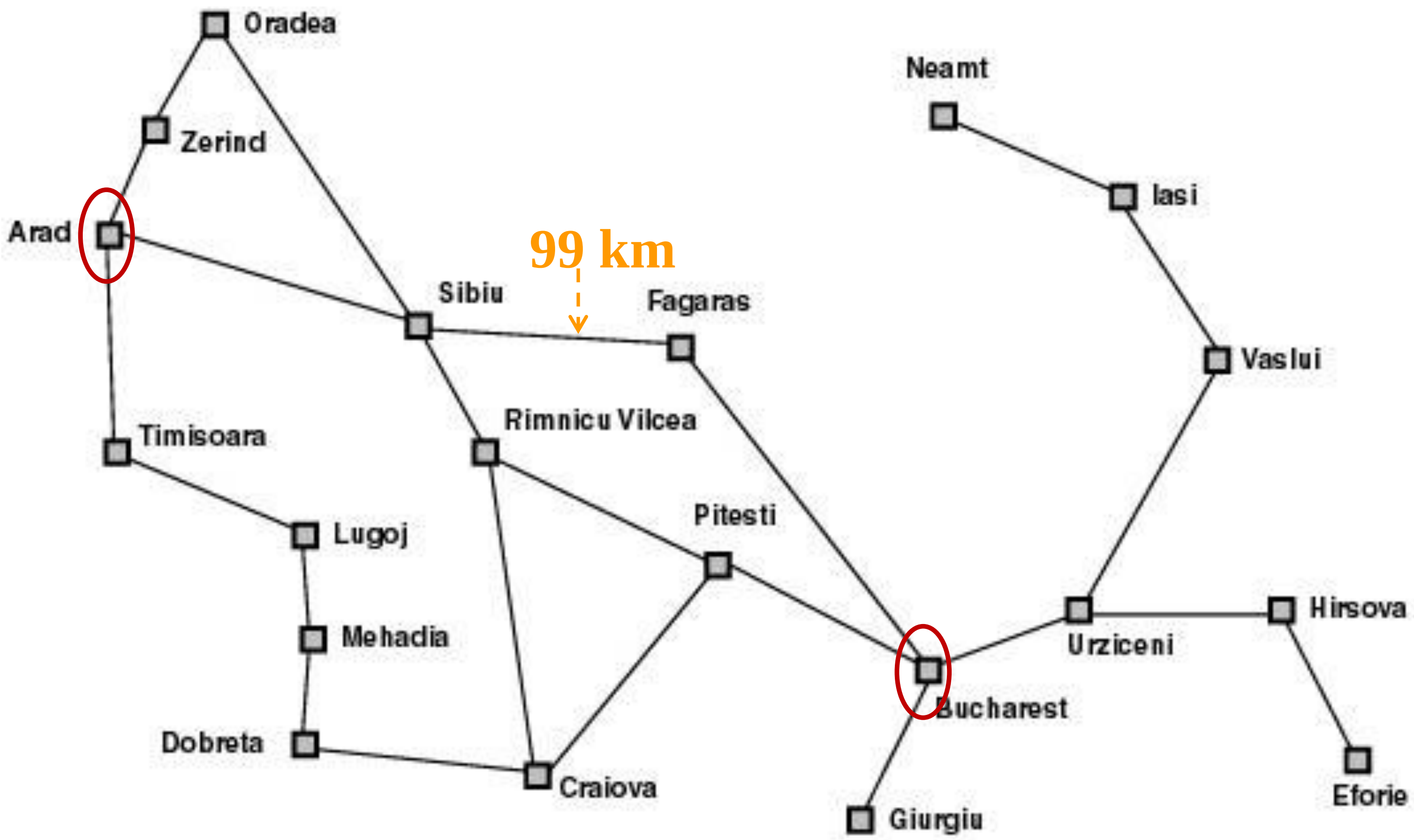
  Trova una soluzione? Quanto costa trovarla? Quanto efficiente è la soluzione?

  Costo totale = costo della ricerca + costo del cammino soluzione

Valuteremo gli alg. sul primo, ottimizzando il secondo

Itinerario: il problema

Caso che vedremo:
trovare il percorso più breve (in km) da una città di partenza a una città di arrivo



Itinerario: la formulazione (la scelta del livello di astrazione)

- *Stati*: <u>le città</u>. Es. *In*(Pitesti)

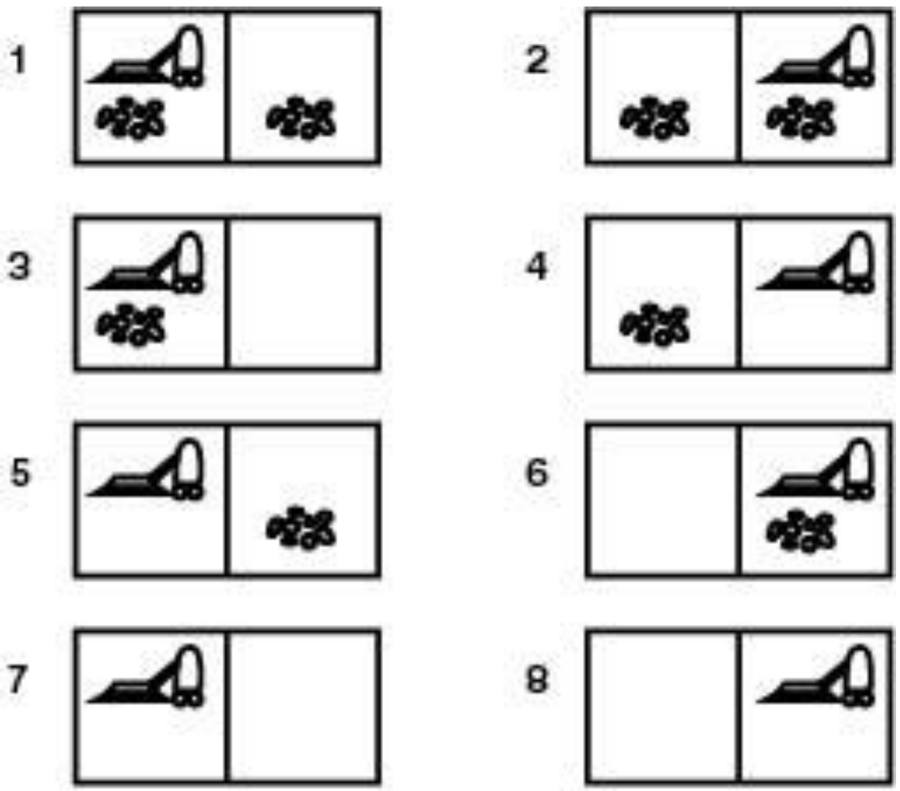
1. *Stato iniziale*: la città da cui si parte. *In*(Arad)
2. *Azioni*: spostarsi su una città vicina collegata
   - Azioni(*In*(Arad)) ={*Go*(Sibiu), *Go*(Zerind) …}
3. Modello di transizione
   - Risultato(*In*(Arad), *Go*(Sibiu)) = *In*(Sibiu)
4. *Test Obiettivo: {In(Bucarest)}*
5. *Costo del cammino*: somma delle lunghezze delle strade

- Lo <u>**spazio degli stati**</u> coincide con la rete (*grafo*) di collegamenti tra città i.e. grafo di stati collegati da azioni, rappresentabile in modo esplicito *in questo caso semplice*, tramite la mappa
- Astrazione dai dettagli: essenziale per "modellare"

Aspirapolvere: il problema (toy problem)

Versione semplice: solo due locazioni, sporche o pulite, l'agente può essere in una delle due

STATI



Percezioni:

Sporco

NonSporco

Azioni:

Sinistra (L)

Destra (R)

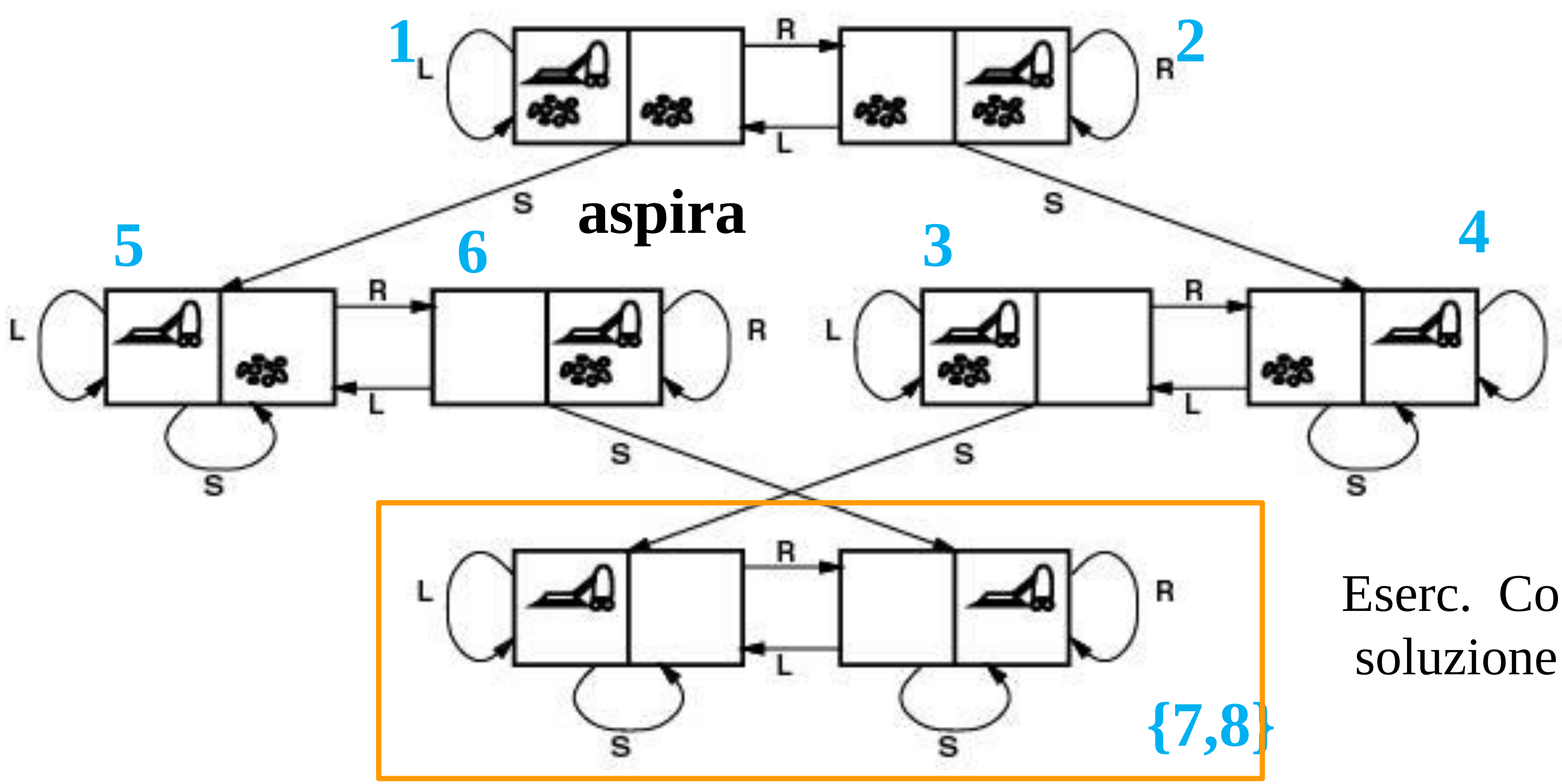
Aspira (S)

Aspirapolvere: formulazione

- *Obiettivo*: rimuovere lo sporco { 7, 8 }
- Ogni azione ha costo 1

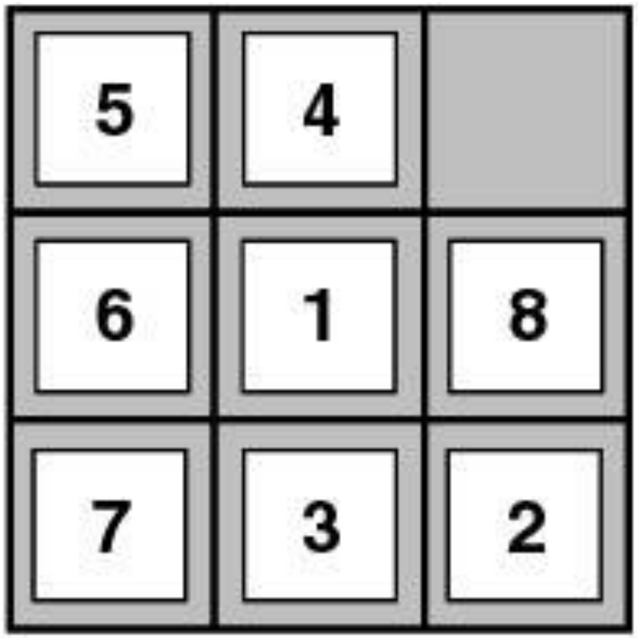
SPAZIO DEGLI STATI :

Grafo

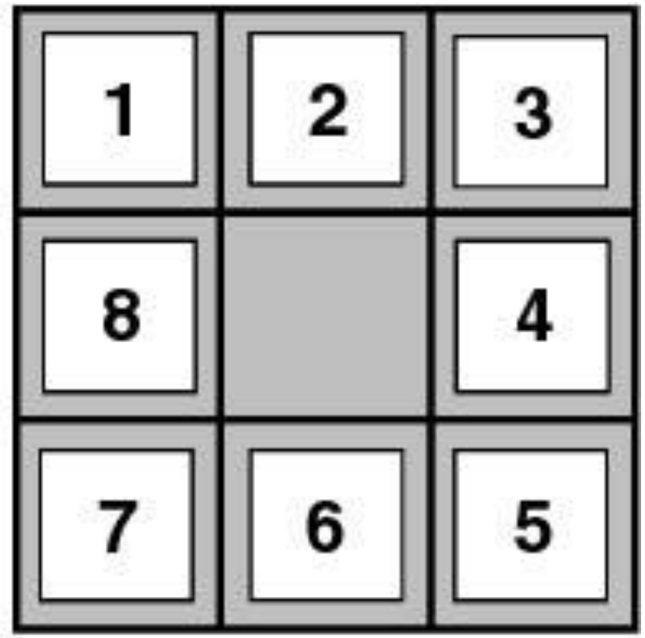


Eserc. Costo di soluzione min.?

Il puzzle dell’otto (o “rompicapo” a 8 tasselli)



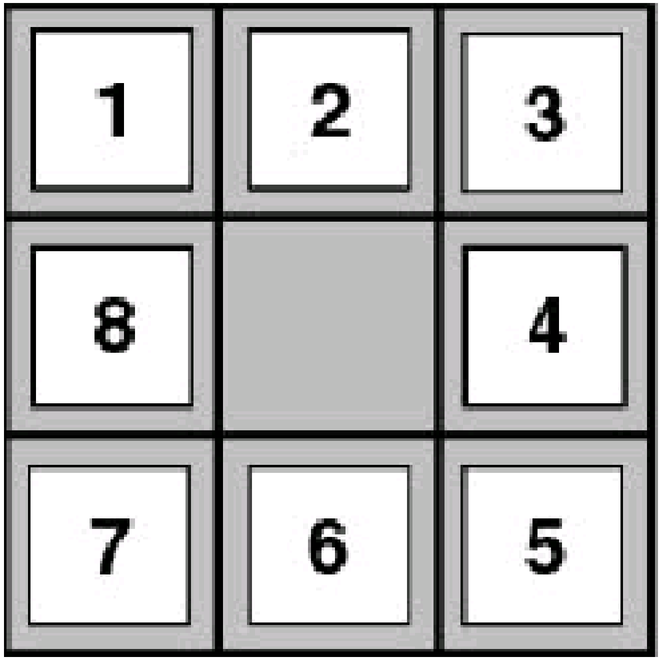
Start State



Goal State

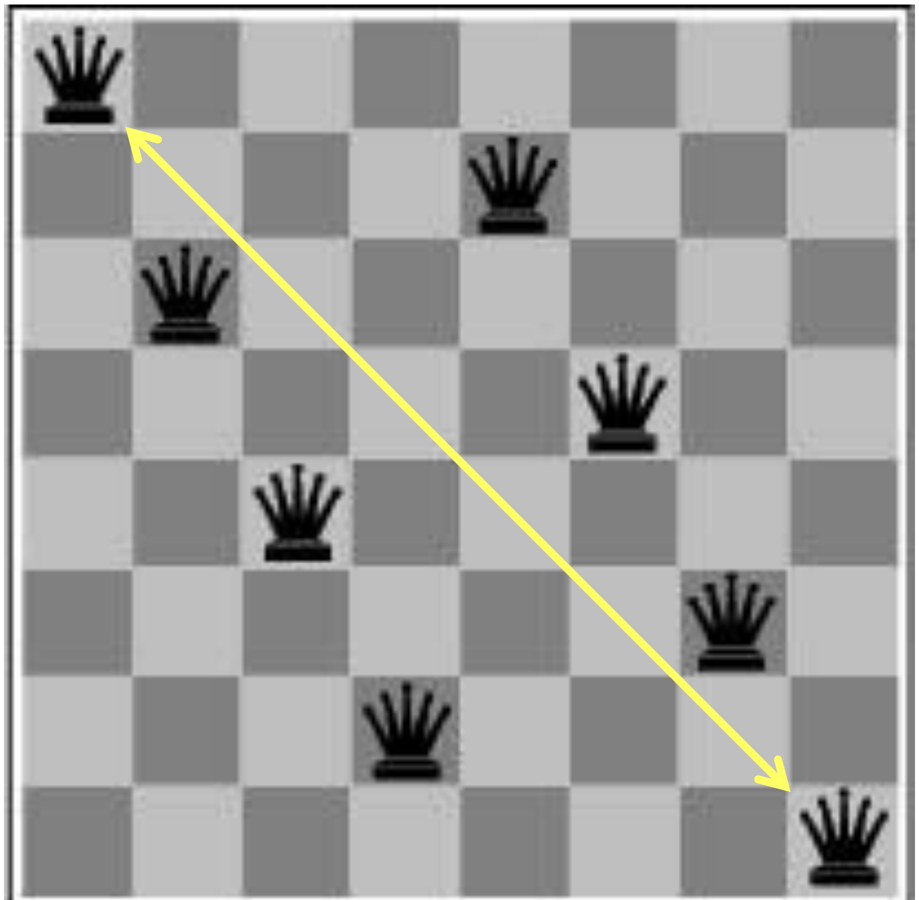
Puzzle dell'otto: formulazione

- *Stati*: possibili configurazioni della scacchiera
- *Stato iniziale*: una configurazione
- *Obiettivo*: una configurazione --->
  Goal-Test: Stato obiettivo? --->
- *Azioni*: mosse della *casella bianca*
  in sù: ↑ in giù: ↓
  a destra: → a sinistra: ←
- *Costo cammino*: ogni passo costa 1



- Lo *spazio degli stati* è un grafo con possibili cicli.
- NP-completo. Per 8 tasselli: 9!/2 =181K stati (*)! Ma risolvibile in poco tempo (ms). Se cresce no! (*)= http://www.cut-the-knot.com/pythagoras/fifteen.shtml

Le otto regine: il problema



Collocare 8 regine sulla scacchiera in modo tale che nessuna regina sia attaccata da altre: **Questa è una soluzione?**

Le otto regine:

Formulazione incrementale 1

Si aggiungono le regine una alla volta

- *Stati*: scacchiere con 0-8 regine
- *Goal-Test*: 8 regine sulla scacchiera, nessuna attaccata
- *Costo cammino*: zero (resta 8, per le 8 mosse effettive, e non è rilevante, interessa solo lo stato finale)
- *Azioni*: aggiungi una regina
- *Spazio stati*: $64 \times 63 \times \ldots \times 57 \sim 1.8 \times 10^{14}$ sequenze possibili da considerare! (quanti miliardi?)

I.e. la ricerca può essere molto onerosa!

Le otto regine:
Formulazione incrementale 2

- *Stati*: scacchiere con 0-8 regine, nessuna minacciata
- *Goal-Test*: 8 regine sulla scacchiera, nessuna minacciata
- *Costo cammino*: zero
- *Azioni*: aggiungi una regina nella colonna vuota più a destra ancora libera in modo che non sia minacciata

2057 sequenze da considerare (*)

Le 8 regine:
Formulazione a stato completo

- *Goal-Test*: 8 regine già sulla scacchiera, nessuna minacciata
- *Costo cammino*: zero
- *Stati*: scacchiere con 8 regine, una per colonna
- *Azioni*: sposta una regina nella colonna, se minacciata
- **Messaggio**: formulazioni diverse → portano a spazi stati diversi

Dimostrazione di teoremi

- Il problema:

  Dato un insieme di premesse

  $$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ v \Rightarrow q,\ t \Rightarrow r,\ s \Rightarrow v\}$$

  dimostrare una proposizione p

- Nel calcolo proposizionale consideriamo un'unica regola di inferenza, il *Modus Ponens (MP)*:

  $$\textit{Se } p \text{ e } p \Rightarrow q \ \textit{ allora } q$$

Dim. teoremi: formulazione

- *Stati*: insiemi di proposizioni
- *Stato iniziale*: un insieme di proposizioni (le premesse).
- *Stato obiettivo*: un insieme di proposizioni contenente il teorema da dimostrare. *Es p*.
- *Operatori:* l'applicazione del MP, che aggiunge teoremi

continua

Dim. teoremi: spazio degli stati

$$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ v \Rightarrow q,\ t \Rightarrow r,\ s \Rightarrow v\}$$

Ha applicato

$$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ v \Rightarrow q,\ t \Rightarrow r,\ \boxed{s \Rightarrow v,}\ v\}$$

$$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ v \Rightarrow q,\ \boxed{t \Rightarrow r,}\ s \Rightarrow v,\ r\}$$

$$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ \boxed{v \Rightarrow q,}\ t \Rightarrow r,\ s \Rightarrow v,\ v,\ q\}$$

Eureka

$$\boxed{\{s,\ t,\ q \Rightarrow p,\ \boxed{r \Rightarrow p,}\ v \Rightarrow q,\ t \Rightarrow r,\ s \Rightarrow v,\ r,\ p\}}$$

$$\boxed{\{s,\ t,\ \boxed{q \Rightarrow p,}\ r \Rightarrow p,\ v \Rightarrow q,\ t \Rightarrow r,\ s \Rightarrow v,\ v,\ q,\ p\}} \longrightarrow \textbf{Eureka}$$

Problemi reali (esempi)

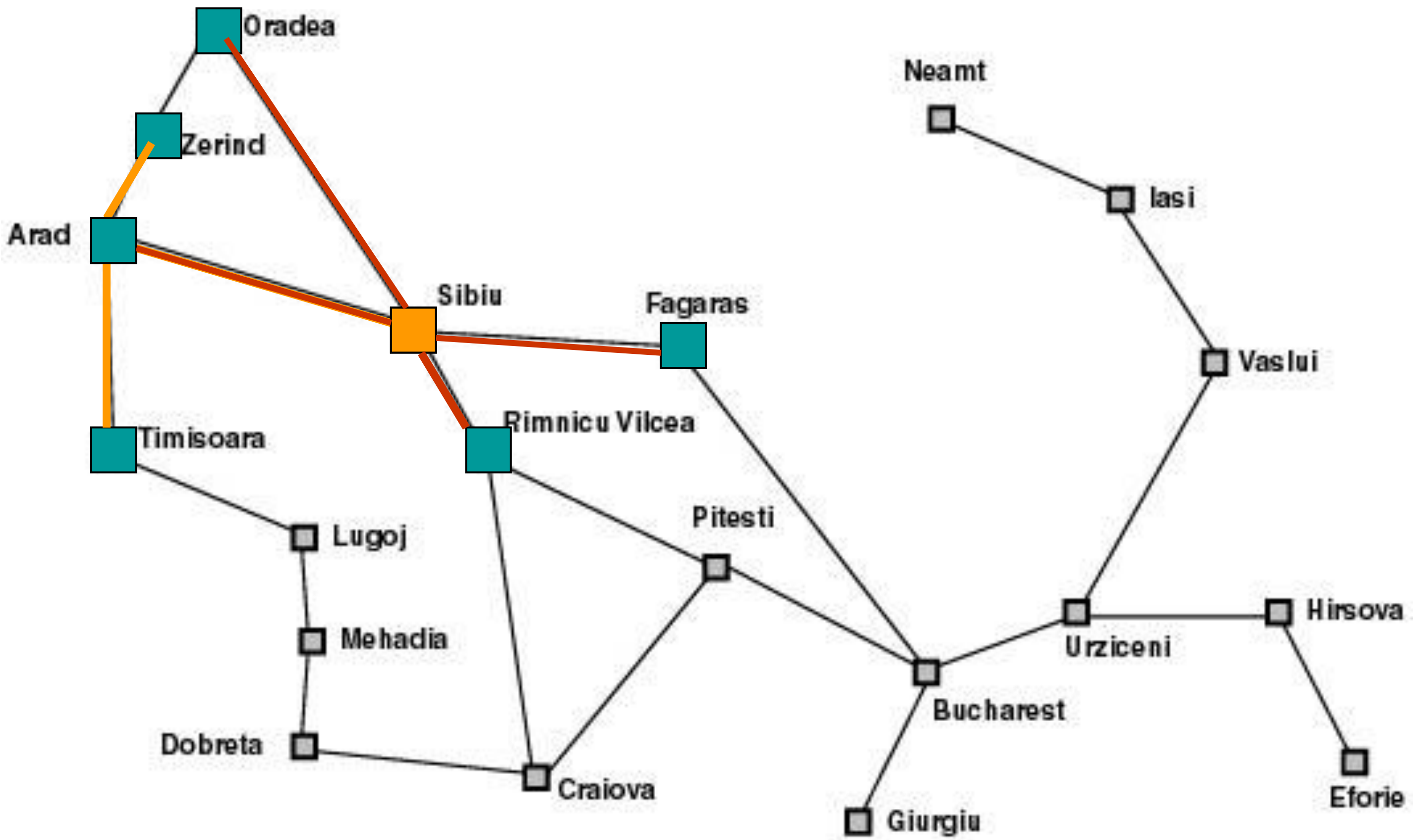
- Pianificazione di viaggi aerei
- Problema del commesso viaggiatore
- Configurazione VLSI
- Navigazione di robot (spazio continuo!)
- Montaggio automatico
- Progettazione di proteine
- …

WARNING – Versioni AIMA

- L'ed. IV AIMA ha cambiato alcune **terminologie** e impostazione (o anche eliminazione di alcuni) degli **algoritmi** (o analisi) rispetto all'ed. III.
- Anche per il 2023 seguiremo la formulazione degli algoritmi qui esposta nel seguito (che corrisponde in larga parte alla **ed. III AIMA**)

Ricerca della soluzione

Generazione di un **albero di ricerca** sovrapposto allo **spazio degli stati** (generato da *possibili* sequenze di azioni)

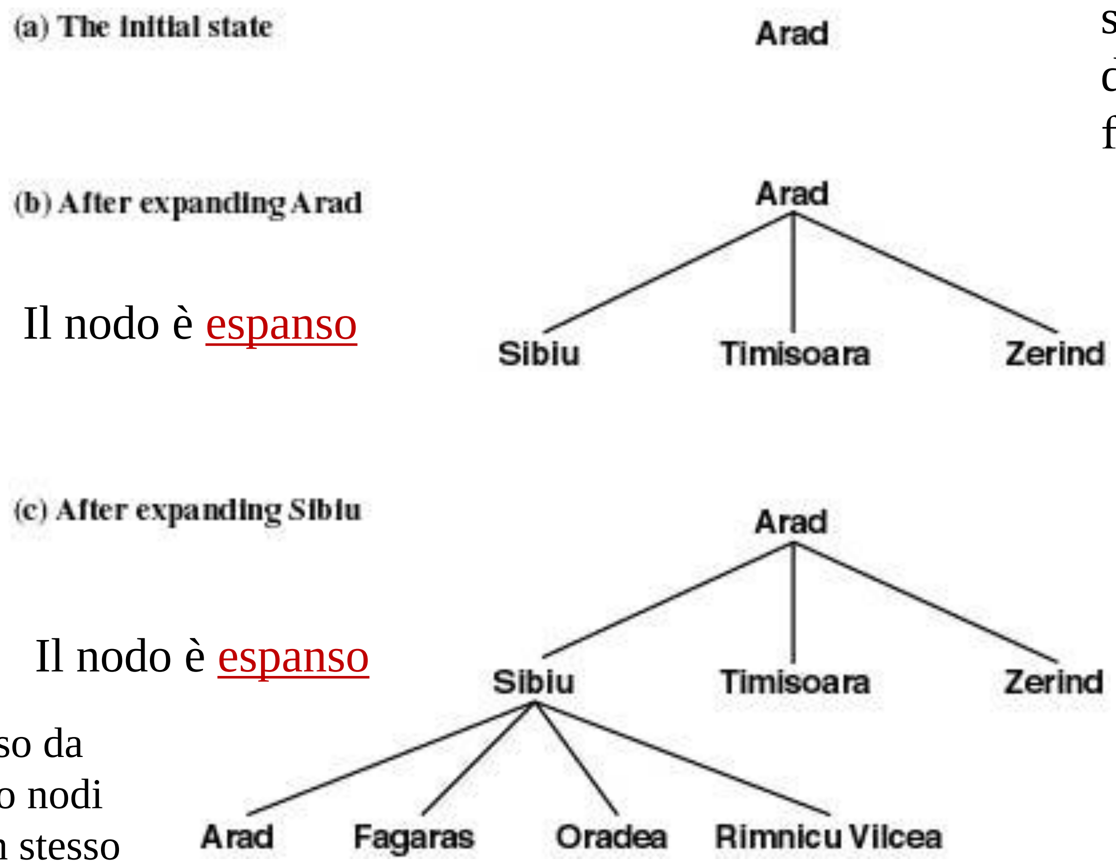


Ricerca: approfondire un'opzione, da parte le altre e ripredenderle se non trova soluzione

Ricerca della soluzione

Generazione di un albero di ricerca sovrapposto allo spazio degli stati

Nota: assumiamo sia noti concetti di padre, figlio, foglie, …



Il nodo è espanso

Frontiera

Il nodo è espanso

Nota: «nodo» diverso da «stato»: e.g. esistono nodi albero di ricerca con stesso stato (città) e.g Arad

Ricerca ad albero

Ossia senza controllare se i nodi (<u>stati</u>) siano già stati esplorati
Vedremo "a/su grafo" con questi controlli

function Ricerca-Albero (*problema)*
  returns soluzione oppure **fallimento**
 Inizializza la frontiera con stato iniziale del problema
 loop do
  if *la frontiera è vuota* **then return fallimento**
  Scegli un nodo foglia da espandere e rimuovilo dalla frontiera*
  if *il nodo contiene uno stato obiettivo*
   then return *la soluzione corrispondente*
  Espandi il nodo e aggiungi i successori alla frontiera

espansione

esamina opzione

passa alle altre opzioni

*<u>Strategia</u>: quale scegliere?

I nodi dell'albero di ricerca

Un nodo *n* è una struttura dati con quattro componenti:

- Uno stato: *n*.stato
- Il nodo padre: *n*.padre
- L'azione effettuata per generarlo: *n*.azione
- Il costo del cammino dal nodo iniziale al nodo: *n*.costo-cammino indicata come $g(n)$ *(=padre.costo-cammino+costo-passo ultimo)*

Struttura dati per la frontiera

- *Frontiera*: lista dei nodi in attesa di essere espansi (le foglie dell'albero di ricerca).
- La frontiera è implementata come una *coda* con operazioni:
  - Vuota?(*coda*)
  - POP(*coda*) estrae il primo elemento
  - Inserisci(elemento, *coda*)
  - Diversi tipi di coda hanno diverse funzioni di inserimento e implementano *strategie* diverse

Diversi tipi di <u>strategie</u> (di ricerca)

- FIFO- First In First Out → BF (Breadth-first)
  - Viene estratto l'elemento più vecchio (in attesa da più tempo); in nuovi nodi sono aggiunti alla fine.
- LIFO-Last In First Out → DF (Depht-first)
  - Viene estratto il più recentemente inserito; i nuovi nodi sono inseriti all'inizio (pila)
- Coda con priorità → UC, et altri successivi
  - Viene estratto quello con priorità più alta in base a una funzione di ordinamento; dopo l'inserimento dei nuovi nodi <u>si riordina.</u>

Strategie non informate (che vedremo)

- Ricerca in ampiezza (BF)
- Ricerca in profondità (DF)
- Ricerca in profondità limitata (DL)
- Ricerca con approfondimento iterativo (ID)
- Ricerca di costo uniforme (UC)

Vs strategie di ricerca euristica (o informata): fanno uso di informazioni riguardo alla distanza stimata dalla soluzione (lez. prossima)

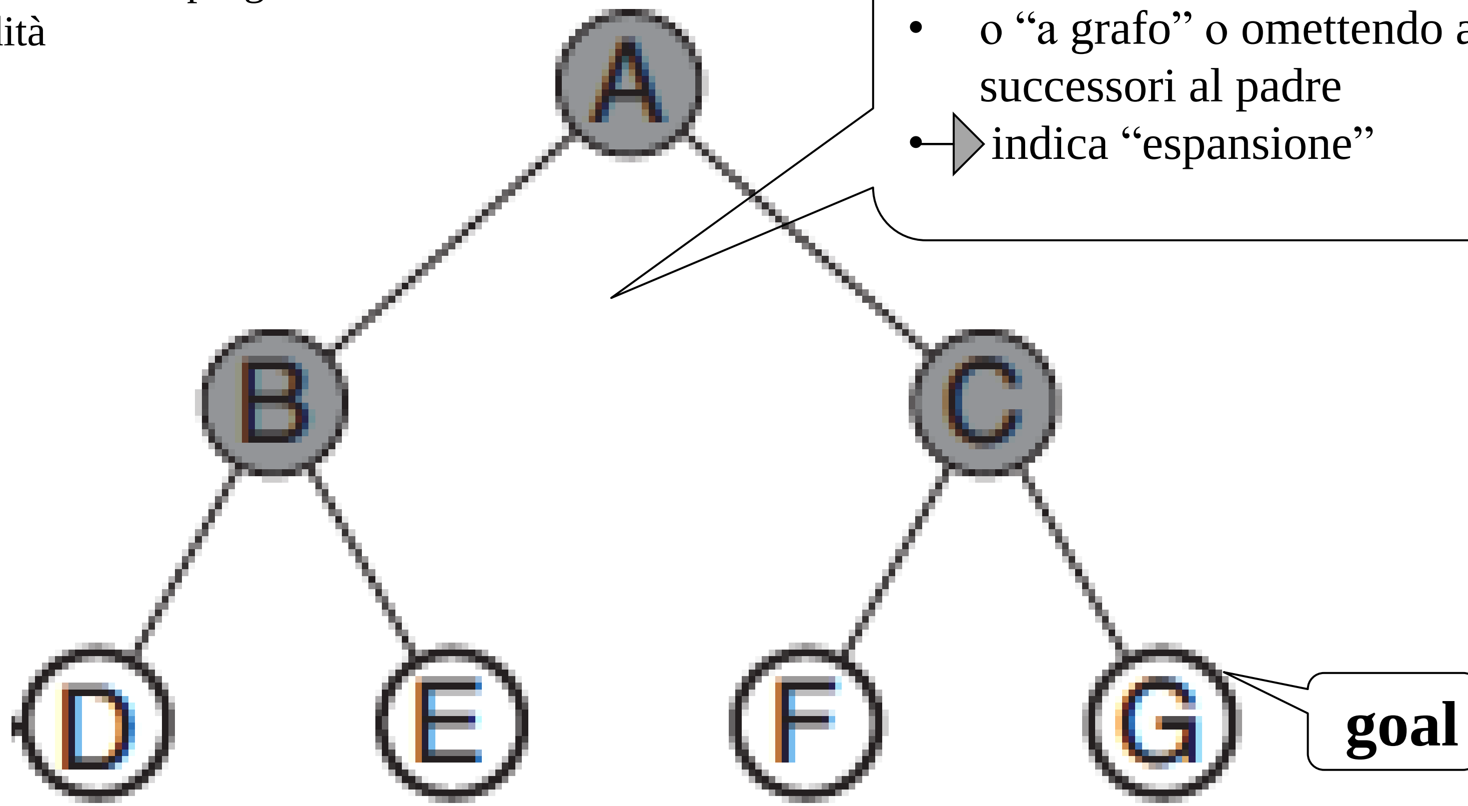
Valutazione di una strategia

- *Completezza*: se la soluzione esiste viene trovata
- *Ottimalità* (ammissibilità): trova la soluzione migliore, con costo minore (per il «costo del cammino soluzione»)
- *Complessità in tempo*: tempo richiesto per trovare la soluzione
- *Complessità in spazio*: memoria richiesta

per il «costo della ricerca»

Ricerca in ampiezza (BF -Breadth-first*)

O come esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità



Note per l'esempio specifico:
- o "a grafo" o omettendo archi da successori al padre
- indica "espansione"

Implementata con una coda che inserisce alla fine (**FIFO**)

* Warning: versus Ed. IV AIMA, per noi BF è Breadth First, Best First sarà dopo

Ricerca in ampiezza - BF (su/ad albero*)

(*) senza gestire problema stati già esplorati

function Ricerca-Ampiezza-A (*problema)*

returns soluzione oppure **fallimento**

nodo = un nodo con *stato il problema.stato-iniziale e costo-di-cammino=0*

if *problema.Test-Obiettivo(nodo.*Stato*)* **then return** Soluzione(*nodo*)

frontiera = una coda FIFO con *nodo* come unico elemento

loop do

if *Vuota?*(*frontiera*) **then return fallimento**

nodo = POP(*frontiera*)

for each azione **in** problema.Azioni(nodo.Stato) **do**

espansione

figlio = Nodo-Figlio(*problema*, *nodo, azione*) [costruttore: vedi AIMA]

if Problema.TestObiettivo(figlio.Stato) **then return** Soluzione(*figlio*)

frontiera = Inserisci(*figlio*, *frontiera*) /* frontiera gestita come coda FIFO

end

Nota che in questa versione i nodo.stato sono goal-tested al momento in cui sono generati, **anticipato→ più efficiente**, si ferma appena trova goal prima di espandere

Ricerca-grafo in ampiezza – BF (su grafo)

(*) evitiamo di espandere (nodi con) stati già esplorati

```
function Ricerca-Ampiezza-g (problema)
    returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    if problema.Test-Obiettivo(nodo.Stato) then return Soluzione(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto
  loop do
    if Vuota?(frontiera) then return fallimento
    nodo = POP(frontiera); aggiungi nodo.Stato  a esplorati
    for each azione in problema.Azioni(nodo.Stato) do
          figlio = Nodo-Figlio(problema, nodo, azione)
          if figlio.Stato non è in esplorati e non è in frontiera then
             if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
             frontiera = Inserisci(figlio, frontiera)   /* in coda
```

Aggiunte in verde per gestire gli stati ripetuti

Nota che in questa versione i nodo.stato sono goal-tested al momento in cui sono generati, anticipato → più efficiente, si ferma appena trova goal prima di espandere

In Python (notate l'aderenza allo slide prima)

```
def breadth_first_search(problem):  """Ricerca-grafo in ampiezza"""
    explored = [] # insieme degli stati già visitati (implementato come una lista)
    node = Node(problem.initial_state)  # il costo del cammino è inizializzato nel
costruttore del nodo
    if problem.goal_test(node.state):
        return node.solution(explored_set = explored)
    frontier = FIFOQueue() # la frontiera e' una coda FIFO
    frontier.insert(node)
    while not frontier.isempty(): # seleziona il nodo per l'espansione
        node = frontier.pop()
        explored.append(node.state) # inserisce il nodo nell'insieme dei nodi esplorati
        for action in problem.actions(node.state):
            child_node = node.child_node(problem,action)
                if (child_node.state not in explored) and (not
              frontier.contains_state(child_node.state)):
              if problem.goal_test(child_node.state):
                         return child_node.solution(explored_set = explored)
                # se lo stato non e' uno stato obiettivo allora inserisci il nodo nella frontiera
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

Analisi complessità spazio-temporale (BF)

- Assumiamo

  b = fattore di ramificazione (**b**ranching)
  (numero max di successori)

  d = profondità del nodo obiettivo più superficiale (**d**epth) [più vicino all' iniziale]

  m = lunghezza massima dei cammini nello spazio degli stati (**m**ax)

Ricerca in ampiezza: analisi

- Strategia *completa*
- Strategia *ottimale* <u>se gli operatori hanno tutti lo stesso costo *k*</u>, cioè $g(n) = k \cdot \text{depth}(n)$, dove $g(n)$ è il costo del cammino per arrivare a *n*
- Complessità nel tempo (nodi generati)

  $T(b, d) = 1 + b + b^2 + \ldots + b^d \rightarrow O(b^d)$ [*b* figli per ogni nodo]

- Esercizio: e se spostassimo il test-obiettivo post-generazione? (vedi primo schema alg.)
- Nota (*): Riflettere che il numero nodi cresce exp., non assumiamo di conoscere già il grafo ne una notazione di linearità nel numero nodi . Questo è tipico dei problemi in AI (pensate a quelli generati per le configurazioni dei giochi, con rappresentazione *implicita* dello spazio stati, non esplicitamente/staticamente in spazi enormi).
- Complessità spazio (nodi in memoria): $O(b^d)$ [<u>frontiera</u>]

Nota: O() notazione per la complessità asintotica

Ricerca in ampiezza: esempio

- *Esempio*: b=10; 1 milione nodi al sec generati; 1 nodo occupa 1000 byte

Piu incisivo!

Profondità	**Nodi**	**Tempo**	**Memoria**
2	110	0,11 ms	107 kilobyte
4	11.100	11 ms	10,6 megabyte
6	10^6	1.1 sec	1 gigabyte
8	10^8	2 min	103 gigabyte
10	10^{10}	3 ore	10 terabyte
12	10^{12}	13 giorni	1 petabyte
14	10^{14}	3,5 anni	1 esabyte

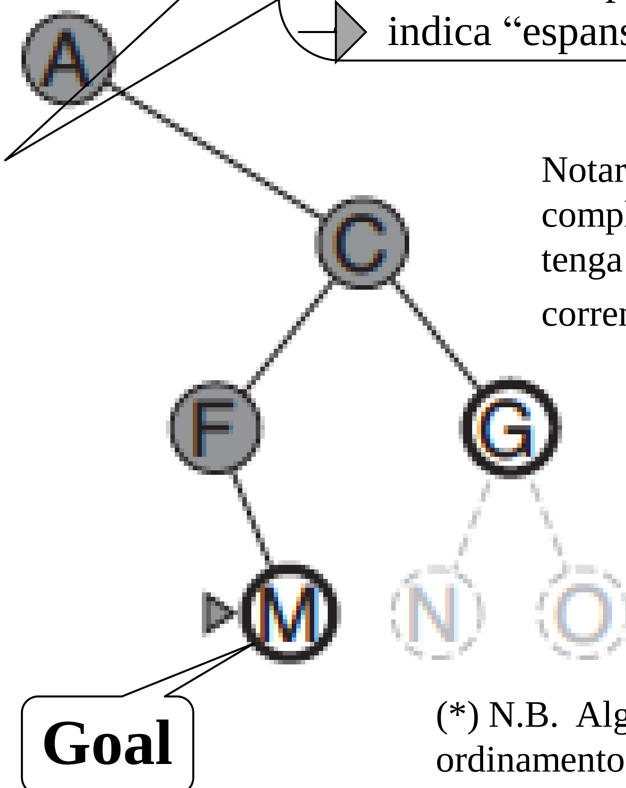
Scala male: solo istanze piccole!

Ricerca in profondità (DF)

Note per l'esempio specifico:

- Qui insert in ordine alfabetico inverso: C, B (*) (o espansione in ordine alfabetico a pari livello)
- "a grafo" o omettendo archi da successori al padre

indica "espansione"



Notare come cancelli rami completamente esplorati ma tenga tutti i fratelli del path corrente: memoria solo ***b*** **x** ***m***

(*) N.B. Alg. Python avrà ordinamento diverso a pari profondità Si vedrà a esercitazione

Implementata da una coda che mette i successori in testa alla lista (**LIFO**, pila o stack). Alg. generale visto all'inizio (a grafo o albero)

Ricerca in profondità: analisi [versione su albero]

- Se *m* → lunghezza massima dei cammini nello spazio degli stati
- *b* → fattore di diramazione
  - Tempo: $O(b^m)$ [che può essere > $O(b^d)$]
  - <u>Occupazione memoria: *bm*</u> [frontiera sul cammino]
- [Versione su/ad albero, caso standard per DF]: Strategia *non completa* (possibili loop) e *non ottimale*.
- Ma … Drastico risparmio in memoria:

  BF d=16 10 esabyte

  DF d=16 156 Kbyte

Ricerca in profondità: analisi [versione su grafo]

- In caso di DF con visita grafo si perdrebbero i vantaggi di memoria: la memoria torna da ***bm*** a tutti i possibili stati (potenzialmente, caso pessimo, esponenziale come BF*) (per mantenere la lista dei visitati/esplorati), ma cosi DF diviene ***completa*** in spazi degli stati finiti (tutti i nodi verranno espansi nel caso pessimo)
- Comunque resta non completa in spazi infiniti
- È possibile controllare anche solo i nuovi nodi rispetto al <u>cammino radice-nodo</u> corrente senza aggravio di memoria (evitando però cosi solo i <u>cicli in spazi finiti</u> ma non i cammini ridondanti: vedi dopo)

*di nuovo: pochi in mappa (20 città), ma si pensi al gioco dell'otto, scacchi etc in cui le possibili mosse generano un enorme numero di configurazioni diverse (stati)

Ricerca in profondità (DF) <u>ricorsiva</u>

- Ancora più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (solo *m* nodi nel caso pessimo)
- Realizzata da un algoritmo ricorsivo "con backtracking" che non necessita di tenere in memoria *b* nodi per ogni livello, ma salva lo stato su uno stack a cui torna in caso di fallimento per fare altri tentativi (generando i nodi fratelli al momento del backtracking).

Ricerca in profondità —DF ricorsiva (su albero)

```
function Ricerca-DF-A (problema)
    returns soluzione oppure fallimento
    return Ricerca-DF-ricorsiva(CreaNodo(problema.Stato-iniziale), problema)

function Ricerca-DF-ricorsiva(nodo, problema)
    returns soluzione oppure fallimento
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    else
    for each azione in problema.Azioni(nodo.Stato) do
        figlio = Nodo-Figlio(problema, nodo, azione)
        risultato = Ricerca-DF-ricorsiva(figlio, problema)
        if  risultato ≠ fallimento then return risultato
    return fallimento
```

In Python

```
def recursive_depth_first_search(problem, node):
    """Ricerca in profondita' ricorsiva """
    # controlla se lo stato del nodo e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    # in caso contrario continua
    for action in problem.actions(node.state):
        child_node = node.child_node(problem, action)
        result = recursive_depth_first_search(problem, child_node)
        if result is not None:
            return result
    return None #con fallimento
```

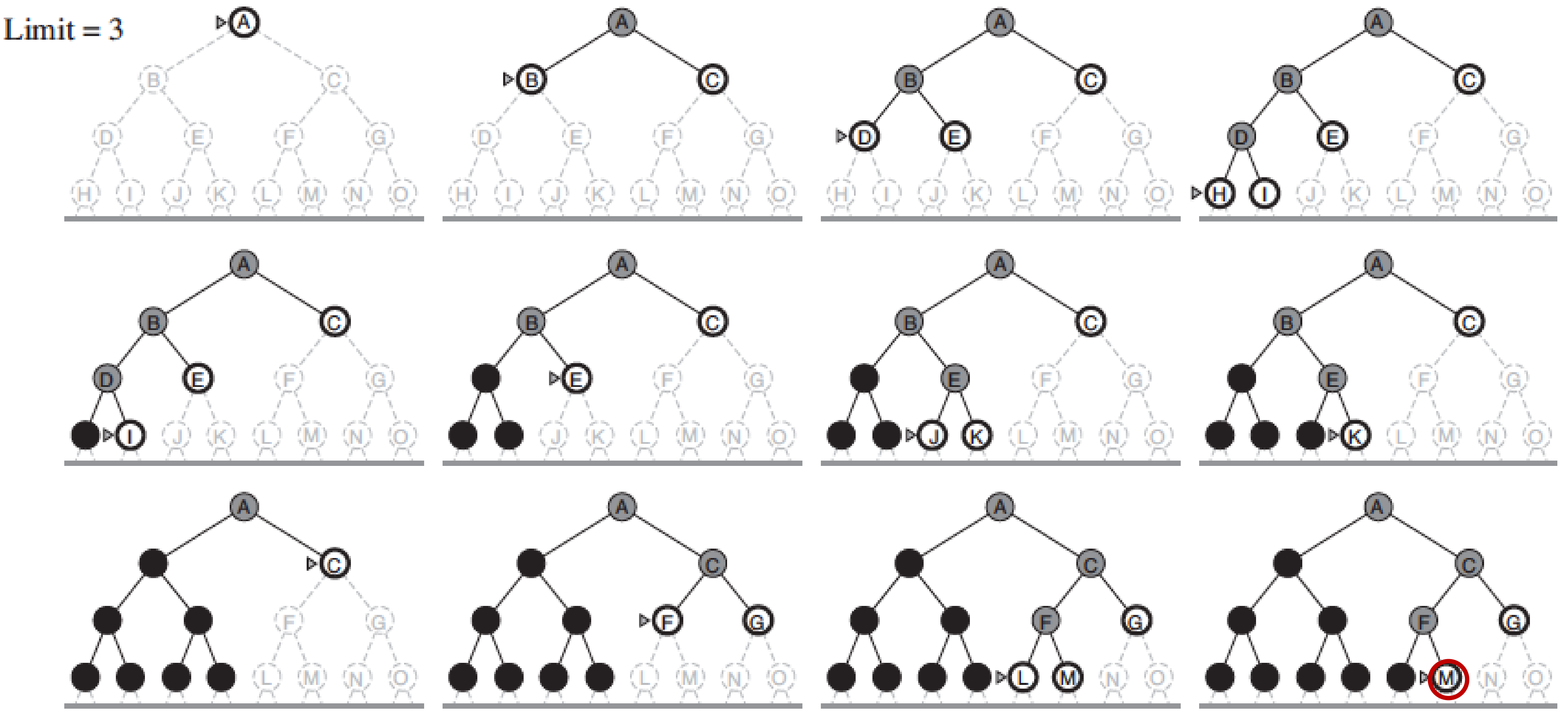
Ricerca in profondità limitata (DL)

- Si va in profondità fino ad un certo livello predefinito ℓ
- *Completa* per problemi in cui si conosce un limite superiore per la profondità della soluzione.

  Es. Route-finding limitata dal numero di città – 1
- Completo: se $d < \ell$ (d profondità nodo obiettivo più superf.)
- Non ottimale
- Complessità tempo: $O(b^{\ell})$
- Complessità spazio: $O(b\ell)$

Approfondimento iterativo (ID)

Si prova DF (DL) con limite di profondità 0, poi 1, poi 2, poi 3, … fino a trovare la soluzione



ID: analisi

- Miglior compromesso tra BF e DF

  BF: $b+b^2+ \ldots +b^{d-1}+b^d$ con $b=10$ e $d=5$

  10+100+1000+10.000+100.000=111.110
- ID: I nodi dell'ultimo livello generati una volta, quelli del penultimo 2, quelli del terzultimo 3 … quelli del primo *d* volte

  ID: $(d)b+(d-1)\,b^2+ \ldots +3b^{d-2}+2b^{d-1}+1b^d$

  = 50+400+3000+20.000+100.000=123450
- Complessità tempo: $O(b^d)$ (se esiste soluzione)
- Spazio: $O(bd)$ (se esiste soluzione) versus $O(b^d)$ della BF

Ergo: Vantaggi della BF (**completo**, **ottimale** se costo fisso oper. K), con tempi analoghi *ma* costo memoria analogo a quello di DF

Direzione della ricerca

Un problema ortogonale alla strategia è la *direzione della ricerca*:

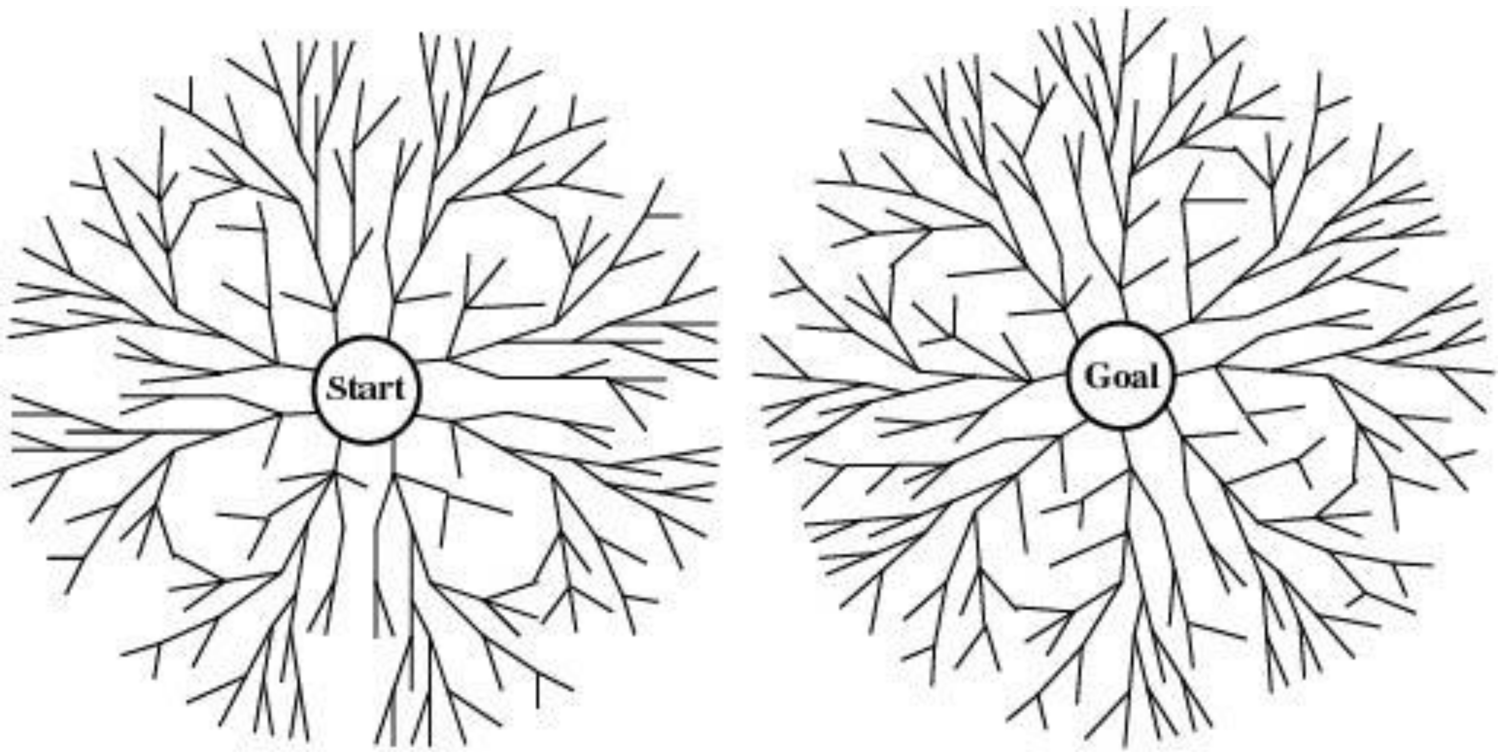
- ricerca *in avanti* o *guidata dai dati*: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo;
- ricerca *all'indietro* o *guidata dall'obiettivo*: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

Quale direzione?

- Conviene procedere nella direzione in cui il fattore di diramazione è minore
- Si preferisce ricerca all'indietro quando, e.g:
  - l'obiettivo è chiaramente definito (e.g. theorem proving) o si possono formulare una serie limitata di ipotesi;
- Si preferisce ricerca in avanti quando, e.g.:
  - gli obiettivi possibili sono molti (design)

Ricerca bidirezionale

Si procede nelle due direzioni fino ad incontrarsi



Ricerca bidirezionale: analisi

- Complessità tempo: $O(b^{d/2})$ [/2 = radice quadrata!]

  (assumendo test intersezione in tempo costante, es. hash table)
- Complessità spazio: $O(b^{d/2})$

  (almeno tutti i nodi in una direzione in memoria, es. usando BF)

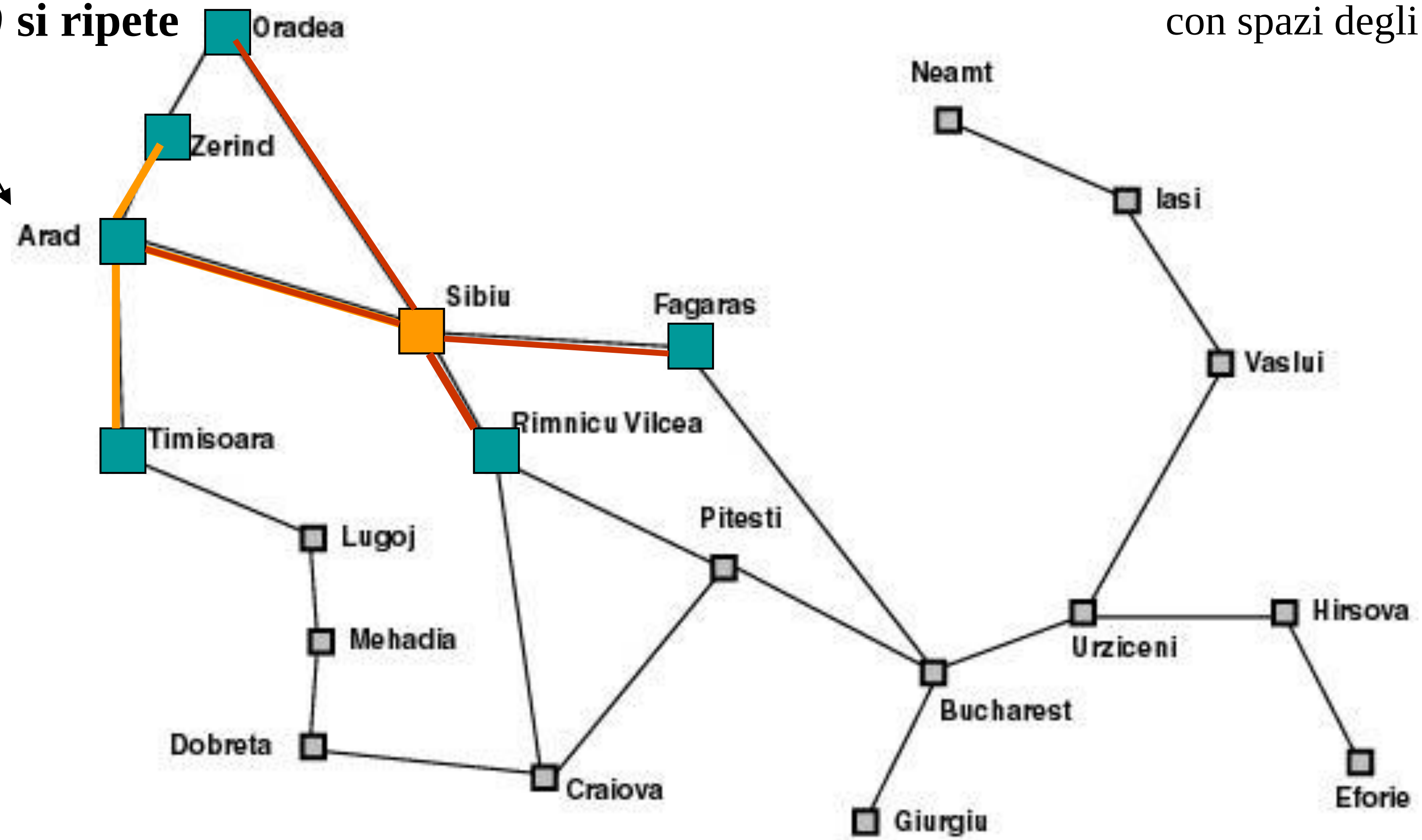
NOTA: non sempre applicabile, es. predecessori non definiti, troppi stati obiettivo …

Ricerca "ad albero"/"a grafo": cammini ciclici

I cammini ciclici rendono gli alberi di ricerca infiniti

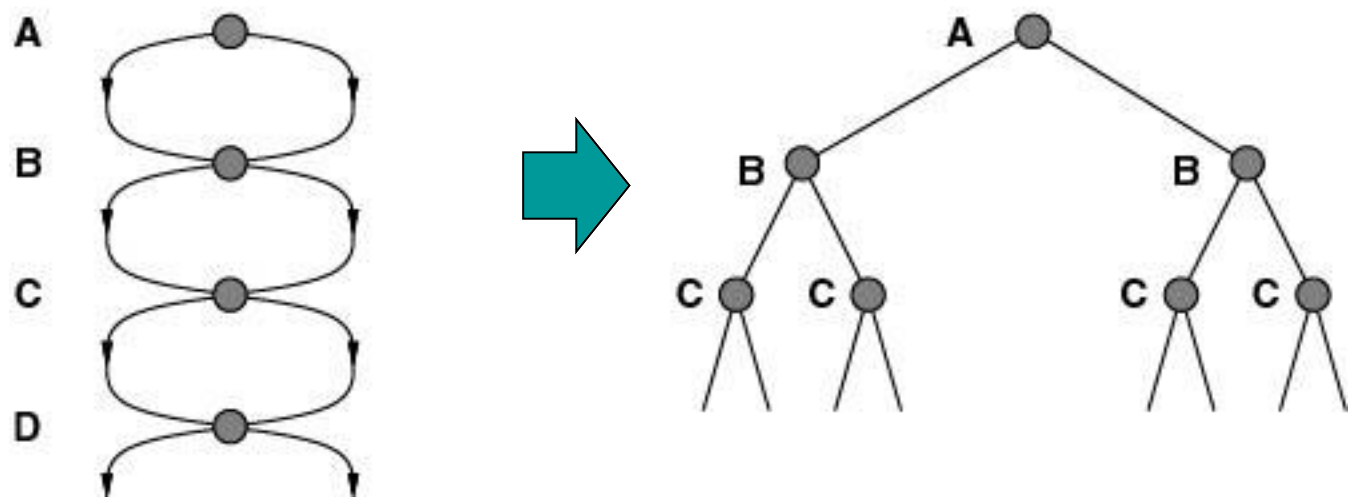
Alberi di ricerca infiniti
con spazi degli stati finiti

***In(Arad)* si ripete**

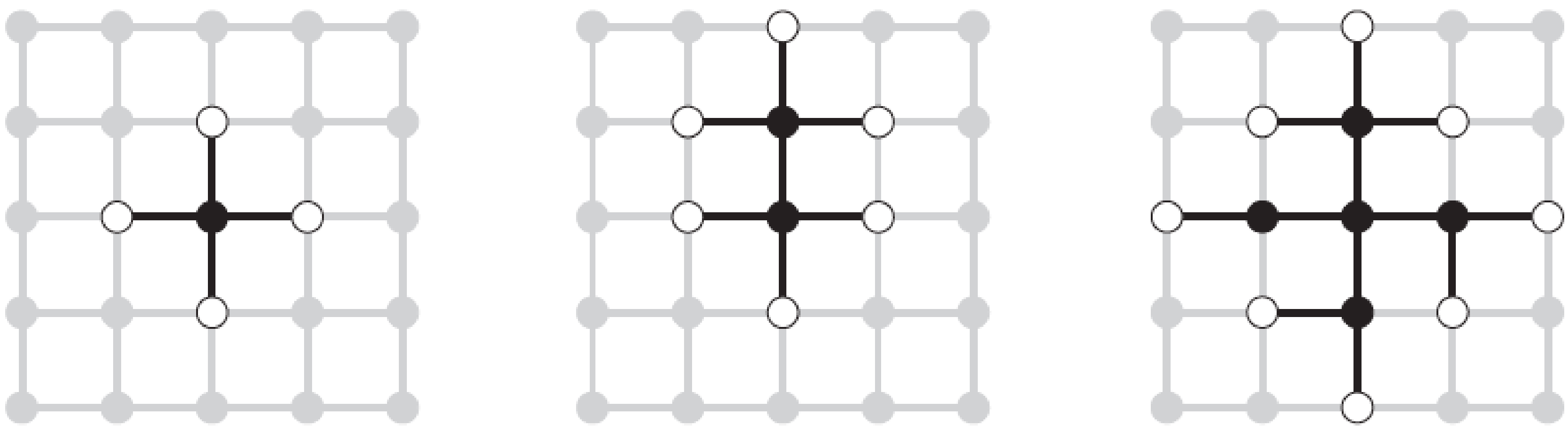


Ricerca su grafi: ridondanze

Su spazi di stati a grafo si possono generare più volte gli stessi nodi (o meglio nodi con stesso stato) nella ricerca, anche in assenza di cicli (cammini ridondanti)



Ridondanza nelle griglie


(a) (b) (c)

Visitare stati già visitati fa compiere lavoro inutile. Come evitarlo?

Costo: 4^d ma solo $\sim 2d^2$ stati distinti

Compromesso tra spazio e tempo

- Ricordare gli stati già visitati occupa spazio (es. lista *eplorati* in visita a grafo) ma ci consente di evitare di visitarli di nuovo
- *Gli algoritmi che dimenticano la propria storia sono destinati a ripeterla!*

Tre soluzioni

In ordine crescente di costo e di efficacia:

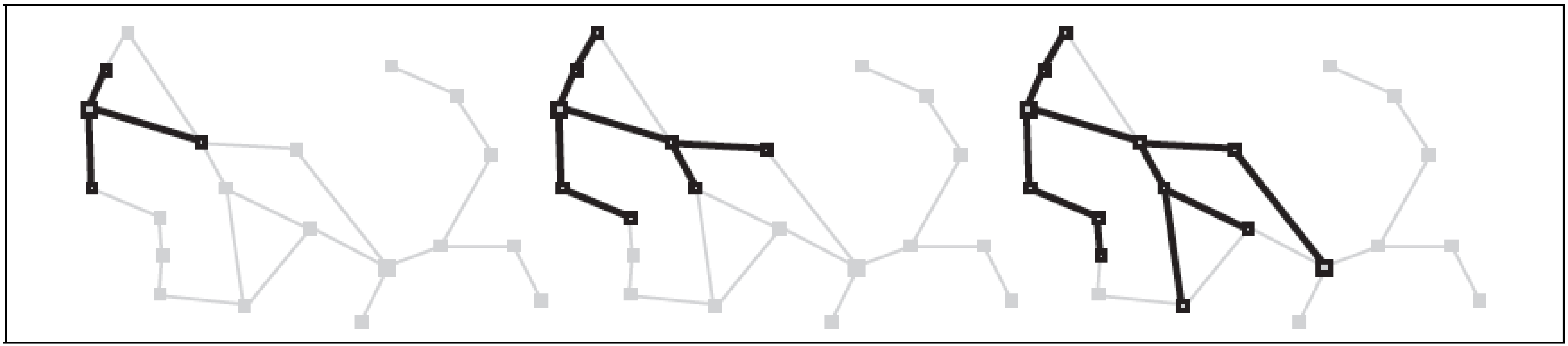
- Non tornare nello stato da cui si proviene: si elimina il genitore dai nodi successori (non evita i cammini ridondanti)
- Non creare cammini con cicli: si controlla che i successori non siano antenati del nodo corrente (detto per la DF)
- Non generare nodi con stati già visitati/*esplorati*: ogni nodo visitato deve essere tenuto in memoria per una complessità O(*s*) dove *s* è il numero di stati possibili (e.g. *hash table* per accesso efficiente).
  - Repetita: Il costo può essere alto: in caso di DF (profon.) la memoria torna da *bm* a tutti gli stati, ma diviene una ricerca *completa* (per spazi finiti). *Ma in molti casi gli stati crescono exp. (gioco otto, scacchi, …)*

Ricerca "su grafi" (repetita!)

- Mantiene una lista dei nodi (<u>stati</u>) visitati/esplorati (anche detta *lista chiusa*) (*)
- Prima di espandere un nodo si controlla se lo stato era stato già incontrato prima o è gia nella frontiera
- Se questo succede, il nodo appena trovato non viene espanso
- Ottimale solo se abbiamo la garanzia che il costo del nuovo cammino sia maggiore o uguale (cioè che il nuovo cammino non conviene) (verrà discusso in seguito)

- (*) Ed. IV AIMA: introduce il termine di insieme di stati "raggiunti" che include sia (gli stati del)la frontiera che la lista degli esplorati

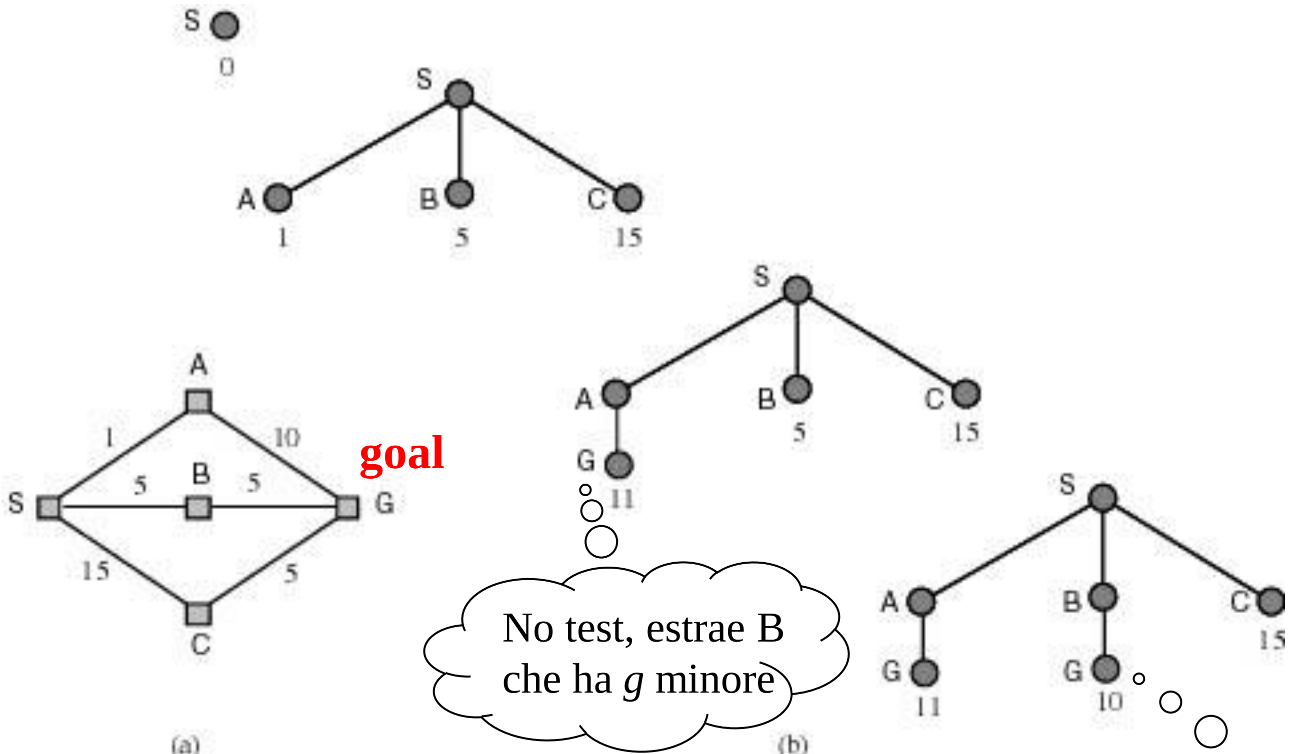
Ricerca sul grafo della Romania

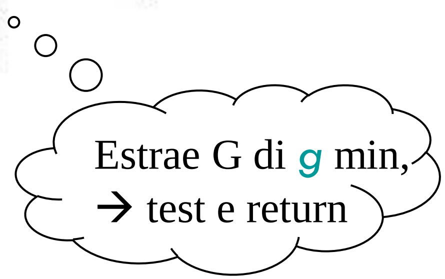


- La ricerca su grafo esplora uno stato al più una volta
- Proprietà: La **frontiera** separa i nodi <u>esplorati</u> da quelli <u>non-esplorati</u> [ogni cammino dallo stato iniziale a inesplorati deve attraversare uno stato della frontiera]

Ricerca di costo uniforme (UC)

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): *si sceglie il nodo di costo minore sulla frontiera* (si intende il costo *g(n)* del cammino), si espande sui contorni di *uguale (o meglio uniforme) costo (e.g. in km)* invece che sui contorni di uguale profondità (BF)





Implementata da una <u>coda ordinata per costo cammino crescente</u> (in cima i nodi di costo minore)

Ricerca UC (su albero)

= primo schema di alg. visto

function Ricerca-UC-A (*problema)*
  returns soluzione oppure **fallimento**
  nodo = un nodo con *stato il problema.stato-iniziale e costo-di-cammino=0*
  frontiera = una **coda con priorità** con *nodo* come unico elemento
 loop do
  if *Vuota?*(*frontiera*) **then return fallimento**
  nodo = POP(*frontiera*)
  if problema.TestObiettivo(*nodo*.Stato) **then return** Soluzione(*nodo*)
  for each azione **in** problema.Azioni(nodo.Stato) **do**
    figlio = Nodo-Figlio(*problema*, *nodo, azione*)
    frontiera = Inserisci(*figlio*, *frontiera*) /* in coda con priorità
end

Posticipato* per vedere il costo minore su *g* (diverso da BF, ma tipico per coda con priorità)

Ricerca-grafo UC

function Ricerca-UC-G (*problema)*

returns soluzione oppure **fallimento**

nodo = un nodo con *stato il problema.stato-iniziale e costo-di-cammino=0*

frontiera = una coda con priorità con *nodo* come unico elemento

esplorati = insieme vuoto

loop do

if *Vuota?*(*frontiera*) **then return fallimento**

nodo = POP(*frontiera*);

if problema.TestObiettivo(*nodo*.Stato) **then return** Soluzione(*nodo*)

aggiungi *nodo*.Stato a *esplorati*

for each azione **in** problema.Azioni(nodo.Stato) **do**

figlio = Nodo-Figlio(*problema*, *nodo, azione*)

if *figlio*.Stato non è in *esplorati* e non è in *frontiera* **then**

frontiera = Inserisci(*figlio*, *frontiera*) /* in coda con priorità

else if *figlio*.Stato è in *frontiera* con Costo-cammino più alto **then**

sostituisci quel nodo frontiera con figlio

Posticipato per vedere il costo minore

Warning: AIMA ed. IV ha usato uno schema di UC diverso e alcune proprietà cambiano

g(n)

In Python

```
def uniform_cost_search(problem):  """Ricerca-grafo UC"""
    explored = [] # insieme (implementato come una lista) degli stati gia' visitati
    node = Node(problem.initial_state) # il costo del cammino e' inizializzato nel costruttore del nodo
    frontier = PriorityQueue(f = lambda x:x.path_cost) # la frontiera e' una coda coda con priorita'
    #lambda serve a definire una funzione anonima a runtime
    frontier.insert(node)
    while not frontier.isempty():
        # seleziona il nodo node = frontier.pop() # estrae il nodo con costo minore, per l'espansione
        if problem.goal_test(node.state):
            return node.solution(explored_set = explored)
        else:  # se non lo e' inserisci lo stato nell'insieme degli esplorati
            explored.append(node.state)
        for action in problem.actions(node.state):
            child_node = node.child_node(problem, action)
            if (child_node.state not in explored) and (not frontier.contains_state(child_node.state)):
                frontier.insert(child_node)
            elif frontier.contains_state(child_node.state) and
(frontier.get_node(frontier.index_state(child_node.state)).path_cost >
                                                child_node.path_cost):
                frontier.remove(frontier.index_state(child_node.state))
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

se lo stato del nodo figlio è già nella frontiera, ma con un costo più alto allora sostituisci quel nodo nella frontiera con il nodo figlio

Costo uniforme: analisi

Ottimalità e *completezza* garantite purché il costo degli archi sia maggiore di $\varepsilon>0$. (vedi lez. prossima)

Assunto C* come il costo della soluzione ottima

$\lfloor C^*/\varepsilon \rfloor$ è il numero di mosse nel caso peggiore, arrotondato per difetto (e.g. attratto ad andare verso tante mosse di costo ε prima di una che parta più alta ma poi abbia un path a costo totale più basso).

Complessità: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

Nota: quando ogni azione ha lo stesso costo UC somiglia a BF ma complessità $O(b^{1+d})$

[causa esame e arresto posticipato, solo dopo aver espanso anche l'ultima frontiera, oltre la profondità del goal*]

Confronto delle strategie (albero)

Criterio	BF	UC	DF	DL	ID	Bidir
Completa?	si	si(^)	*no*	si (+)	*si*	si (£)
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Ottimale?	si(*)	si(^)	no	no	si(*)	si (£)

(*) se gli operatori/archi hanno tutti lo stesso costo

(^) per costi degli archi $\geq \varepsilon > 0$

(+) per problemi per cui si conosce un limite alla profondità della soluzione (se l >d)

(£) usando UC (o BF)

Suggerimento: riprovare a riempire la tabella come esercizio

Conclusioni

- Un agente per "problem solving" adotta un paradigma generale di risoluzione dei problemi:
  - Formula il problema   Nota: parte non-automatica
  - Ricerca la soluzione nello spazio degli stati (diventa automatico)
- Strategie "non informate" per la ricerca della soluzione
- Prossima volta: come si può ricercare "meglio"
- BIB (bibliografia): AIMA ed. III Cap 3 (fino a 3.4)

Per informazioni

Alessio Micheli

micheli@di.unipi.it

Dipartimento di Informatica
Università di Pisa - Italy

Computational Intelligence &
Machine Learning Group