Ricerca euristica

Alessio Micheli

a.a. 2021/2022

Credits: Maria Simi

Russell-Norvig

Ricerca euristica

- La ricerca esaustiva non è praticabile in problemi di complessità esponenziale (e.g. 10^{120} configurazioni in scacchi).
- Noi usiamo conoscenza del problema ed esperienza per riconoscere i cammini più promettenti.
  - Usiamo una stima del costo futuro
  - Evitando di generare gli altri (*pruning*)!
- La *conoscenza euristica* (dal greco "eureka") aiuta a fare scelte "oculate"
  - non evita la ricerca ma la riduce
  - consente in genere di trovare una buona soluzione in tempi accettabili.
  - sotto certe condizioni garantisce completezza e ottimalità

Funzioni di valutazione euristica

Conoscenza del problema data tramite una *funzione di valutazione f*, che include *h* detta funzione di valutazione *euristica*:

$h : n \rightarrow \mathrm{R}$

La funzione si applica al nodo ma dipende solo dallo stato (*n*.Stato)

(manteniamo la notazione in *n* per uniformità con *g*; *g* dipendeva anche dal cammino fino al *nodo*)

$f(n) = g(n) + h(n)$, *ove g(n) è il costo cammino visto con UC*

Esempi di euristica *h*

Per procedere preferibilmente verso il percorso migliore, seguendo «problem-specific information», di stima del costo futuro

- La città più vicina (o la città più vicina alla mèta in linea d'aria – tabella esterna) nel problema dell'itinerario
- Il numero delle caselle fuori posto nel gioco dell'otto
- Il vantaggio in pezzi nella dama o negli scacchi

- Esempio:

Mappa Romania

Dist. in linea d'aria

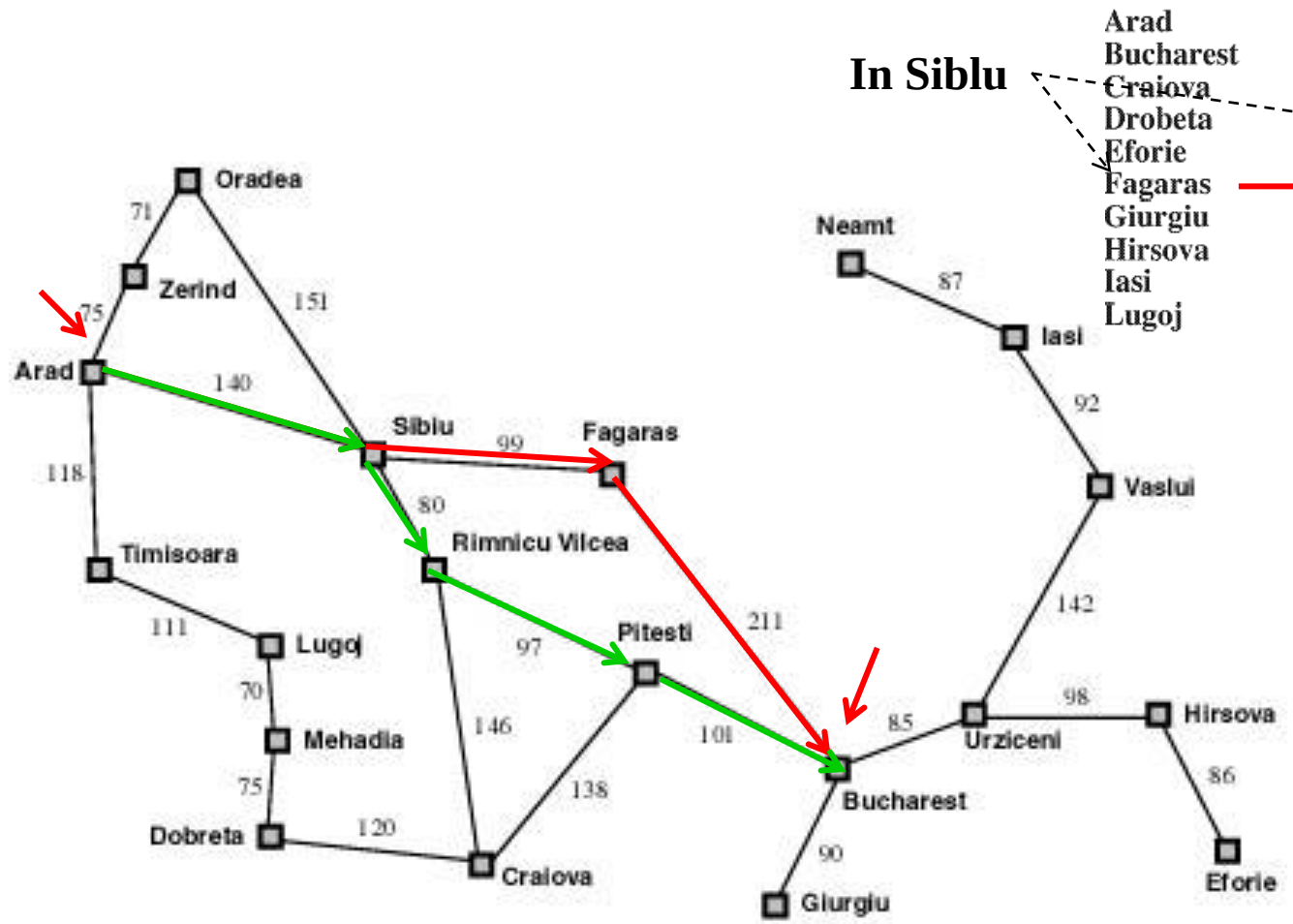
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Algoritmo di ricerca Best-First

- **Best first** - heuristic con stesso algoritmo di UC ma con uso di *f* (stima di costo) per la *coda con priorità*
- Scelta di *f* determina la strategia di ricerca
- A ogni passo si sceglie il nodo sulla frontiera per cui il valore della *f* è migliore (*il nodo più promettente*).
  - Nota: Migliore significa 'minore' in caso di un'euristica che stima la distanza della soluzione
  - Caso speciale: **greedy best-first** , si usa solo *h* (*f*=*h*)

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

Ricerca greedy best-first: esempio *f=h*



Da Arad a Bucarest …

Greedy best-first: Arad, Sibiu, Fagaras, Bucharest (450)

ma non è l 'Ottimo: Arad, Sibiu, Rimnicu, Pitesti, Bucarest (418)

Algoritmo A: definizione

- Si può dire qualcosa di *f* per avere garanzie di completezza e ottimalità?
- Def: Un algoritmo A è un algoritmo Best First con una funzione di valutazione dello stato del tipo:

$$f(n) = g(n) + h(n), \text{ con } h(n) \geq 0 \text{ e } h(goal)=0$$

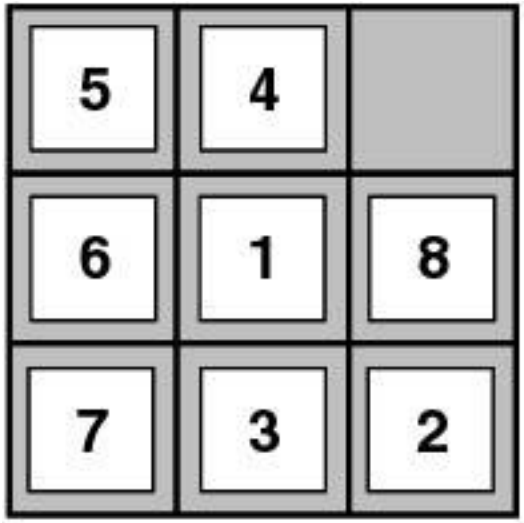
  - $g(n)$ è il costo del cammino percorso per raggiungere n
  - $h(n)$ una stima del costo per raggiungere da n un nodo goal

Vedremo come casi particolari dell'algoritmo A:

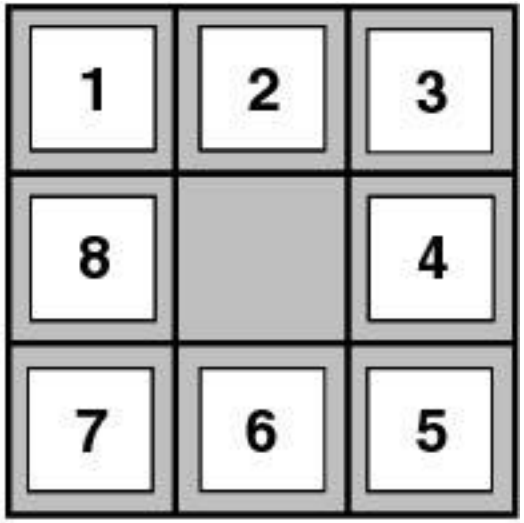
- Se $h(n) = 0$ [$f(n) = g(n)$] si ha Ricerca Uniforme (UC)
- Se $g(n) = 0$ [$f(n) = h(n)$] si ha Greedy Best First

Algoritmo A: esempio per $f = g + h$

Esempio nel gioco dell'otto



Start State



Goal State

f(n) = #mosse fatte + #caselle-fuori-posto

f(Start) = 0 + 7

f(goal state)=?+0

Dopo ←,↓, ↑,→ $f = 4 + 7$

stesso stato, g è cambiata

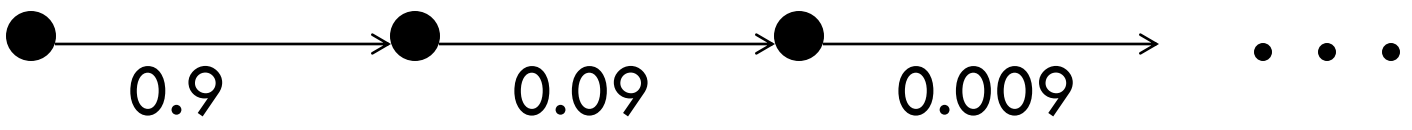
L' algoritmo A è completo

Teorema: L'algoritmo A con la condizione

$$g(n) \geq d(n) \cdot \varepsilon \quad (\varepsilon > 0 \text{ costo minimo arco})$$

è *completo*. profondità Dim: next slide

Nota: la condizione ci garantisce che non si verifichino situazioni strane del tipo



e che il costo lungo un cammino non cresca "abbastanza"

- se cresce abbastanza possiamo fermare quel path per costo alto di *g*.

Completezza di A: dimostrazione

- Sia [n_0 n_1 n_2 … n'… n_k=*goal*] un cammino soluzione.
- Sia n' un nodo della frontiera su un cammino soluzione: n' prima o poi sarà espanso.
  - Infatti esistono solo un numero finito di nodi x che possono essere aggiunti alla frontiera con $f(x) \leq f(n')$;
  - [è la condizione sulla crescita di g dello slide precedente, t.c. non esista una catena infinita di archi e nodi che possa aggiungere con costo sempre $\leq$ f(n')]
- Quindi, se non si trova una soluzione prima, n' verrà espanso e i suoi successori aggiunti alla frontiera. Tra questi anche il suo successore sul cammino soluzione.
- Il ragionamento si può ripetere fino a dimostrare che anche il nodo *goal* sarà selezionato per l'espansione

Algoritmo A*: la stima ideale

Funzione di valutazione ideale (*oracolo*):

$$f^*(n) = g^*(n) + h^*(n)$$

$g^*(n)$ costo del cammino minimo da radice a n

$h^*(n)$ costo del cammino minimo da n a goal

$f^*(n)$ costo del cammino minimo da radice a goal, attraverso n

Normalmente:

Costo cammino ≥ Cammino migliore

$g(n) \geq g^*(n)$ e $h(n)$ è una stima di $h^*(n)$

Si può andare in sottostima (e.g. linea d'aria) o sovrastima della distanza dalla soluzione

Algoritmo A*: definizione

Definizione: euristica ammissibile

Lower-bound condition

$\forall n \,.\, h(n) \leq h^*(n)$ *h è una sottostima*

Es. l'euristica della distanza in linea d'aria

Definizione: **Algoritmo A***

Un algoritmo A in cui *h* è una funzione euristica ammissibile.

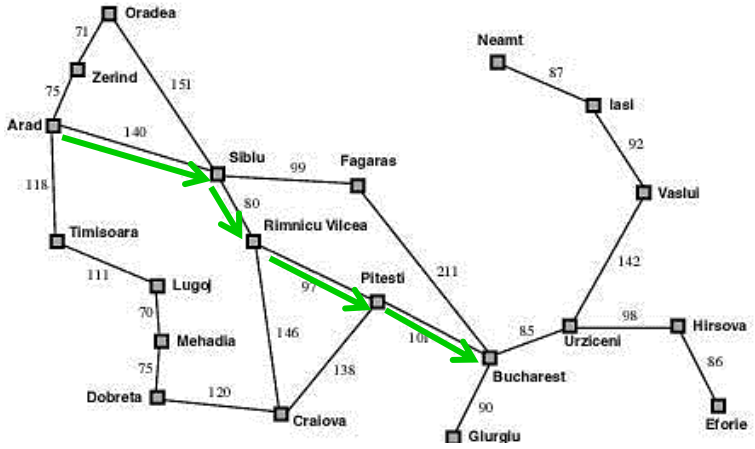
Teorema: gli algoritmi A* sono *ottimali*.

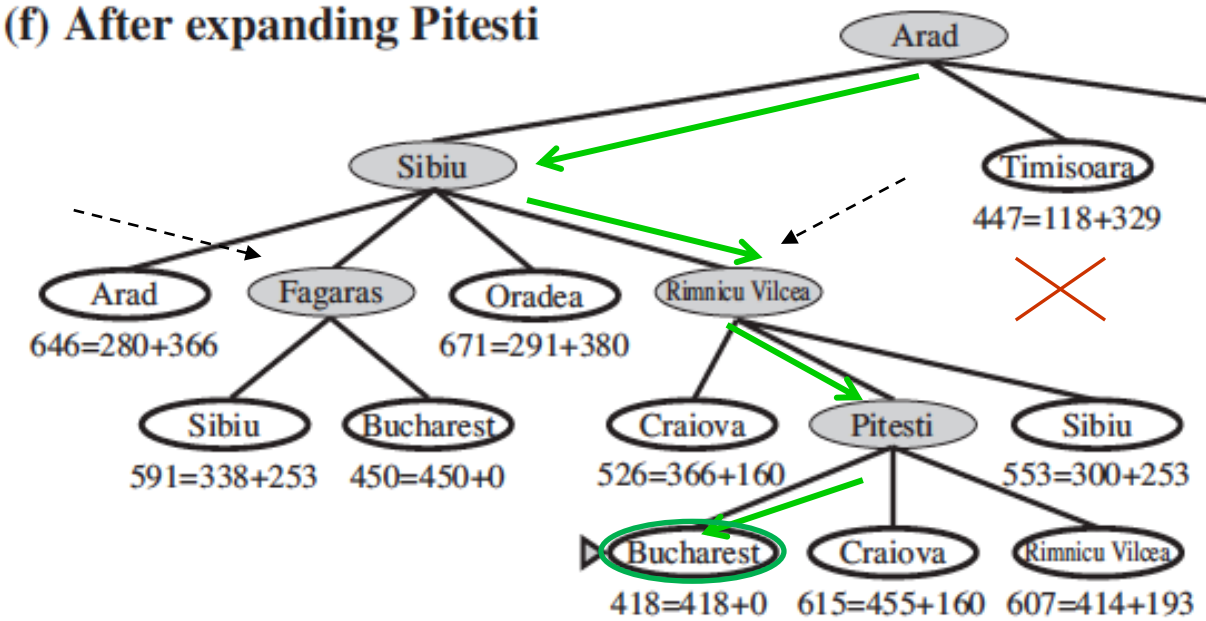
Corollario: BF(+) e UC sono ottimali ($h(n)=0$)

[(+) BF con passi a costo costante]

Itinerario con A*

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374





Ottimo

Nota: Qui rami non espansi: bel pruning !

Suggerimento: ripetere come esercizio autonomamente (vanno fatte le somme g+h)! [usa A* versione ad albero]

Osservazioni su A*

1. Rispetto a greedy best-first, la componente *g* fa sì che si abbandonino cammini che vanno troppo in profondità
2. H sotto- o sovra-stima?
   a) Una sottostima (*h*) può farci compiere del lavoro inutile (tenendo anche candidati non buoni), però non ci fa perdere il cammino migliore (quando prendo nodo goal è il cammino migliore)
   b) Una funzione che qualche volta sovrastima può farci perdere la soluzione ottimale

      (taglio per causa di sovrastima, invece era buona)

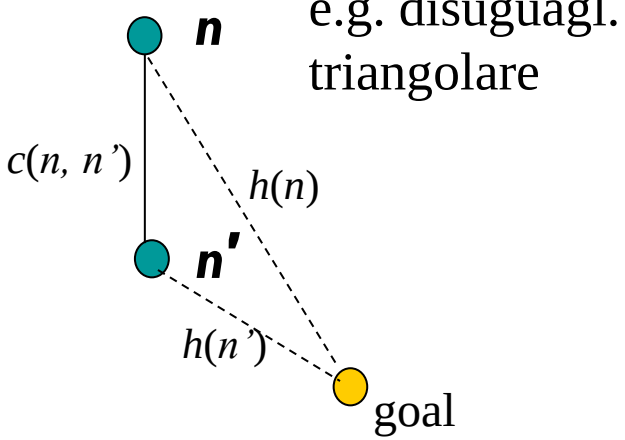
Ottimalità di A*

- Nel caso di ricerca a/su albero l'uso di un'euristica *ammissibile* è sufficiente a garantire l'ottimalità di A*
- Nel caso di ricerca su grafo serve una proprietà più forte: la consistenza (detta anche monotonicità)
  - Per evitare rischio di scartare candidati ottimi (stato già incontrato)
    (si vuol evitare, causa uso della lista esplorati di far «sparire», o meglio non considerare al momento dell'espansione, candidati ottimali)
  - Cerchiamo quindi condizioni per garantire che il primo *espanso* sia il migliore
  - Warning: AIMA ed. IV ha usato uno schema di UC diverso e alcune proprietà cambiano

Euristica consistente o monotòna

- *Definizione*: euristica consistente
  - [h(goal) = 0]
  - $\forall n.\ h(n) \leq c(n, a, n') + h(n')$ Consistenza locale

    dove *n'* è *un successore di n*
  - Ne segue che $f(n) \leq f(n')$ *[dim: dopo]*



- Nota: se *h* è consistente la *f* non decresce mai lungo i cammini, da cui il termine monotòna (v. dopo)

Euristiche monotòne: proprietà

- *Teorema*: Un'euristica monotona è ammissibile
- Esistono euristiche ammissibili che non sono monotone, ma sono rare*.
- Le euristiche monotone garantiscono che la soluzione meno costosa venga trovata per prima e quindi sono ottimali anche nel caso di *ricerca su grafo*.
  - *Non si devono recuperare tra gli antenati nodi con costo minore*
  - *Lista degli esplorati, stato già esplorato è sul cammino ottimo → posso evitare di inserire il corrente ripetuto senza perdere l'ottimalità*
    - if figlio.Stato non è in esplorati e non è in frontiera then
      - frontiera = Inserisci(figlio, frontiera)
  - *Per la frontiera, volendo evitare stati ripetuti, resta "if" finale di UC*
    - if *figlio*.Stato è in *frontiera* con Costo-cammino più alto **then**
      - sostituisci quel nodo frontiera con figlio

Ottimalità di A* (teorema, con h consistente)

1. Se $h(n)$ è consistente i valori di $f(n)$ lungo un cammino sono non decrescenti:

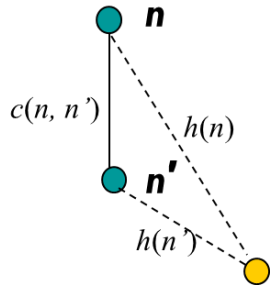
Se $h(n) \leq c(n, a, n') + h(n')$ *def. consistenza*

$g(n) + h(n) \leq g(n) + c(n, a, n') + h(n')$ *sommando g(n)*

ma siccome $g(n) + c(n, a, n') = g(n')$

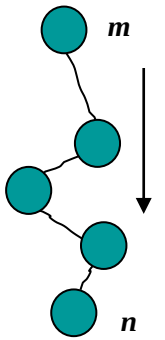
$g(n) + h(n) \leq g(n') + h(n')$

$f(n) \leq f(n')$ → f monotòna



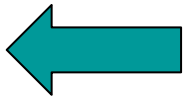
2. Ogni volta che A* seleziona un nodo (n) per l'*espansione*, il cammino ottimo a tale nodo è stato trovato:

se così non fosse, ci sarebbe un altro nodo m della frontiera sul cammino ottimo (a n, ancora da trovare con un cammino ottimo), con $f(m)$ minore (per la monot. e n successore di m); ma ciò non è possibile perché tale nodo sarebbe già stato espanso (** si espande prima un nodo con f minore)



3. Quando seleziona nodo goal è cammino ottimo [h=0, f=C*]

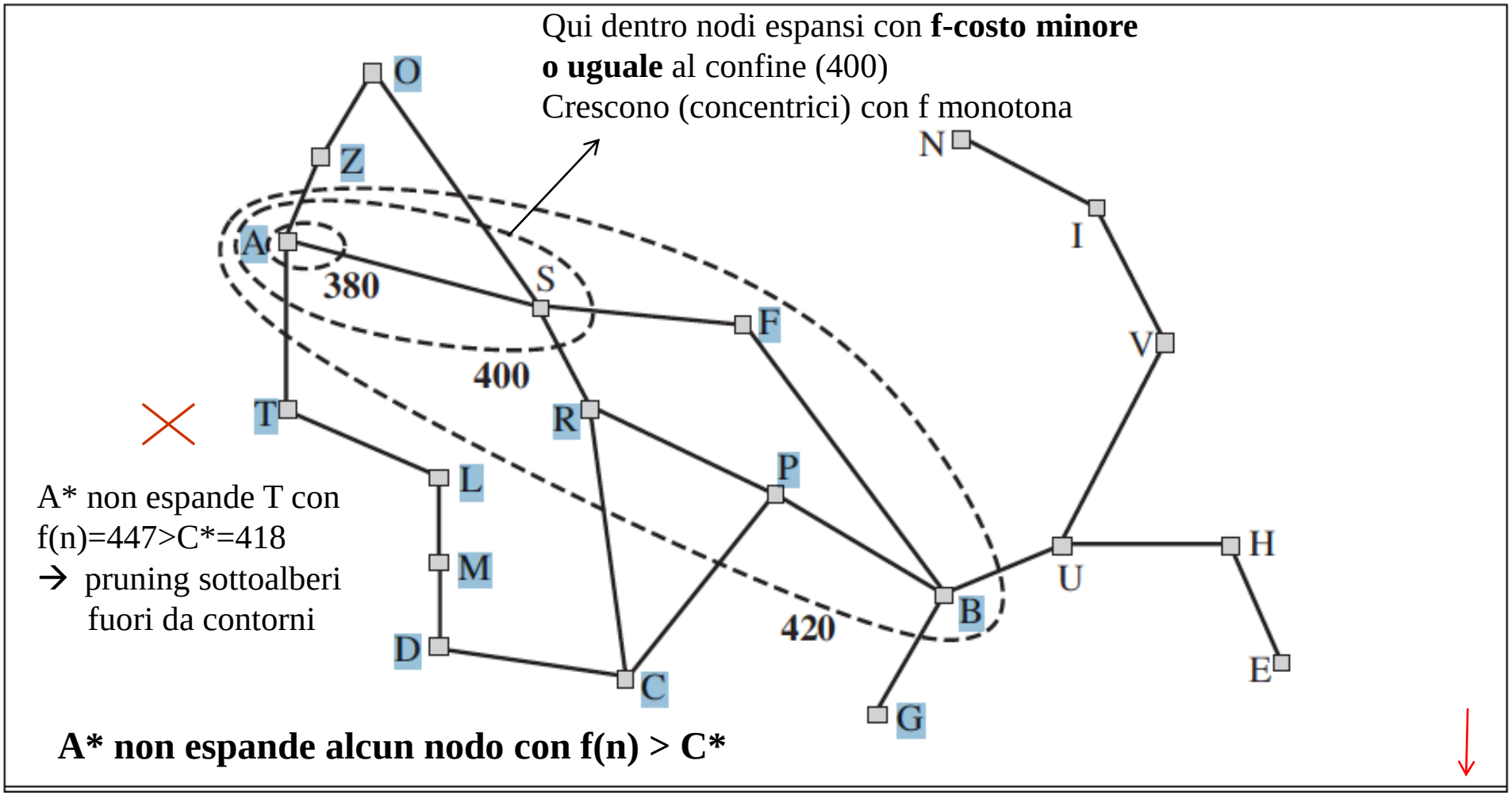
Perché A* è vantaggioso?



- A* espande tutti i nodi con f(n) <C* (C* = costo ottimo)
- A* espande alcuni nodi con f(n) =C*
- **A* non espande alcun nodo con f(n) > C***

- Quindi alcuni nodi (e suoi sottoalberi) non verranno considerati per l' espansione (ma restiamo ottimali): *pruning* (*h* opportuna, più alta possibile tra le ammissibili, fa tagliare molto)

I contorni nella ricerca A*



Perché A* è vantaggioso?

- A* espande tutti i nodi con f(n) <C*
- A* espande alcuni nodi con f(n) =C*
- **A* non espande alcun nodo con f(n) > C***

Commenti:

- Più f è aderente a stima ottimale, più taglio! Ovali più stretti
- Cercheremo quindi una *h* il più alta possibile tra le ammissibili
- Se molto bassa molti (sino a tutti i) nodi restano minore di C* → espando tutti (a cerchi) (esempio?)
- Il pruning sotto-alberi è il punto focale: non li abbiamo già in memoria e *evitiamo di generarli* (decisivo per i problemi di AI a spazio stati esponenziali)

Resume: A* algorithm

- L'algoritmo è quello degli schemi usati per UC (lezione 3 problem solving: schede 59-61)
- Usando $f = g+h$ per la *coda con priorità*
- Ove h e g soddisfano quanto allo slide 9 [A] e
- Ove h è una funzione euristica ammissibile [A*].
- E considerando le condizioni dette per ottenere l'ottimalità su grafi (next slide)

Bilancio su A*

- A* è completo: discende dalla completezza di A (A* è un algoritmo A particolare)
- A* con euristica monotona è ottimale
- A* è ottimamente efficiente: a parità di euristica nessun altro algoritmo espande meno nodi (senza rinunciare a ottimalità)
- Problemi?
  - Quale euristica?
  - e ancora l'occupazione di memoria che nel caso pessimo resta $O(b^{d+1})$, causa frontiera

Su f: Due sotto-casi speciali

UC e Greedy Best First

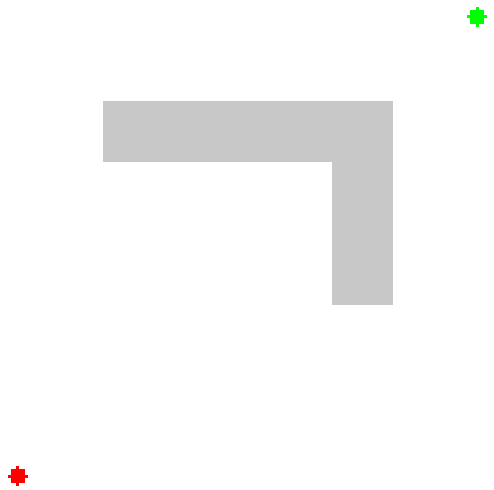
Casi particolari dell'algoritmo A:

- Se $h(n) = 0$ [$f(n) = g(n)$] si ha Uniform Cost
- Ossia g non basta (si può migliorare)
- Se $g(n) = 0$ [$f(n) = h(n)$] si ha Greedy Best First
- Ossia h non basta (già visto all'inizio)

- Esercizio: ricavare BF e DF con *f* apposita
- Esercizio: Greedy Best First ad albero è completo? [non ovvio]

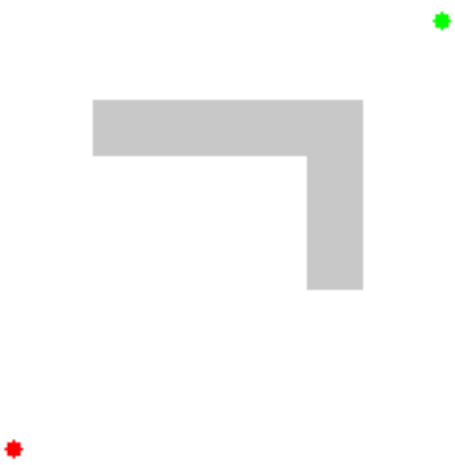
Dijkstra [1959], UC versus A*

- Illustration of Dijkstra's algorithm search for finding path from a start node (lower left, red) to a goal node (upper right, green) in a robot motion planning problem. Open nodes represent the "tentative" set. Filled nodes are visited ones, with color representing the distance: the greener, the farther. Nodes in all the different directions are explored uniformly, appearing as a more-or-less circular wavefront as Dijkstra's algorithm uses a **heuristic identically equal to 0.** → **UC !!!**
- https://en.wikipedia.org/wiki/Dijkstra's_algorithm#/media/File:Dijkstras_progress_animation.gif



Dijkstra [1959], UC versus A* [1968]

- Illustration of A* search for finding path from a start node to a goal node in a robot motion planning problem. The empty circles represent the nodes in the *open set*, i.e., those that remain to be explored, and the filled ones are in the closed set. Color on each closed node indicates the distance from the start: the greener, the farther. **One can first see the A* moving in a straight line in the direction of the goal**, then when hitting the obstacle, it explores alternative routes through the nodes from the open set → frontiera
- https://en.wikipedia.org/wiki/A*_search_algorithm#/media/File:Astar_progress_animation.gif



The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if a lower bound on the "distance" to the goal (*h*) is available

(Costruire) le euristiche di A*

Valutazione di funzioni euristiche

A parità di ammissibilità, una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore (visitare meno nodi)

Questo dipende da quanto *informata* è l'euristica (dal *grado di informazione posseduto*)

$h(n)=0$ minimo di informazione (BF o UC)

$h^*(n)$ massimo di informazione (oracolo)

In generale, per le euristiche ammissibili:

$$0 \leq h(n) \leq h^*(n)$$

Più informata, più efficiente

Teorema: Se $h_1 \leq h_2$, i nodi espansi da A* con h_2 sono un sottoinsieme di quelli espansi da A* con h_1.

[A* espande tutti i nodi con f(n)= g(n)+h(n) <C*, e sono meno per h maggiore (h maggiore fa andare più nodi oltre C*)]

Se $h_1 \leq h_2$, A* con h_2 è almeno efficiente quanto A* con h_1

- Un'euristica più informata (accurata) riduce lo spazio di ricerca (è più efficiente), ma è tipicamente più costosa da calcolare (e.g. un caso estremo ?)

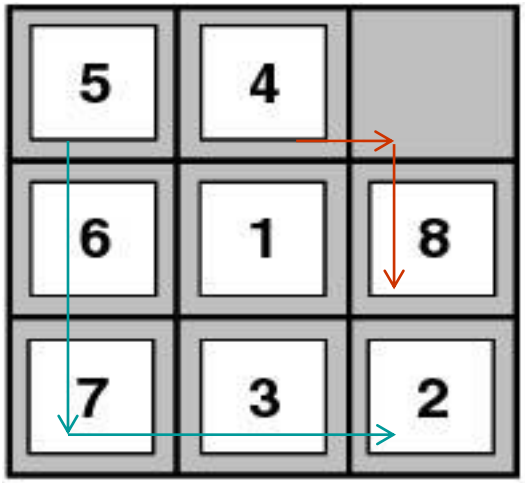
Confronto di euristiche ammissibili

- Due euristiche ammissibili per il gioco dell'8

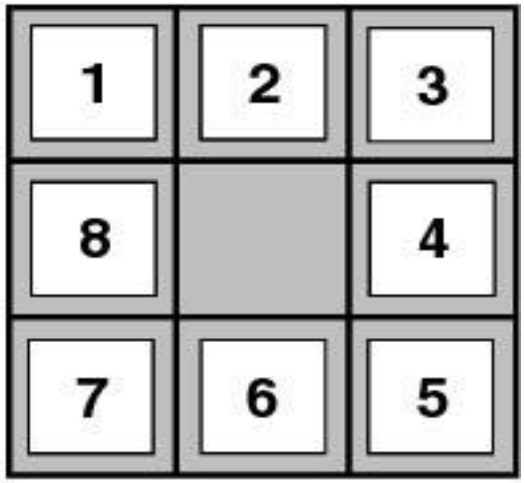
  h_1: conta il numero di caselle fuori posto

  h_2: somma delle distanze Manhattan(*) [orizz./vert.] delle caselle fuori posto dalla posizione finale

- h_2 *è più informata di* h_1 infatti $\forall n \,.\, h_1(n) \leq h_2(n)$
- ***Definizione***: h_2 *domina* h_1



Start State



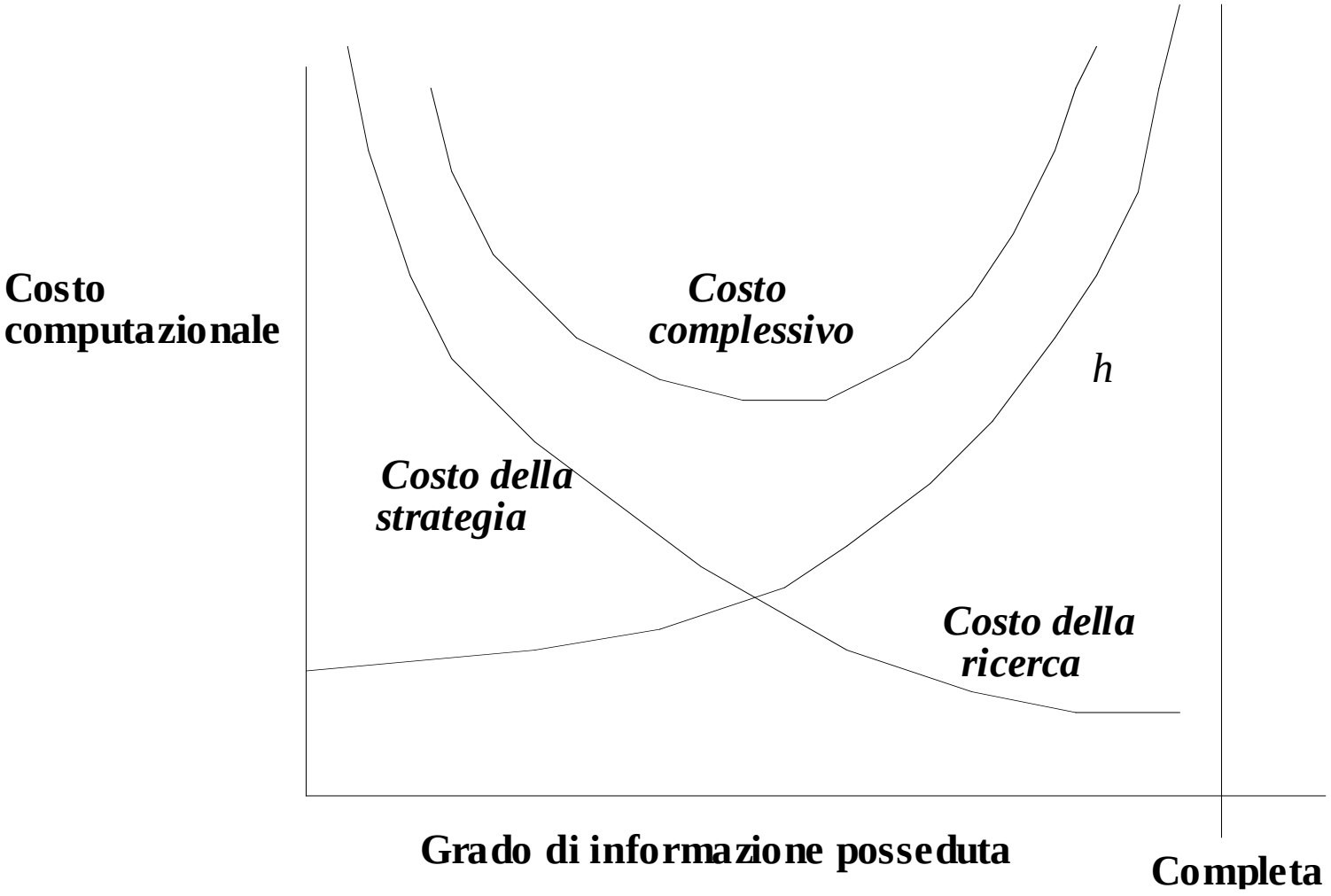
Goal State

h1 = 7

h2= 4+2+2+2+
+2+0+3+3 = 18

(*) $h((x, y)) = MD((x, y), (x_g, y_g)) = |x - x_g| + |y - y_g|$

Costo ricerca vs costo euristica



[figura da Nilsson 1980]

Misura del potere euristico

Come valutare gli algoritmi di ricerca euristica ...

Fattore di diramazione effettivo b^*

N: numero di nodi generati

d: profondità della soluzione

Esempio:

d=5; N= 52

b*= 1.92

b^* è il fattore di diramazione di un albero uniforme con N+1 nodi; soluzione dell'equazione

$$N + 1 = b^* + (b^*)^2 + \ldots + (b^*)^d$$

Sperimentalmente una buona euristica ha un b^* abbastanza vicino a 1 (< 1.5)

Ricordate la larghezza degli ellissodi?

Esempio: dal gioco dell'otto

d	ID (appr. it. non inf)	A*(h1)	A*(h2)
2	10 (2,43)	6 (1,79)	6 (1,79)
4	112 (2,87)	13 (1,48)	12 (1,45)
6	680 (2,73)	20 (1,34)	18 (1,30)
8	6384 (2,80)	39 (1,33)	25 (1,24)
10	47127 (2,79)	93 (1,38)	39 (1,22)
12	3644035 (2,78)	227 (1,42)	73 (1,24)
14	- Nodi generati b^*	539 (1,44)	113 (1,23)
…	-	…	…

Sono riportati: Nodi generati e fattore di diramazione effettivo (b^*, verde)
I dati sono mediati, per ogni d, su 100 istanze del problema [AIMA]

Capacità di esplorazione

Influenza di b^*:

Con $b=2$

$d=6$ N=100

$d=12$ ⟵ N=10.000

ma con $b=1.5$

$d=12$ N=100

$d=24$ ⟵ N=10.000

… migliorando di poco l'euristica si riesce, a parità di nodi espansi, a raggiungere una profondità doppia di esplorazione mosse!

Quindi . . .

1. Tutti i problemi dell'IA (o quasi) sono di complessità esponenziale … (nel generare nodi, i.e. configurazioni possibili) ma c'è esponenziale e esponenziale!
2. L'euristica può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca
3. Migliorando anche di poco l'euristica si riesce ad esplorare uno spazio molto più grande (più in profondità).

Come si inventa un'euristica?

- Alcune strategie per ottenere euristiche ammissibili:
  - Rilassamento del problema
  - Massimizzazione di euristiche
  - Database di pattern disgiunti
  - Combinazione lineare
  - Apprendere dall'esperienza

Rilassamento del problema

- Nel gioco dell'8 mossa da A a B possibile se ...
  1. B adiacente a A
  2. B libera

uno spazio degli stati con archi aggiunti

- h_1 e h_2 sono calcoli della *distanza esatta* della soluzione in versioni semplificate del puzzle:
  - h_1 (nessuna restrizione , ne 1 ne 2): sono sempre ammessi scambi a piacimento tra caselle (si muove ovunque) → # caselle fuori posto
  - h_2 (solo restrizione 1): sono ammessi spostamenti anche su caselle occupate, purché adiacenti → somma delle distanze Manhattan

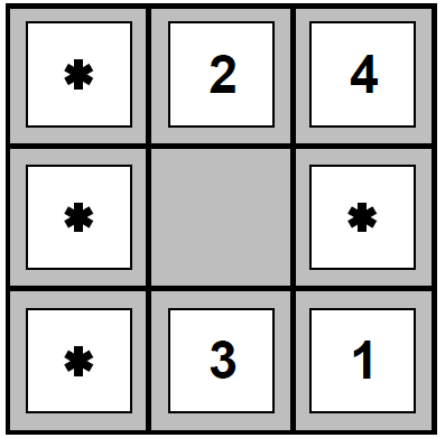
Massimizzazione di euristiche

- Se si hanno una serie di euristiche ammissibili $h_1, h_2, \ldots h_k$ senza che nessuna "domini" un' altra allora conviene prendere il massimo dei loro valori:

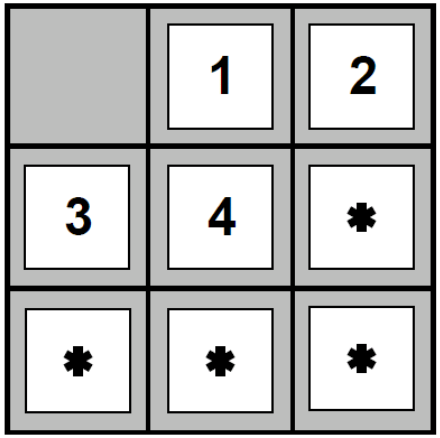
$$h(n)=\max(h_1(n), h_2(n), \ldots, h_k(n))$$

- Se le h_i sono ammissibili anche la h lo è
- La h domina tutte le altre.

Euristiche da sottoproblemi



Start State



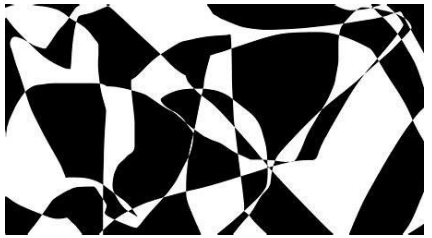
Goal State

- Costo della soluzione ottima al sottoproblema (di sistemare 1,2,3,4) è una sottostima del costo per il problema nel suo complesso
- (e.g. rilevatesi più accurata della Manhattan)
- *Database di pattern*: memorizzare ogni istanza del sottoproblema con relativo costo della soluzione
- Usare questo database per calcolare h_{DB} (estraendo dal DB la configurazione corrispondente allo stato completo corrente)

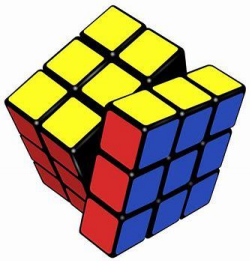
Sottoproblemi multipli

- Potremmo poi fare la stessa cosa per altri sottoproblemi: 5-6-7-8, 2-4-6-8 … ottenendo altre euristiche ammissibili
- Poi prendere il valore massimo: ancora una euristica ammissibile
- Ma potremmo sommarle e ottenere un'euristica ancora più accurata?

Pattern disgiunti



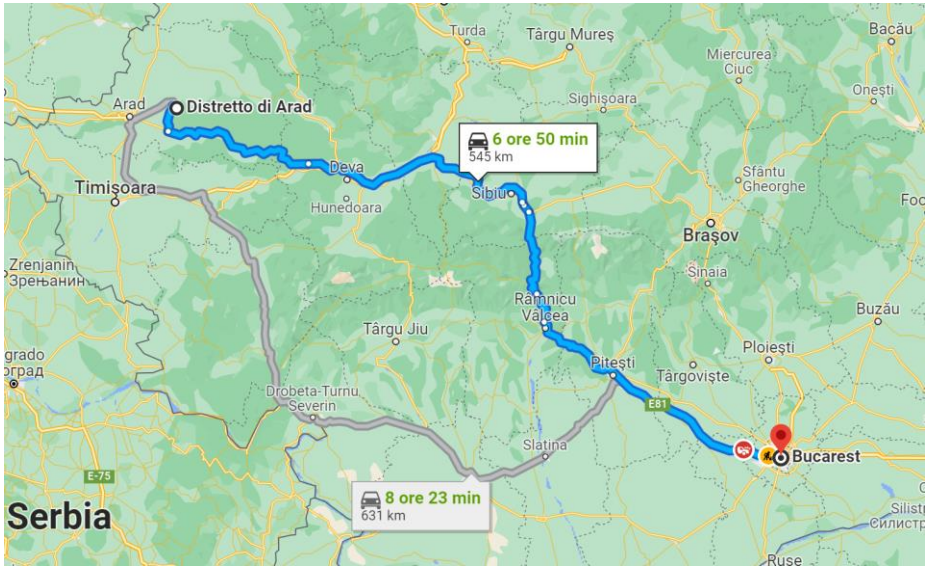
- In generale no perchè le soluzioni ai sottoproblemi interferiscono (condividono alcune mosse, se sposto 1-2-3-4, spostero anche 4-5-6-7) e la somma delle euristiche in generale non è ammissibile (potremmo sovrastimare avendo avuto aiuti mutui)
- Si deve eliminare il costo delle mosse che contribuiscono all'altro sottoproblema
- Database di pattern *disgiunti* consentono di sommare i costi (euristiche additive) [e.g. solo costo mosse su1-2-3-4]
- Sono molto efficaci: gioco del 15 in pochi ms
- Difficile scomporre per cubo Rubik



Aggiornamento: Mappe Reali

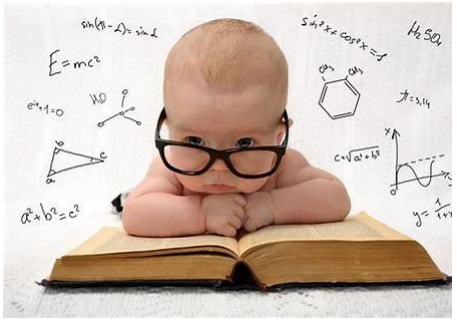
Per gli interessati a euristiche per problemi di mappe stradali on-line (servizi web) si rimanda alla sez. 3.6.4 AIMA ed. IV.

- Per-calcolo di costi di cammini ottimi tra coppie di vertici
- E per grafi grandi: con punti di riferimento (pivot) e scorciatoie (anche usando conoscenze specifiche come le strade più frequentate)



Best solution? ;-)

Apprendere dall'esperienza



- Far girare il programma, raccogliere dati: coppie <*stato*, *h**>
- Usare i dati per apprendere a predire la *h* con algoritmi di apprendimento induttivo (da istanze note stimiamo *h in generale*)
- Gli algoritmi di apprendimento si concentrano su caratteristiche salienti dello stato (*feature*, x_i) [e.g. apprendiamo che da numero tasselli fuori posto 5 → costo~14, etc]

Combinazione di euristiche

- Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare

$$h(n) = c_1\, x_1(n) + c_2\, x_2(n) + \ldots + c_k\, x_k(n)$$

Gioco dell'8:

$h(n) = c_1$ #fuori-posto $+ c_2$ #coppie-scambiate

Scacchi:

$h(n) = c_1$ vant-pezzi $+ c_2$ pezzi-attacc. + c3 regina + …

- Il peso dei coefficienti può essere aggiustato con l'esperienza, anche qui apprendendo automaticamente da esempi di gioco
- h(goal) =0 (e.g. gioco dell'8) ma ammissibilità e consistenza *non* automatiche

Algoritmi evoluti basati su A*

Migliorare l'occupazione di memoria

Indice:

- Beam search
- A* con approfondimento iterativo (IDA*)
- Ricerca best-first ricorsiva (RBFS)
- A* con memoria limitata (MA*) in versione semplice (SMA*)

Beam search

- Nel Best First viene tenuta tutta la frontiera; se l'occupazione di memoria è eccessiva si può ricorrere ad una variante: la *Beam search*.
- La *Beam Search* tiene ad ogni passo solo *i k nodi più promettenti*, dove *k* è detto *l'ampiezza del raggio (beam)*.
- La *Beam Search* non è completa.

IDA*

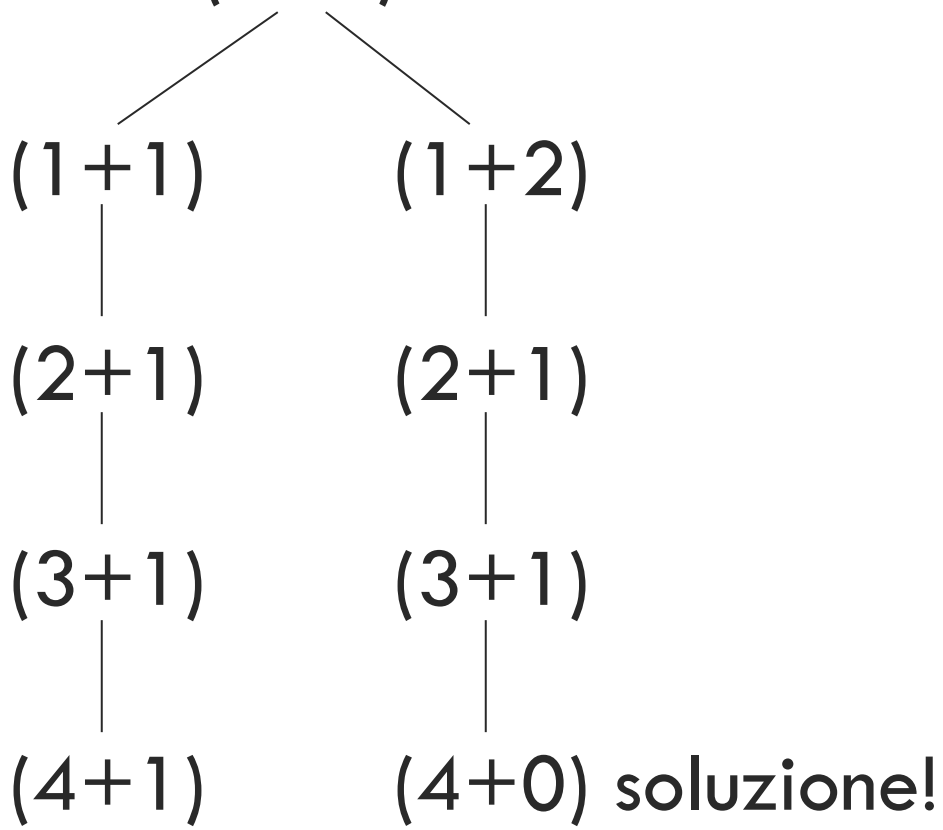
A con approfondimento iterativo*

- IDA* combina A* con ID: ad ogni iterazione si ricerca in profondità con un limite (cut-off) dato dal valore della funzione *f* (e non dalla profondità)
- il limite *f-limit* viene aumentato ad ogni iterazione, fino a trovare la soluzione.
- Punto critico: di quanto viene aumentato *f-limit*

Tipo: DF/ID con cut-off dato da *f*

Esempio

Iteraz. *4*



f-limit=4

Quale incremento?

- Cruciale la scelta dell'incremento per garantire l'ottimalità
  - Nel caso di costo delle azioni fisso è chiaro: il limite viene incrementato del costo delle azioni.
  - Nel caso che i costi delle azioni siano variabili?
    - costo minimo
    - si potrebbe ad ogni passo fissare il limite successivo al valore minimo delle *f* scartate (in quanto superavano il limite) all'iterazione precedente.

Analisi IDA*

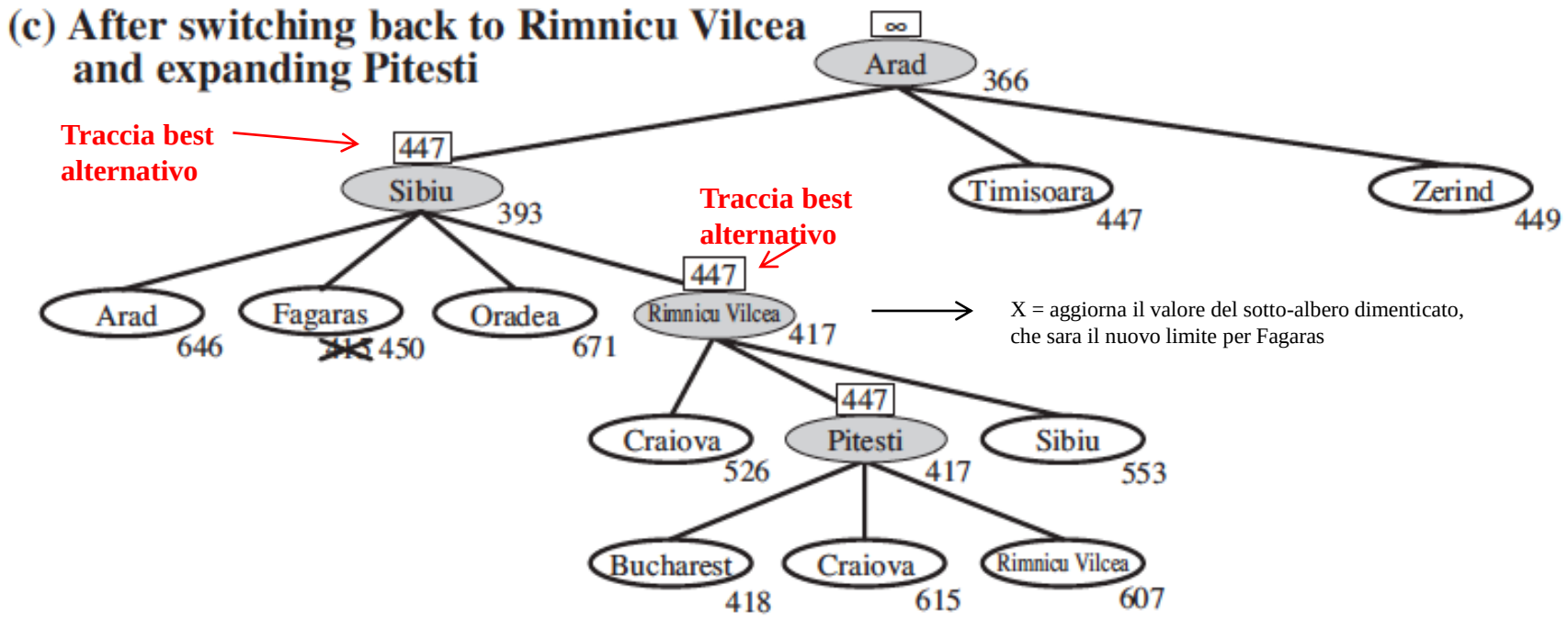
- IDA* completo e ottimale
  - Se le azioni hanno costo costante k (caso tipico 1) e *f-limit* viene incrementato di k
  - Se le azioni hanno costo variabile e l'incremento di *f-limit* è $\leq \varepsilon$ (minimo costo degli archi)
  - Se il nuovo *f-limit* = min. valore *f dei nodi generati ed* esclusi all'iterazione precedente
- Occupazione di memoria O(bd) [dall' alg. DF]

Fin qui 2022

Best-first ricorsivo (RBFS)

- Simile a DF ricorsivo: cerca di usare meno memoria, facendo del lavoro in più
- Tiene traccia ad ogni livello del migliore percorso alternativo
- Invece di fare backtracking in caso di fallimento (DF si ferma solo in fondo) interrompe l'esplorazione quando trova un nodo meno promettente (secondo *f*)
- Nel tornare indietro si ricorda il miglior nodo che ha trovato nel sottoalbero esplorato, per poterci eventualmente tornare
- Memoria: lineare nella profondita delle sol. ottima

Best first ricorsivo: esempio



Best First ricorsivo: algoritmo

```
function Ricerca-Best-First-Ricorsiva(problema)
    returns soluzione oppure fallimento
    return RBFS(problema, CreaNodo(problema.Stato-iniziale), ∞)      // all'inizio f-limite è un valore molto grande

function RBFS (problema, nodo, f-limite)
    returns soluzione oppure fallimento e un nuovo limite all' f-costo      // restituisce due valori
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    successori = [ ]
    for each azione in problema.Azioni(nodo.Stato) do
            aggiungi Nodo-Figlio(problema, nodo, azione) a successori          // genera i successori
    if successori è vuoto then return fallimento, ∞
    for each s in successori do                                                // valuta i successori
            s.f = max(s.g + s.h, nodo.f)              // un modo per rendere monotona f
    loop do
            migliore = il nodo con f minimo tra i successori
            if migliore.f > f_limite then return fallimento, migliore.f
            alternativa = il secondo nodo con f minimo tra i successori
            risultato, migliore.f = RBFS(problema, migliore, min(f_limite, alternativa))
            if risultato ≠ fallimento then return risultato
```

A* con memoria limitata

Versione semplice

- L'idea è quella di utilizzare al meglio la memoria disponibile
- SMA* procede come A* fino ad esaurimento della memoria disponibile
- A questo punto "dimentica" il nodo peggiore, dopo avere aggiornato il valore del padre.
- A parità di *f* si sceglie il nodo migliore più recente e si dimentica il nodo peggiore più vecchio.
- Ottimale se il cammino soluzione sta in memoria.

Considerazioni

- In algoritmi a memoria limitata (IDA* e SMA*) le limitazioni della memoria possono portare a compiere molto lavoro inutile [esp. ripetuta stessi nodi]
- Difficile stimare la complessità temporale effettiva
- *Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale*

Conclusione

- Agenti in ambienti determ., osservabili,statici, compl. noti.
- Ricerca come scelta della sequenza di azioni (cammino in spazio degli stati) che raggiunge l'obiettivo.

  Il cammino è la soluzione.
- Attività necessarie:
  - La formulazione del problema
  - Scelta dell'algoritmo di ricerca adeguato
  - Identificazione della funzione di valutazione euristica più efficace
- BIB: AIMA (3 ed.) Cap.3.5 a fine

Per informazioni

Alessio Micheli

micheli@di.unipi.it

Dipartimento di Informatica
Università di Pisa - Italy

Computational Intelligence &
Machine Learning Group