Inteligenta Artificiala

Universitatea Politehnica Bucuresti
Anul universitar 2016-2017

Adina Magda Florea

Curs 2

Strategii de cautare

- Reprezentarea solutiei problemei
- Strategii de cautare de baza
- Strategii de cautare informate
- Strategii de cautare informate cu memorie limitata
- Determinarea functiei euristice

1 Reprezentarea solutiei problemei

- Reprezentare prin spatiul starilor
- Reprezentare prin grafuri SI/SAU
- Echivalenta reprezentarilor
- Caracteristicile mediului de rezolvare

- Pentru a reprezenta si gasi o solutie:
  - Structura simbolica
  - Instrumente computationale
  - Metoda de planificare

Rezolvarea problemei reprezentata prin spatiul starilor

- Stare
- Spatiu de stari
- Stare initiala
- Stare/stari finala/finale
- (S_i, O, S_f)
- Solutia problemei
- Caracteristicile mediului

8-puzzle

S_i

2		3
1	8	4
7	6	5

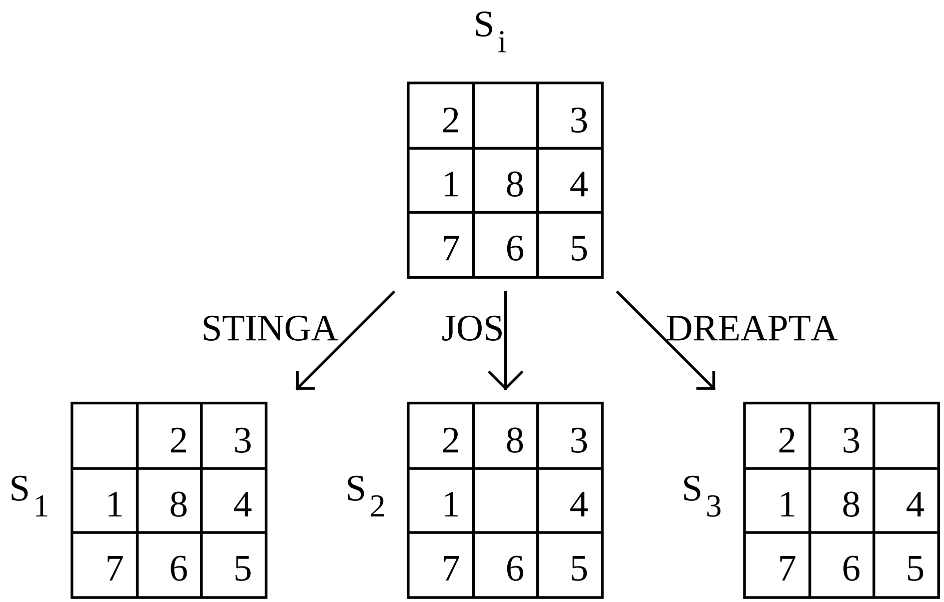
→

S_f

1	2	3
8		4
7	6	5

SUS - Mutare patrat liber in sus
STINGA - Mutare patrat liber la stinga
JOS - Mutare patrat liber in jos
DREAPTA - Mutare patrat liber la dreapta

(a) Stare initiala (b) Stare finala (c) Operatori

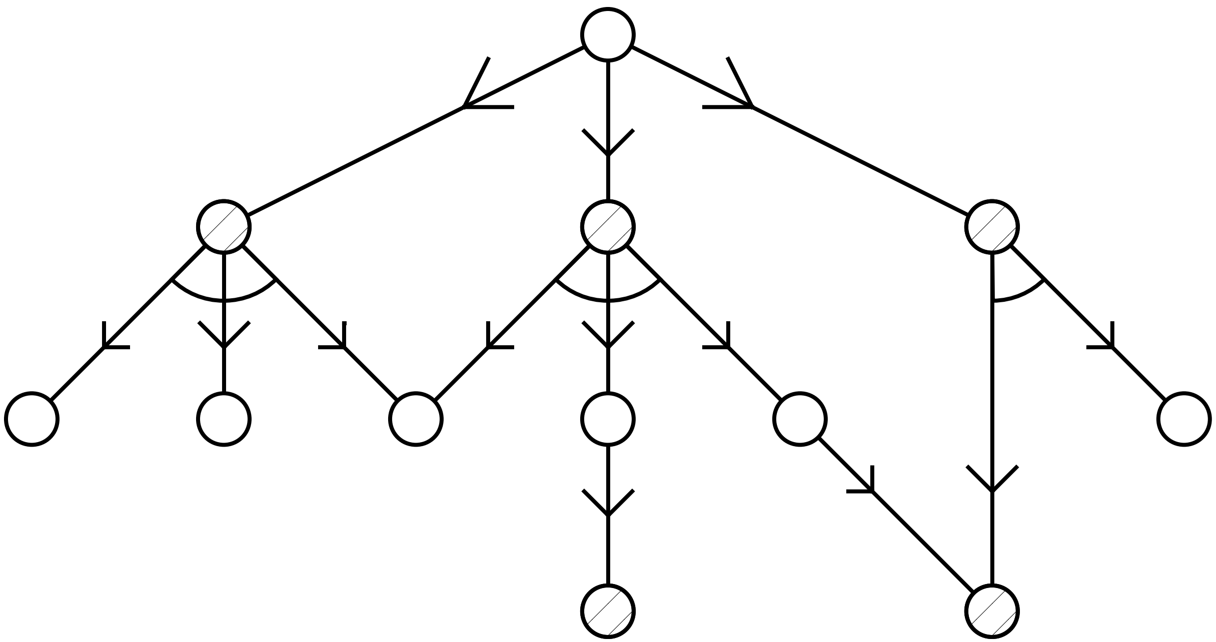


(d) Tranzitii posibile din starea S_i

Rezolvarea problemei reprezentata prin grafuri SI/SAU

- (P_i, O, P_e)
- Semnificatie graf SI/SAU
- Nod rezolvat
- Nod nerezolvabil
- Solutia problemei

Graf SI/SAU

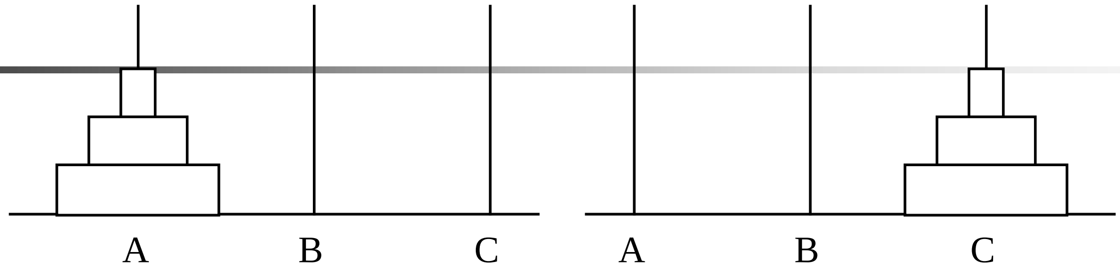


Nod SAU

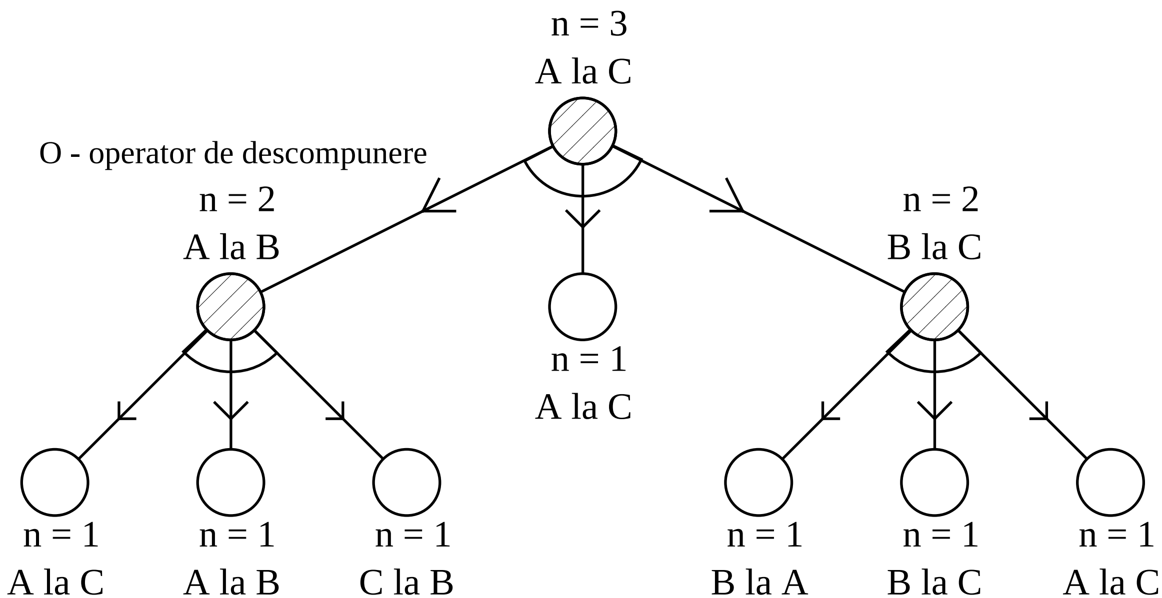
Noduri SI

Noduri SAU

Turnurile din Hanoi

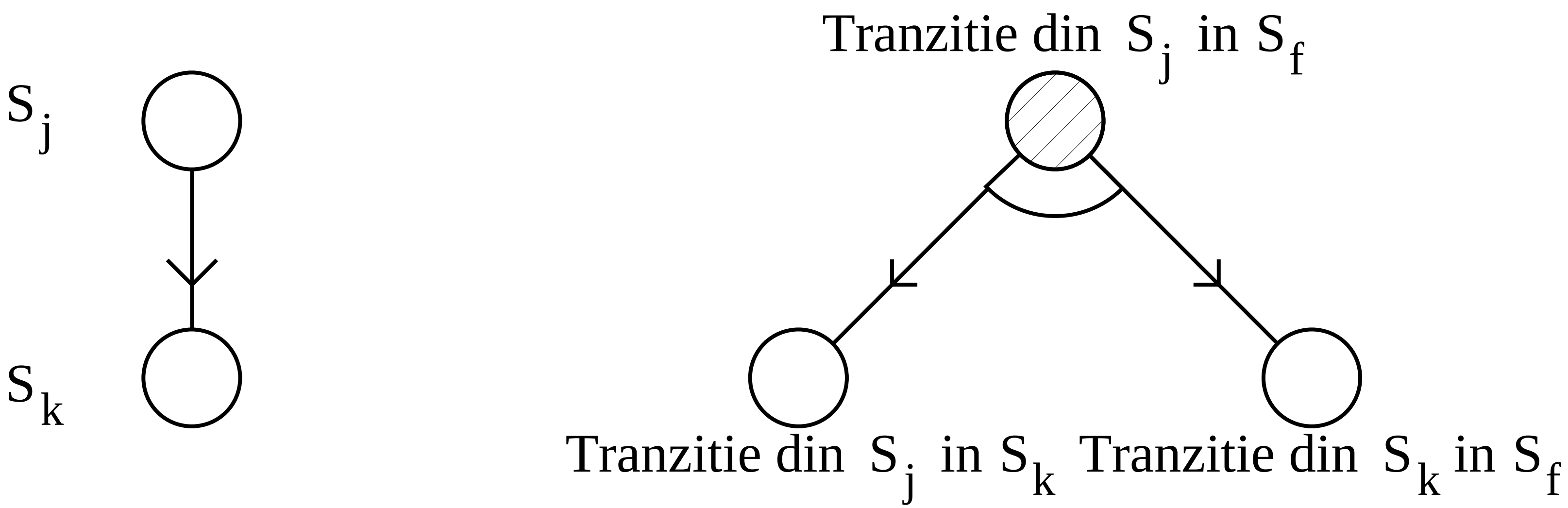


(a) Stare initiala (b) Stare finala



(c) Arborele SI/SAU de descompunere in subprobleme

Echivalenta reprezentarilor



S_j, S_k - stari intermediare S_f - stare finala

(a) Spatiul starilor (b) Descompunerea problemei in subprobleme

Caracteristicile mediului

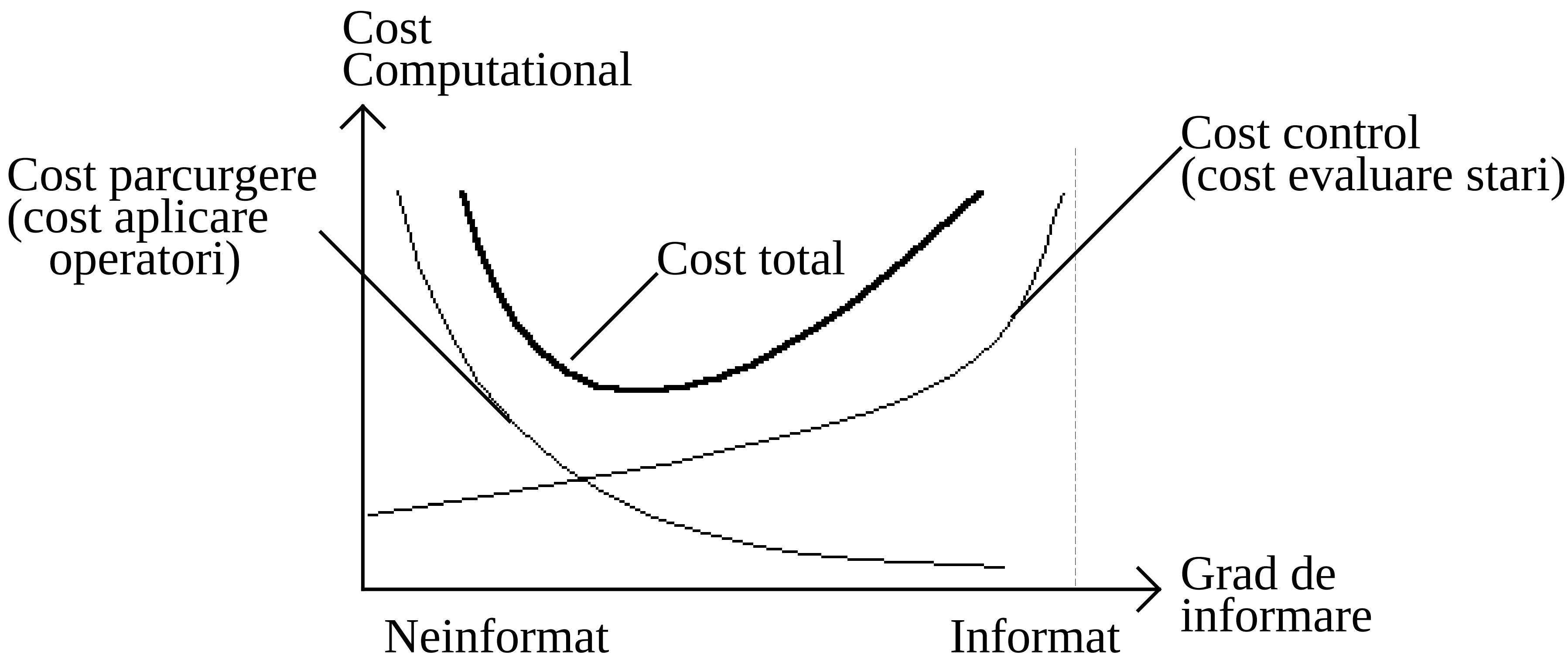
- Observabil / neobservabil
- Discret / continuu
- Finit / infinit
- Determinist / nedeterminist

2. Strategii de cautare de baza

Criterii de caracterizare

- Completitudine
- Optimalitate
- Complexitate
- Capacitatea de revenire
- Informare

Costuri ale cautarii



Cautari neinformate in spatiul starilor

- Gasirea unei cai sau a tuturor cailor, cu cost sau fara cost
- Cautarea pe nivel si cautarea in adancime se refera la doua metode de cautare neinformate in care parcurgerea se face in ordinea nodurilor succesoare starii curente in cautare:
  - Cautarea pe nivel – cele mai apropiate intai
  - Cautarea in adancime – cele mai departate intai
- Algoritmul lui Dijkstra rezolva problema celei mai scurte cai daca toate costurile arcelor sunt ≥ 0
- Algoritmul Bellman-Ford rezolva problema gasirii tuturor cailor de cost minim unde costurile arcelor pot fi si negative
- Algoritmul Floyd-Warshall rezolva problema gasirii tuturor cailor de cost minim

Caracteristici cautari pe grafuri nespecificate explicit

- Cautarea pe nivel (BFS)
- Cautare in adancime (DFS)
- Cautare in adancime cu nivel iterativ (iterative deepening)
- Cautare de tip backtracking
- Cautare bidirectionala

- **Care strategie este mai buna ?**

Caracteristici cautari pe grafuri nespecificate explicit

- Cele mai multe implementari bazate pe Open / FRONTIERA si Closed / TERITORIU
- DFS – Open – stiva (LIFO)
- BFS – Open – coada (FIFO)
- In ambele Closed este implementata ca o tabela de dispersie (Hash)
- Open este o coada de prioritati (prority queue)

Algoritm NIV: **Strategia cautarii <u>pe nivel</u> in spatiul starilor**

1. Initializeaza listele FRONTIERA $\leftarrow$ {S_i}, TERITORIU $\leftarrow$ {}
2. **daca** FRONTIERA = {}

   atunci intoarce INSUCCES
3. Elimina primul nod S din FRONTIERA si insereaza-l in TERITORIU
4. Expandeaza nodul S

   4.1. Genereaza toti succesorii directi S_j ai nodului S

   4.2. **pentru** fiecare succesor S_j al lui S **executa**

   4.2.1. Stabileste legatura $S_j \rightarrow S$

   4.2.2. **daca** S_j este stare finala

   atunci

   i. Solutia este (S_j, S, .., S_i)

   ii. **intoarce** SUCCES

   4.2.3. Insereaza S_j in FRONTIERA, ***la sfarsit***
5. **repeta de la** 2

sfarsit.

- Caracteristici cautare pe nivel
- Algoritmul presupune spatiul de cautare arbore si nu graf
- Pentru un spatiu de cautare graf se insereaza pasul 3’

3’. **daca** S $\in$FRONTIERA $\cup$ TERITORIU **atunci repeta de la** 2

Strategia cautarii in adancime in spatiul starilor

- Intr-o reprezentare a solutiei problemei prin spatiul starilor *adancimea unui nod* se defineste astfel:
- $Ad(S_i) = 0$, unde S_i este nodul stare initiala,
- $Ad(S) = Ad(S_p)+1$, unde S_p este nodul predecesor nodului S.

Algoritm ADANC(AdMax): Strategia cautarii <u>in adancime</u> in spatiul starilor

1. Initializeaza listele FRONTIERA $\leftarrow \{S_i\}$, TERITORIU $\leftarrow \{\}$
2. **daca** FRONTIERA = {}

   atunci intoarce INSUCCES
3. Elimina primul nod S din FRONTIERA si insereaza-l in TERITORIU

3'. **daca** Ad(S) = AdMax **atunci repeta de al 2**

4. Expandeaza nodul S
   - 4.1. Genereaza toti succesorii directi S_j ai nodului S
   - 4.2. **pentru** fiecare succesor S_j al lui S **executa**
     - 4.2.1. Stabileste legatura $S_j \rightarrow S$
     - 4.2.2. **daca** S_j este stare finala

       atunci
       - i. Solutia este $(S_j, .., S_i)$
       - ii. **intoarce** SUCCES
     - 4.2.3. Insereaza S_j in FRONTIERA, ***la inceput***
5. **repeta de la** 2

sfarsit.

- Caracteristici cautare in adancime

- **Cautare in adincime cu nivel iterativ (ID)**

  pentru AdMax=1, m **executa**

  ADANC(AdMax)

Caracteristici cautare in adancime **cu nivel iterativ**

- **Cautare de tip backtracking**
- **Cautare bidirectionala**

Algoritm: **Backtracking nerecursiv**

1. Initializeaza FRONTiERA cu $\{S_i\}$ /* S_i este starea initiala */
2. **daca** FRONTIERA = { }
   atunci intoarce INSUCCES /* nu exista solutie /*
3. Fie S prima stare din FRONTIERA
4. **daca** toate starile succesoare ale lui S au fost deja generate
   atunci
   4.1. Elimina S din FRONTIERA
   4.2. **repeta de la** 2
5. **altfel**
   5.1. Genereaza S', noua stare succesoare a lui S
   5.2. Introduce S' la începutul listei FRONTIERA
   5.3. Stabileste legatura S'$\rightarrow$ S
   5.4. Marcheaza în S faptul ca starea succesoare S' a fost generata
   5.5. **daca** S' este stare finala
   atunci
   5.5.1. Afiseaza calea spre solutie urmarind legaturile S'$\rightarrow$ S ..
   5.5.2. **întoarce** SUCCES /* s-a gasit o solutie */
   5.6. **repeta de la** 2

sfarsit.

2.2. Cautari neinformate in grafuri SI/SAU

Adancimea unui nod

- $Ad(S_i) = 0$, unde S_i este nodul problema initiala,
- $Ad(S) = Ad(S_p) + 1$ daca S_p este nod SAU predecesor al nodului S,
- $Ad(S) = Ad(S_p)$ daca S_p este nod SI predecesor al nodului S.

Algoritm NIV-SI-SAU: **Strategia cautarii pe nivel in arbori SI/SAU.**

1. Initializeaza listele FRONTIERA $\leftarrow$ $\{S_i\}$, TERITORIU $\leftarrow$ $\{\}$
2. Elimina primul nod S din FRONTIERA si insereaza-l in TERITORIU
3. Expandeaza nodul S
   - 3.1. Genereaza toti succesorii directi S_j ai nodului S
   - 3.2. **pentru** fiecare succesor S_j al lui S **executa**
     - 3.2.1. Stabileste legatura $S_j \rightarrow S$
     - 3.2.2. **daca** S_j reprezinta o multime de cel putin 2 subprobleme **atunci** /* este nod SI */
       - i. Genereaza toti succesorii subprobleme S^k_j ai lui S_j
       - ii. Stabileste legaturile intre nodurile $S^k_j \rightarrow S_j$
       - iii. Insereaza nodurile S^k_j in FRONTIERA, ***la sfirsit***
     - 3.2.3. **altfel** insereaza S_j in FRONTIERA, ***la sfirsit***

4. **daca** nu s-a generat nici un succesor al lui S in pasul precedent (3)

   atunci

   4.1. **daca** S este nod terminal etichetat cu o problema neelementara

   atunci

   4.1.1. Eticheteaza S nerezolvabil

   4.1.2. Eticheteaza cu nerezolvabil toate nodurile predecesoare lui S care devin nerezolvabile

   datorita lui S

   4.1.3. **daca** nodul S_i este nerezolvabil

   atunci intoarce INSUCCES /* problema nu are solutie */

   4.1.4. Elimina din FRONTIERA toate nodurile care au predecesori nerezolvabili

4.2. **altfel** /* S este nod terminal etichetat cu o problema elementara */

4.2.1. Eticheteaza S rezolvat

4.2.2. Eticheteaza cu rezolvat toate nodurile predecesoare lui S care devin rezolvate datorita lui S

4.2.3. **daca** nodul S_i este rezolvat

atunci

i. Construieste arborele solutie urmarind legaturile

ii. **intoarce** SUCCES /* s-a gasit solutia */

4.2.4. Elimina din FRONTIERA toate nodurile rezolvate si toate nodurile care au predecesori rezolvati

5. **repeta de la** 2

sfarsit.

Complexitatea strategiilor de cautare

- B - *factorul de ramificare* al unui spatiu de cautare

8-puzzle

- Numar de miscari:
- 2 m pt colt = 8
- 3 m centru lat = 12
- 4m centru $\Rightarrow$ **24 miscari**
- **B = nr. misc. / nr. poz. p. liber = 2.67**
- Numar de miscari:
- 1 m pt colt = 4
- 2 m centru lat = 8
- 3m centru $\Rightarrow$ **15 miscari** $\Rightarrow$ **B = 1.67**

Complexitatea strategiilor de cautare

- **B** - *factorul de ramificare*
- **d-** adancimea celui mai apropiat nod solutie
- **m** – lungimea maxima a oricarei cai din spatiul de cautare

Rad – B noduri, B^2 pe niv 2, etc.

- Numarul de stari posibil de generat pe un nivel de cautare *d* este B^d
- T - numarul total de stari generate intr-un proces de cautare, *d* – adancime nod solutie

$$\mathbf{T = B + B^2 + \ldots + B^d = O(B^d)}$$

Complexitatea strategiilor de cautare

- **Cautare pe nivel**

  Numar de noduri generate

  $\mathbf{B + B^2 + \ldots + B^d = O(B^d)}$

  Complexitate timp, spatiu

- **Cautare in adancime**

  Numar de noduri generate

  B*m – daca nodurile expandate se sterg din TERITORIU

  Complexitate timp, spatiu

Complexitatea strategiilor de cautare

- **Cautare backtracking**

  Numar de noduri generate **m** – daca se elimina TERITORIU

  Complexitate timp, spatiu

- **Cautare cu nivel iterativ**

  Numar de noduri generate

  $\mathbf{d*B+(d-1)*B^2+ \ldots + (1)*B^d = O(B^d)}$

  Complexitate timp, spatiu

  B=10, d=5

  N(BFS)=111110, N(IDS) = 123 450

b = 10
1 mil. noduri/sec
1000 bytes/nod

Adancime	Nr noduri	Timp	Memorie
2	110	.11 milisec	107 KB
4	11 100	11 milisec	10.6 MB
6	10^6	1.1 sec	1 GB
8	10^8	2 min	103 GB
10	10^{10}	3 h	10 TB
12	10^{12}	13 zile	1 petabytes
14	10^{14}	3.5 ani	99 petabytes

Complexitatea strategiilor de cautare

Criteriu	Nivel	Adancime	Adanc. limita	Nivel iterativ	Bidirectionala
Timp	B^d	B^d	B^m	B^d	$B^{d/2}$
Spatiu	B^d	$B*d$	$B*m$	B^d	$B^{d/2}$
Optimalitate?	Da	Nu	Nu	Da	Da
Completa?	Da	Nu	Da daca $m \geq d$	Da	Da

B – factor de ramificare, **d** – adancimea solutiei,
m – adancimea maxima de cautare (AdMax)

3. Strategii de cautare informate

Cunostintele euristice pot fi folosite pentru a creste eficienta cautarii in trei moduri:

- **Selectarea nodului urmator de expandat in cursul cautarii.**
- In cursul expandarii unui nod al spatiului de cautare se poate decide pe baza informatiilor euristice care dintre succesorii lui vor fi generati si care nu
- Eliminarea din spatiul de cautare a anumitor noduri generate

Cautare informata de tip "best-first"

- Evaluarea cantitatii de informatie
- Calitatea unui nod este estimata de *functia de evaluare euristica*, notata **w(n)** pentru nodul n
- Presupuneri pentru functia w(n)
- Strategia de cautare a alpinistului
- Strategia de cautare “best-first”

Algoritm BFS: **Strategia de cautare "best-first" in spatiul starilor**

Intrari: Starea initiala S_i *si functia* w(S) *asociata starilor*

Iesiri: SUCCES *si solutia sau* INSUCCES

1. Initializeaza listele OPEN $\leftarrow \{S_i\}$, CLOSED $\leftarrow \{\}$
2. Calculeaza w(S_i) si asociaza aceasta valoare nodului S_i
3. **daca** OPEN = {}

   atunci intoarce INSUCCES
4. Elimina nodul S cu w(S) minim din OPEN si insereaza-l in CLOSED
5. **daca** S este stare finala

   atunci

   i. Solutia este (S,.., S_i)

   ii. **intoarce** SUCCES
6. Expandeaza nodul S

   6.1. Genereaza toti succesorii directi S_j ai nodului S

6.2. **pentru** fiecare succesor S_j al lui S **executa**

6.2.1 Calculeaza $w(S_j)$ si asociaza-l lui S_j

6.2.2. Stabileste legatura $S_j \rightarrow S$

6.2.3. **daca** $S_j \notin OPEN \cup CLOSED$

atunci introduce S_j in OPEN cu $w(S_j)$ asociat

6.2.4. **altfel**

i. Fie S'_j copia lui S_j din OPEN sau CLOSED

ii. **daca** $w(S_j) < w(S'_j)$

atunci

- Elimina S'_j din OPEN sau CLOSED (de unde apare copia)

- Insereaza Sj cu w(Sj) asociat in OPEN

iii. **altfel** ignora nodul S_j

7. **repeta de la 3**

sfarsit.

Cazuri particulare

- Strategia de cautare "best-first" este o generalizare a strategiilor de cautare neinformate

  - strategia de cautare pe nivel w(S) = Ad(S)

  - strategia de cautare in adincime w(S) = -Ad(S)

- **Strategia de cautare de cost uniform / Dijkstra**

$$w(S_j) = \sum_{k=i}^{j-1} cost\_arc(S_k, S_{k+1})$$

- **Minimizarea efortului de cautare – cautare euristica**

  w(S) = functie euristica

Cautarea solutiei optime in spatiul starilor.

Algoritmul A*

w(S) devine **f(S)** cu 2 comp:

- g(S), o functie care estimeaza costul real g*(S) al caii de cautare intre starea initiala S_i si starea S,
- h(S), o functie care estimeaza costul real h*(S) al caii de cautare intre starea curenta S si starea finala S_f.
- $f(S) = g(S) + h(S)$
- $f^*(S) = g^*(S) + h^*(S)$

Calculul lui f(S)

- Calculul lui g(S)

$$g(S) = \sum_{k=i}^{n} cost\_arc(S_k, S_{k+1})$$

- Calculul lui h(S)
- Trebuie sa fie admisibila
- O functie euristica h se numeste *admisibila* daca pentru orice stare S, **h(S) ≤ h*(S).**
- Definitia stabileste ***conditia de admisibilitate*** a functiei h si este folosita pentru a defini ***proprietatea de admisibilitate*** a unui algoritm A*.

A* admisibil

Fie un algoritm A* care utilizeaza cele doua componente **g** si **h** ale functiei de evaluare **f**. Daca

- (1) functia **h** satisface conditia de admisibilitate
- (2) $cost\_arc(S,S') \geq c$

  pentru orice doua stari S, S', unde **c** > 0 este o constanta si costul **c** este finit
- atunci ***algoritmul A* este*** ***admisibil***, adica este garantat sa gaseasca calea de cost minim spre solutie.
- Completitudine – garantat sa gaseasca solutie daca solutia exista si costurile sunt pozitive

Implementare A*

Strategia de cautare "best-first" se modifica:

…

2. Calculeaza **$w(S_i)=g(S_i) + h(S_i)$** si asociaza aceasta valoare nodului S_i
3. **daca** OPEN = {}

   atunci intoarce INSUCCES - *nemodificat*
4. Elimina nodul S cu **w(S)** minim din OPEN si insereaza-l in CLOSED - *nemodificat*

…..

6.2.4. **altfel**

i. Fie S'_j copia lui S_j din OPEN sau CLOSED

ii. **daca** **$g(S_j) < g(S'_j)$**

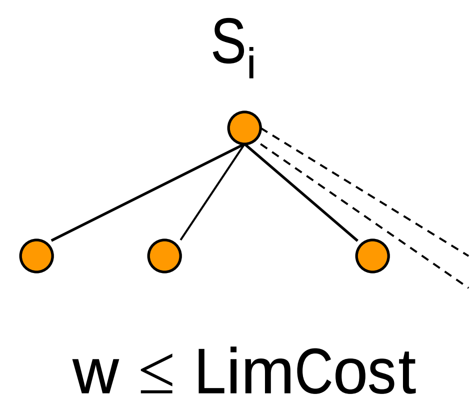
atunci …

4. Cautari informate cu memorie limitata

- A* se termina intotdeauna gasind o solutie optima si poate fi aplicat pe probleme generale
- Cu toate acestea, cantitatea de memorie necesara creste repede pe masura avansului algoritmului.
- Presupunem 100 bytes pt a memora o stare si atributele ei; rezulta aproximativ 10 mbytes/sec => o memorie de 1G se utilizeaza in mai putin de 2 minute
- Algoritmii clasici pentru cautare cu memorie limitata sunt:
  - Depth first iterative deepening (DFID)
  - Iterative deepening A* (IDA*)

DFID

- Cautarea realizeaza BFS cu o serie de DFS care opereaza pe o frontiera de cautare care creste succesiv
- Cautarea in adancime este modificata a.i. sa utilizeze o **limita de cost** in loc de o limita a adancimii
- Fiecare iteratie expandeaza nodurile din interiorul unui **contur de cost** pentru a vedea care sunt nodurile de pe urmatorul contur
- Daca cost arce 1 – DFID fara cost



DFID

- Algoritmul utilizeaza **doua limite U si U'** pentru urmatoarea iteratie. Apeleaza repetitiv functia DFID care cauta o cale optima (secventa de stari) **p**.
- DFID actualizeaza **variabila globala U' la valoarea minima a costului cailor** generate pana intr-un moment al cautarii
- Daca spatiul de cautare nu contine solutia si este infinit, algoritmul nu se termina

Algoritm DFID: Strategia de cautare depth first iterative deepening

Foloseste

- functia iterativa **BuclaDFID**
- functia recursiva **DFID**
- Functia **Expand** pentru generarea succesorilor unui nod
- Functia **Goal** care testeaza daca stare finala

BuclaDFID

Intrari: Starea initiala s si functia de cost w(s) asociata starilor

Iesiri: Calea de la s la starea finala sau {}

U' ← 0,   bestPath ← {}

cat timp (bestPath = {} si U'≠ inf) **repeta**

        U ← U',   U' ← inf

        bestPath ←DFID(s, 0, U)

intoarce bestPath

sfarsit

DFID(s, g, U)

Intrari: starea s, costul caii g, limita superioara U

Iesiri: calea de la s la starea finala sau {}

Efect lateral: Actualizarea lui U’

daca (Goal(s)) **atunci intoarce** s

Suc(s) $\leftarrow$ Expand(s)

pentru fiecare v in Suc(s) **repeta**

    daca g + w(s,v) $\leq$ U /* starea este in limita U */

    atunci p $\leftarrow$ DFID(v, g+w(s,v), U)

        daca p $\neq$ {} **atunci intoarce** (s,p) /* s-a gasit solutie */

    altfel

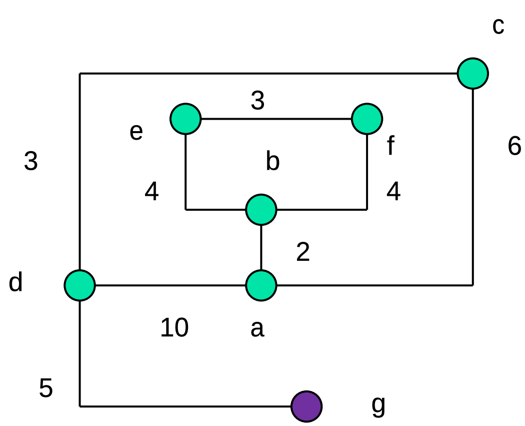
        daca g + w(s,v) < U’ **atunci** U’ $\leftarrow$ g + w(s,v) /* noua limita */

intoarce {}

sfarsit

Cautare DFID

Pas	Iteratie	Selectie	Apel	U	U’	Obs
1	1	{}	{(a,0)}	0	inf	
2	1	a	{}	0	2	g(b), g(c) si g(d) > U
3	2	{}	{(a,0)}	2	inf	Incepe o noua iteratie
4	2	a	{(b,2)}	2	6	g(c) si g(d) > U
5	2	b	{}	2	6	g(e) si g(f) > U
6	3	{}	{(a,0)}	6	inf	Incepe o noua iteratie
7	3	a	{(b,2),(c,6)}	6	10	g(d) > U
…………..						…………..
55	7	g	{(b,13)}	14	15	Scop atins

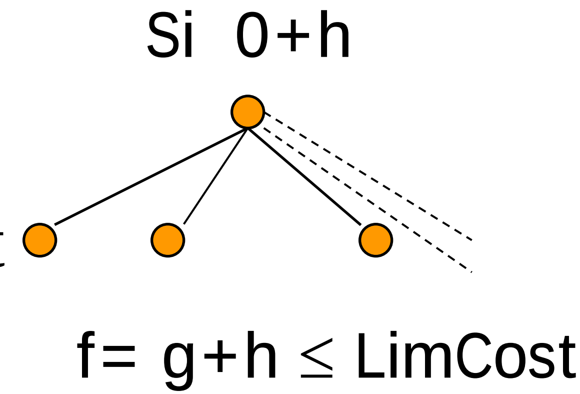


cat timp (bestPath = {} si U’≠ inf) **repeta**
U ← U’, U’ ← inf
bestPath ←DFID(s, 0, U)

pentru fiecare v in Suc(s) **repeta**
daca g + w(s,v) ≤ U
atunci p ← DFID(v, g+w(s,v), U)
daca p ≠ {} **atunci intoarce** (s,p)
altfel
daca g + w(s,v) < U’ **atunci** U’ ← g + w(s,v)

IDA*

- Iterative deepening A*
- Bazat pe DFID
- Garantat sa gaseasca solutia de cost minim
- f(s) = g(s) + h(s)



Algoritm IDA*: Strategia de cautare iterative deepening A*

Foloseste

- functia iterativa **BuclaIDA***
- functia recursiva **IDA***
- Functia **Expand** pentru generarea succesorilor unui nod
- Functia **Goal** care testeaza daca stare finala

BuclaIDA*

Intrari: Starea initiala s, functia de cost w(s) si h euristica asociata starilor

Iesiri: Calea de la s la starea finala sau {}

U' ← h(s),   bestPath ← {}

cat timp (bestPath = {} si U'≠ inf) **repeta**

        U ← U',   U' ← inf

        bestPath ←IDA*(s, 0, U)

intoarce bestPath

sfarsit

IDA*(s, g, U)

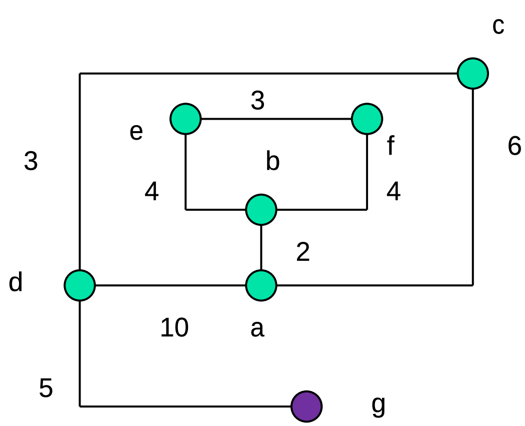
Intrari: starea s, costul caii g, limita superioara U

Iesiri: calea de la s la starea finala sau {}

Efect lateral: Actualizarea lui U'

```
daca (Goal(s)) atunci intoarce s
Suc(s) ← Expand(s)
pentru fiecare v in Suc(s) repeta
        daca     g + w(s,v) + h(v) > U           /* cost mai mare decat limita veche U */
              daca  g + w(s,v) + h(v) < U'      /* cost mai mic decat noua limita  */
              atunci U' ← g + w(s,v) + h(v)    /* actualizez noua limita */
        altfel
              p ← IDA*(v, g +w(s,v), U)
              daca p ≠ {} atunci intoarce (s,p)    /* s-a gasit solutie */
intoarce {}
sfarsit
```

Pas	Iteratie	Selectie	Apel	U	U’	Obs
1	1	{}	{(a,11)}	11	inf	h(a)
2	1	a	{}	11	14	f(b), f(d) si f(c) > U
3	2	{}	{(a,11)}	14	inf	Incepe o noua iteratie
4	2	a	{(c,14)}	14	15	f(b) si f(d) > U
5	2	c	{(d,14)}	14	15	
6	2	d	{(g,14)}	14	15	f(a) > U
7	2	g	{}	14	15	Scop atins



Cautare IDA*

5. Determinarea functiei de evaluare f

- In functie de problema
- Transformare abstracta prin relaxarea unor restrictii ale problemei – se poate automatiza
- Pattern databases – baze de date de sabloane generate din solutii partiale – se pot calcula prin program
- Ne intereseaza o functie euristica h(s) cat mai apriape de h*(s)
- Am dori si un effort cat mai mic

h(S) aproape de h*(S)?

- Fie doi algoritmi A*, A1 si A2, cu functiile de evaluare h_1 si h_2 admisibile, $g_1=g_2$

$$f_1(S) = g_1(S) + h_1(S) \qquad f_2(S) = g_2(S) + h_2(S)$$

- Se spune ca algoritmul **A2 este *mai informat* decat** algoritmul **A1** daca pentru orice stare S cu $S \neq S_f$

$$h_2(S) > h_1(S)$$

h_2 domina h_1

h(S) monotona?

- **Monotonia functiei h(S)**

Daca

$$h(S) \leq h(S') + cost\_arc(op, S, S')$$

pentru orice doua stari S si S' succesor al lui S, cu S' diferit de S_f, din spatiul de cautare

atunci se spune ca h(s) este **monotona (sau consistenta)**

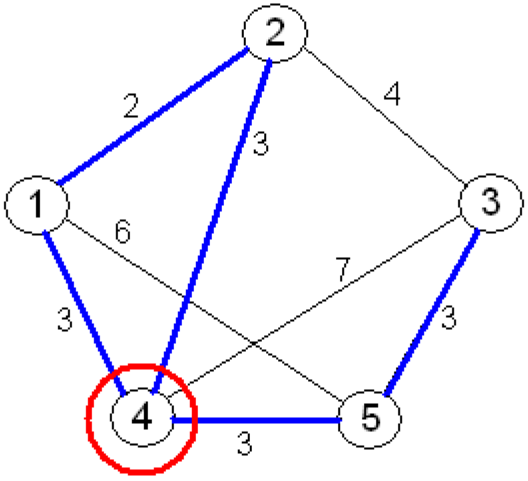
- Daca ***h este monotona*** atunci avem garantia ca un nod introdus in CLOSED nu va mai fi niciodata eliminat de acolo si reintrodus in OPEN iar implementarea se poate simplifica corespunzator

Exemple de functii euristice

- **Problema comis-voiajorului**

$$h_1(S) = cost\_arc(S_i, S)$$

- h2(S) = costul arborelui de acoperire de cost minim al oraselor neparcurse pana in starea S



Exemple de functii euristice

- **8-puzzle** $$h_1(S) = \sum_{i=1}^{8} t_i(S)$$

$$h_2(S) = \sum_{i=1}^{8} \text{Distanta}(t_i)$$

Distanta Manhattan
dx=|nod.x-scop.x|
dy=|nod.y-scop.y|
h(nod) = D*(dx+dy)

Relaxarea conditiei de optimalitate a algoritmului A*

- O functie euristica h se numeste **ε-admisibila** daca

$$h(S) \le h^*(S) + \varepsilon \qquad\qquad \text{cu } \varepsilon > 0$$

- Algoritmul A* care utilizeaza o functie de evaluare f cu o componenta ***h*** ε -admisibila gaseste intotdeauna o solutie al carei cost depaseste costul solutiei optime cu cel mult ε.
- Un astfel de algoritm se numeste *algoritm A* ε - admisibil* iar solutia gasita se numeste *solutie ε - optimala*.

Relaxarea conditiei de optimalitate a algoritmului A*

- 8-puzzle

$$f_3(S) = g(S) + h_3(S) \qquad h_3(S) = h_2(S) + 3 \cdot T(S)$$

$$T(S) = \sum_{i=1}^{8} Scor[t_i(S)]$$

$$Scor[t_i(S)] = \begin{cases} 2 & \text{daca patratul } t_i \text{ in starea S nu este urmat de} \\ & \text{succesorul corect din starea finala} \\ 0 & \text{pentru orice pozitie a lui } t_i \text{ diferita de centru} \\ 1 & \text{pentru } t_i \text{ aflat la centrul mozaicului} \end{cases}$$

Cum putem gasi o funcie euristica?

- **Variante “relaxate” ale problemei**
- *h1* si *h2* din 8 puzzle reprezinta de fapt distante dintr-o versiune simplificata a problemei

  O piesa poate fi mutata de la A la B **daca:**

  A este adiacent cu B pe verticala sau orizontala **si**

  B este liber

(1) O piesa poate fi mutata de la A la B daca A si B sunt adiacente

(2) O piesa poate fi mutata de la A la B daca B este liber

Cum putem gasi o functie euristica?

- A* in jocuri pt parcurgerea unui teritoriu

Distanta Manhattan

dx=|nod.x-scop.x|

dy=|nod.y-scop.y|

h(nod) = D*(dx+dy)

Distanta Euclidiana

dx=|nod.x-scop.x|

dy=|nod.y-scop.y|

h(nod) = D*rad(dx^2+dy^2)

Cum putem gasi o functie euristica?

- 2 tipuri de sol: campie (1) si munte (3)
- A* va cauta de 3 ori mai departe pe campie decat pe munte
- Aceasta deoarece este posibil sa existe o cale pe campie care merge pe langa munte
- Viteza algoritmului poate creste cu o distanta de 2 in loc de 3 – cauta numai de 2 ori mai mult pe campie
- g(s) = 1 + factor(t) *(g(s) -1)
- factor = 0 costul terenului compelt ignorat
- factor = 1 se va folosi costul real

Pattern database pt euristici

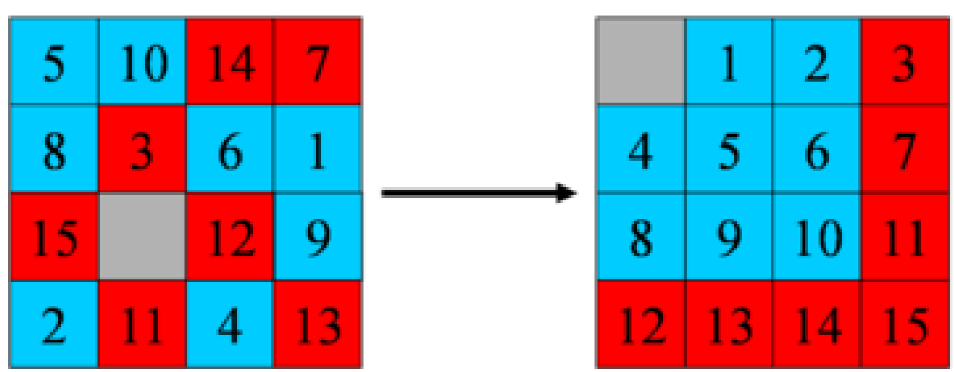
- Memoreaza o colectie de solutii a unor subprobleme care trebuie rezolvate pt a rezolva problema
- Functie euristica precalculata si memorata
- **Pattern** – o specificare partiala a unei stari
- **Target pattern** – a specificare partiala a starii scop
- **Pattern database** – multimea tuturor pattern-urilor care pot fi obtinute prin relaxari sau permutari ale target pattern

Pattern database pt euristici

- Pentru fiecare pattern din baza de date calculam distanta (nr minim de mutari) fata de target pattern folosind analiza inversa.
- Distanta este costul pattern-ului

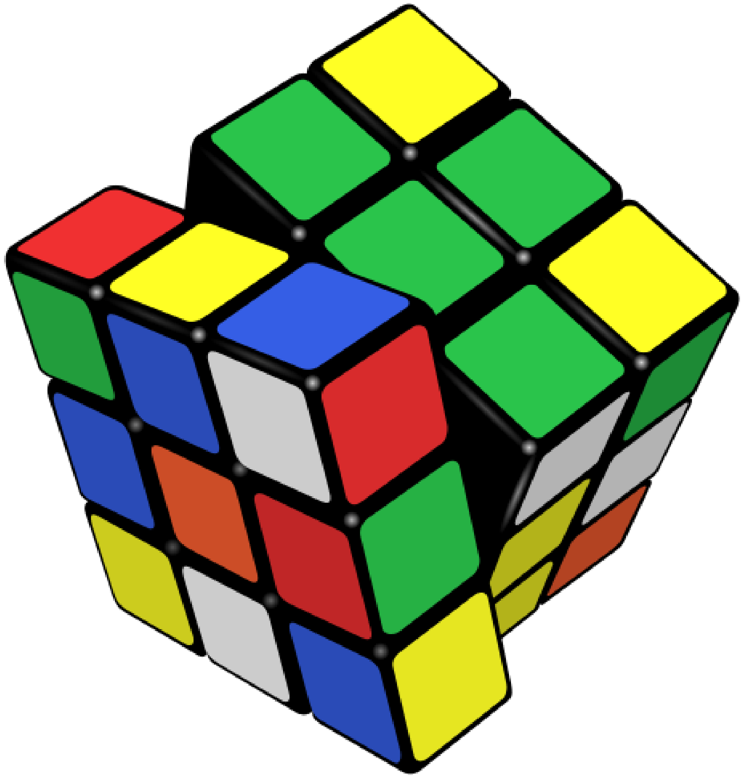
Pattern database pt euristici

- 15 puzzle
- Baza de date va indica numarul minim de mutari necesare pt a duce la locul bun 3, 7, 11, 12, 13, 14 si 15; apoi se rezolva 8-puzzle (albastru)
- 31 mutari pt a rezolva piesele rosii (22 mutari pentru a rezolva piesele albastre)



Cubul lui Rubik

- 9 patrate cu 6 culori diferite
- Cea mai buna solutie IDA*



Cubul lui Rubik

Euristici

- ***Distanta Manhattan 3D*** = Calculeaza distanta liniara intre 2 puncte in R3 prin insumarea distantelor punctului in fiecare dimensiune
- Distanta Manhattan 3D intre punctele p1 si p2

  $md3d(p1, p2) = |x1-x2|+|y1-y2|+|z1-z2|$
- Poate fi calculata in timp liniar
- Trebuie impartita la 8 – fiecare miscare muta 4 colturi si 4 muchii – pentru a fi admisibila

Cubul lui Rubik

- *Dureaza mult*
- Adancime 18 – am 250 ani
- **Pattern database**
- Se memoreaza intr-o tabela numarul de miscari necesare pt a rezolva colturile cubului sau subprobleme

Cea mai buna?

- Avem mai multe euristici bune
- Pe care o alegem?
- $h(n) = \max(h1(n), \ldots h_k(n))$