Inteligență Artificială

Universitatea Politehnica Bucuresti
Anul universitar 2021-2022

Adina Magda Florea

Curs 2

Strategii de căutare

- Reprezentarea soluției problemei
- Strategii de căutare de bază
- Strategii de căutare informate
- Strategii de căutare informate cu memorie limitată
- Determinarea funcției euristice

1 Reprezentarea soluției problemei

- Grafuri reprezentate explicit
- Grafuri reprezentate implicit

- Graful asociat unei probleme de cautare
- Graf neponderat sau graf ponderat

Reprezentarea soluției problemei

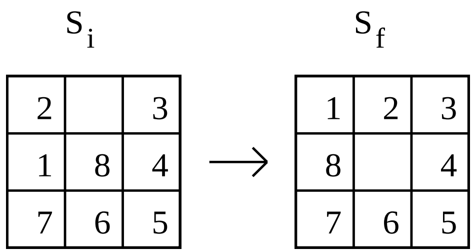
- Reprezentare prin spatiul starilor
- Reprezentare prin grafuri SI/SAU (AND-OR)
- Echivalenta reprezentarilor
- Caracteristicile mediului de rezolvare

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

1.1 Rezolvarea problemei reprezentată prin spațiul stărilor

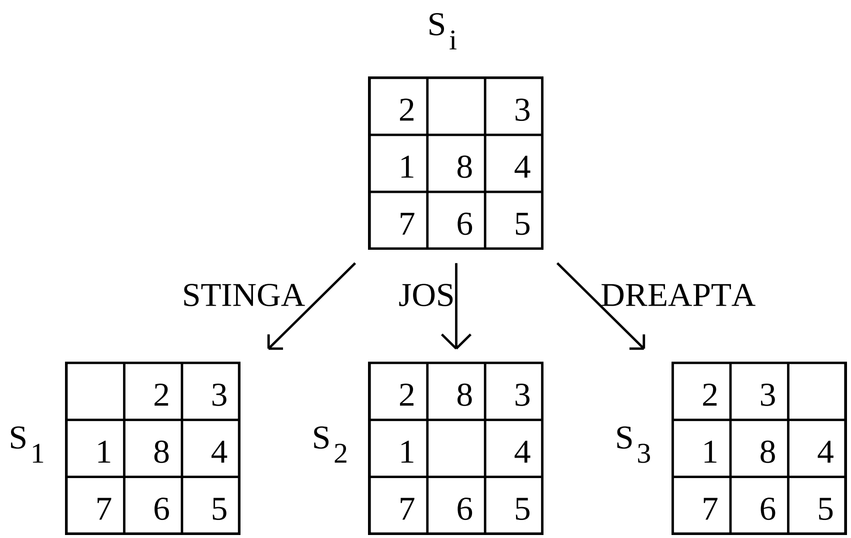
- (S_i, O, S_f)
- Stare
- Spatiu de stari (S_Set)
- Stare initiala
- Stare/stari finala/finale
- Solutia problemei

8-puzzle



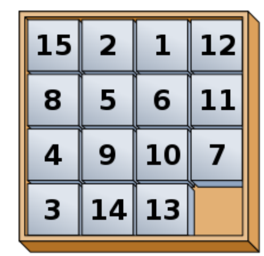
SUS - Mutare patrat liber in sus
STINGA - Mutare patrat liber la stinga
JOS - Mutare patrat liber in jos
DREAPTA - Mutare patrat liber la dreapta

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



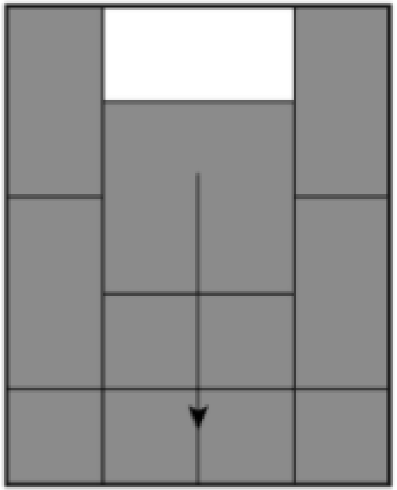
(d) Tranzitii posibile din starea S_i

15-puzzle

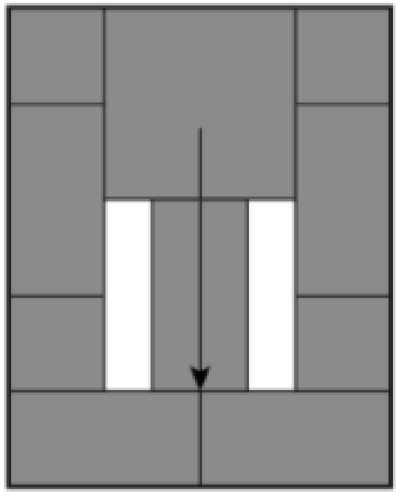


$(n^2\text{-}1)$-puzzle

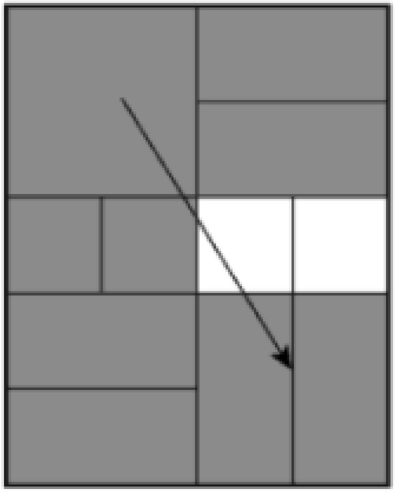
General sliding tile puzzle



Donkey' puzzle

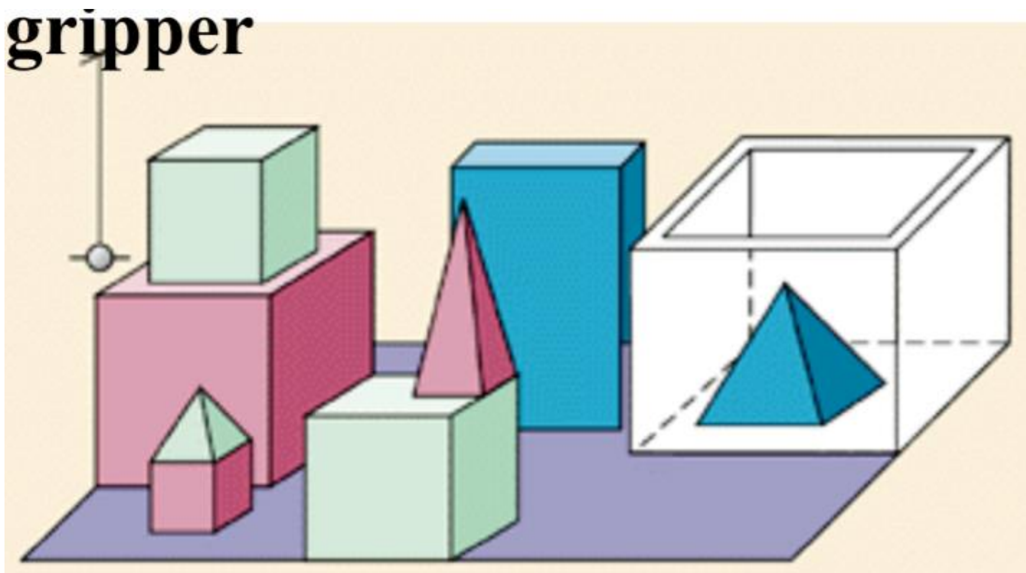


Century puzzle

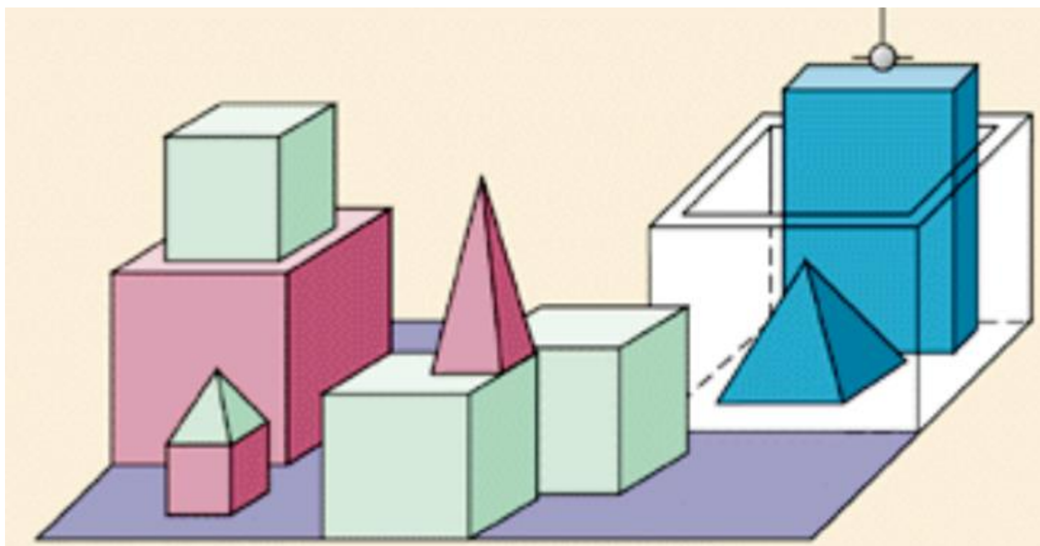


Dad' puzzle

Block world



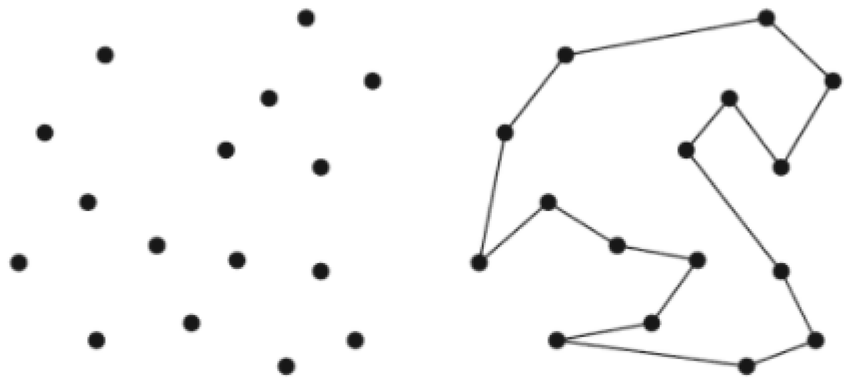
(a) "Pick up a big red block."



(b) "Find a block which is taller than the one you are holding and put it into the box."

Probleme cu cost

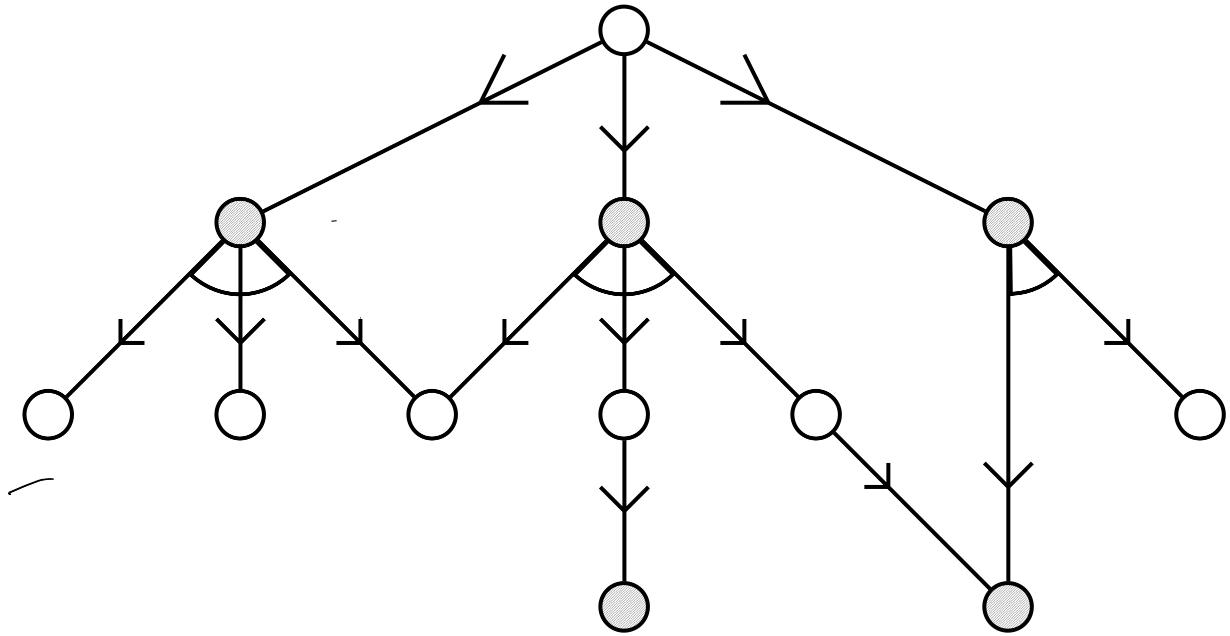
- Planificarea rutelor
- Problema comisvoiajorului



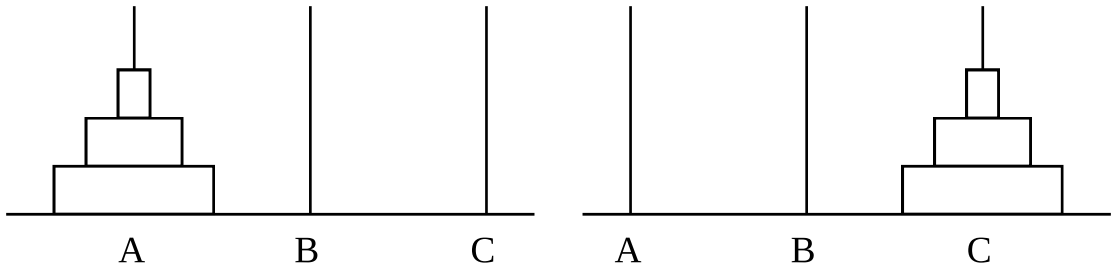
1.2 Rezolvarea problemei reprezentată prin grafuri SI/SAU

- (P_i, O, P_e)
- Nod problema
- Spatiul problemelor (P_Set)
- Problema initiala
- Problema elementara / probleme elementare
- Semnificatie graf SI/SAU (AND-OR)
- Nod rezolvat
- Nod nerezolvabil
- Solutia problemei

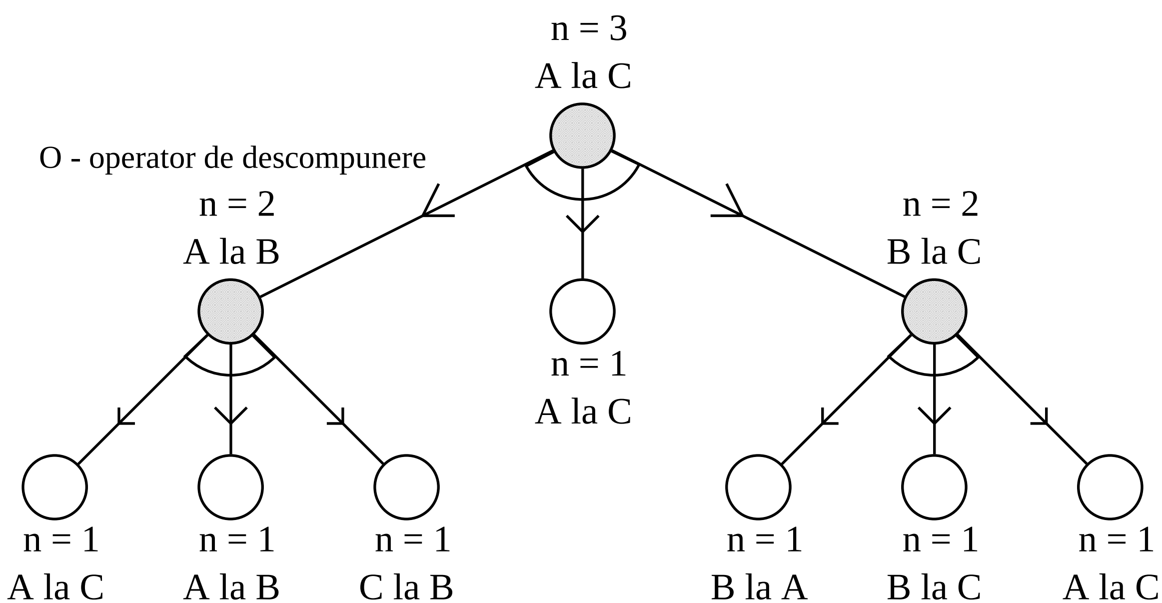
Graf SI/SAU



Turnurile din Hanoi



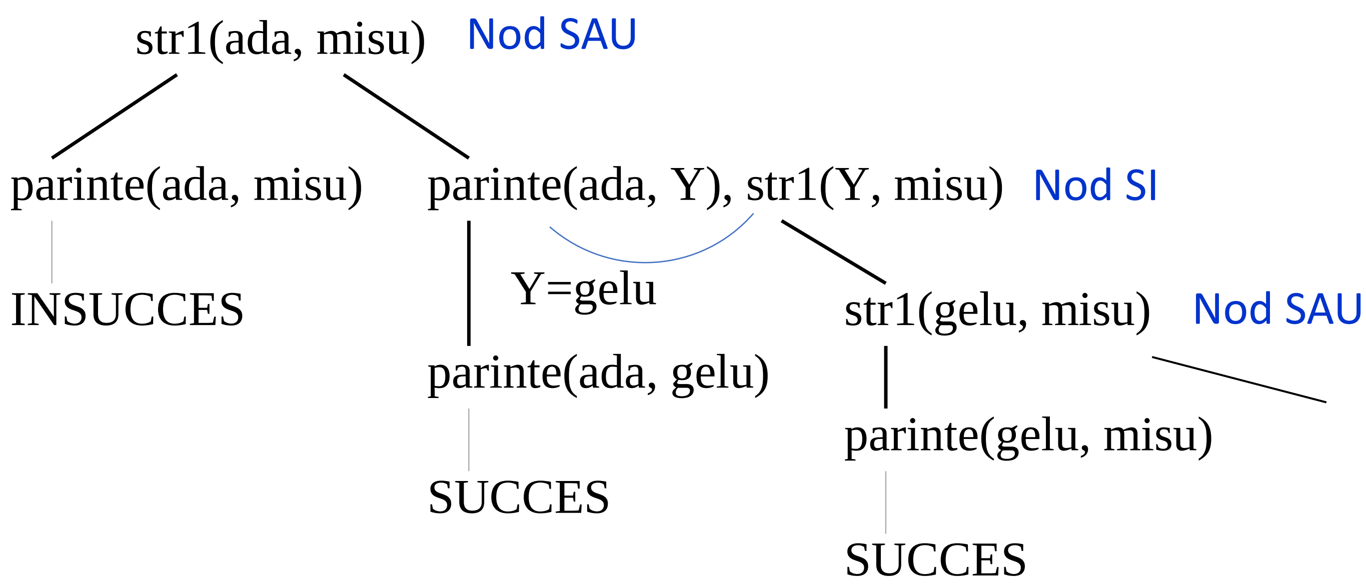
(a) Stare initiala (b) Stare finala



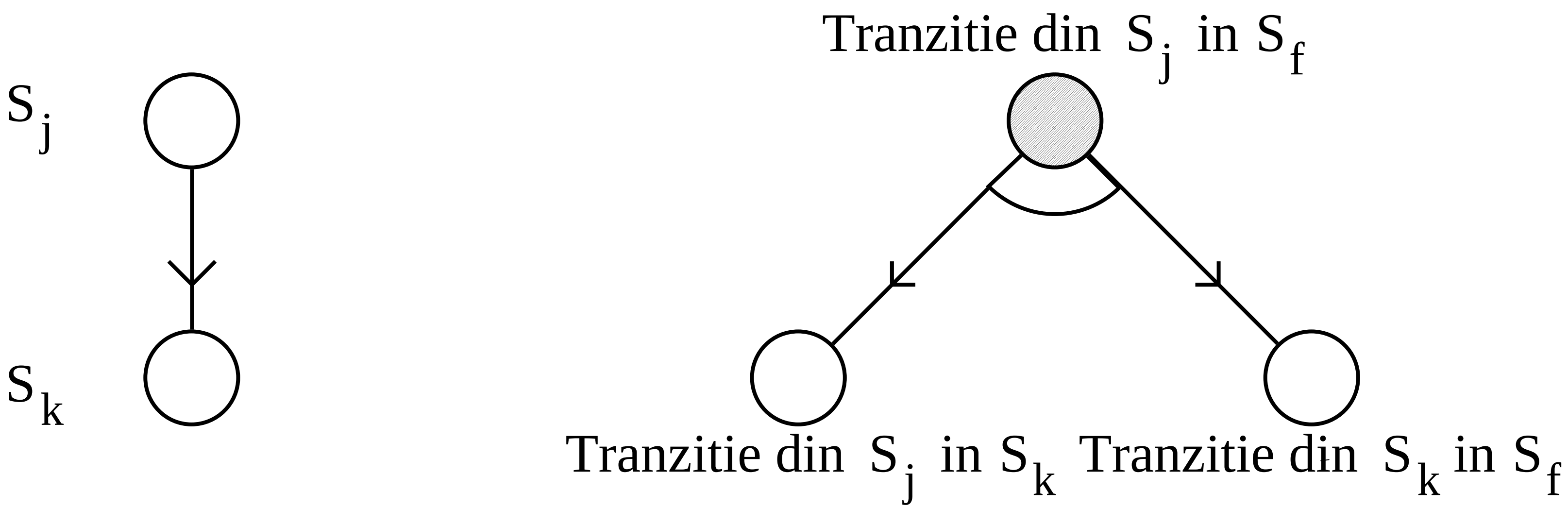
(c) Arborele SI/SAU de descompunere in subprobleme

PROLOG

```
parinte(ada,gelu)
parinte(gelu, misu).
str1(X, Z) :- parinte(X, Z).
str1(X, Z) :- parinte(X, Y), str1(Y, Z)
?-str1(ada,misu).
```



Echivalența reprezentărilor



S_j, S_k - stari intermediare S_f - stare finala

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

1.3 Caracteristicile mediului

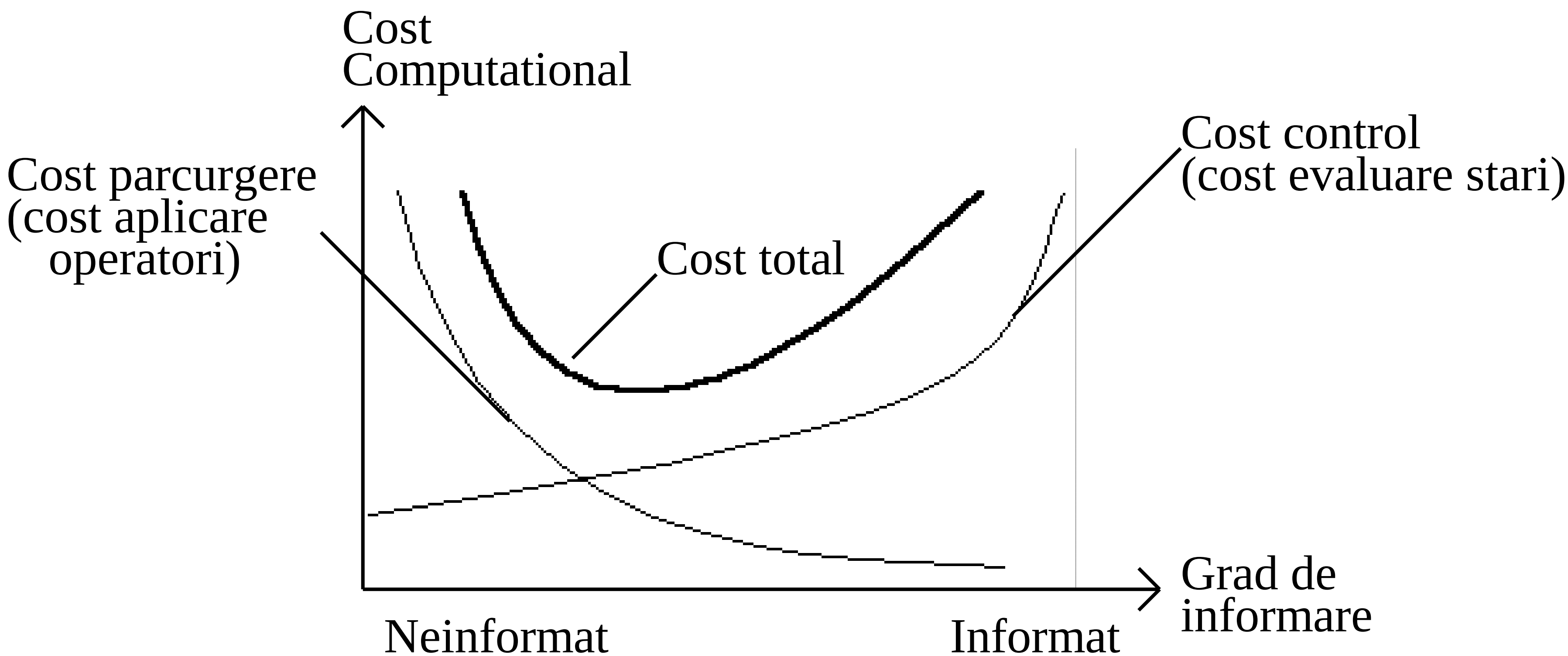
- Observabil / neobservabil
- Discret / continuu
- Finit / infinit
- Determinist / nedeterminist

2. Strategii de căutare de bază

Criterii de caracterizare

- Completitudine
- Optimalitate
- Complexitate: timp, spatiu
- Capacitatea de revenire
- Informare

Costuri ale cautarii



2.1 Căutari neinformate

- Gasirea unei cai sau a tuturor cailor, cu cost sau fara cost

Grafuri specificate explicit

- **Cautarea pe nivel** si **cautarea in adancime** - 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 cai de cost minim daca toate costurile arcelor sunt ≥ 0
- **Algoritmul Bellman-Ford** rezolva problema cai de cost minim dar costurile arcelor pot fi si negative (de la un nod sursa la toate)
- **Algoritmul Floyd-Warshall** rezolva problema gasirii tuturor cailor de cost minim, costuri pozitive si negative dar cicluri ne-negative

Căutari neinformate pentru grafuri specificate implicit / probleme de cautare

- Cautarea pe nivel (Breadth First Search - BFS)
- Cautare in adancime (Depth First Search - DFS)
- Cautare in adancime cu nivel iterativ (Iterative Deepening - ID)
- Cautare de tip backtracking
- Cautare bidirectionala

Căutari neinformate în spațiul stărilor

- 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.
- Cele mai multe implementari bazate pe utilizarea a 2 liste: OPEN si CLOSED
- DFS – Open – stiva (LIFO)
- BFS – Open – coada (FIFO)
- In cele mai multe implementari CLOSED este implementata ca o tabela de dispersie (Hash)

Algoritm BFS / DFS(AdMax): Cautare pe nivel/in adancime in spatiul starilor

1. Initializeaza listele OPEN $\leftarrow \{S_i\}$, CLOSED $\leftarrow \{\}$
2. **daca** OPEN = {}

   atunci intoarce INSUCCES
3. Elimina primul nod S din OPEN si insereaza-l in CLOSED
4. **daca** S $\in$ OPEN $\cup$ CLOSED (inainte de inserare S)

   atunci repeta de la 2

4'. **daca** Ad(S) = AdMax **atunci repeta de al 2** /* pt DFS */

5. Expandeaza nodul S

   5.1. Genereaza toti succesorii directi S_j ai nodului S

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

   5.2.1. Stabileste legatura $S_j \rightarrow S$

   5.2.2. **daca** S_j este stare finala

   atunci

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

   ii. **intoarce** SUCCES

   5.2.3. Insereaza S_j in OPEN, ***la sfarsit*** / ***la inceput***

6. **repeta de la** 2

sfarsit.

Algoritm DFS(AdMax): Cautare in adancime cu adancime limitata

Intrari: Starea initiala S_i

Iesiri: SUCCES *si solutia sau* INSUCCES sau *AdMax*

1. Initializeaza lista OPEN $\leftarrow \{S_i\}$,
2. **daca** OPEN = {} **atunci intoarce** INSUCCES
3. Elimina primul nod S din OPEN
4. **daca** S este stare finala

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

   ii. **intoarce** SUCCES
5. **daca** Ad(S) > AdMax **atunci intoarece** AdMax
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 Stabileste legatura $S_j \rightarrow S$

   6.2.2. **daca** $S_j \notin$ OPEN

   atunci introduce S_j in OPEN
7. **repeta de la 2**

sfarsit.

Algoritm Iterative Deepening: Cautare in adancime cu nivel iterativ

Intrari: Starea initiala S_i

Iesiri: SUCCES *si solutia sau* INSUCCES

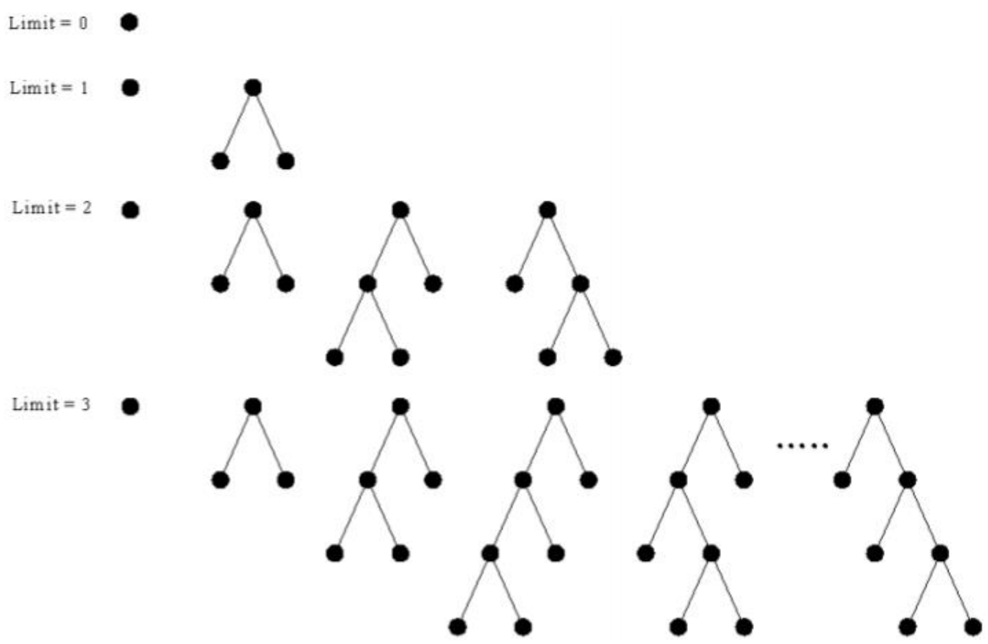
1. **pentru** AdMax = 0 la inf **repeta**

   1.1 **rezultat** = DFS(AdMax)

   1.2 **daca** rezultat <> AdMax

   atunci intoarece rezultat

sfarsit



Abordare hibrida

Căutari neinformate în grafuri SI/SAU

- Se folosesc notiunile de nod rezolvat, nerezolvabil
- BFS, DFS
- Notiunea de adancime a unui nod este diferita
- Intr-o reprezentare a solutiei problemei prin grafuri SI/SAU *adancimea unui nod* se defineste astfel:
  - $Ad(S_i) = 0$, 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 BFS-AND-OR: Cautare pe nivel in grafuri SI/SAU

1. Initializeaza listele OPEN $\leftarrow$ {S_i}, CLOSED $\leftarrow$ {}
2. Elimina primul nod S din OPEN si insereaza-l in CLOSED
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 OPEN, ***la sfarsit***
      3.2.3. **altfel** insereaza S_j in OPEN, ***la sfarsit***

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 OPEN 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 OPEN toate nodurile rezolvate si toate nodurile care au predecesori rezolvati

5. **repeta de la** 2

sfarsit.

2.2 Complexitatea strategiilor de căutare

- 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 ⇒ **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 ⇒ **15 miscari** ⇒ **B = 1.67**

Complexitatea strategiilor de căutare

- **B** - *factorul de ramificare*
- **d**- adancimea celui mai apropiat nod solutie (de cost minim daca exista costuri)
- **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 căutare

- **Cautare pe nivel**

  Numar de noduri generate

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

- **Cautare in adancime**

  Numar de noduri generate - **B*d** – daca nodurile expandate se sterg din CLOSED

  Sau B*m cu AdMax=m

Complexitatea strategiilor de căutare

- **Cautare backtracking**

  Numar de noduri generate **m**

- **Cautare in adancime cu nivel iterativ**

  Numar de noduri generate

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

Complexitatea strategiilor de căutare

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 căutare 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

3.1 Căutare 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
- O functie euristica w(n) este o evaluare a unui nod care mapeaza acel nod la o valoare reala >=0
- Exemple
  - Strategia de cautare “best-first”
  - Strategia de cautare a alpinistului

Algoritm BestFS: 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

3.2 Căutarea soluției optime în spațiul 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) & h(S_f)=0**
- Definitia stabileste ***conditia de admisibilitate*** a functiei h si este folosita pentru a defini ***proprietatea de admisibilitate*** a unui algoritm A*.

A* - proprietatea de admisibilitate

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) $\text{cost\_arc}(S, S') \geq c$

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 …

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

Consistenta si monotonia functiei euristice

- **Euristica consistenta**

O functie euristica h este **consistenta** daca

$$h(S) \leq h(S') + cost\_arc(op, S, S') \;\; \& \; h(S_f)=0$$

pentru orice doua stari S si S' succesor al lui S din spatiul de cautare

- **Euristica monotona**

O functie euristica h este **monotona** daca

$f(S_j) \geq f(S_i)$ pentru orice $j > i$, $0 \leq i,j \leq n$, cu $S_n=S_f$

si $f(S)=g(S)+h(S)$

Estimarea costului total al caii de cautare este nedescrescatoare de la un nod la succesorii lui

Consistenta si monotonia functiei euristice

Teorema 1

O functie euristica h este **consistenta** daca si numai daca este **monotona**

Consistenta si monotonia functiei euristice

Teorema 2

O functie euristica **consistenta** este **admisibila**

Consistenta si monotonia functiei euristice

- Maximul a 2 functii admisibile este o fct admisibila
- Maximum a 2 functii consistente este o fct consistenta
- Fct euristice admisibile nu sunt necesare si consistente
- 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

Relaxarea condiției de optimalitate a algoritmului A*

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

$$h(S) \leq h^*(S) + \varepsilon$$

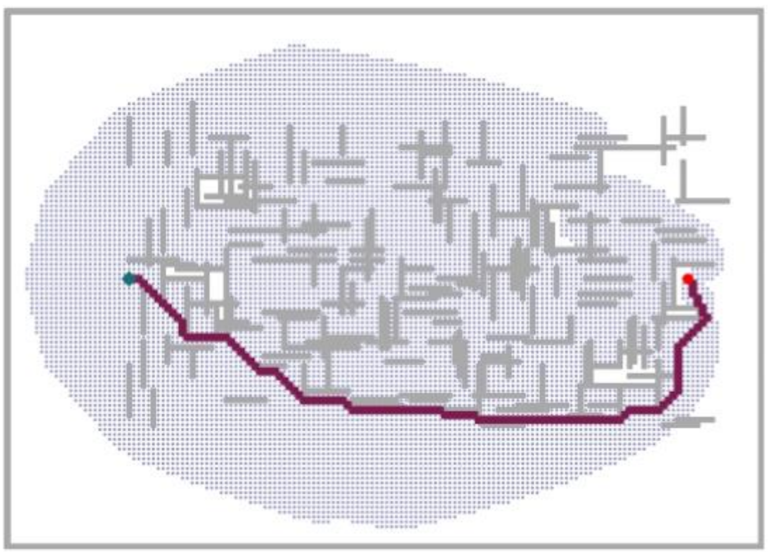
cu $\varepsilon > 0$

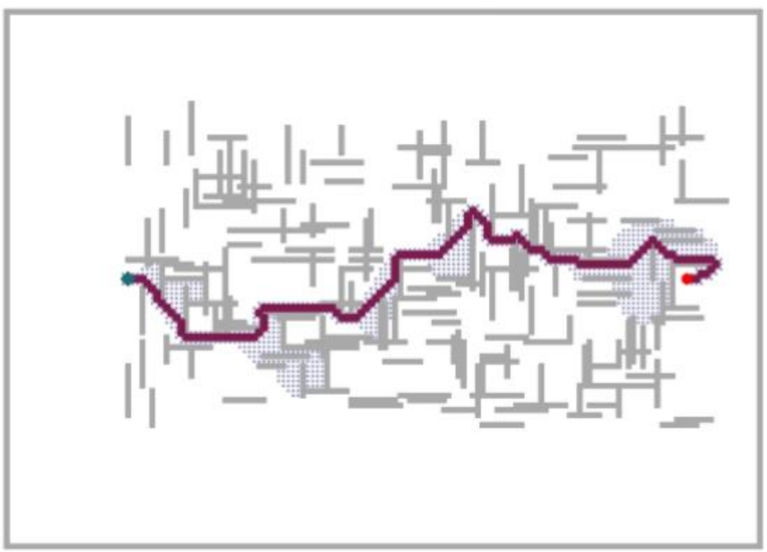
- Algoritmul A* care utilizeaza o functie de evaluare f cu o componenta ***h*** ε -admisibila gaseste intotdeauna o solutie al carei cost depaseste costul solutiei optime cu cel mult ε.
- Un astfel de algoritm se numeste *algoritm A* ε -admisibil* iar solutia gasita se numeste *solutie ε -optimala*.

Relaxarea condiției de optimalitate

- De fapt putem utiliza o functie euristica ne-admisibila
- A* ponderat (Weighted A* search)

$$f(S) = g(S) + W * h(S), \quad W>1$$


(a)

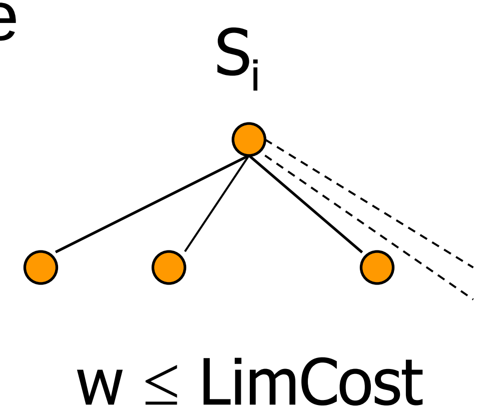

(b)

4. Strategii de căutare informată cu memorie limitată

- 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.
- Algoritmi pentru cautare solutiei de cost optim cu memorie limitata sunt:
  - Beam search
  - Depth first iterative deepening cu cost (DFID)
  - Iterative deepening A* (IDA*)
  - Memory bound A* (MA*)
  - Simplified memory bound A* (SMA*)

DFID cu cost

- 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 atunci DFID fara cost



DFID cu cost

- Algoritmul utilizeaza **doua limite U si U’** pentru urmatoarea iteratie.
- Apeleaza repetitiv functia DFID care cauta o cale de cost minim **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: Depth first iterative deepening cu cost

Foloseste

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

BuclaDFID

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

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

U’ $\leftarrow$ 0, bestPath $\leftarrow$ {} /* initializare limita globala si cale */

cat timp (bestPath = {} si U’$\neq$ inf) **repeta** **/*** nu s-a gasit solutie, exploreaza */

U $\leftarrow$ U’, U’ $\leftarrow$ inf /* reset limita, init limita globala noua */

bestPath $\leftarrow$**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** Cale(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** (p) /* s-a gasit solutie */

    altfel /* cost mai mare decat limita veche U */

        daca g + w(s,v) < U' /* cost mai mic decat limita globala */

        atunci U' $\leftarrow$ g + w(s,v) /* seteaza limita globala noua */

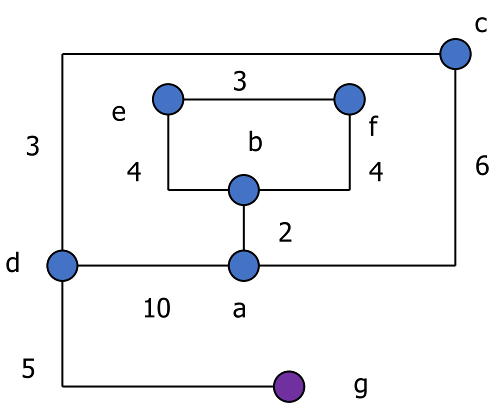
intoarce {}

sfarsit

Cautare DFID

Pas	Iteratie	Selectie	Apeluri	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
8	3	b	{(e,6),(f,6),(c,6)}	6	10	
9	3	e	{(f,6),(c,6)}	6	10	g(f) > U
10	3	f	{(c,6)}	6	10	g(e) > U
11	3	c	{}	6	9	g(d)
12	4	{}	{(a,0)}	9	inf	Incepe o noua iteratie

…………..

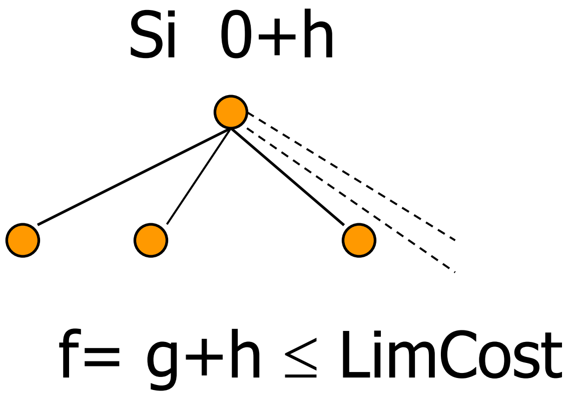


pentru fiecare v in Suc(s) **repeta**
  daca g + w(s,v) ≤ U
  atunci p ← DFID(v, g+w(s,v), U)
    daca p ≠ {} **atunci intoarce** (p)
  altfel
    daca g + w(s,v) < U' **atunci** U' ← g + w(s,v)

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

IDA*

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



Algoritm IDA*: 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** Cale(s)

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

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

    daca g + w(s,v) + h(v) ≤ U /* starea este in limita U */

    atunci

        p $\leftarrow$ IDA*(v, g +w(s,v), U)

        daca p ≠ {} **atunci intoarce** p /* s-a gasit solutie */

    altfel /* cost mai mare decat limita veche U */

        daca g + w(s,v) + h(v) < U' /* cost mai mic decat noua limita */

        atunci U' $\leftarrow$ g + w(s,v) + h(v) /* actualizez noua limita */

intoarce {}

sfarsit

5. Determinarea funcției de evaluare h

- Specifica problemei, determinate manual (hand crafted)

SAU

- Generare automata de euristici
- **Transformare abstracta** prin relaxarea unor restrictii ale problemei
- Pattern databases – precalculeaza si memoreaza distanta pana la solutie intr-un spatiu abstract generat de relaxarea restrictiilor impuse miscarilor/actiunilor
- Ne intereseaza o functie euristica h(s) cat mai apriape de h*(s)
- Am dori si un effort cat mai mic

Exemple de funcții 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 funcții 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) = (dx+dy)

Relaxarea condiției de optimalitate pentru 8-puzzle

- 8-puzzle

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

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

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

Transformari abstracte pt functii euristice

- O transformare abstracta F:S→S’, spatiu abstract, mapeaza stari ***s*** din S (problema reala) in stari F(s) din S’ si mutari/actiuni ***a*** din problema reala in actiuni in spatial abstract
- Daca distanta intre oricare 2 stari F(u) si F(v) in spatial abstract este mai mica sau egala decat distanta intre u si v, atunci aceasta distanta poate fi utilizata ca o euristica admisibila
- Calculate pe parcursul cautarii sau stocate in pattern databases

Transformari abstracte pt functii euristice

- **Variante “relaxate” ale problemei**

O piesa poate fi mutata de la X la Y daca X si Y sunt adiacente si X este blank

O piesa poate fi mutata de la X la Y daca X si Y sunt adiacente

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

O piesa poate fi mutata de la X la Y

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

Cum putem găsi o funcție euristică?

- A* in jocuri pt parcurgerea unui teritoriu

Distanta Manhattan

dx=|nod.x-scop.x|

dy=|nod.y-scop.y|

h(nod) = (dx+dy)

Distanta Euclidiana

dx=|nod.x-scop.x|

dy=|nod.y-scop.y|

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

Cum putem găsi o funcție euristică?

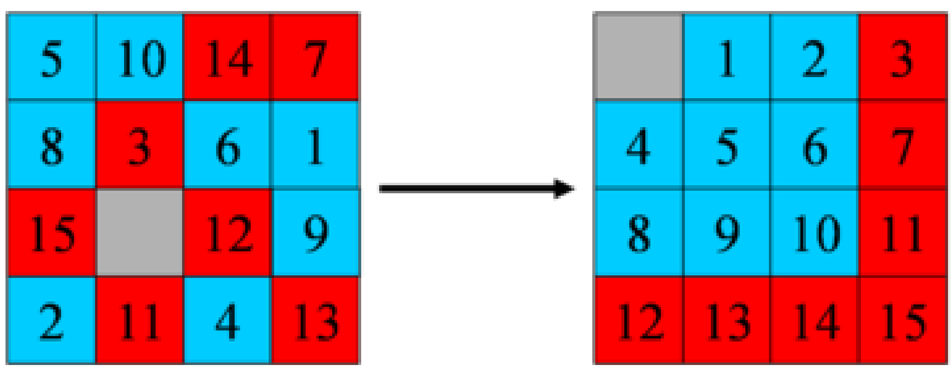
- **Pattern database pt euristici**
- Memoreaza o colectie de solutii ale unor subprobleme care trebuie rezolvate pentru a rezolva problema
- Fiecare solutie de subproblema are o functie euristica precalculata (costul cautarii) si memorata
- **Pattern** – o specificare partiala a unei stari
- **Target pattern** – a specificare partiala a starii scop

Pattern database pt euristici

- **Pattern database** – multimea tuturor pattern-urilor care pot fi obtinute prin relaxari sau permutari ale target pattern
- 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

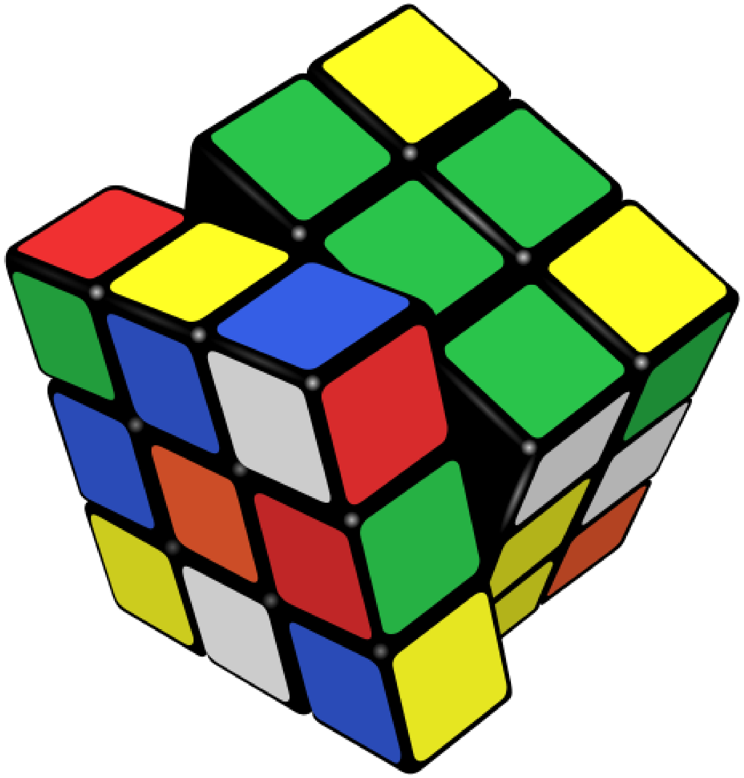
Pattern database pt euristici

- 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 pentru a fi admisibila deoarece fiecare miscare muta 4 colturi si 4 muchii

Cubul lui Rubik

- *Dureaza mult*
- Adancime 18 – aprox 100 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?
- $h(n) = \max(h1(n), \ldots h_k(n))$