

Problémamegoldó ágensek

# Problémamegoldó ágens

- Célorientált ágens
- Célorientált ágens egyik típusa
- olyan cselekvéssorozatot keresnek, amelyek a kívánt állapotokba vezetnek
- **célmegfogalmazás (goal formulation):**
  - Problémamegoldás első lépése
  - Pillanatnyi helyzeten és az ágens hasznosságértékén alapul
  - Korlátozza az ágens által elérendő dolgok számát → döntési probléma nagy mértékben leegyszerűsödik

# Problémamegoldó ágens

- **problémamegfogalmazás (problem formulation)**: az a folyamat, amely során eldöntjük, hogy mely cselekvéseket és állapotokat vegyük figyelembe, ha egy cél adott.
- **keresés (search)**: lehetséges cselekvéssorozatok előállításának folyamata
  - bemenete egy probléma
  - kimenete pedig egy cselekvéssorozat formájában előálló **megoldás (solution)**: kiinduló állapotból a célállapotba vezet
  - **végrehajtási (execution) fázis**: a megoldásban szereplő cselekvéssorozat végrehajtása

# Problémamegoldó ágens

- Összefoglalva a tervezési folyamatot:
  - Először a célt és a problémát fogalmazza meg
  - Megkeresi a problémát megoldó cselekvéssorozatot
  - végül a cselekvéseket egyenként végrehajtja
  - Amikor kész vele, egy másik célt fogalmaz meg, és az egészet újratekinti

# Ágens környezetének jellemzői

- **Statikus**: környezet változásáról nem vesz tudomást
- **Megfigyelhető**: kezdeti állapot megállapítása könnyebb
- **Diszkrét**: problémamegfogalmazásnál megadott cselekvések lehetnek
- **Determinisztikus**: megoldások egyedi cselekvéssorozatok → váratlan eseményeket nem tudják figyelembe venni

# Probléma megfogalmazása

4 komponens szükséges:

1, **kiinduló állapot (initial state)**: ebből kezdi az ágens a cselekvéseit

2, lehetséges **cselekvések (actions)** halmaza: általában az  
az **állapotátmenet-függvényt (successor function)** alkalmazza

**Állapotátmenet-függvény**: egy adott  $x$  állapot esetén visszaadja a rendezett  $\langle \text{cselekvés}, \text{utódállapot} \rangle$  párok halmazát, ahol minden cselekvés az  $x$  állapotban legális cselekvések egyike, és minden utódállapotot egy cselekvésnek az  $x$  állapotra való alkalmazásával nyerünk.

# Probléma megfogalmazása

## Állapottér (state space):

kezdeti állapot és az állapotátmenet-függvény együttesen implicit módon definiálják

azon állapotok halmazát, amelyek a kiinduló állapotból elérhetőek

Az állapottér egy **útja (path)**: az állapotok egy sorozata, amely állapotokat a cselekvések egy sorozata köt össze

# Probléma megfogalmazása

**3, célteszt (goal test):** meghatározza, hogy egy adott állapot célállapot-e

- Néha létezik a lehetséges célállapotok egy explicit halmaza, és a teszt egyszerűen megnézi, hogy az ágens elérte-e ezek egyikét.

**4, útköltség- (path cost) függvény:** minden úthoz hozzárendel egy költséget

- ágens azt a költségfüggvényt fogja választani, amely a saját hatékonysági mértékének felel meg
- A megoldás kvalitását az útköltségfüggvény méri, és egy **optimális megoldásnak (optimal solution)** a megoldások közt a legkisebb lesz az útköltsége.

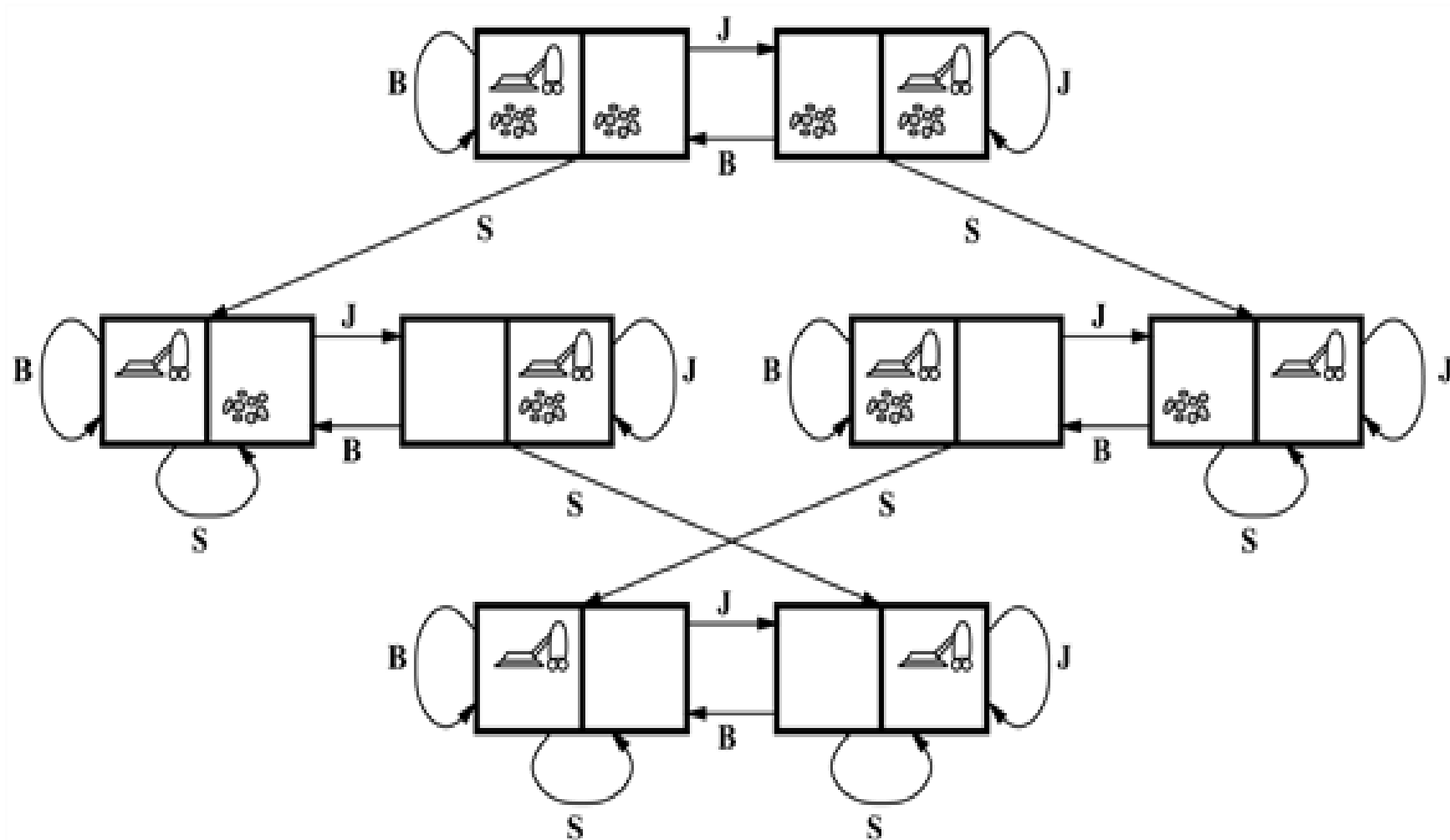


# Példa: „Porszívóvilág”

Problémamegfogalmazás:

- **Állapotok:** az ágens két hely egyikében lehet. Mindegyik lehet piszkos, de lehet tiszta is. Így  $2 \times 2^2 = 8$  lehetséges állapotról beszélhetünk.
- **Kezdeti állapot:** akármelyik állapot lehet kezdeti állapot.
- **Állapotátmenet-függvény:** a három (*Balra*, *Jobbra*, *Szív*) cselekvés alkalmazásából adódó legális állapotokat generálja. A teljes állapottér a 3.3. ábrán látható.
- **Célteszt:** ellenőrzi, hogy minden négyzet tiszta-e.
- **Útköltség:** minden lépés költsége 1, így az út költsége megegyezik az út lépéseinek a számával.

# „Porszívóvilág” állapottere



# Példa: 8-as kirakójáték (8-puzzle)

- Cél: egy meghatározott állás, mint például az ábra jobb oldalán látható állás elérése

7	2	4
5		6
8	3	1

Kiinduló állapot

	1	2
3	4	5
6	7	8

Célállapot

# 8-as kirakójáték (8-puzzle)

NP-teljes probléma: polinomiális időben  $O(x^n)$  nem megoldható  
gyakran új kereső algoritmusok tesztelésére alkalmazzák a kirakójátékot  
8-as kirakójátéknak  $9!/2 = 181\,440$  elérhető állapota van  
15-ös kirakójátéknak (a  $4 \times 4$ -es táblán) kb. 10 billió állapota van

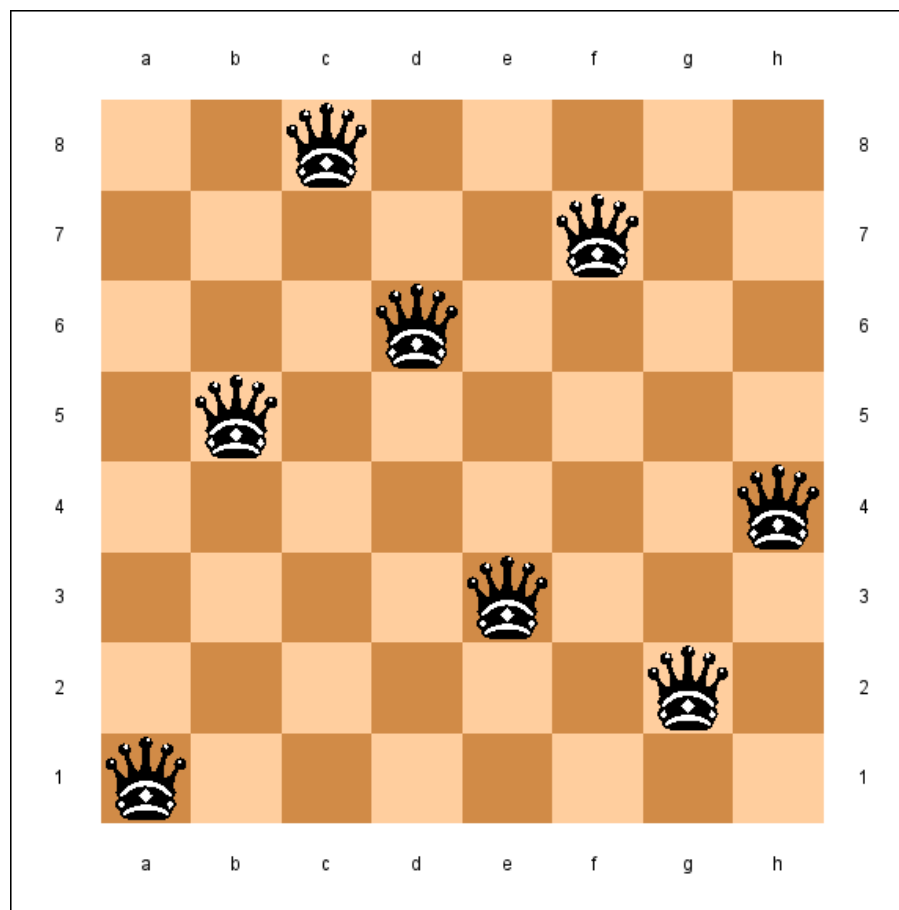
# 8-as kirakójáték (8-puzzle)

Problémamegfogalmazás:

- **Állapotok:** az állapotleírás meghatározza mind a nyolc kocka és az üres hely pozícióját a kilenc lehetséges pozíció egyikében.
- **Kezdeti állapot:** akármelyik állás lehet kezdeti állapot.
- **Állapotátmenet-függvény:** a négy cselekvés (üres hely megy *Balra, Jobbra, Fel, Le*) alkalmazásából adódó *legális állapotokat generálja*.
- **Célteszt:** *ellenőrzi, hogy az állapot megegyezik-e a célállapottal (más célkonfiguráció is lehetséges).*
- **Útköltség:** *minden lépés költsége 1, így az út költsége megegyezik az út lépéseinek a számával.*

# 8-királynő probléma (8-queens problem)

- Cél: 8 királynőt úgy helyezzünk el egy sakktáblán, hogy egyáltalán ne támadják egymást.



# 8-királynő probléma (8-queens problem)

- 2 fő megfogalmazása létezik:
  - **inkrementális megfogalmazás (incremental formulation):** a királynőket egyenként helyezzük el a sakktáblán
  - **teljes állapot leírás (complete-state formulation):** először felhelyezzük mind a 8 királynőt, majd mozgatjuk őket

# 8-királynő probléma (8-queens problem)

Problémamegfogalmazás inkrementális megközelítésben:

- **Állapotok:** egy állapot a 0 ... 8 királynő, tetszőleges elrendezése a táblán
- **Kezdeti állapot:** a táblán nincs egy királynő sem
- **Állapotátmenet-függvény:** helyezz egy új királynőt egy üres mezőre
- **Célteszt:** 8 királynő a táblán és egyik sincs támadás alatt

Ebben a megfogalmazásban  $64 \times 63 \times \dots \times 57 \approx 1,8 \times 10^{14}$  lehetséges vizsgálandó sorozatunk van.  $\rightarrow$  módosítsuk úgy, hogy ne helyezzünk királynőt támadt mezőre



# 8-királynő probléma (8-queens problem)

Problémamegfogalmazás módosítása:

- **Állapotok:**  $n$  ( $0 \leq n \leq 8$ ) királynő olyan elrendezése a táblán, hogy az  $n$  bal oldali oszlopban oszloponként egy található úgy, hogy nem támadják egymást.
- **Állapotátmenet-függvény:** helyezz egy királynőt a bal szélső, még üres oszlopba úgy, hogy azt ne támadja egyetlen királynő sem.

probléma állapottere  $1,8 \times 10^{14}$ -ről 2057 méretű térré csökkent

# Valós világbeli problémák

- **útkeresési problémák (route-finding problem):** számítógép-hálózatokban útvonalkeresésre, katonai műveletek tervezésénél és légi útvonaltervező rendszerekben is
- **körutazási problémák (touring problems):** kiindulási állapotba térjünk vissza
- **Az utazó ügynök probléma (Travelling Salesperson Problem – TSP):** egy körutazási probléma, amelyben minden várost pontosan egyszer kell meglátogatni  
cél: legrövidebb út megkeresése

# Valós világbeli problémák

- **robotnavigáció (robot navigation):** útkeresési probléma általánosítása

Az utak diszkrét halmaza helyett a robot egy folytonos térben (elvben) végtelen cselekvés- és állapothalmazzal

- **interneten kereső (Internet searching)** szoftverrobotok: kérdésekre választ, kapcsolódó információkat, vagy kereskedelmi üzletek lehetőségét kutatják fel

az internethez egy gráfszerű kép rendelhető, ahol a csomópontok (weboldalak) linkekkel össze vannak kapcsolva

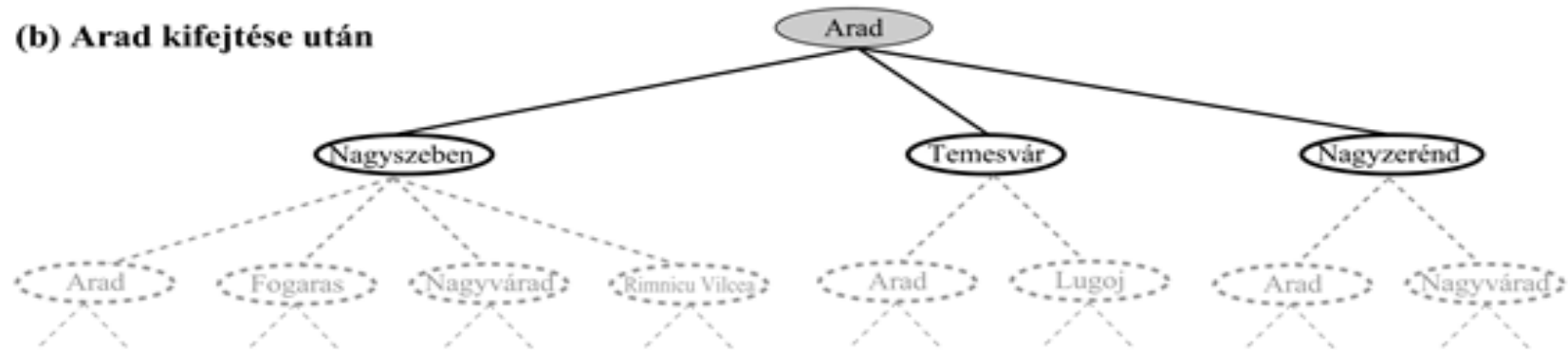
# Megoldás keresése keresési fa alkalmazásával

- **keresési fát (search tree)** az állapotteret együttesen definiáló kezdeti állapotból és az állapotátmenet-függvényből generálnak
- aktuális állapot **kifejtésével (expanding)**: állapotátmenet-függvénynek az aktuális állapotra történő alkalmazásával, amivel az állapotok egy új halmazát **generáljuk (generating)**.
- A keresés lényege a következő: egy lehetőséget kiválasztani, és a többi későbbre halasztani arra az eshetőségre, ha az első választás nem vezetne megoldásra.
- A kifejtendő állapot kiválasztását a **keresési stratégia (search strategy)** határozza meg.

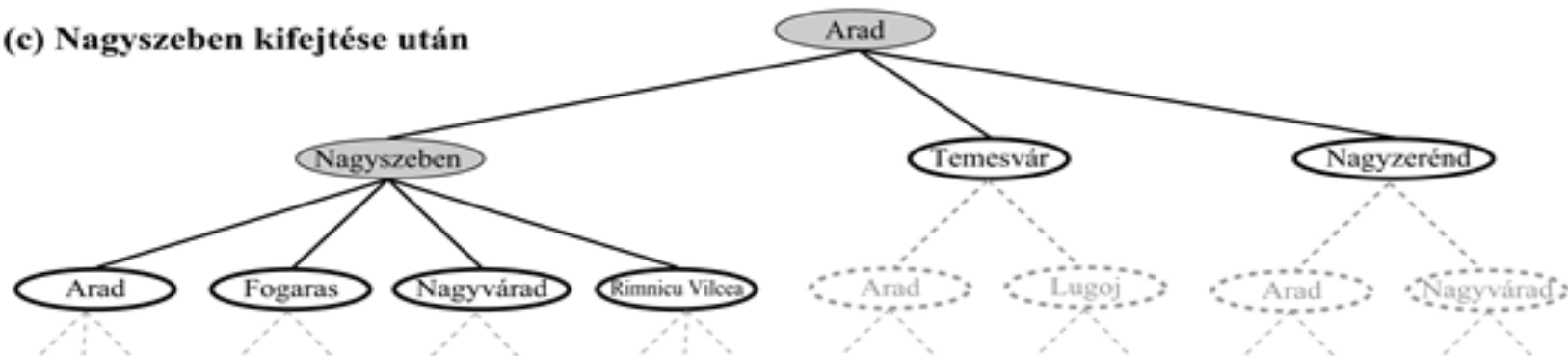
**(a) Kezdeti állapot**



**(b) Arad kifejtése után**



**(c) Nagyszeben kifejtése után**



# Keresési fa

```
function FA-KERESÉS(probléma, stratégia) returns egy megoldás vagy kudarc  
  a probléma kezdeti állapotából kiindulva inicializáld a keresési fát  
  loop do  
    if nincs kifejtendő csomópont then return kudarc  
    a stratégiának megfelelően válassz ki kifejtésre egy levélcsomópontot  
    if a csomópont célállapotot tartalmaz then return a hozzá tartozó megoldás  
    else fejtsd ki a csomópontot és az eredményül kapott csomópontokat, és add a keresési fához  
  end
```

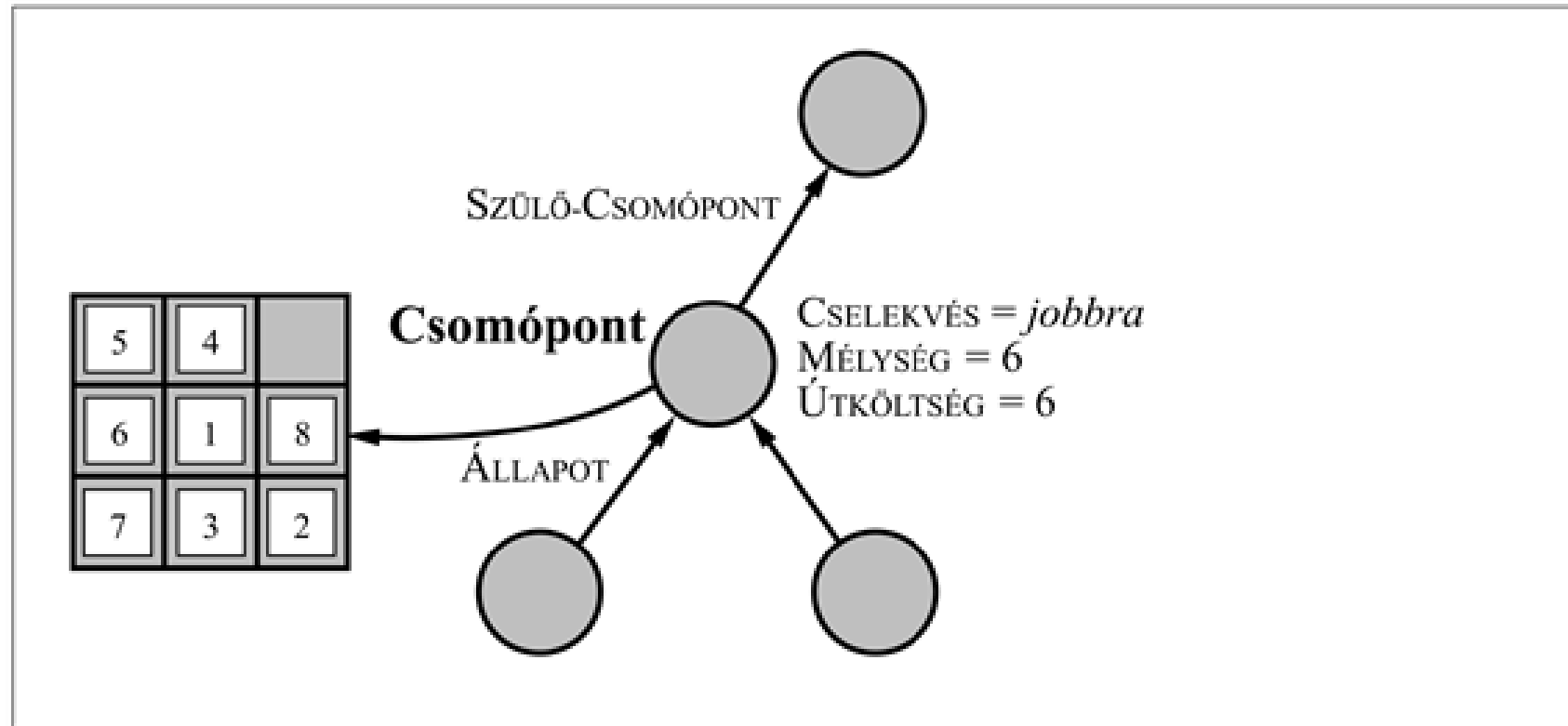
# Keresési fa

egy csomópontot egy öt komponensből álló adatszerkezettel reprezentálhatunk:

- ÁLLAPOT: az állapottérnek a csomóponthoz tartozó állapota
- SZÜLŐ-CSOMÓPONT: a keresési fa azon csomópontja, amely a kérdéses csomópontot generálta
- CSELEKVÉS: a csomópont szülő-csomópontjára alkalmazott cselekvés
- ÚT-KÖLTSÉG: a kezdeti állapotból a kérdéses csomópontig vezető út általában  $g(n)$ -nel jelölt költsége, ahogy ezt a szülőmutatók jelzik
- MÉLYSÉG: a kezdeti állapotból vezető út lépéseinek a száma

**perem (frontier):** lista, amely a leggenerált, kifejtésre váró csomópontokat nyilvántartja

# Keresési fa





# Algoritmus hatékonyságának mérése

- **Teljesség (completeness):** az algoritmus garantáltan megtalál egy megoldást, amennyiben létezik megoldás?
- **Optimalitás (optimality):** a stratégia megtalálja az optimális megoldást?
- **Időigény (time complexity):** mennyi ideig tart egy megoldás megtalálása?
- **Tárigény (space complexity):** a keresés elvégzéséhez mennyi memóriára van szükség?

# Keresési algoritmusok csoportosítása

- **Nem informált keresés:**

semmilyen információjuk nincs az állapotokról a probléma definíciójában megadott információn kívül.

Működésük során mást nem tehetnek, mint a következő állapotok generálása és a célállapot megkülönböztetése a nem célállapottól.

- **Informált vagy heurisztikus keresés:**

tudják, hogy az egyik közbülső állapot „ígéretesebb”, mint egy másik közbülső állapot

problémaspecifikus információkat is figyelembe veszünk

# Nem informált keresés

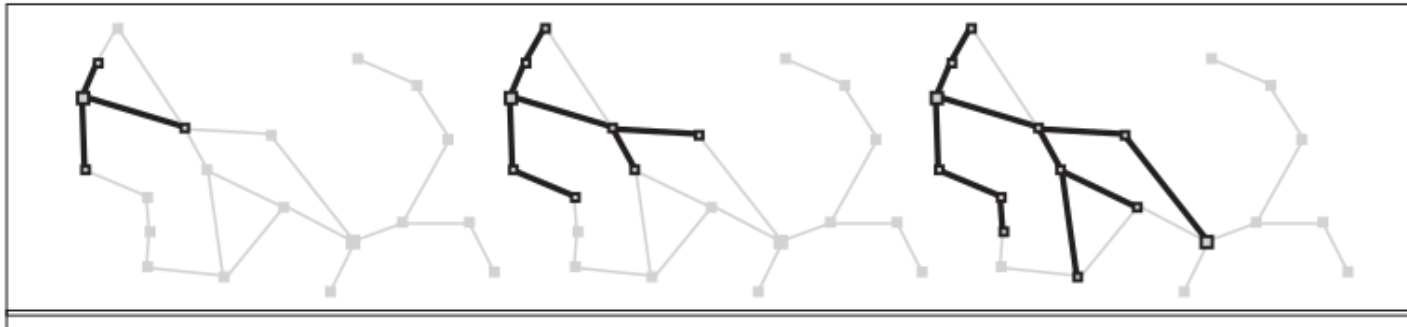
- Szélességi keresés
- Egyenletes költségű keresés
- Mélységi keresés
- Mélységkorlátozott keresés
- Iteratívan mélyülő mélységi keresés
- Kétirányú keresés

# Nem informált keresés

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```



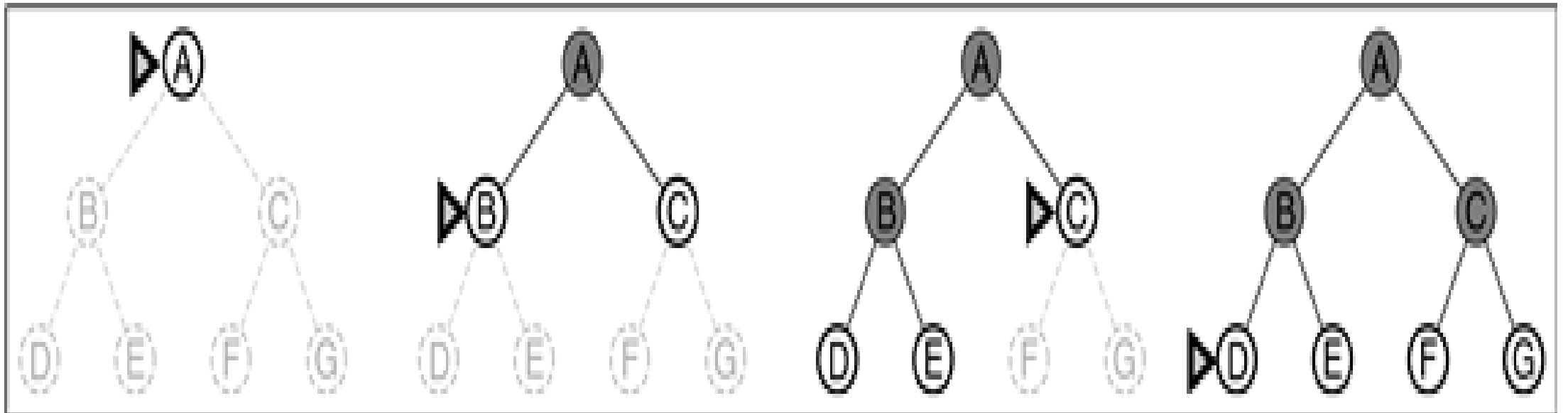
# Szélességi keresés

- a fában igyekszik először a gyökérhez közelebbi csúcsokat átvizsgálni, és csak azután folytatja a keresést a mélyebb csúcsokon.
- Megvalósítás: olyan üres peremmel, amely egy először-be-először-ki (first-in-first-out – FIFO) sor, biztosítva ezzel, hogy a korábban meglátogatott csomópontokat az algoritmus korábban fejti ki.

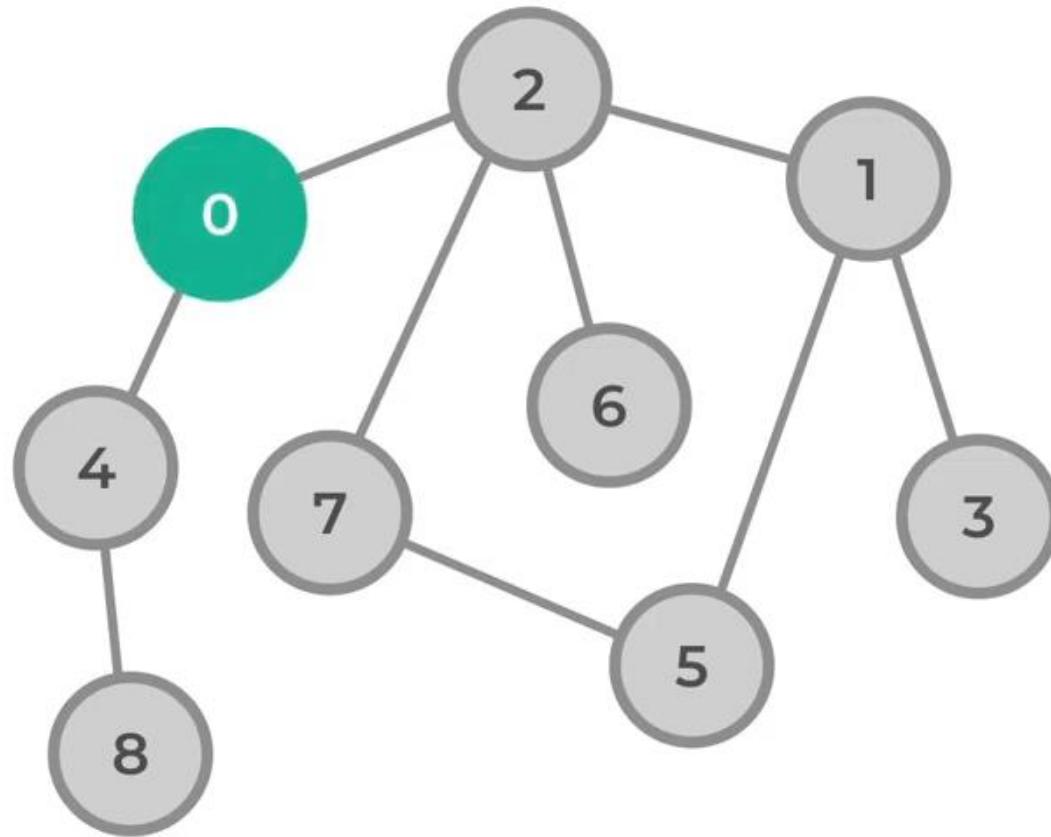
# Szélességi keresés gráfon

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

# Szélességi keresés



# Szélességi keresés gráfon





# Szélességi keresés tulajdonságai

- Teljes: Ha a legsekélyebb célcsomópont valamilyen véges  $d$  mélységben fekszik, a szélességi keresés eljut hozzá az összes nála sekélyebben fekvő csomópontot kifejtve
- Nem feltétlenül optimális: optimális, ha az útköltség a csomópont mélységének nem csökkenő függvénye (például ha minden cselekvésnek ugyanannyi a költsége).
- Magas tárigény:
  1. Legyen  $b$  az elágazási tényező, megoldás  $d$  mélységben található

Összes csomópont száma:  $b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$
- Minden generált csomópontot a memóriában el kell tárolni, mert vagy a perem eleme, vagy egy perembeli csomópont őse.

# Szélességi keresés fán: idő- és tárigény

$b = 10$  elágazási tényezővel rendelkező szélességi keresés

Ha másodpercenként 10 000 csomópontot generálunk, illetve egy csomópont tárolásához 1000 bájtra van szükség

Mélység	Csomópontok	Időigény	Tárigény
2	1100	0,11 másodperc	1 Mbájt
4	111100	11 másodperc	106 Mbájt
6	$10^7$	19 perc	10 Gbájt
8	$10^9$	31 óra	1 Tbájt
10	$10^{11}$	129 nap	101 Tbájt
12	$10^{13}$	35 év	10 Pbájt
14	$10^{15}$	3523 év	1 Ebájt

# Egyenletes költségű keresés

- A szélességi keresés optimális, ha minden lépés költsége azonos
- Egyenletes költségű keresés tetszőleges lépésköltség mellett optimális
- mindig a legkisebb útköltségű  $n$  csomópontot fejt ki először, nem pedig a legkisebb mélységű csomópontot.
- mindig végtelen hurokba kerül, ha egy csomópont kifejtése zérus költségű cselekvéshez és ugyanahhoz az állapothoz való visszatérést eredményez (például a *NoOp* cselekvés).
- Optimalitás elégséges feltétele: minden lépés költsége egy kis pozitív  $\epsilon$  konstansnál nagyobb, vagy azzal egyenlő
- Idő – és tárkomplexitás:  $O(b^{1+\lceil C^*/\epsilon \rceil})$
- Ha minden lépés egyenlő költségű  $\rightarrow$  szélességi keresés  $O(b^{d+1})$  idő- és tárkomplexitással

# Egyenletes költségű keresés

**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

*frontier*  $\leftarrow$  a priority queue ordered by PATH-COST, with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the lowest-cost node in *frontier* \*/

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

    add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

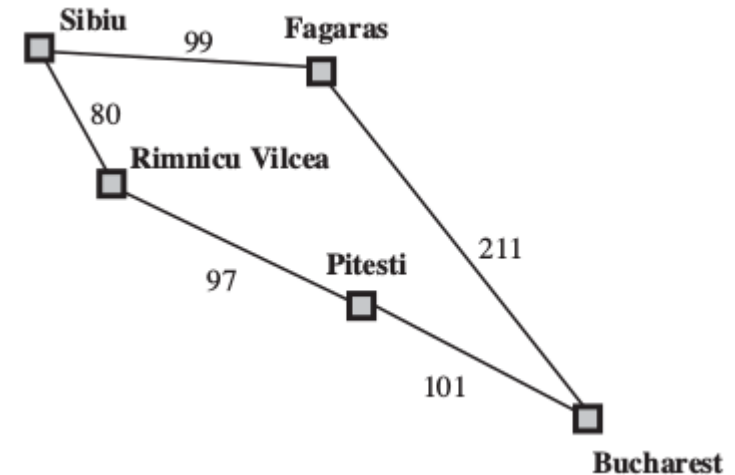
*child*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

*frontier*  $\leftarrow$  INSERT(*child*, *frontier*)

**else if** *child*.STATE is in *frontier* with higher PATH-COST **then**

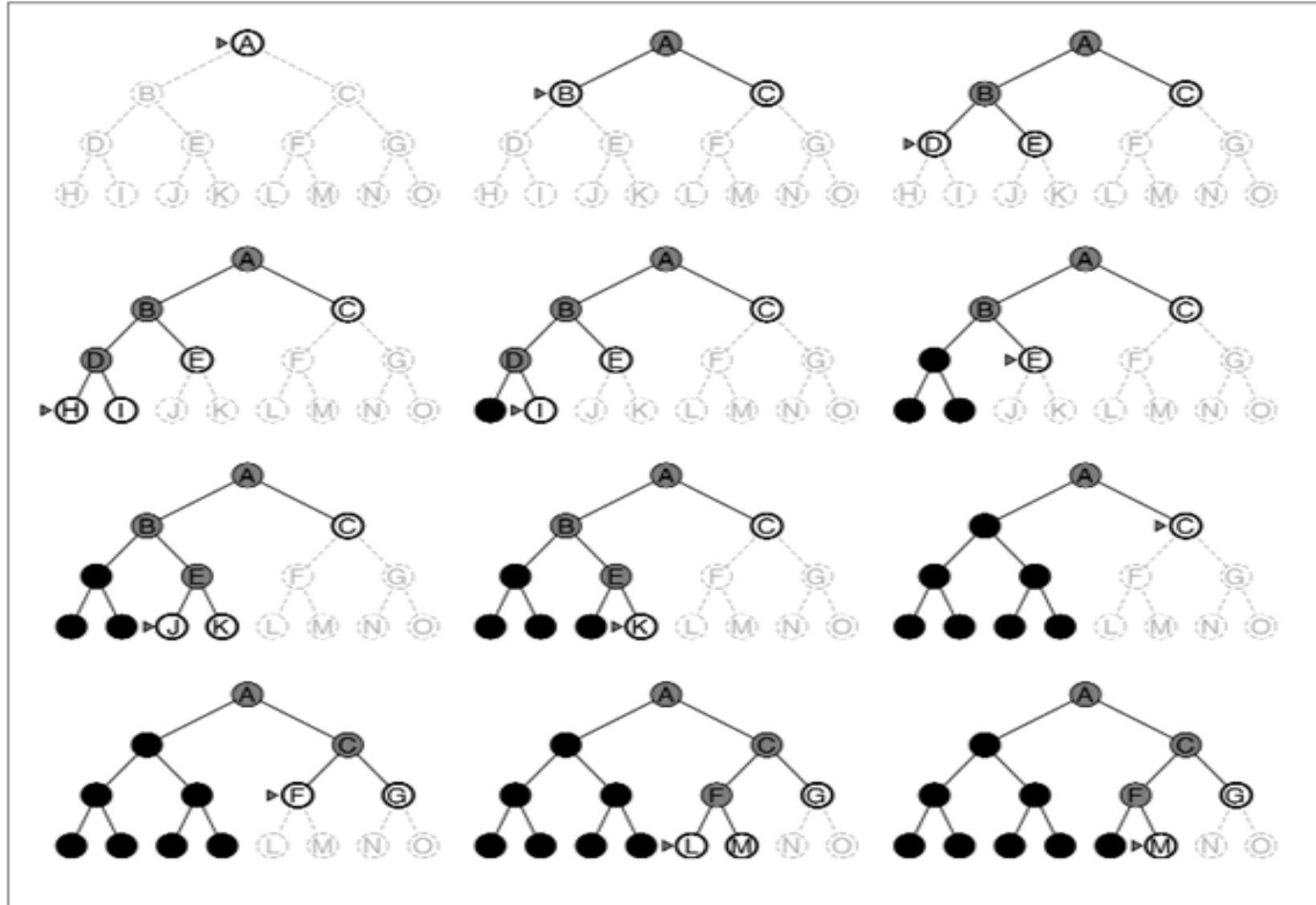
            replace that *frontier* node with *child*



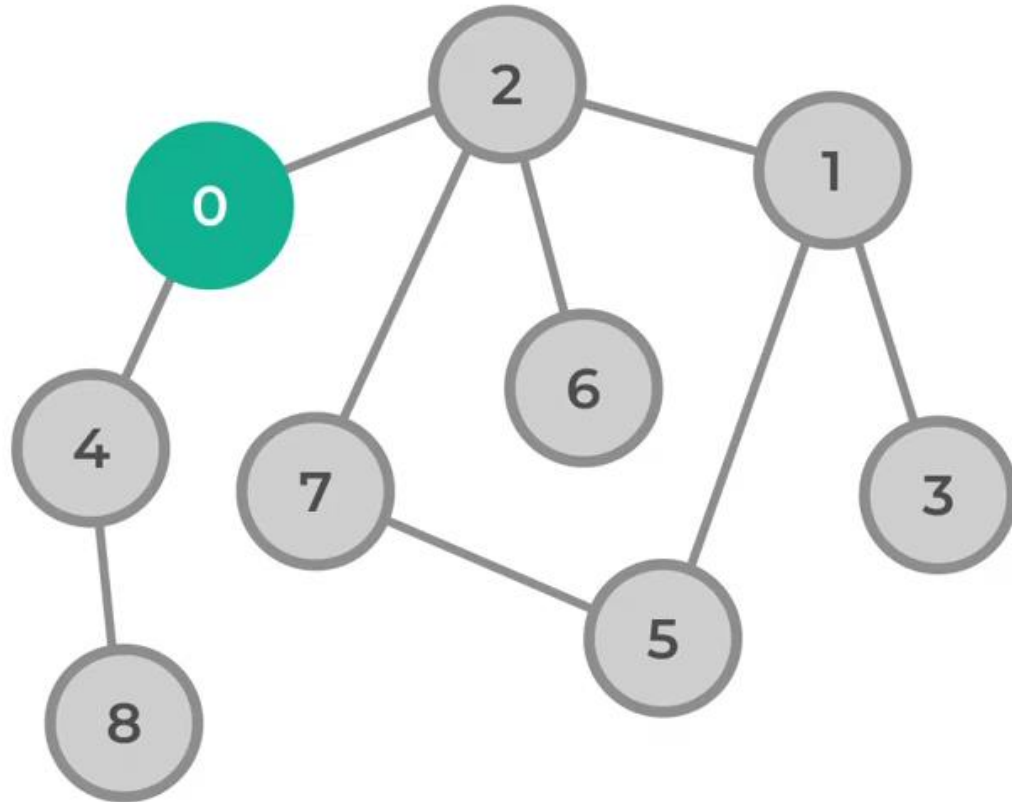
# Mélységi keresés

- A keresés azonnal a fa legmélyebb szintjére jut el, ahol a csomópontoknak már nincsenek követőik. Kifejtésüket követően kikerülnek a peremből és a keresés „visszalép” ahhoz a következő legmélyebben fekvő csomóponthoz, amelynek vannak még ki nem fejtett követői.

# Mélységi keresés



# Mélységi keresés gráfon



# Mélységi keresés tulajdonságai

- Nem teljes: Ha a bal oldali részfa korlátlanul mély lenne és nem tartalmazna megoldást, a mélységi keresés soha nem állna meg
- Nem optimális: egy rossz választással egy hosszú (akár végtelen) út mentén lefelé elakadhat, miközben például egy más döntés elvezetne a gyökérhez közeli megoldáshoz
- Alacsony tárigény: Csak egyetlen, a gyökércsomóponttól egy levélcsomópontig vezető utat kell tárolnia, kiegészítve az út minden egyes csomópontja melletti kifejtetlen csomópontokkal. Egy kifejtett csomópont el is hagyható a memóriából, feltéve, hogy az összes leszármazottja meg lett vizsgálva.



# Mélységi keresés tulajdonságai

- Tárigény:  $b \cdot m + 1$ ,

ahol  $b$ : elágazási tényező

$m$ : maximális mélység az állapottérben

A mélységi keresés **visszalépéses keresésnek (backtracking search)** nevezett változata még kevesebb memóriát használ. A visszalépéses keresés az összes követő helyett egyidejűleg csak egy követőt generál. Minden részben kifejtett csomópont emlékszik, melyik követője jön a legközelebb.

Ily módon csak  $O(m)$  memóriára van szükség.

# Mélységkorlátozott keresés

- Végtelen fák problémájának kiküszöbölésére
- az utak maximális mélységére egy  $\ell$  korlátot ad: Az  $\ell$  mélységben lévő csomópontokat úgy kezeli, mintha nem is lennének követőik.

Tulajdonságai:

- Nem teljes: ha  $\ell < d$  -t választunk, azaz, ha a legsekélyebb célcsomópont a mélységkorláton túl van
- Nem optimális: A mélységkorlátozott keresés  $\ell > d$  választással sem lesz optimális

# Iteratívan mélyülő mélységi keresés

- fokozatosan növeli a mélységkorlátot – legyen az először 0, majd 1, majd 2 stb. – amíg a célt meg nem találja. Ez akkor következik be, ha a mélységkorlát eléri a  $d$ -t, a legsekélyebben fekvő célcsomópont mélységét.
- Szélességi és mélységi keresés előnyös tulajdonságait ötvözi:
  - Teljes
  - Optimális, ha minden cselekvésnek azonos a költsége
  - Alacsony tárigény:  $O(bd)$

# Iteratívan mélyülő keresés

- Az iteratívan mélyülő keresésben a legmélyebb szinten ( $d$  mélység) található csomópontokat csak egyszer fejtjük ki, egy szinttel feljebb kétszer stb. egészen a gyökér gyerekeiig, amelyeket  $d$ -szer fejtünk ki.
- konkretizálva, például  $b = 10$  és  $d = 5$  esetén ezek a számok:

$$Cs(IMK) = 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,450$$

$$Cs(SZK) = 10 + 100 + 1000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$$

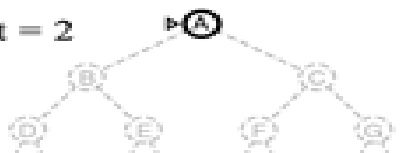
*Általánosságban nagy keresési térrel rendelkező problémák esetén és ha a megoldás mélysége nem ismert, a nem informált módszerek köréből az iteratívan mélyülő keresés a javasolt.*

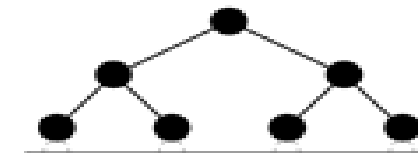
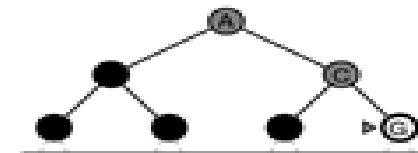
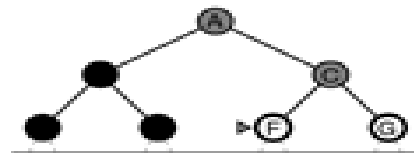
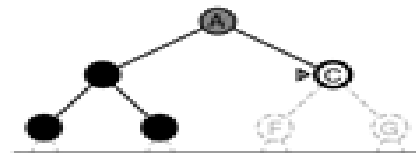
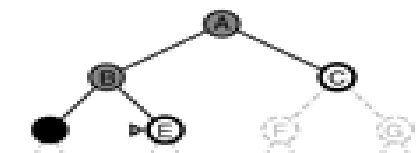
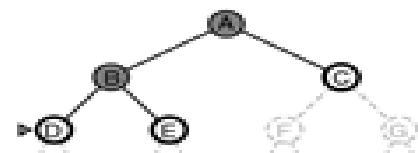
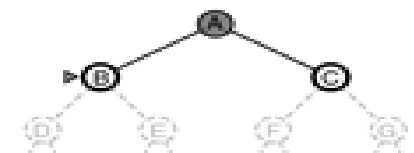
Korlát = 0 

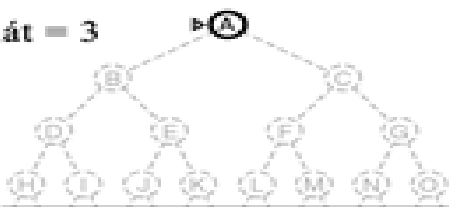


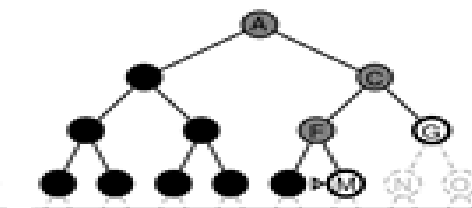
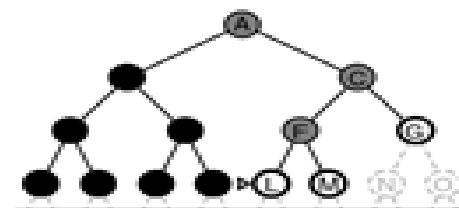
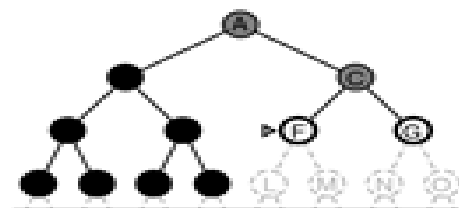
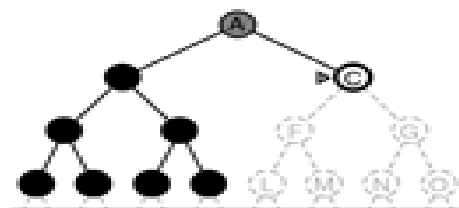
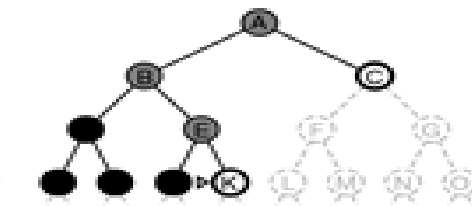
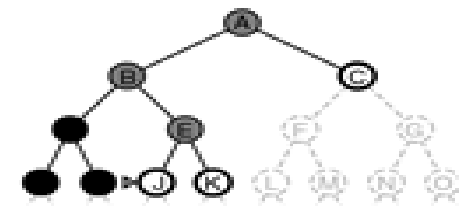
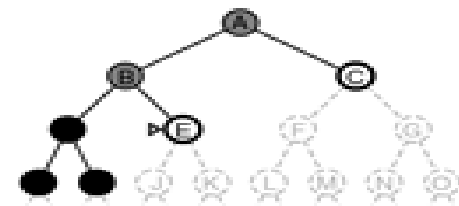
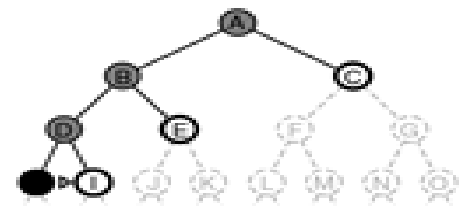
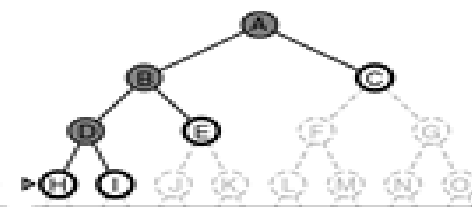
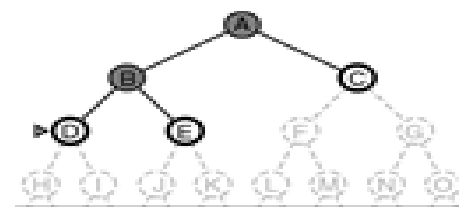
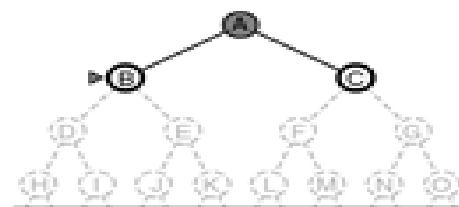
Korlát = 1 



Korlát = 2 



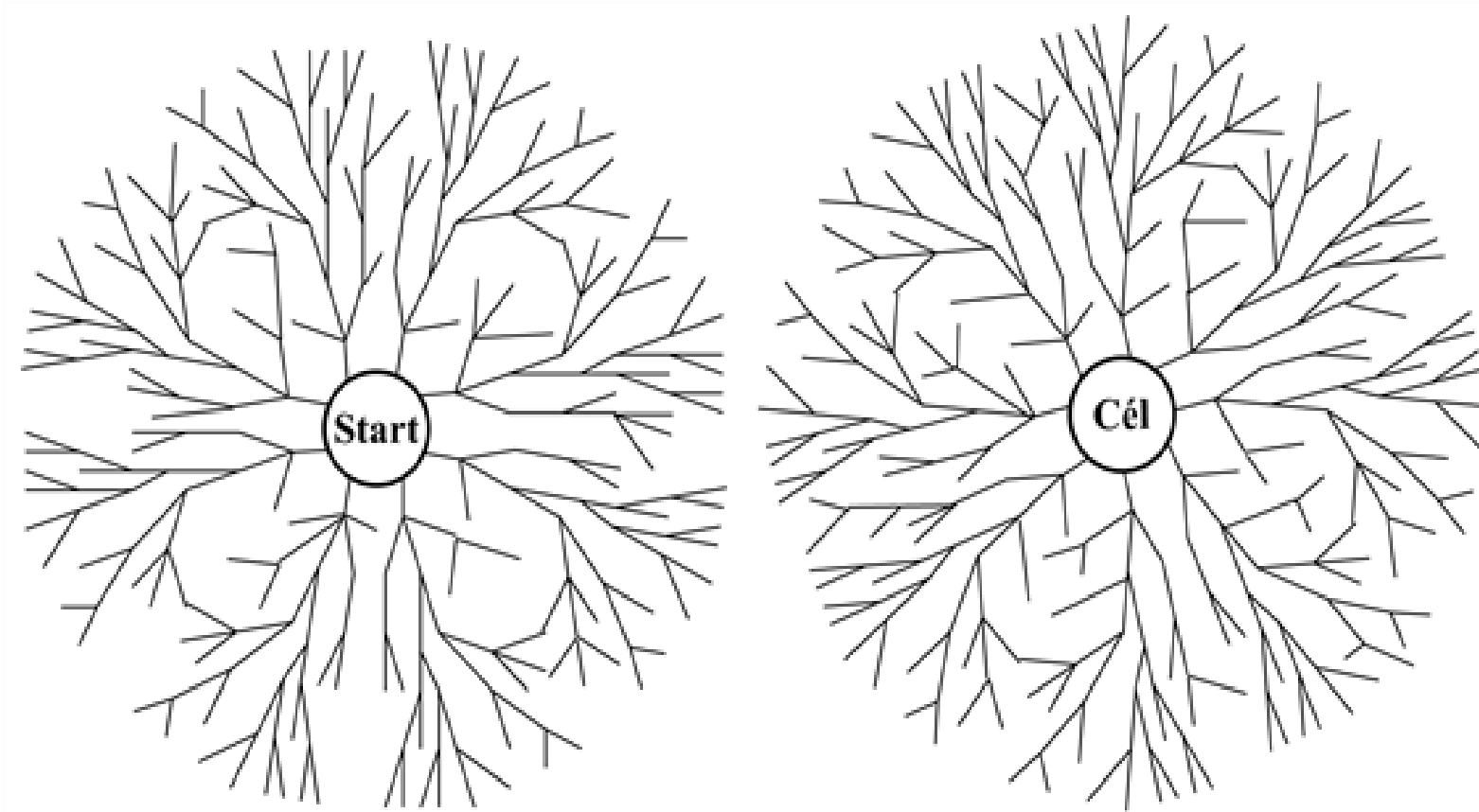
Korlát = 3 



# Kétirányú keresés

- egyszerre el lehet indítani egy keresést előre felé a kiinduló állapotból, illetve hátrafelé a célállapotból, és a keresés akkor fejeződik be, ha a két keresés valahol találkozik.
- A kétirányú keresést úgy implementálják, hogy az egyik vagy mindkét keresés egy csomópont kifejtése előtt megvizsgálja, hogy az nem része-e a másik keresési fa peremének. Ha igen, megvan a cél.
- Ha a probléma például  $d = 6$  megoldás mélységű, a két keresés a legrosszabb esetben akkor találkozik, ha mindegyik algoritmus a 3-as mélységben egy csomópont kivételével minden csomópontot kifejtett.
- A  $b = 10$  esetén ez 2220 csomópont generálását jelenti a standard szélességi keresés által generált 1111110 csomóponthoz képest. Annak ellenőrzését, hogy egy csomópont a másik keresési fához tartozik-e, egy hash-táblával konstans időben meg lehet oldani. A kétirányú keresés idő- és tárkomplexitása így  $O(b^{d/2})$ .

# Kétirányú keresés



# Nem informált keresések összehasonlítása

Kritérium	Szélességi	Egyenletes költségű	Mélységi	Mélységi korlátozott	Iteratívan	Kétirányú (amennyiben alkalmazható)
Teljes?	Igen <sup>a</sup>	Igen <sup>a, b</sup>	Nem	Nem	Igen <sup>a</sup>	Igen <sup>a, d</sup>
Időigény	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Tárigény	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimális?	Igen <sup>c</sup>	Igen	Nem	Nem	Igen <sup>c</sup>	Igen <sup>c, d</sup>



# Informált keresés

- **legjobbat-először keresés (best-first search):**
- az általános FA-KERESÉS vagy GRÁF-KERESÉS algoritmusok olyan speciális esete, ahol egy csomópont kifejtésre való kiválasztása egy  $f(n)$  kiértékelő függvénytől (evaluation function) függ.
- egy prioritási sor segítségével implementálható, ami egy olyan adatstruktúra, mely a peremet a növekvő  $f$ -értékek szerint rendezi.
- egy keresési algoritmus család, amelynek az elemeit az eltérő kiértékelő függvények különböztetik meg.

# Informált keresés

- Heurisztikus függvény: légvonalbeli távolságok Bukaresttől

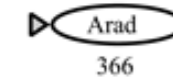
<b>Arad</b>	366	<b>Mehadia</b>	241
<b>Bukarest</b>	0	<b>Neamt</b>	234
<b>Craiova</b>	160	<b>Nagyvárad</b>	380
<b>Dobreta</b>	242	<b>Pitești</b>	100
<b>Eforie</b>	161	<b>Rimniu Vilcea</b>	193
<b>Fogaras</b>	176	<b>Nagyszeben</b>	253
<b>Giurgiu</b>	77	<b>Temesvár</b>	329
<b>Hirsova</b>	151	<b>Csalános</b>	80
<b>Iasi</b>	226	<b>Vaslui</b>	199
<b>Lugos</b>	244	<b>Nagyzerénd</b>	374

# mohó legjobbat-először keresés (greedy best-first search)

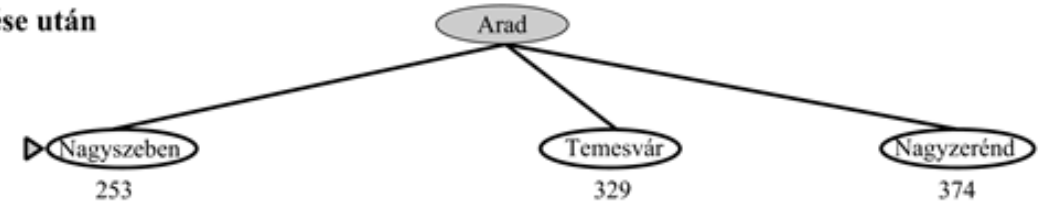
- azt a csomópontot fejt ki a következő lépésben, amelyiknek az állapotát a legközelebbinek ítéli a célállapothoz, abból kiindulva, hogy így gyorsan megtalálja a megoldást.
- Tulajdonságai:
  - Nem teljes: elindulhat egy végtelen úton és soha sem tér vissza újabb lehetőségeket kipróbálni
  - Nem optimális
  - Worst-case idő – és tárigény:  $O(b^m)$ , ahol  $m$  a keresési tér maximális mélysége

Feladat: Aradból eljutni  
Bukarestbe

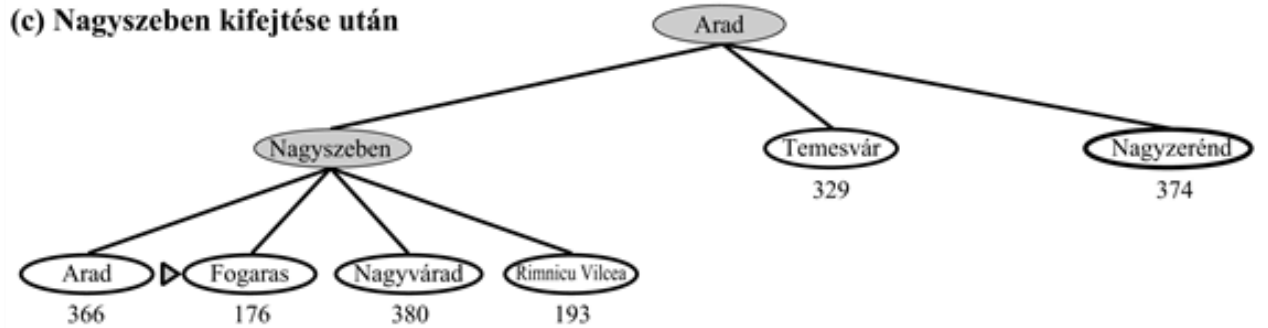
(a) Kezdeti állapot



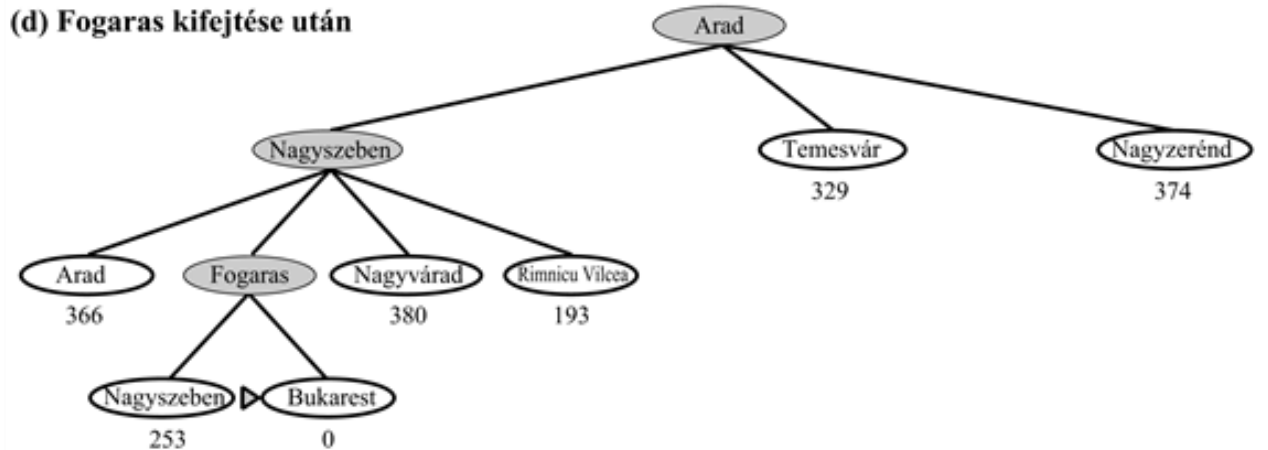
(b) Arad kifejtése után



(c) Nagyszeben kifejtése után



(d) Fogaras kifejtése után



# A\* keresés

- legjobbat-először keresés leginkább ismert változata
- A csomópontokat úgy értékeli ki, hogy összekombinálja  $g(n)$  értékét – az aktuális csomópontig megtett út költsége – és  $h(n)$  értékét – vagyis az adott csomóponttól a célhoz vezető út költségének becslőjét:

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

- $f(n)$  = a legolcsóbb, az  $n$  csomóponton keresztül vezető megoldás becsült költsége
- A fa-keresés optimális, ha  $h(n)$  elfogadható heurisztika soha nem becsüli felül a cél eléréséhez szükséges költséget

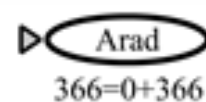
# Heurisztikus függvények konzisztenciája

- Gárfkeresés esetén az optimalitás feltétele: A  $h(n)$  heurisztikus függvény konzisztens, ha minden  $n$  csomópontra és annak egy tetszőleges  $a$  cselekvéssel generált minden  $n'$  utódcsomópontjára az  $n$  csomóponttól elért cél becsült költsége nem kisebb, mint az  $n'$ -be kerülés lépésköltsége és az  $n'$  csomóponttól elért cél becsült költsége:

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

általános háromszög egyenlőtlenség (triangle inequality) egy formája, amely azt fejezi ki, hogy egy háromszög egy oldala sem lehet hosszabb, mint a két másik oldal összege. Itt a háromszöget az  $n$ , az  $n'$  és az  $n$ -hez legközelebbi cél határozza meg.

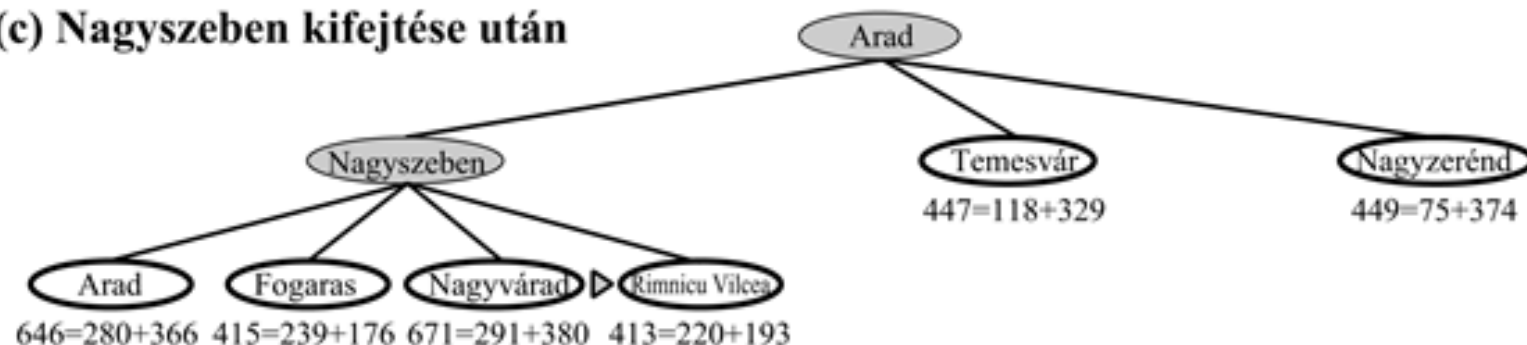
**(a) Kezdeti állapot**



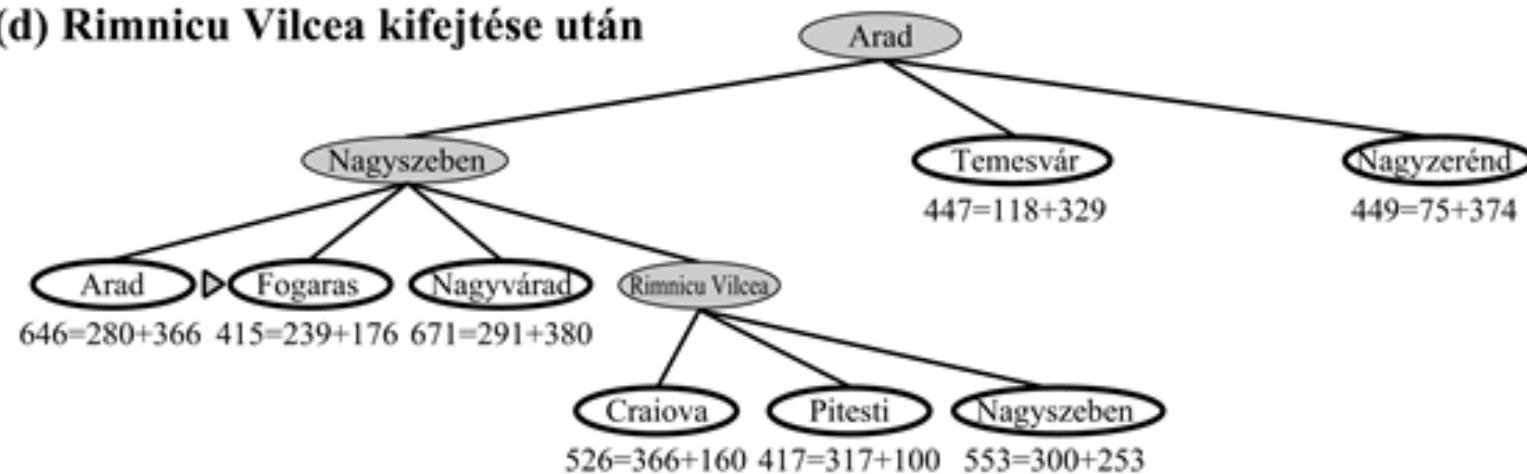
**(b) Arad kifejtése után**



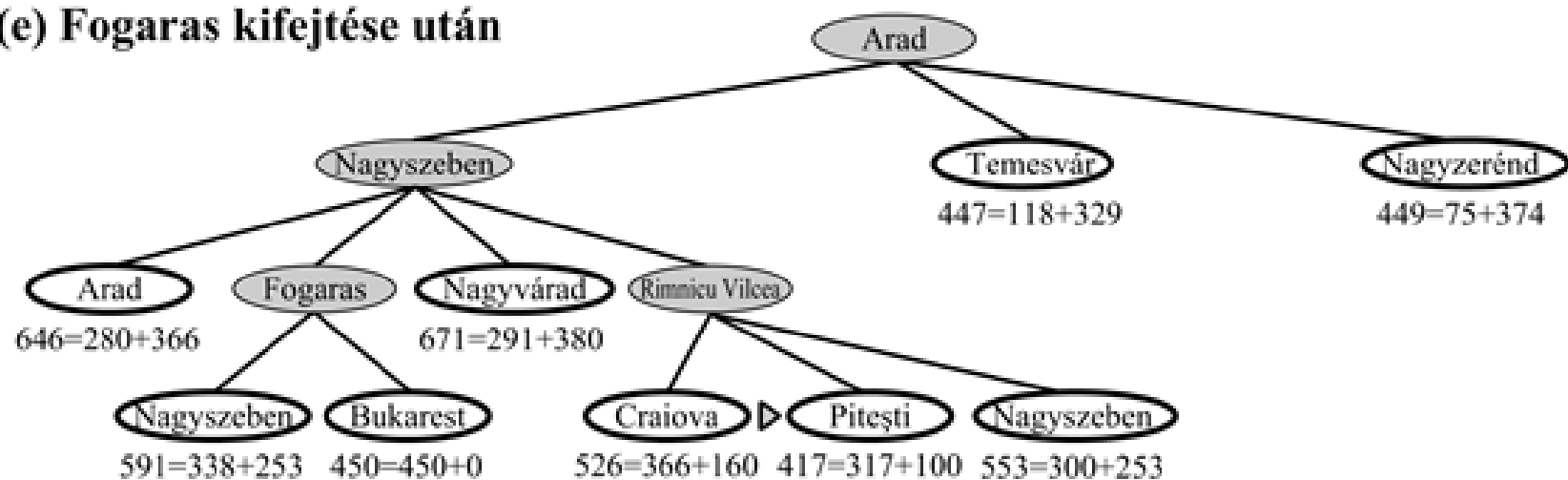
**(c) Nagyszeben kifejtése után**



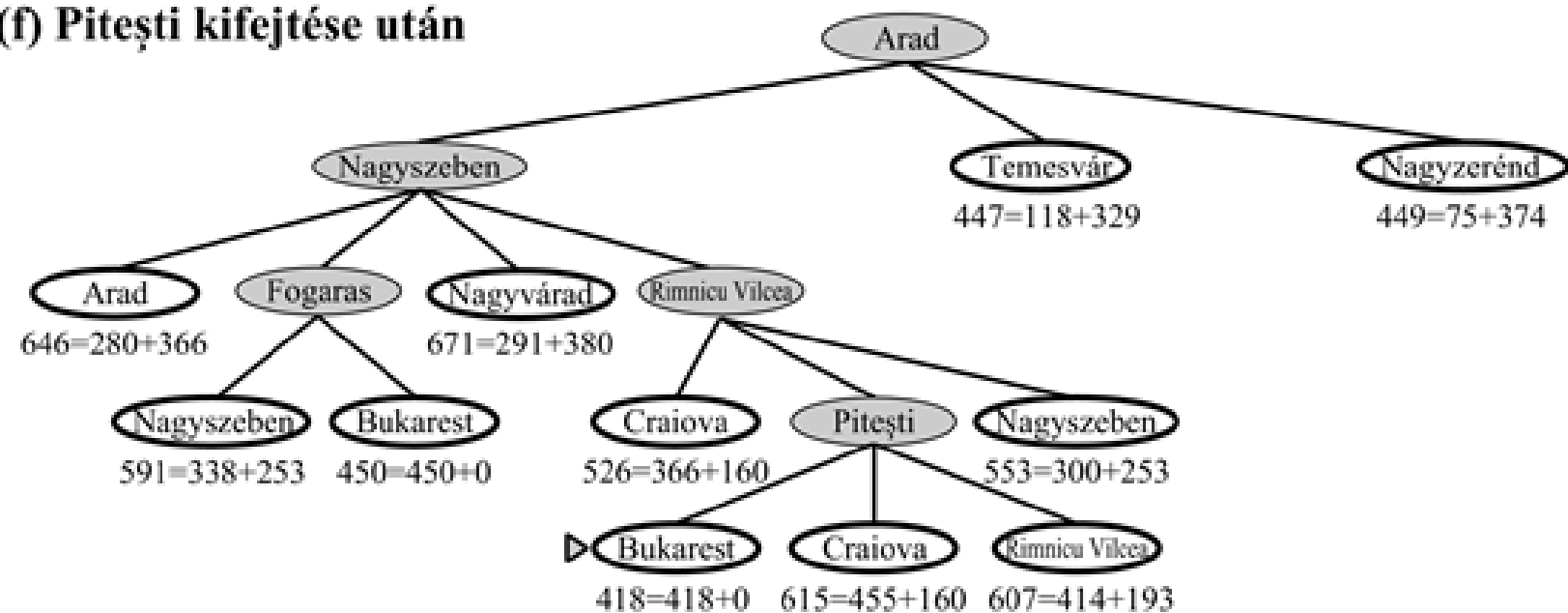
**(d) Rimnicu Vilcea kifejtése után**



**(e) Fogaras kifejtése után**



**(f) Pitești kifejtése után**

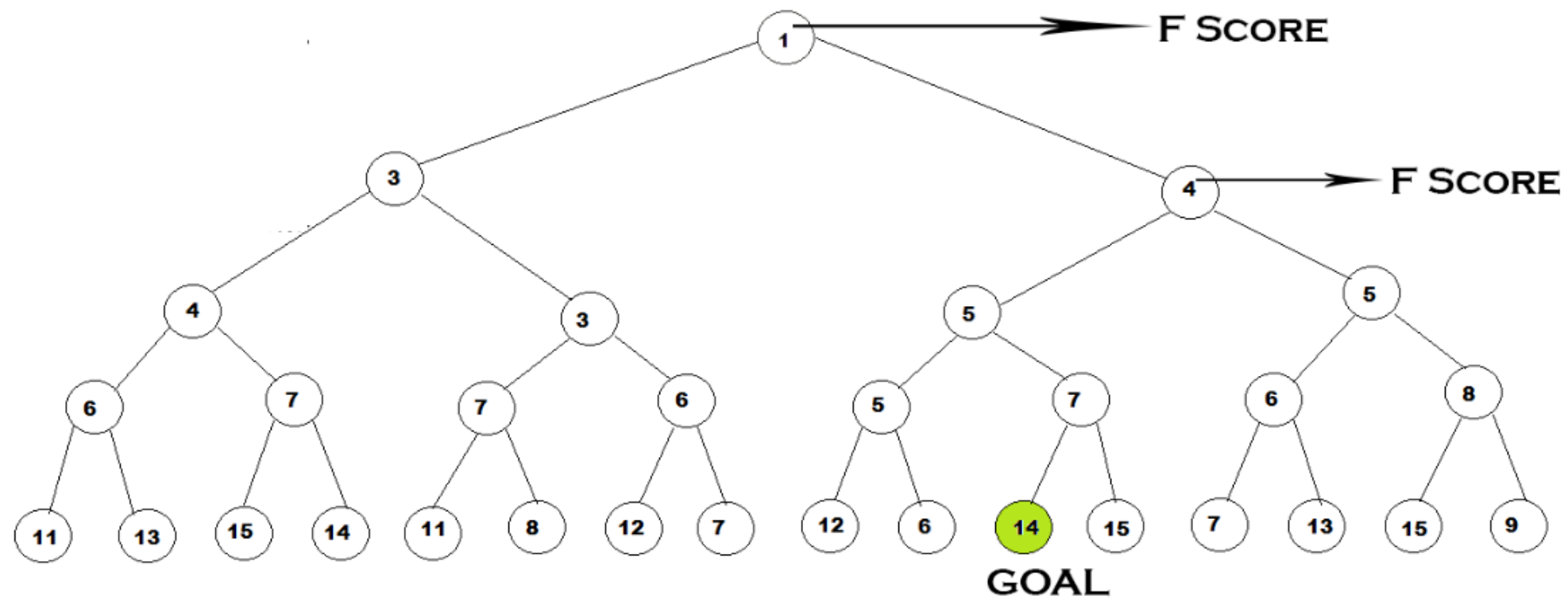




# Memóriakorlátozott heurisztikus keresés

- A\* algoritmus worst-case idő- és tárkomplexitás:  $O(b^d)$  Exponenciális
- Az A\* memóriaigényének mérséklésére a legegyszerűbb módszer az iteratívan mélyülő algoritmus adaptálása heurisztikus keresés környezetre. iteratívan mélyülő A\* algoritmus – IMA\* – (iterative deepening A\*, IDA\*).
- Az IMA\* és a közönséges iteratívan mélyülő algoritmus közötti fő különbség az, hogy a vágási mechanizmus nem a mélységen, hanem az  $f$  költségén ( $g + h$ ) alapul. Ezáltal minden egyes iterációban a vágási érték az a legkisebb  $f$  költség, ami az előbbi iterációban használt vágási értéknél nagyobb.
- Az IMA\* praktikus megoldás számos olyan probléma esetén, ahol egységnyi a lépésköltség
- IMA\* worst-case tárkomplexitás: a megoldás mélységével lineáris
- IMA\* teljes, és optimális keresés

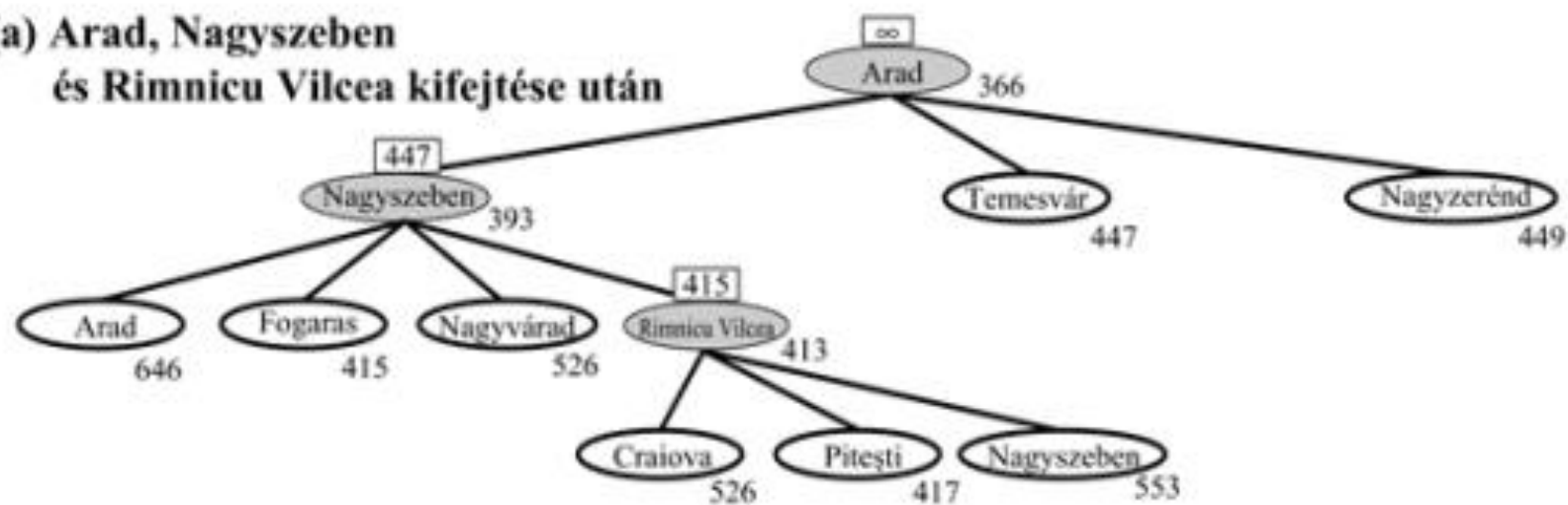
# IMA\*



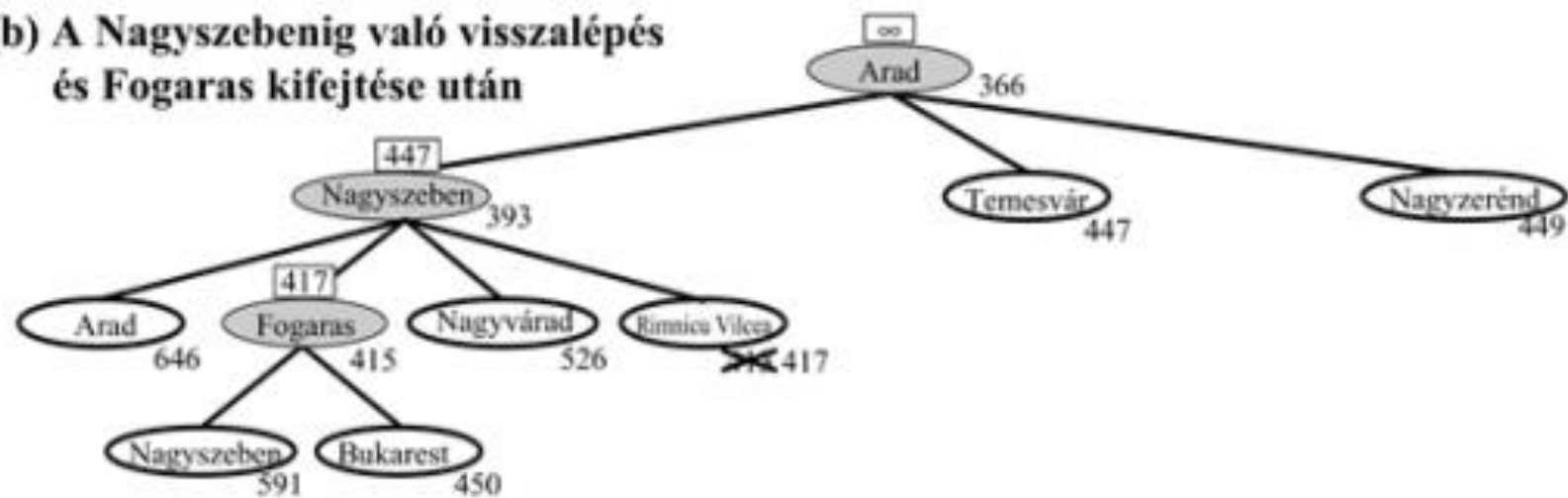
# Rekurzív legjobbat-először keresés

- A struktúrája hasonlít a rekurzív mélységi keresésre, azonban ahelyett, hogy az algoritmus egy utat a végtelenségig folytatna az aktuális pálya mentén, figyeli az aktuális csomóponthoz az elődeitől vezető eddigi legjobb alternatív út  $f$ -értékét. Ha az aktuális csomópont ezt az értéket túlhaladja, a rekurzió az alternatív útra lép vissza. Ahogy a rekurzió visszalép, az RLEK minden csomópont  $f$ -értékét a pálya mentén a gyerekeinek legjobb  $f$ -értékével helyettesíti.
- Tárkomplexitása  $O(bd)$

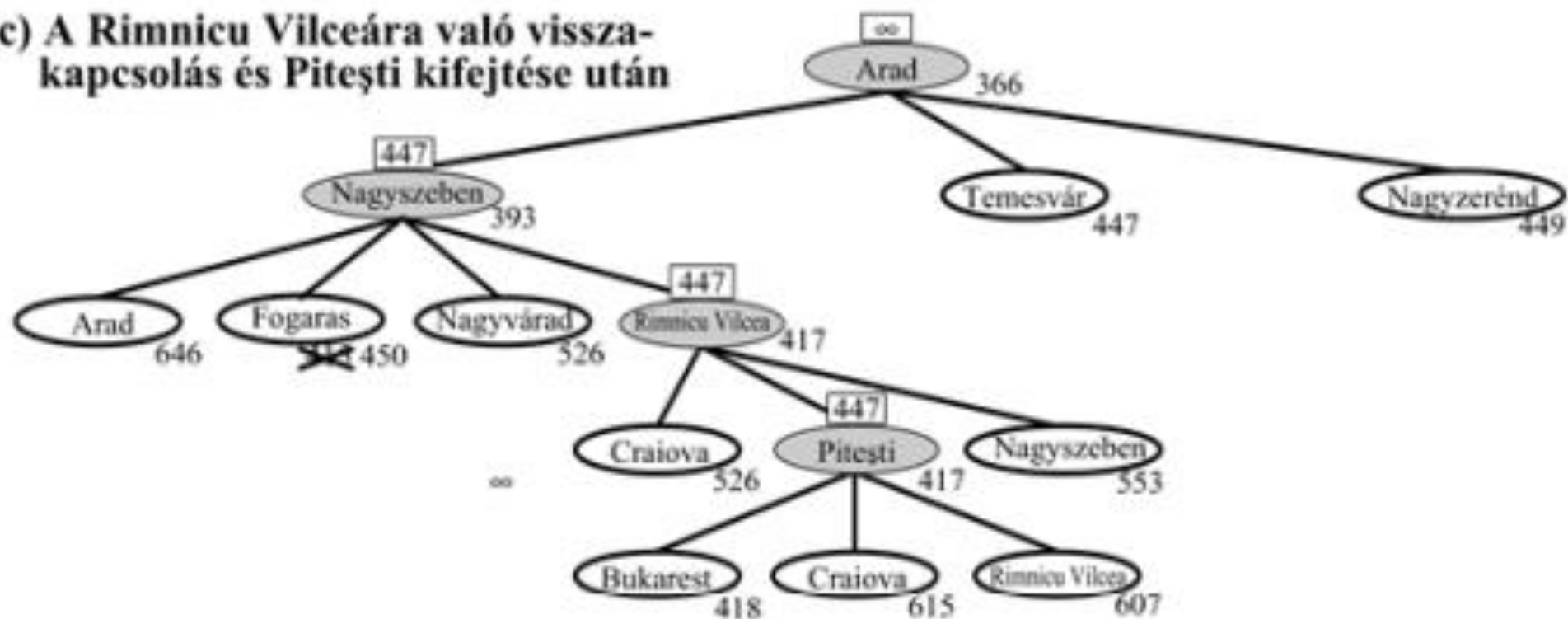
**(a) Arad, Nagyszeben és Rimnicu Vilcea kifejtése után**



**(b) A Nagyszebenig való visszalépés és Fogaras kifejtése után**



(c) A Rimnicu Vilceára való vissza-  
kapcsolás és Pitești kifejtése után



# Lokális keresési algoritmusok

- Számos problémában azonban a célhoz vezető út érdektelen.

Pl: 8-királynő probléma

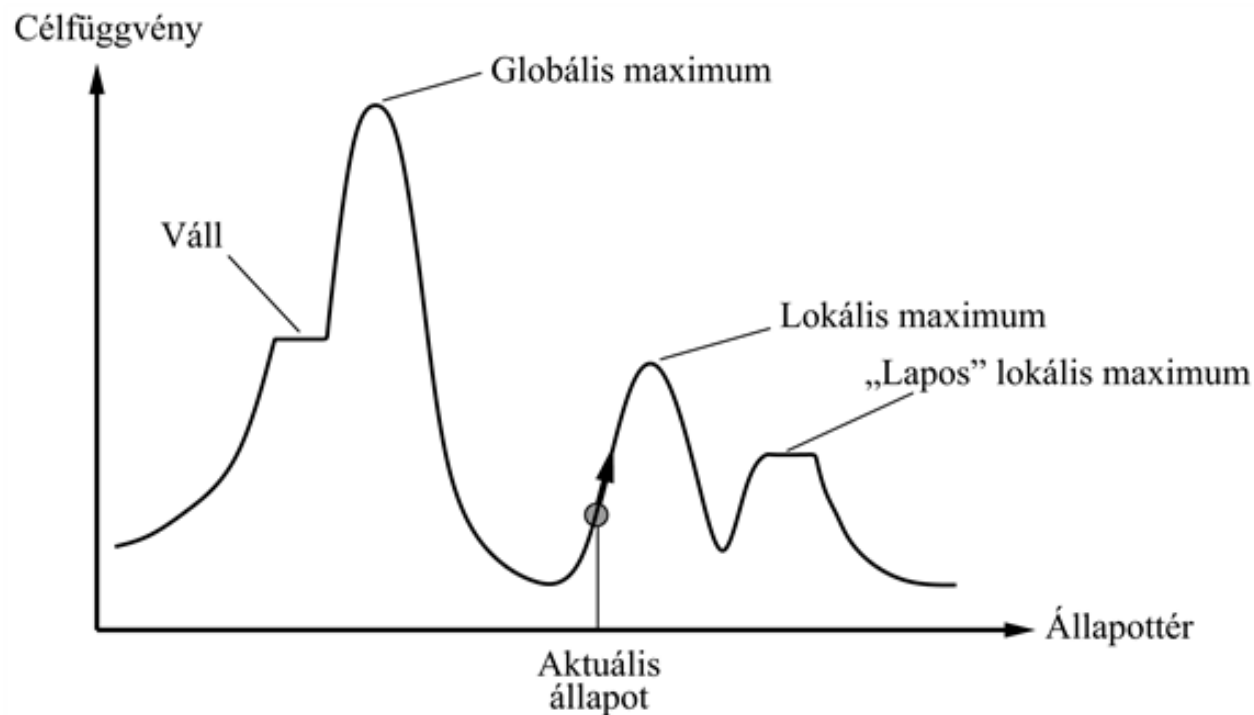
- **lokális keresési algoritmusok (local search):**
  - csak egy aktuális állapotot (current state) vesznek figyelembe (a többszörös utak helyett) és általában csak ennek az állapotnak a szomszédjaira lépnek tovább
  - A keresés által követett utat tipikusan nem is tárolják el

# Lokális keresési algoritmusok

- 2 alapvető előnyük van:
  - igen kevés – általában konstans mennyiségű – memóriát használnak
  - sokszor nagy vagy végtelen (folytonos) keresési térben elfogadható megoldást produkálnak ott, ahol a szisztematikus algoritmusok alkalmatlanok lennének a nagy tár- és időigényük miatt

# Állapotfelszín

- Célfüggvény esetén: cél a globális maximum megtalálása
- Költségfüggvény esetén: cél a globális minimum megtalálása





# Hegymászó algoritmus (hill-climbing)

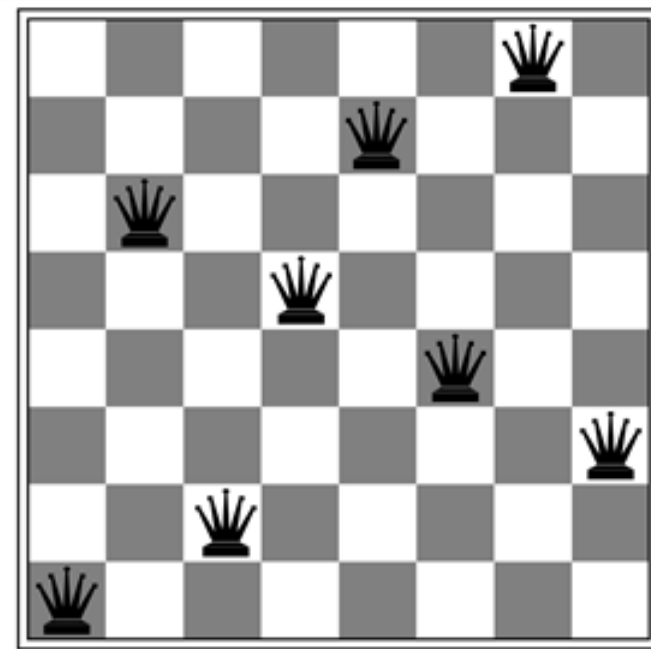
- A keresés egyszerűen csak egy ciklus, ami mindig javuló értékek felé – azaz felfelé – lép
- Minden lépésben az aktuális csomópontot a legjobb szomszédjával cseréli le
- Az algoritmus megáll, amikor felér a csúcsra, ahol nincsenek már magasabb értékű szomszédjai
- nem tart nyilván keresési fát, ezért a csomópontot leíró adatszerkezetnek csak az állapotot és a célfüggvény értékét kell nyilvántartania
- nem néz előre az aktuális állapotot közvetlenül követő szomszédokon túl

# Példa: 8-királynő probléma

- A  $h$  heurisztikus függvény a közvetlenül vagy közvetett módon egymást támadó királynőpárok száma.
- Következő állapot: egy királynőt ugyanabban az oszlopban egy másik mezőre mozgatunk → minden állapotnak 56 követője van

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♚	13	16	13	16
♚	14	17	15	♚	14	16	16
17	♚	16	18	15	♚	15	♚
18	14	♚	15	15	14	♚	16
14	14	13	17	12	14	12	18

(a)



(b)

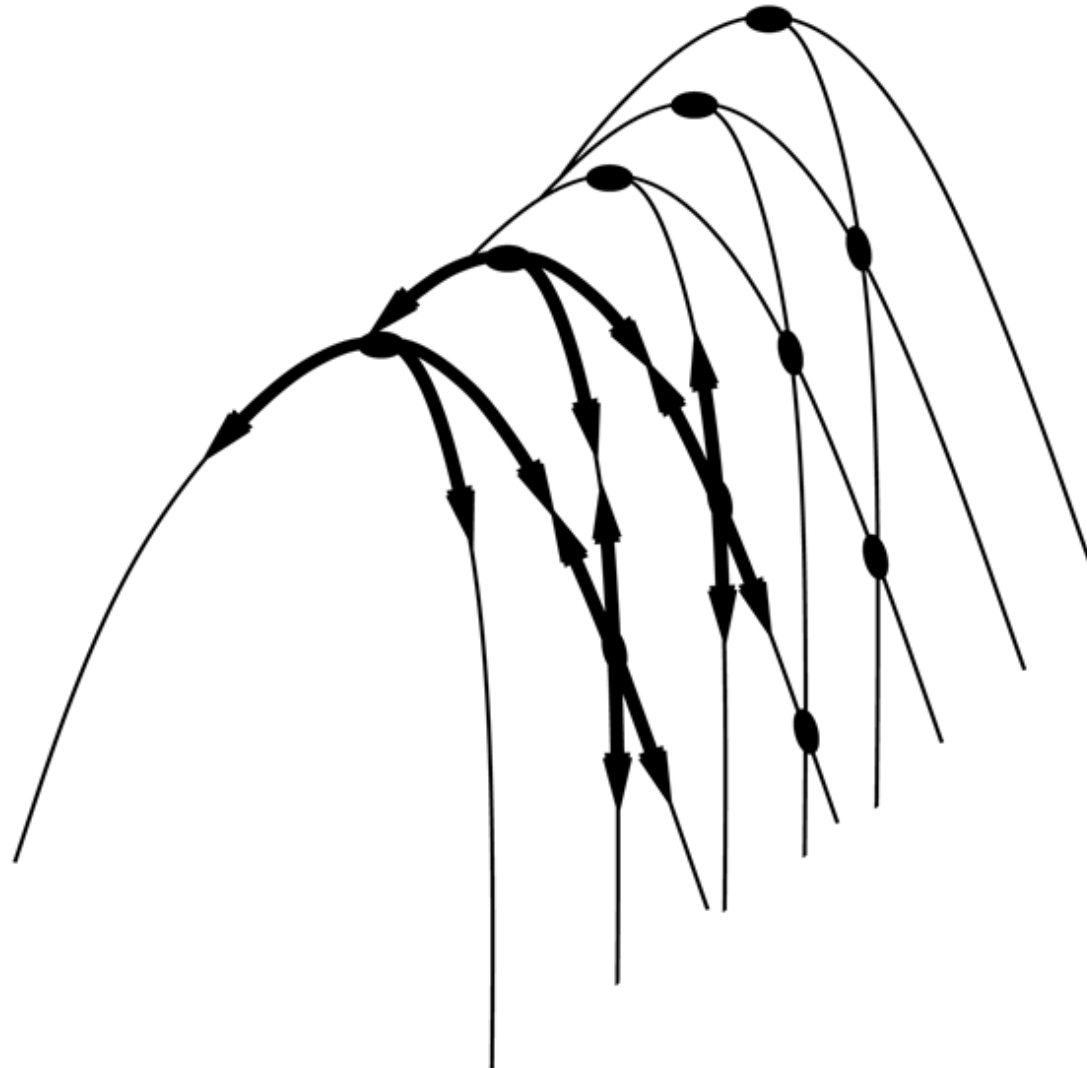
# Hegymászó algoritmus

- hegymászás gyakran megakad az alábbi problémák miatt:
    - **Lokális maximumok:** egy csúcs, amely minden szomszédjánál magasabb, de a globális maximumnál alacsonyabb
    - **Hegygerincek (ridges):** egy olyan lokális maximum sorozatot eredményez, ahol egy mohó algoritmusnak igen nehéz navigálnia maximumok nincsenek közvetlen módon egymással összekapcsolva
- Minden egyes lokális maximumból az összes lehetséges cselekvés a lejtőn lefelé mutat.

# Hegymászó algoritmus

- **Fennsík (plateaux):** fennsík az állapottérnek egy olyan területe, ahol a kiértékelő függvény gyakorlatilag lapos.  
Lehet ez egy lapos lokális maximum, amelyből nincs tovább felfelé, de lehet egy **váll (shoulder)**, ahonnan még lehetséges az előrehaladás

# Hegygerinc



# Hegymászó algoritmus

- **sztochasztikus hegymászó keresés (stochastic hill climbing):** a felfelé mutató irányokból véletlen módon választ
- **elsőnek-választott hegymászó algoritmus (first-choice hill climbing)** sztochasztikus hegymászó keresést használ
  - a követőket véletlen módon addig generálva, amíg az az aktuális állapotnál nem lesz jobb
  - jó stratégia lehet, ha egy állapotnak sok követője van
- **véletlen újraindítású hegymászás (random-restart hill-climbing):**  
Véletlenül generált kiinduló állapotokból hegymászó keresést végez, amíg célba nem ér → teljes, de nem hatékony

# Szimulált hűtés

- Hatékony és teljes algoritmus
- legjobb lépés megtétele helyett azonban egy véletlen lépést tesz.
- Ha a lépés javítja a helyzetet, akkor az mindig végrehajtásra kerül.
- Ellenkező esetben az algoritmus a lépést csak valamilyen 1-nél kisebb valószínűséggel teszi meg. A valószínűség exponenciálisan csökken a lépés „rosszaságával” – azzal a  $\Delta E$  mennyiséggel, amivel a kiértékelő függvény értéke romlott. A valószínűség a  $T$  „hőmérséklet” csökkenésével is csökken. A „rossz” lépések az indulásnál  $T$  magasabb értékeinél valószínűbbek,  $T$  csökkenésével egyre valószínűtlenebbé válnak

# Szimulált hűtés

```
function SZIMULÁLT-LEHŰTÉS(probléma, lehűtési terv) returns egy megoldási állapot
  inputs: probléma, egy probléma
           lehűtési terv, egy leképzés időről „hőmérsékletre”
  local variables: aktuális, egy csomópont
                     következő, egy csomópont
                     T, egy „hőmérséklet”, ami a lefelé lépések valószínűségét szabályozza

  aktuális  $\leftarrow$  CSOMÓPONTOT-LÉTREHOZ(KIINDULÓ-ÁLLAPOT[probléma])
  for  $t \leftarrow 1$  to  $\infty$  do
    T  $\leftarrow$  lehűtési terv[ $t$ ]
    if  $T = 0$  then return aktuális
    következő  $\leftarrow$  az aktuális egy véletlenszerűen kiválasztott követő csomópontja
     $\Delta E = \text{ÉRTÉK}[\textit{következő}] - \text{ÉRTÉK}[\textit{aktuális}]$ 
    if  $\Delta E > 0$  then aktuális  $\leftarrow$  következő
    else aktuális  $\leftarrow$  következő csak  $e^{\Delta E/T}$  valószínűséggel
```



# lokális nyaláb keresés algoritmus

- nem egy, hanem  $k$  állapotot követ nyomon
- Az algoritmus  $k$  véletlen módon generált állapottal indul. Minden lépésben a  $k$  állapot mindegyikének összes követőit kifejti.
- Ha ezek valamelyike egy cél, az algoritmus leáll. Egyébként a teljes listából kiválasztja a legjobb  $k$  követőt, és ezt az eljárást ismétli
- Hátrány: állapotok gyorsan koncentrálódhatnak a tér egy kicsi részében →
- **sztochasztikus nyaláb keresés (stochastic beam search)**:  $k$  legjobb követő megválasztása helyett az algoritmus a  $k$  követőt véletlen módon választja ki, ahol egy adott követő kiválasztásának valószínűsége az állapot értékének növekvő függvénye.

# Online kereső ágensek

- Eddig bemutatott algoritmusok: kiszámítanak egy teljes megoldás, majd érzékelések nélkül végrehajtják
- Online kereső ágensek: a számítás és a végrehajtás átlapolódik (interleaving) → először végrehajtanak egy cselekvést, majd megfigyelik a környezetüket és kiszámítják a következő cselekvést
  - Dinamikus, szemidinamikus környezetekben
  - Sztochasztikus környezetekben
  - Felfedezési problémák esetén: az állapotok és a cselekvések ismeretlenek az ágens számára

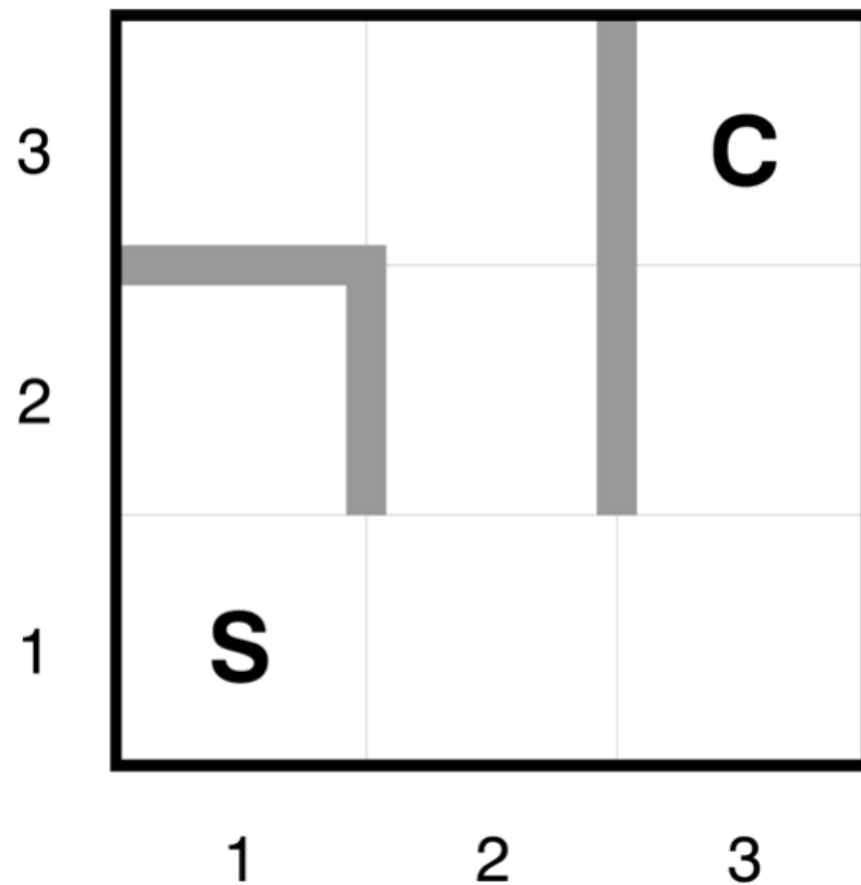
# Online kereső ágens

- Az ágens az alábbiakat tudja:
  - CSELEKVÉSEK ( $s$ ), amely az  $s$  állapotban engedélyezett cselekvések listáját adja vissza
  - A lépésköltség  $c(s, a, s')$
  - CÉL-TESZT ( $s$ )
- Ágens nem képes egy állapot követőit másképpen elérni, mint úgy, hogy az adott állapotban az összes cselekvését kipróbálja
- Egy online kereső ágens minden cselekvés után érzékeli, hogy milyen állapotba került. Ebből az információból felépítheti környezetének térképét.

# Online kereső ágenssek

- Ágens célja: elérjen egy célállapotot, és eközben minimalizálja a költségeket
- **kompetitív arány (competitive ratio)**: ágens által megtett tényleges út teljes költségének és az aktuális legrövidebb út aránya

# Online kereső ágenssek



# Online mélységi keresés

```
function ONLINE-MÉLYSÉGI-ÁGENS( $s'$ ) returns egy cselekvés
  inputs:  $s'$ , egy érzékelés, amely az aktuális állapotot azonosítja
  static: eredmény, cselekvésekkel és állapotokkal indexelt táblázat, kezdetben üres
           nemfeltárt, egy táblázat, amely minden meglátogatott állapotra listázza
             a még nem próbált cselekvéseket
           nemvisszalépett, egy táblázat, amely minden meglátogatott állapotra listázza
             a még nem próbált visszalépéseket
            $s$ ,  $a$ , az előbbi állapot és cselekvés, kezdetben nulla

  if CÉL-TEST( $s'$ ) then return stop
  if  $s'$  egy új állapot then nemfeltárt[ $s'$ ]  $\leftarrow$  CSELEKVÉSEK( $s'$ )
  if  $s$  nem nulla then do
    eredmény[ $a$ ,  $s$ ]  $\leftarrow s'$ 
     $s$  hozzáadása a nemvisszalépett[ $s'$ ] elejéhez
  if nemfeltárt[ $s'$ ] üres then
  if nemvisszalépett[ $s'$ ] üres then return stop
  else  $a \leftarrow$  egy olyan  $b$  cselekvés, hogy eredmény[ $b$ ,  $s'$ ] = POP(nemvisszalépett[ $s'$ ])
  else  $a \leftarrow$  POP(nemfeltárt[ $s'$ ])
   $s \leftarrow s'$ 
  return  $a$ 
```

Köszönöm a figyelmet!