A mesterséges intelligencia alapjai

problémamegoldó ágensek, kereső algoritmusok

Áttekintés

- problémamegoldó ágensek
- problémák típusai
- probléma megfogalmazása
- példák problémákra
- alapvető keresési algoritmusok

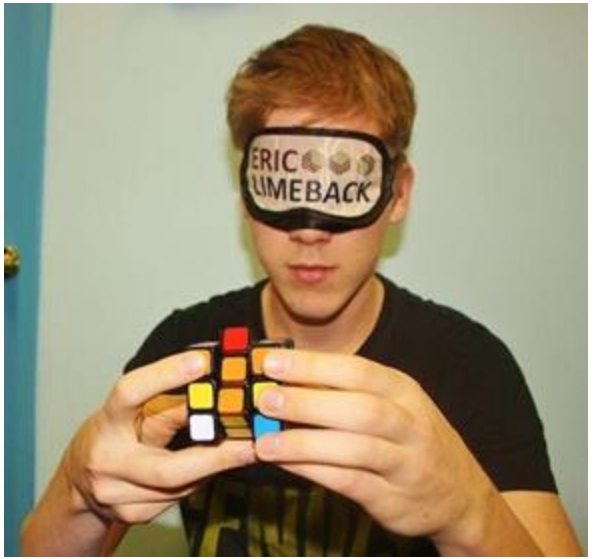
Problémamegoldó ágensek

```
function Simple-Problem-Solving-Agent(percept): művelet
    static:   seq: cselekvéssorozat, kezdetben üres
              state: a világ aktuális állapotának leírása
              goal: cél, kezdetben nincs (null)
              problem: a probléma megfogalmazása

    state = Update-State(state, percept)
    if seq is empty then
        goal = Formulate-Goal(state)
        problem = Formulate-Problem(state, goal)
        seq = Search(problem)
    action = Recommendation(seq, state)
    seq = Remainder(seq, state)
    return action
```

Problémamegoldó ágensek

Ez **offline** problémamegoldás: a megoldás során a *szemek csukva vannak.*



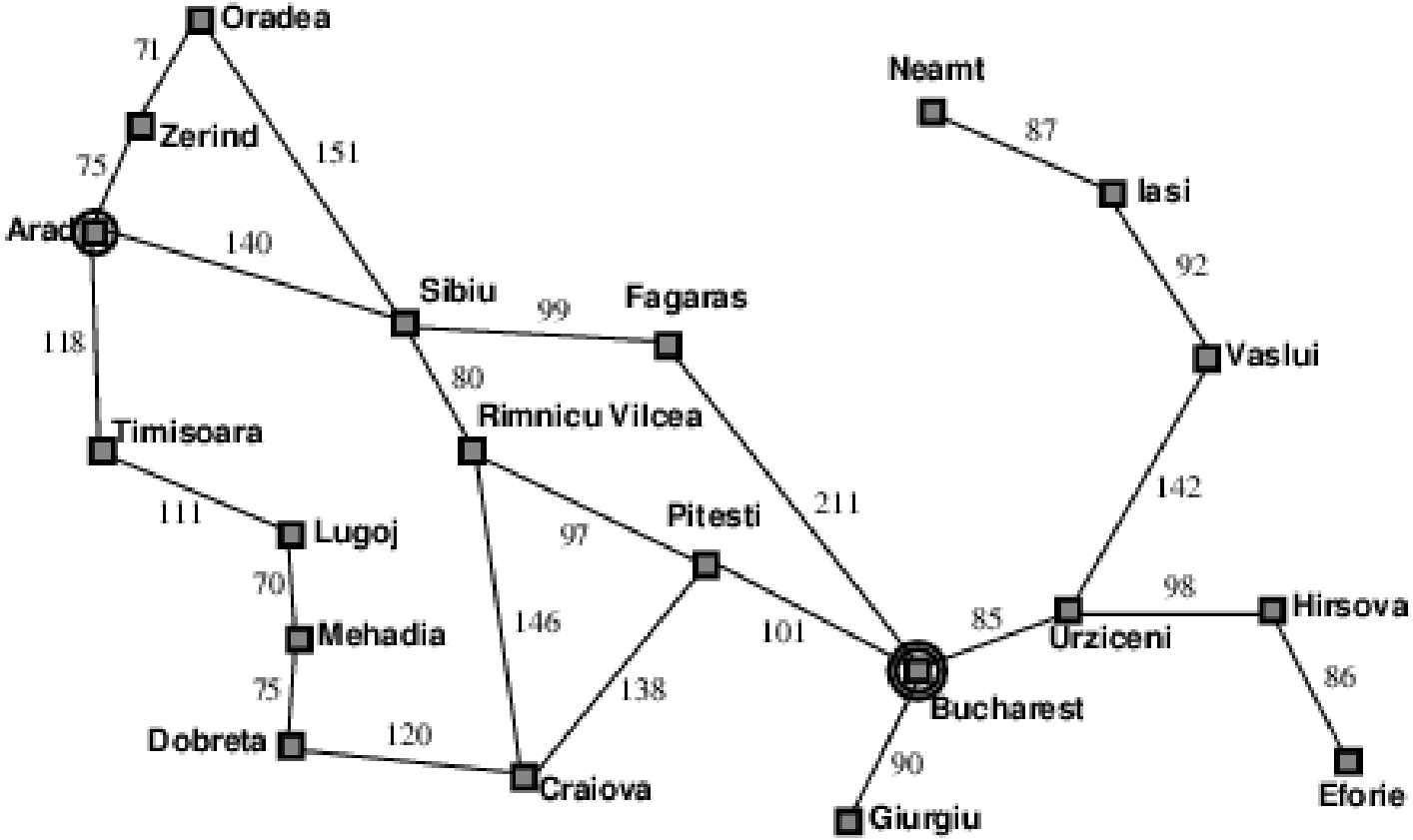
Online problémamegoldás során a teljes tudás hiányában cselekszünk, menetközben új információkat szerzünk.

Példa: Romániai kirándulás

Nyaraláson van Romániában, pontosabban Aradon. A repülő holnap indul Bukarestből.

- **Cél megfogalmazása**: legyen Bukaresten.
- **Probléma megfogalmazása**:
  - állapotok: romániai városok
  - műveletek/cselekvések: átutazik az egyik városból a másikba
- **Megoldás megkeresése**:
  - városok egy sorozata A-ból B-be
  - pl. Arad, Nagyszeben, Fogaras, Bukarest

Példa: Romániai kirándulás



Probléma típusok

- determinisztikus, teljesen megfigyelhető ⇒ egyetlen állapot probléma
  - az ágens pontosan tudja, hogy melyik állapotban van
  - a megoldás egy cselekvéssorozat

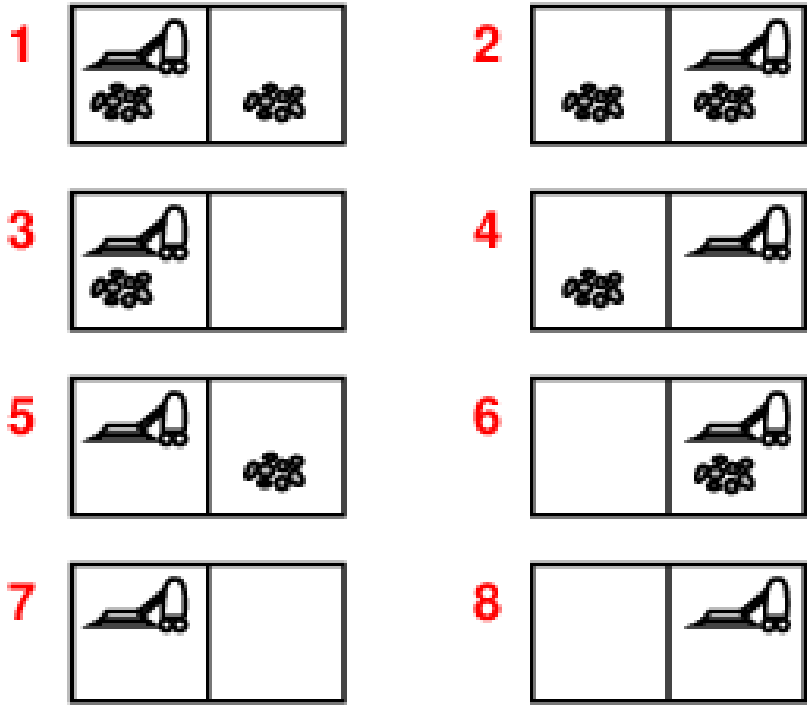
- nem megfigyelhető ⇒ sensorless probléma
  - az ágens nem tudja, hogy hol van
  - a megoldás (ha egyáltalán létezik) egy cselekvéssorozat
- nemdeterminisztikus vagy részben megfigyelhető ⇒ eshetőségi problémák
  - az észlelés új információt az aktuális állapotról
  - a megoldás egy irányelv függő terv, melynek része a keresés
- ismeretlen állapottér esetén ⇒ felfedezéses probléma (online)
  - online probléma

Keresés részleges információ mellett

Példa: porszívó világ

determinisztikus, teljesen megfigyelhető

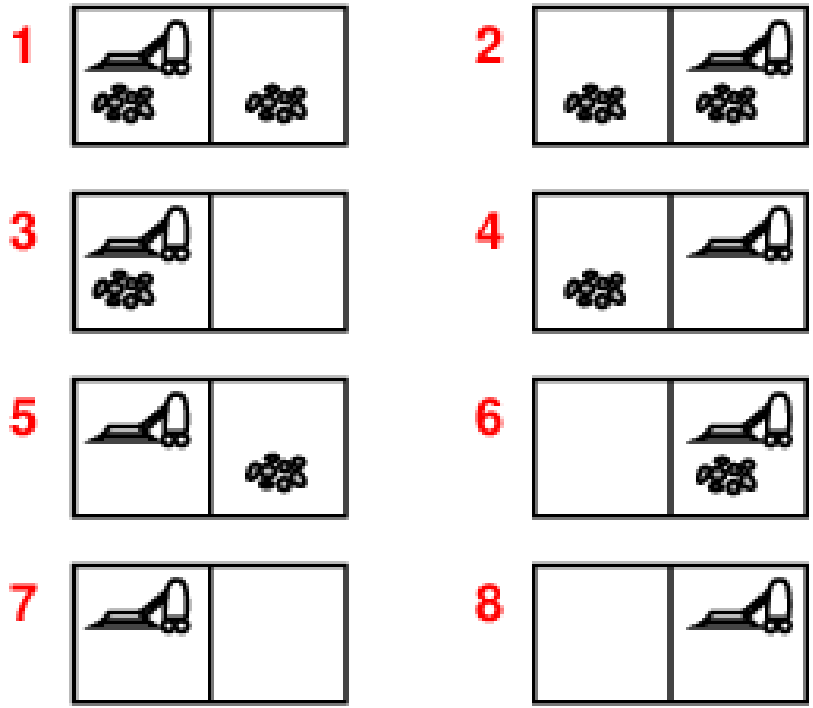
- 5-ös állapotból indul
- [jobbra, takarít]



Példa: porszívó világ

szenzormentes

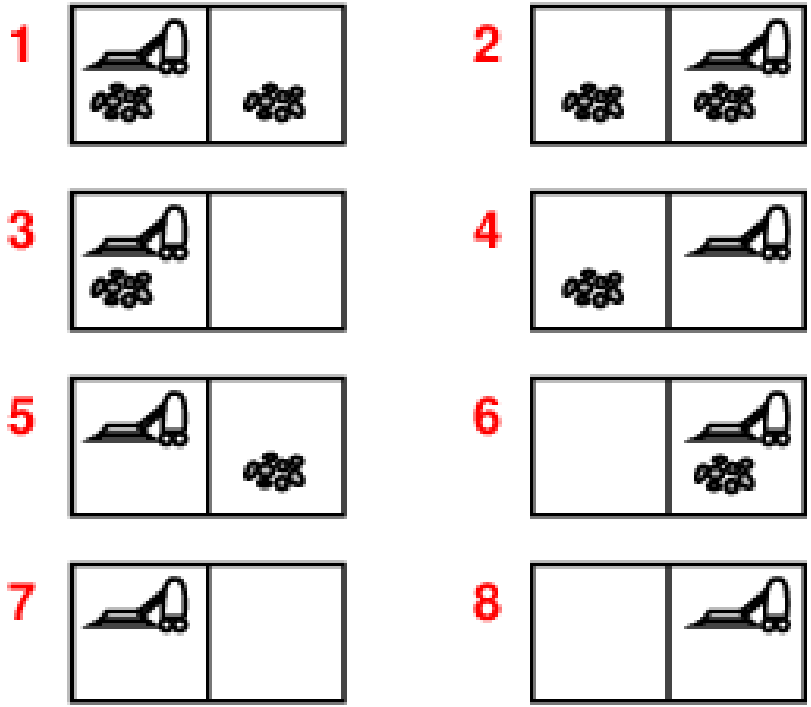
- {1,2,3,4,5,6,7,8} állapotból indul
- jobbra hatására átkerül {2,4,6,8}
- [jobbra, takarít, balra, takarít]



Példa: porszívó világ

eshetőségi probléma

- 5-ös állapotból indul
- Murphy törvénye: a takarítás összekoszolhat egy tiszta szőnyeget.
- helyi észlelés:
  - kosz és pozíció
- [jobbra, *ha* koszos, *akkor* takarít]



Probléma megfogalmazás (Determinisztikus, teljesen megfigyelhető)

- **kezdeti állapot**
  - Aradon van
- S(x) **állapotátmenet függvény**
  - cselekvés-állapot párok
  - S(Arad)={<Arad-Zerind, Zerind>, …}
- **célteszt**
  - explicit (célállapotok halmaza) x=Bukarest
  - implicit (absztrakt tulajdonsággal definiálva) NoDirt(x)
- **útköltség** (additív)
  - pl. távolságok összege, cselekvések száma, stb.
  - c(x,a,y) a lépés költsége, nemnegatív

Megoldás: cselekvéssorozat, mely a kezdőállapotból egy célállapotba vezet.

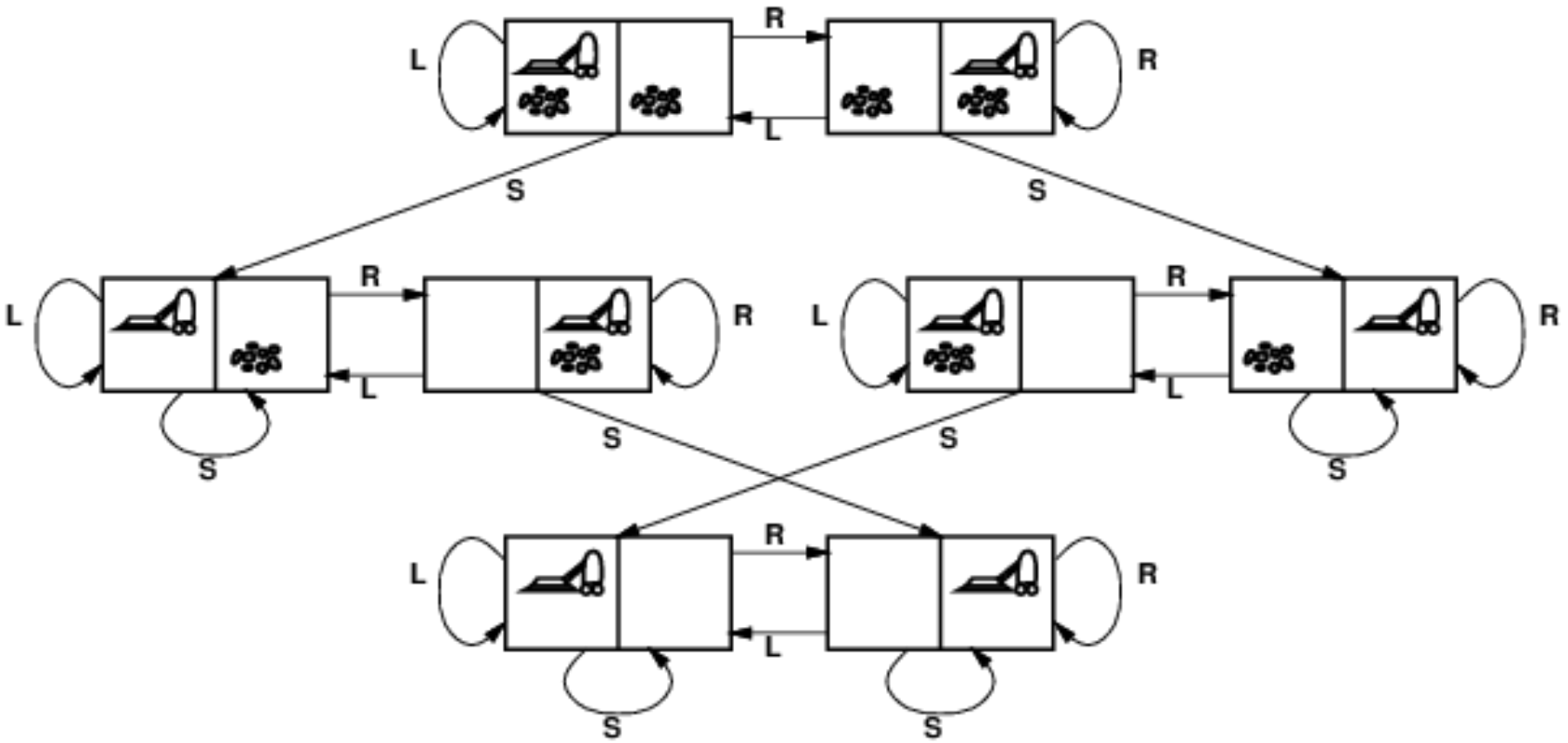
Állapottér megválasztása

- a valós világ elképesztően komplex
  - az állapottér megfogalmazásából a felesleges részleteket elhagyjuk (absztrakció)
- (absztrakt) állapot – valós állapotok halmaza
- (absztrakt) cselekvés – valós cselekvések komplex kombinációja
  - Arad→Zerind művelet tartalmazhat elterelés, ebédszünetet, stb.
- (absztrakt) megoldás – valós utak (valós világbeli megoldások) halmaza
- az absztrakt műveleteknek egyszerűbbnek kell lenni az eredeti problémánál

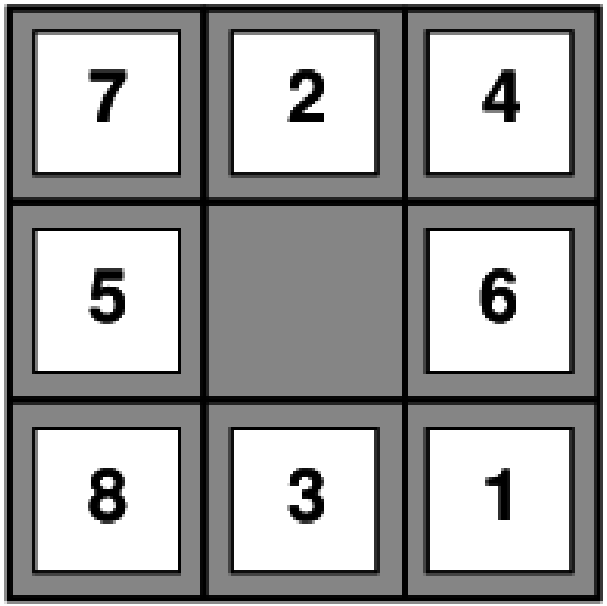
Példa: porszívóvilág problémája

- állapotok
  - kosz és robot helyzete (egészekkel leírható)
- műveletek
  - balra, jobbra, takarít, NoOp
- célteszt
  - már nincs kosz sehol
- útköltség
  - 1 műveletenként (NoOp 0)

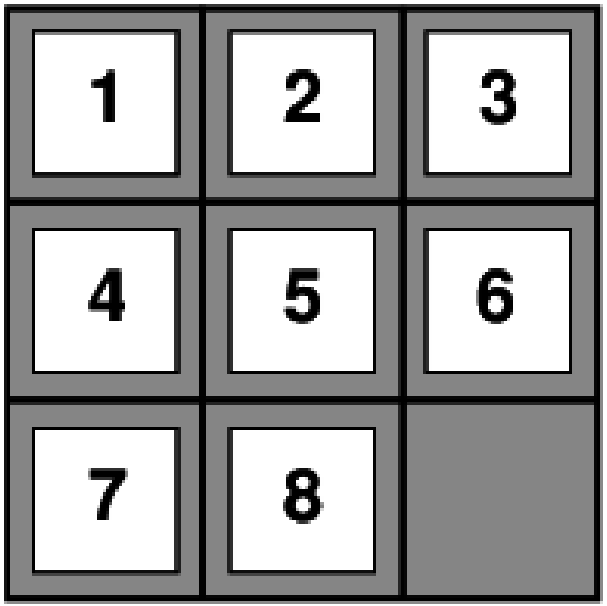
Példa: porszívóvilág állapottere



Példa: nyolcas játék



Start State



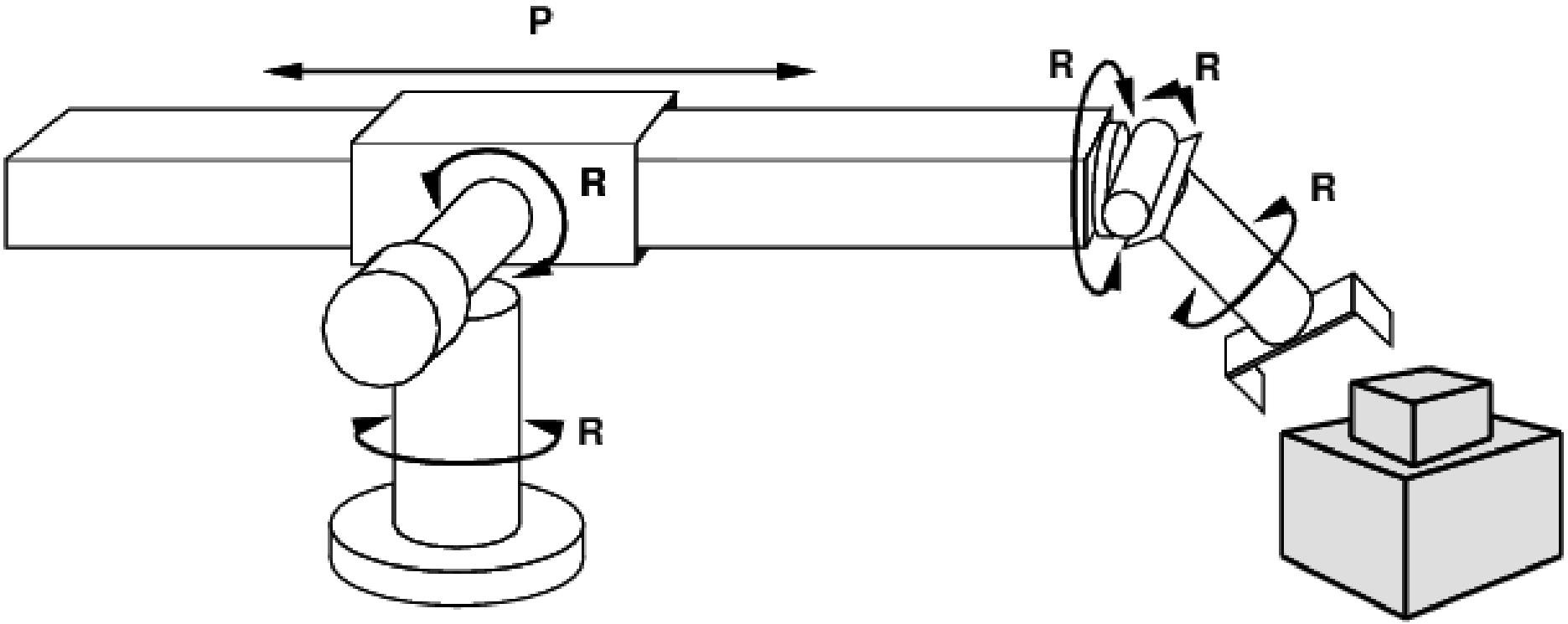
Goal State

Példa: nyolcas játék

- állapotok
  - egyes lapok helyzete (egészekkel leírható)
- műveletek
  - az üres hely mozgatása balra, jobbra, fel, le
- célteszt
  - az aktuális állapot megegyezik a megadottal? (explicit)
- útköltség
  - lépésenként 1

Az ilyen feladatok optimális megoldása NP-nehéz

Példa: robotkar



Példa: robotkar

- állapotok
  - a kar csuklóinak (R) és tengelyének (P) koordinátái (valós számokkal leírható)
  - az összeillesztendő alkatrész darabjainak helyzete
- műveletek
  - a csuklók és tengelyek folytonos mozgása
- célteszt
  - az alkatrész össze van szerelve
- útköltség
  - összeszereléshez felhasznált idő

Fakereső algoritmusok

- offline,
- az állapottér szisztematikus felfedezése
- a már felfedezett állapotok rákövetkezőinek generálása: **kiterjesztés**

Fakereséső algoritmusok

function Tree-Search(problem, strategy): megoldás vagy „sikertelen”

kereső fa inicializálása a probléma kezdőállapotával

loop do

if nincs kiterjeszthető csúcs

then return „sikertelen”

válassz a stratégia alapján egy levél csúcsot

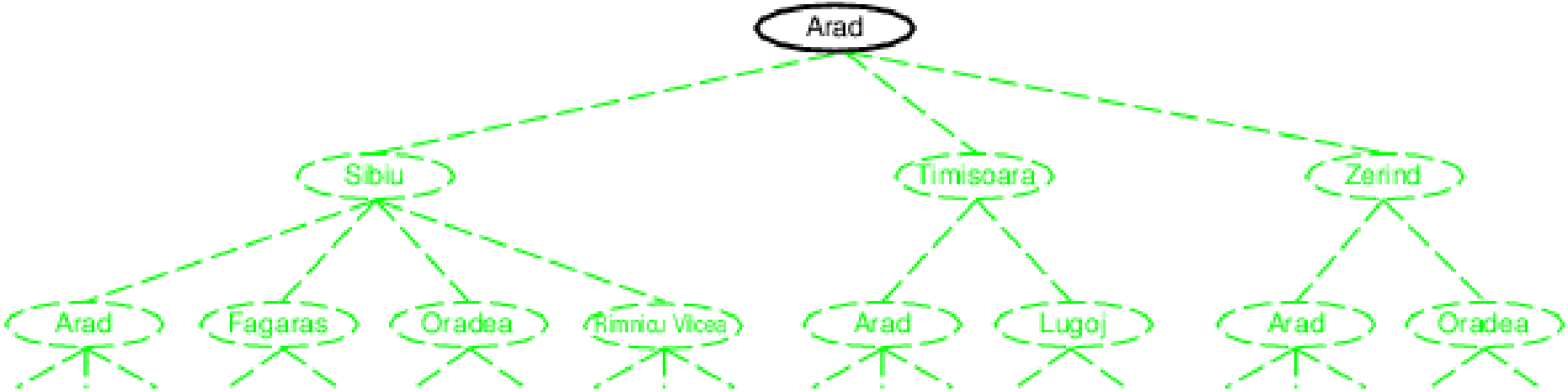
if a csúcs célállapotot tartalmaz

then return kapcsolódó megoldást

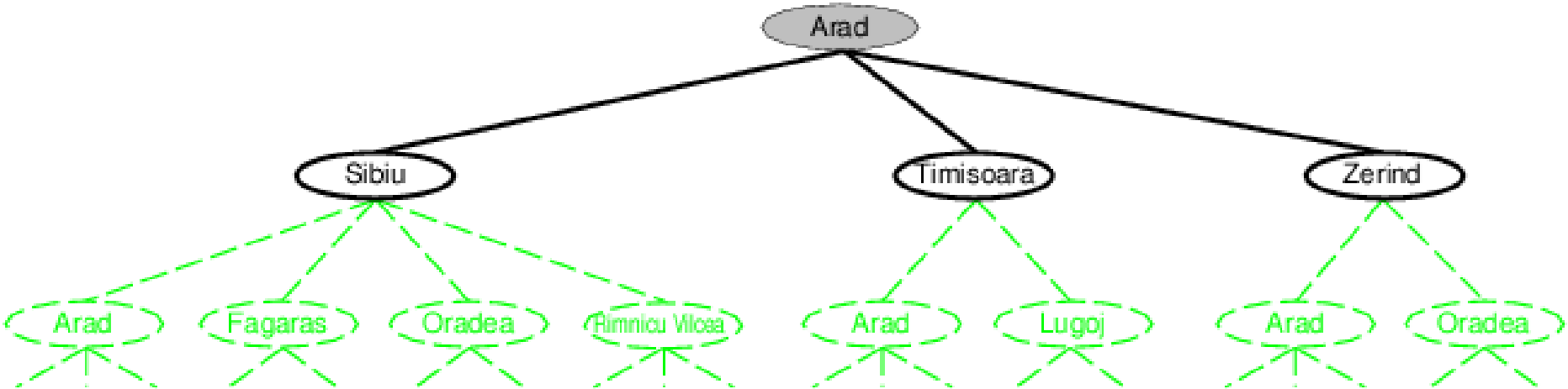
else terjeszd ki a csúcsot, és a gyerekcsúcsokat add a keresőfához

end

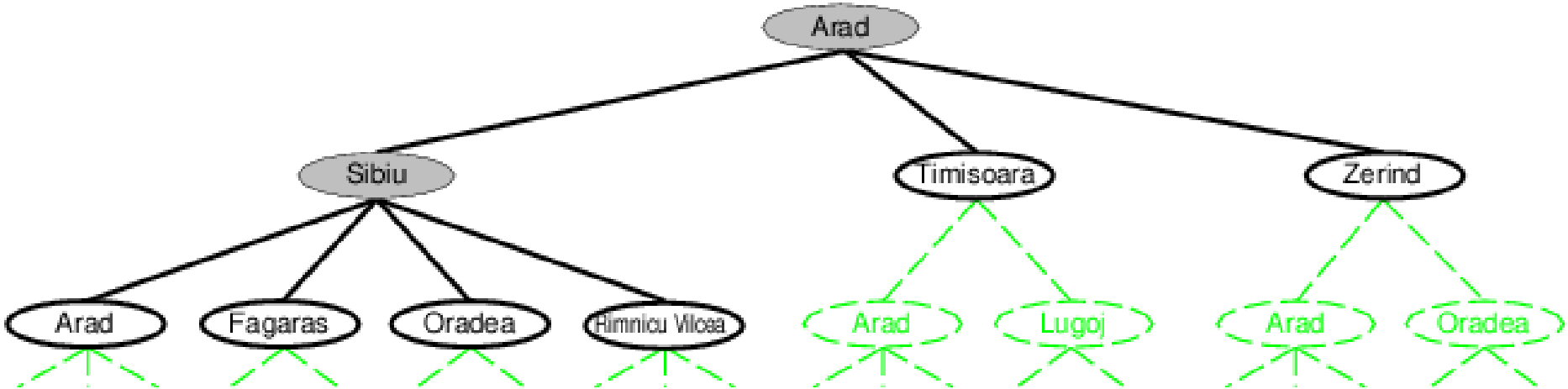
Példa: romániai túra



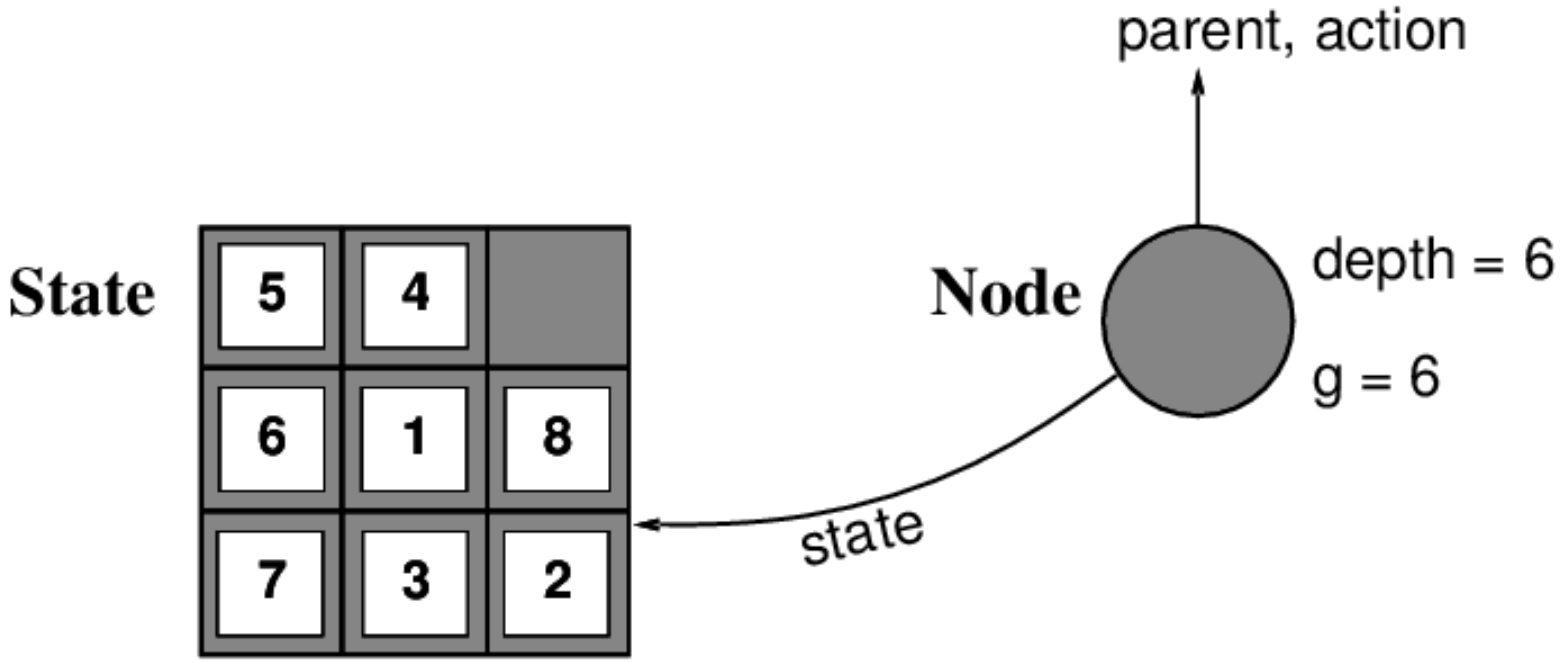
Példa: romániai túra



Példa: romániai túra



Implementáció: állapotok és csúcsok



Implementáció: állapotok és csúcsok

- az **állapot** a fizikai konfiguráció (reprezentációja)
- a **csúcs** egy adatszerkezet, melyből felépül a keresőfa
  - tartalmaz: szülő, gyerekek, mélység, útköltség
- az állapotnak nincs szülője, gyereke, mélysége, útköltsége
- az **expand** (kiterjeszt) függvény új csúcsokat generál, feltölti azok különböző mezőit, és a **SuccessorFn** (rákövetkező) függvényt használja fel, hogy megkonstruálja a megfelelő állapotokat

Fakereső algoritmusok (AIMA kód)

```
def tree_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""
    frontier.append(Node(problem.initial))
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

Implementáció: kiterjesztés – pszeudokód

```
function Expand(node, problem)): rákövetkező csúcsok
    successors  = ∅
    for each action, result in Successor-Fn(problem, State[node]) do
        s = new Node
        Parent-Node[s] = node
        Action[s] = action
        State[s] = result
        Path-Cost[s] = Path-Cost[node] + Step-Cost(State[node],action,result)
        Depth[s] = Depth[node] + 1
    successors += s
    return successors
```

Implementáció: kiterjesztés (AIMA kód)

```
def expand(self, problem):
    "List the nodes reachable in one step from this node."
    return [self.child_node(problem, action)
            for action in problem.actions(self.state)]

def child_node(self, problem, action):
    "[Figure 3.10]"
    next = problem.result(self.state, action)
    return Node(next, self, action,
                problem.path_cost(self.path_cost, self.state,
                                  action, next))
```

Keresési stratégiák

A stratégiát a kiterjesztéshez kapcsolódó kiválasztás határozza meg.

A stratégiákat a következő dimenziók mentén csoportosíthatjuk:

- **teljesség**
  - Ha van megoldás, akkor megtaláljuk?
- **időbonyolultság**
  - Hány csúcsot generál/terjeszt ki?
- **tárbonyolultság**
  - Egyszerre mennyi csúcsot kell a memóriában tárolni?
- **optimalitás**
  - Mindig a legkisebb költségű megoldást találja meg?

Keresési stratégiák

Az idő és tár bonyolultságánál a következő konstansokat használjuk:

- b – maximális elágazási faktor a kereső fában
- d – a legkisebb költségű megoldás mélysége
- m – az állapottér maximális mélysége (esetleg ∞)

Nem informált keresési stratégiák

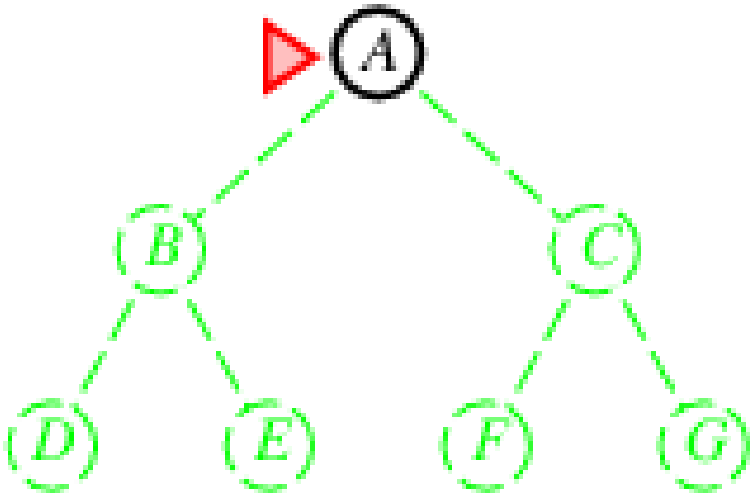
A nem informált keresési stratégiák csak a probléma definiálásakor megadott információkat használja fel

- szélességi keresés
- egyenletes költségű keresés (~ best first)
- mélységi keresés
- mélységkorlátozott keresés
- iteratívan mélyülő mélységi keresés

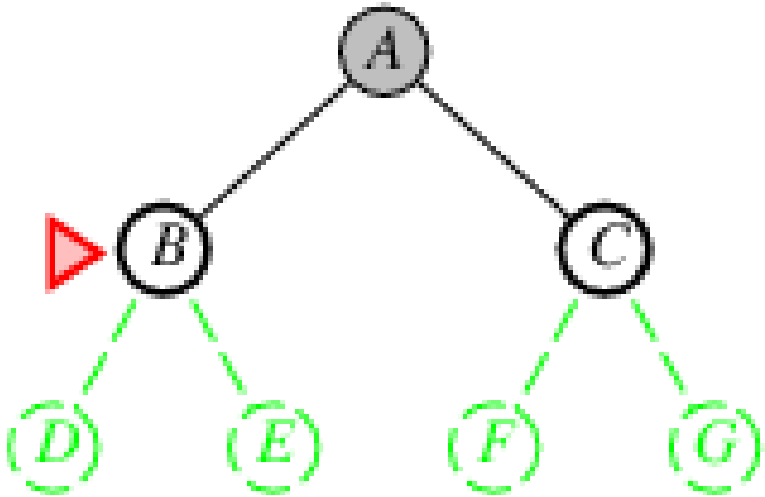
Szélességi keresés

A legsekélyebb (legkisebb mélységű) még ki nem terjesztett csúcsot terjeszti ki

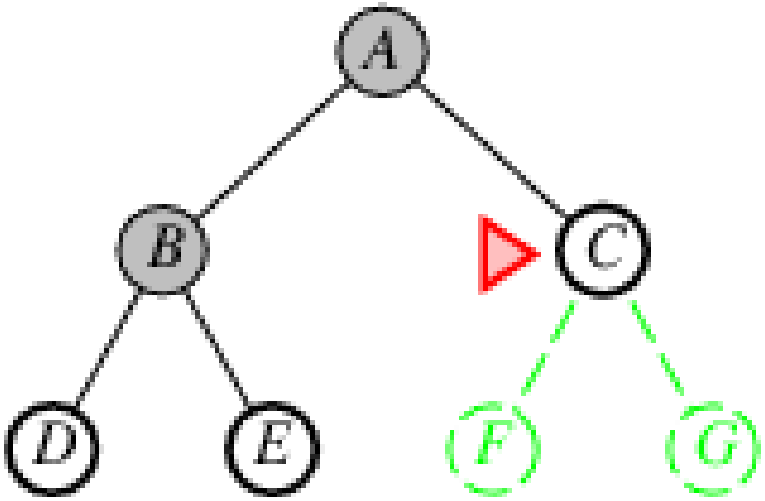
A perem egy sor adatszerkezet (FIFO), az új rákövetkezők a sor végére kerülnek, a sor első eleme kerül kiterjesztésre



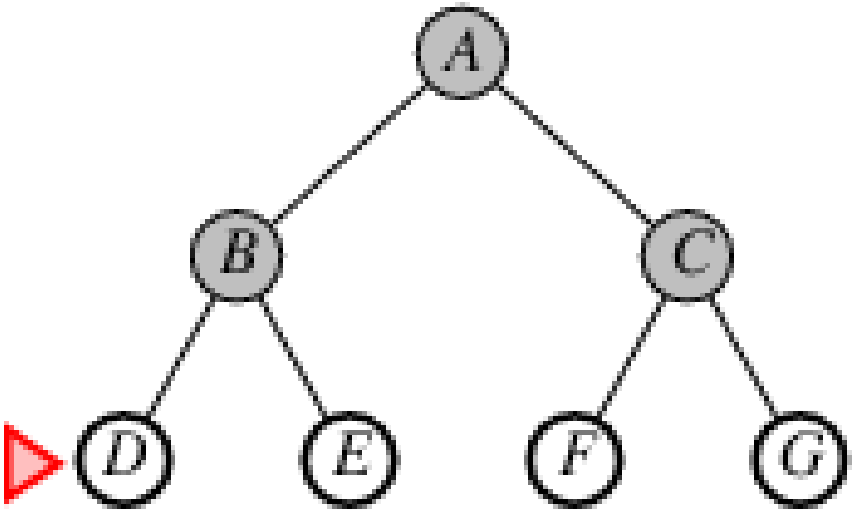
Szélességi keresés



Szélességi keresés



Szélességi keresés



Szélességi keresés tulajdonságai

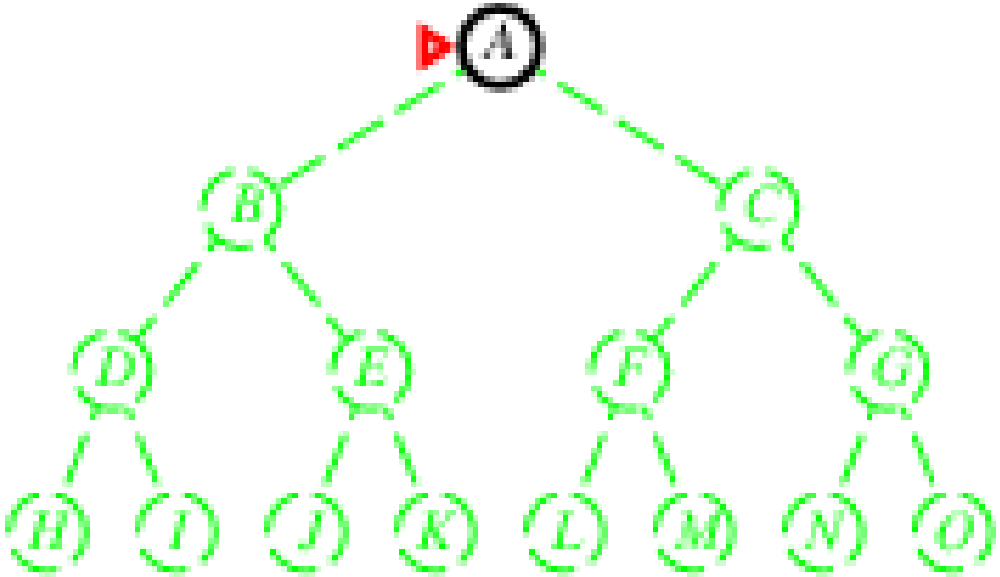
- Teljes?
  - Igen, ha b véges
- Időbonyolultság?
  - $1+b+... + b(b^d-1) = O(b^{d+1})$, azaz d-ben exponenciális
- Tárbonyolultság?
  - $O(b^{d+1})$, minden csúcsot a memóriában tart
- Optimális?
  - Igen, ha minden lépés azonos költségű
  - általános esetben nem
- a tárigény jelentős probléma, ha 100MB/s a csúcsok generálásának sebessége, az egy nap 8640GB

Egyenletes költségű keresés tulajdonságai

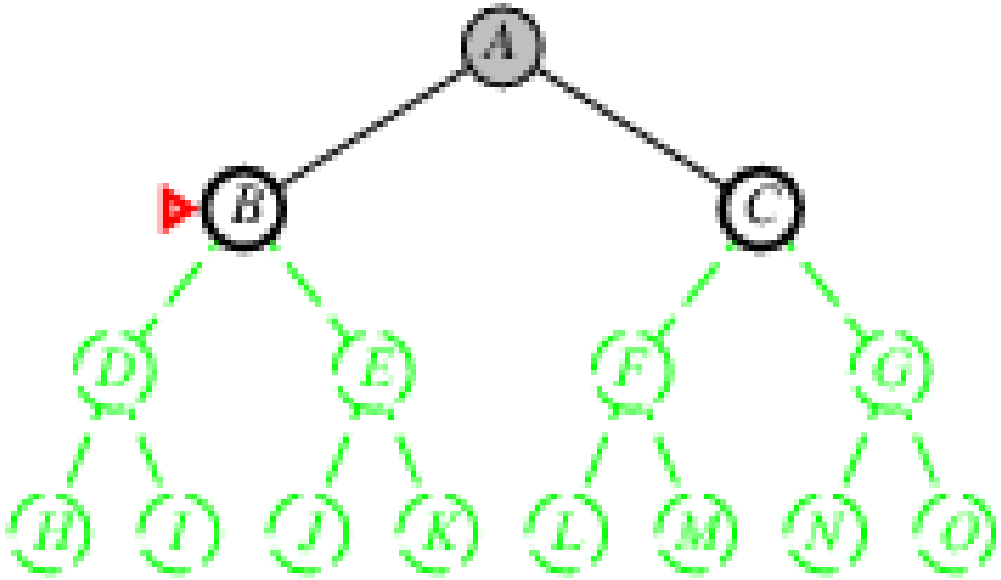
- A legkisebb költségű még ki nem terjesztett csúcsot terjeszti ki.
- Perem implementációja: prioritás sor az útköltség szerint (növekvő)
- Ha minden élköltség azonos, egybeesik a szélességi kereséssel
- Teljes?
  - Igen, ha az élköltségek pozitívak és nagyobbak ε-nál
- Idő- és tárbonyolultság
  - azon csúcsok száma, melyek útköltsége kisebb vagy egyenlő az optimális megoldás költségénél $O(b^{\lceil C^*/\varepsilon \rceil})$
- Optimális?
  - igen, a csúcsok a g(n) útköltség szerint növekvő sorrendben lesznek kifejtve

Mélységi keresés

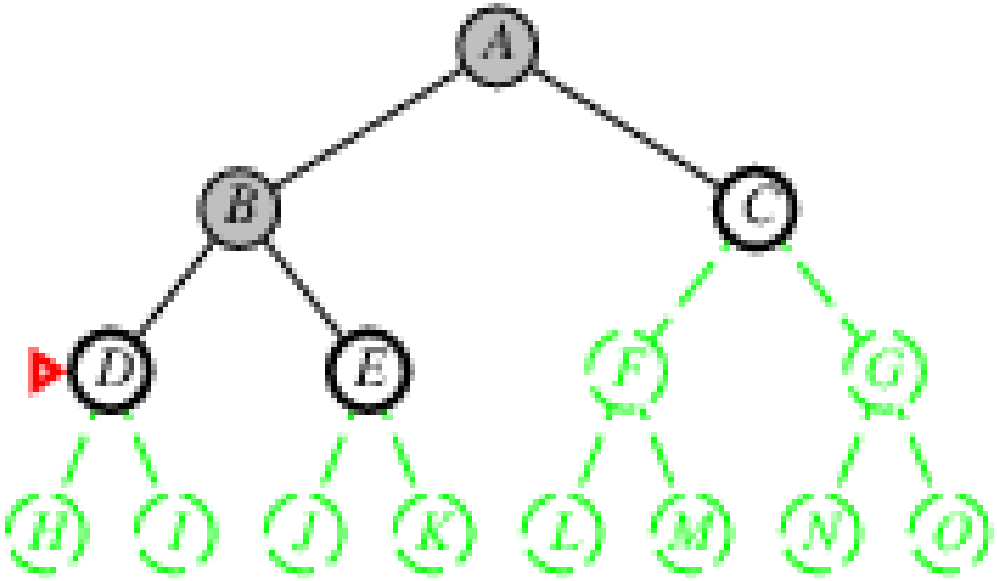
- a legmélyebb ki nem fejtett csúcsot fejti ki
- perem implementációja: verem (LIFO), a rákövetkezőket a verem tetejére teszi



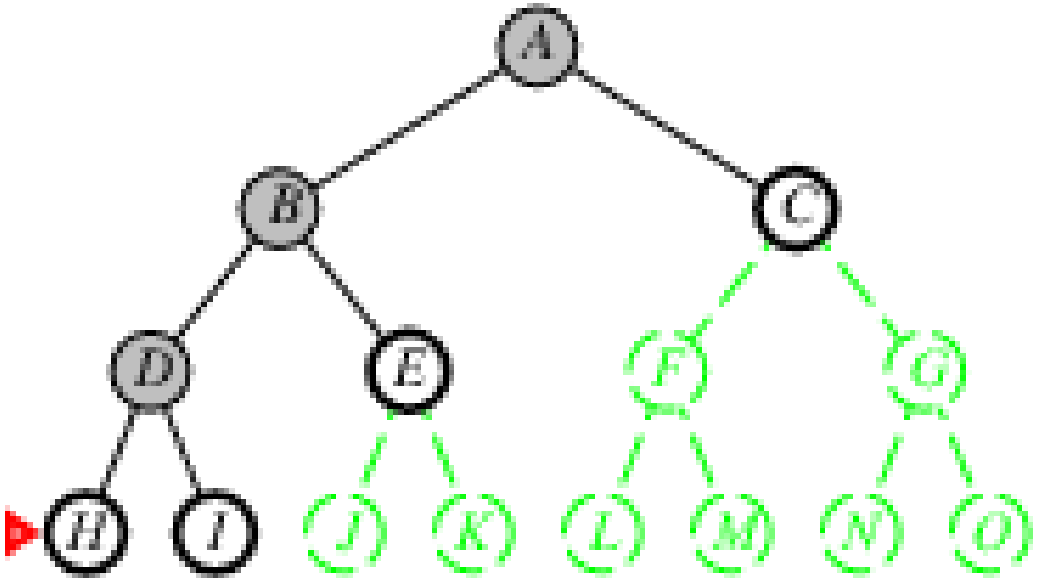
Mélységi keresés



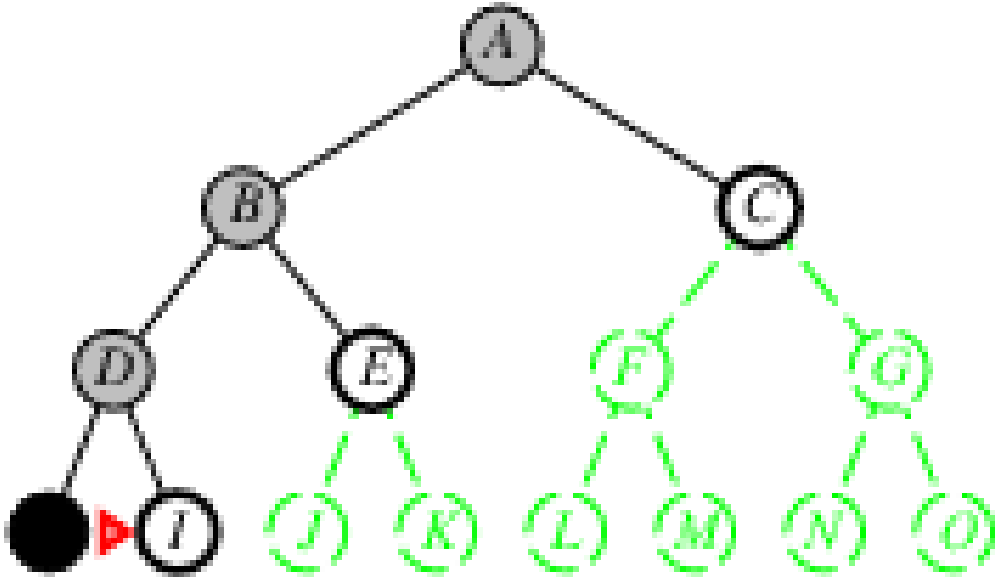
Mélységi keresés



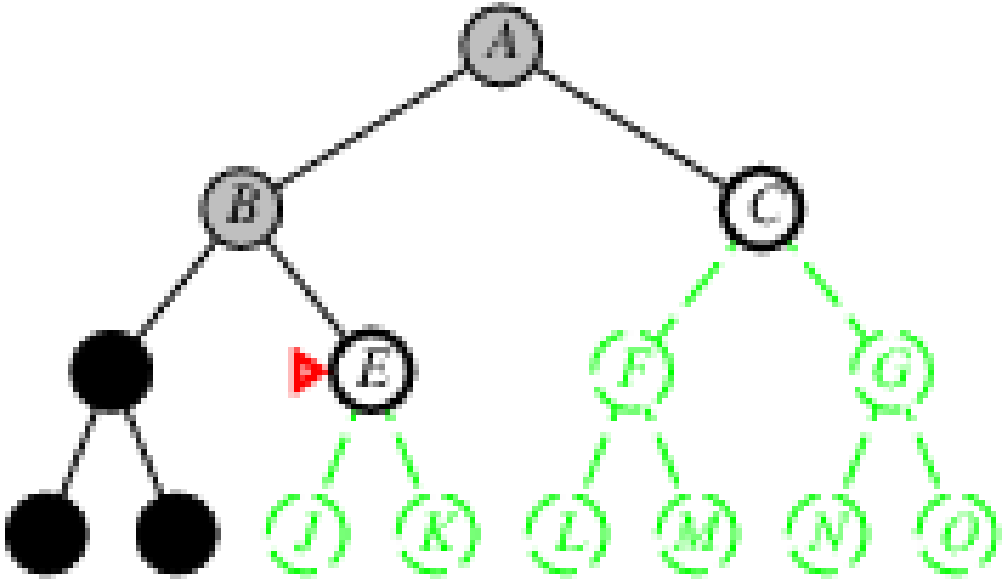
Mélységi keresés



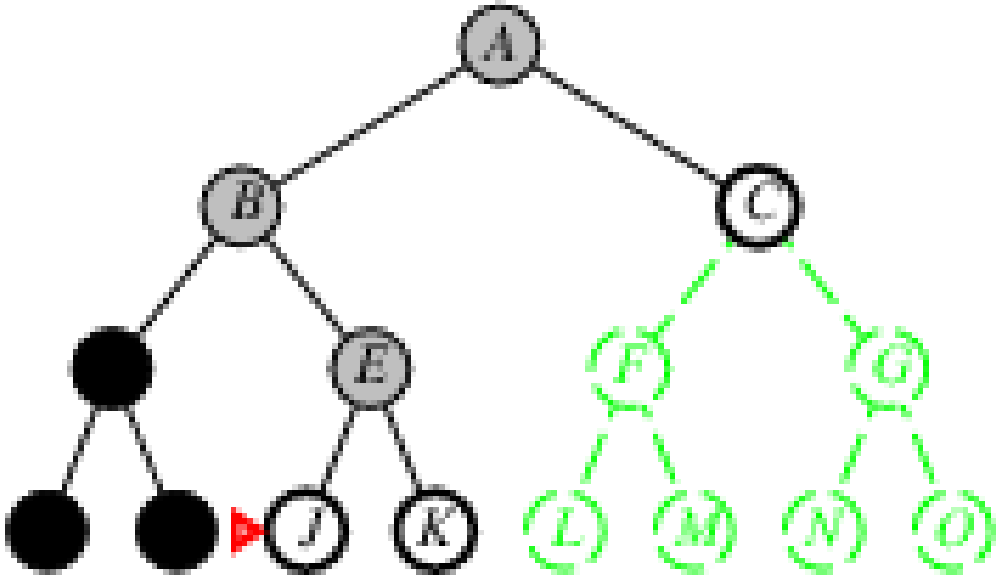
Mélységi keresés



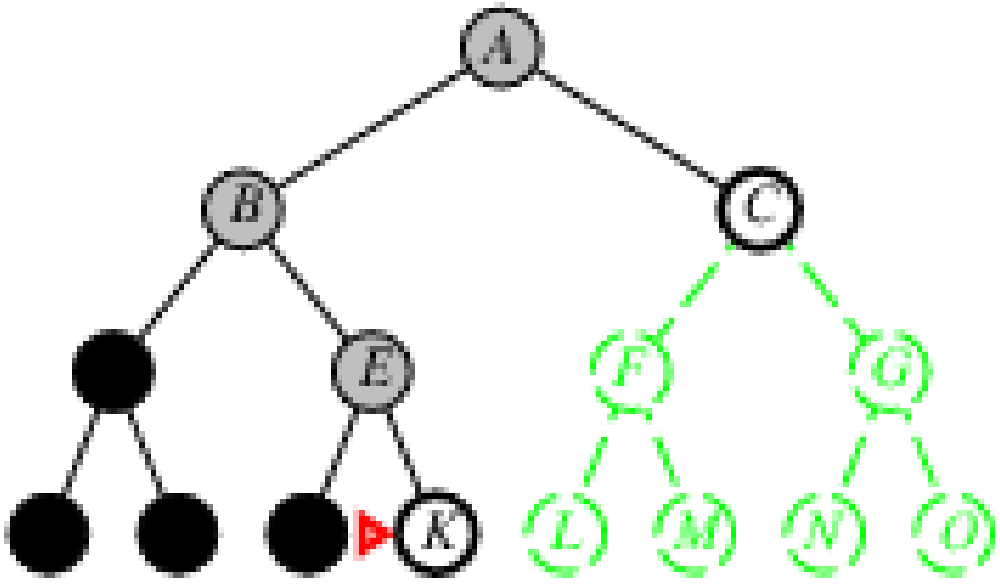
Mélységi keresés



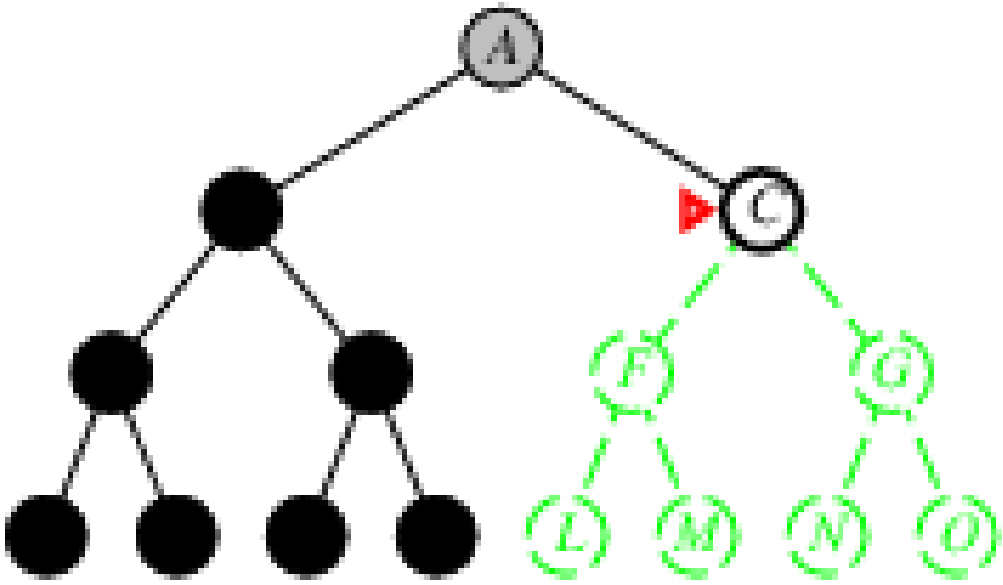
Mélységi keresés



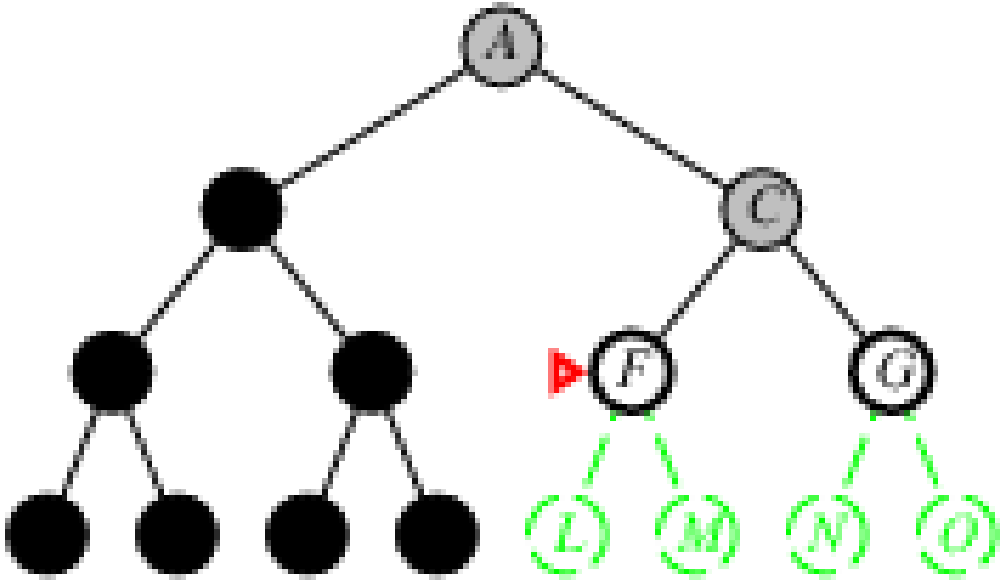
Mélységi keresés



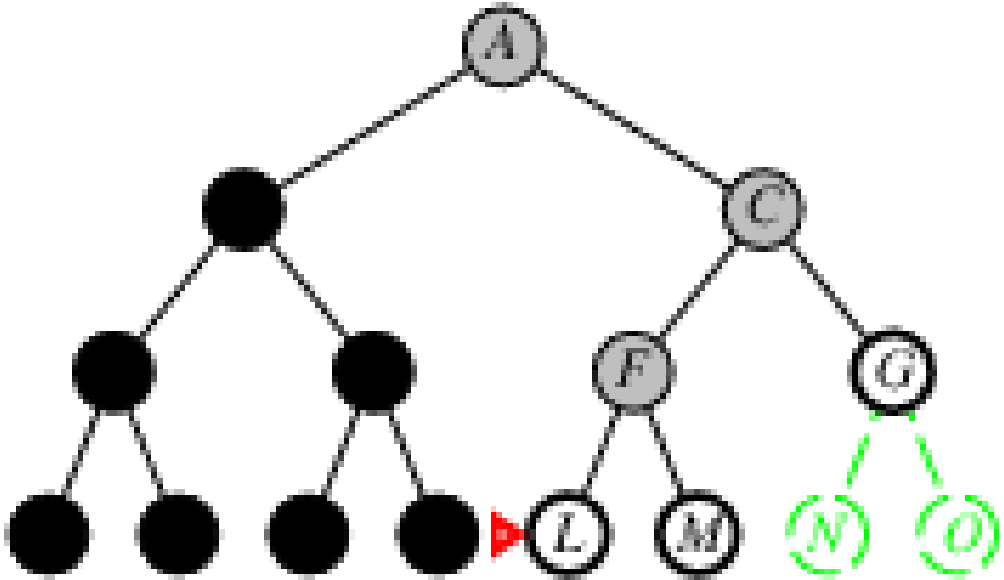
Mélységi keresés



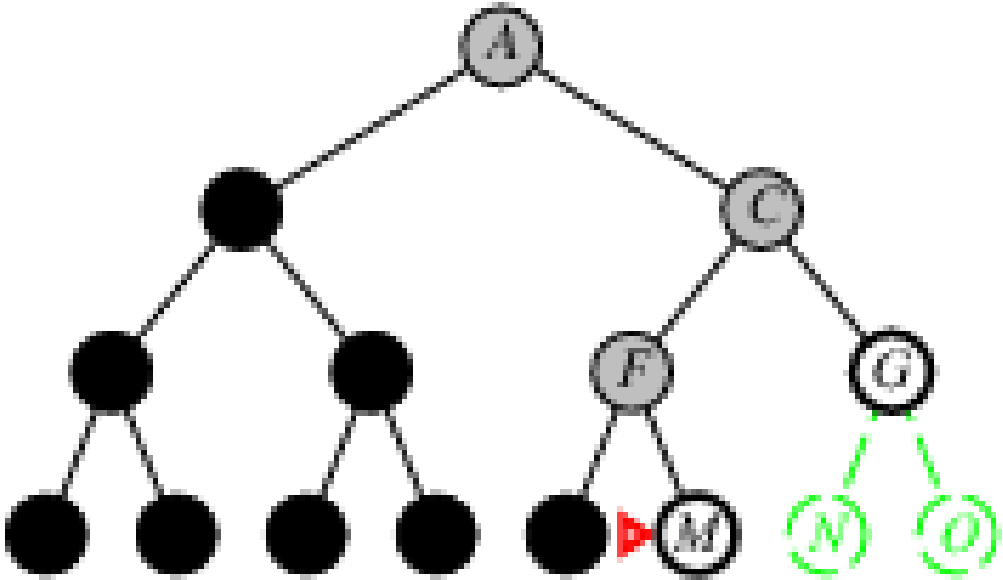
Mélységi keresés



Mélységi keresés



Mélységi keresés



Mélységi keresés tulajdonságai

- Teljes?
  - nem, végtelen vagy ciklust tartalmazó állapotterekben általában nem talál megoldást
  - módosítható, hogy felismerje a ciklusokat
  - véges állapotterekben teljes
- Időbonyolultság?
  - $O(b^m)$: rettenetes, ha m sokkal nagyobb mint d, de ha sűrűn találhatóak megoldások, gyorsabb lehet, mint a szélességi keresés
- Tárbonyolultság?
  - O(bm), lineáris!
- Optimális?
  - Nem

Youtube vizualizáció

Mélységkorlátozott keresés

- a mélységi keresés változata
- az *l* mélységben lévő csúcsoknak nincs rákövetkezőjük.

Mélységkorlátozott keresés – rekurzív implementáció

function Depth-Limited-Search(problem, limit): megoldás, sikertelen vagy vágás

Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

Mélységkorlátozott keresés – rekurzív implementáció

```
function Recursive-DLS(node, problem, limit): megoldás, sikertelen vagy vágás
    cutoff-occurred? = false
    if Goal-Test(problem, State[node]) then return node
    else if Depth[node] = limit then return vágás
        else
            for each successor in Expand(node, problem) do
                result = Recursive-DLS(successor, problem, limit)
                if result = vágás then cutoff-occurred? = true
                else if result != sikertelen then return result
        if cutoff-occurred? then return vágás
        else return sikertelen
```

AIMA kód

```
def depth_limited_search(problem, limit=50):
    "[Figure 3.17]"
    def recursive_dls(node, problem, limit):
        if problem.goal_test(node.state):
            return node
        elif limit == 0:
            return 'cutoff'
        else:
            cutoff_occurred = False
            for child in node.expand(problem):
                result = recursive_dls(child, problem, limit - 1)
                if result == 'cutoff':
                    cutoff_occurred = True
                elif result is not None:
                    return result
            return 'cutoff' if cutoff_occurred else None

    # Body of depth_limited_search:
    return recursive_dls(Node(problem.initial), problem, limit)
```

Iteratívan mélyülő mélységi keresés

```
function Iterative-Deepening-Search(problem): megoldás
    for depth = 0 to ∞ do
        result = Depth-Limited-Search(problem, depth)
        if result != cutoff then return result
    end
```

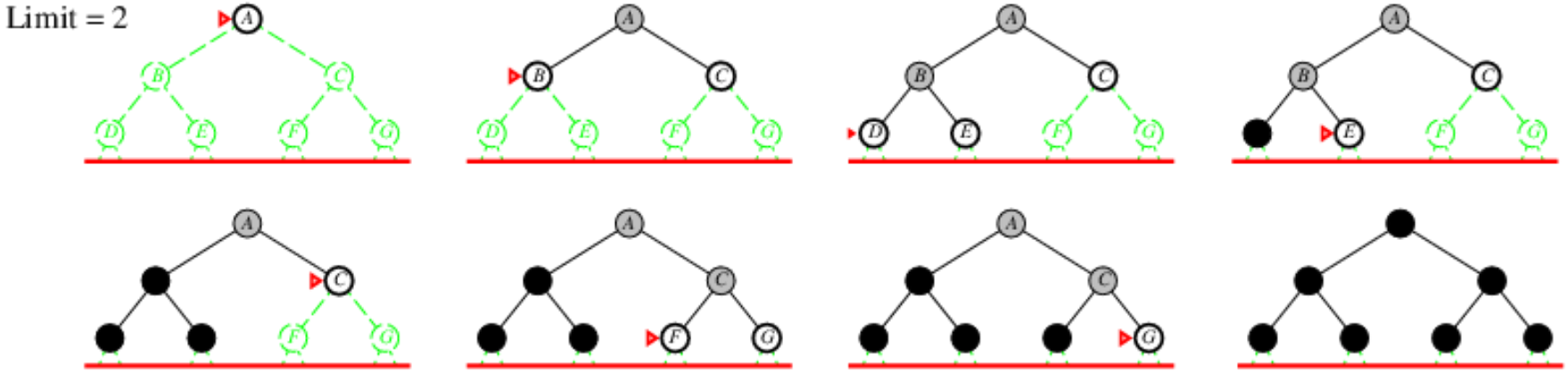
Iteratívan mélyülő mélységi keresés

Limit = 0

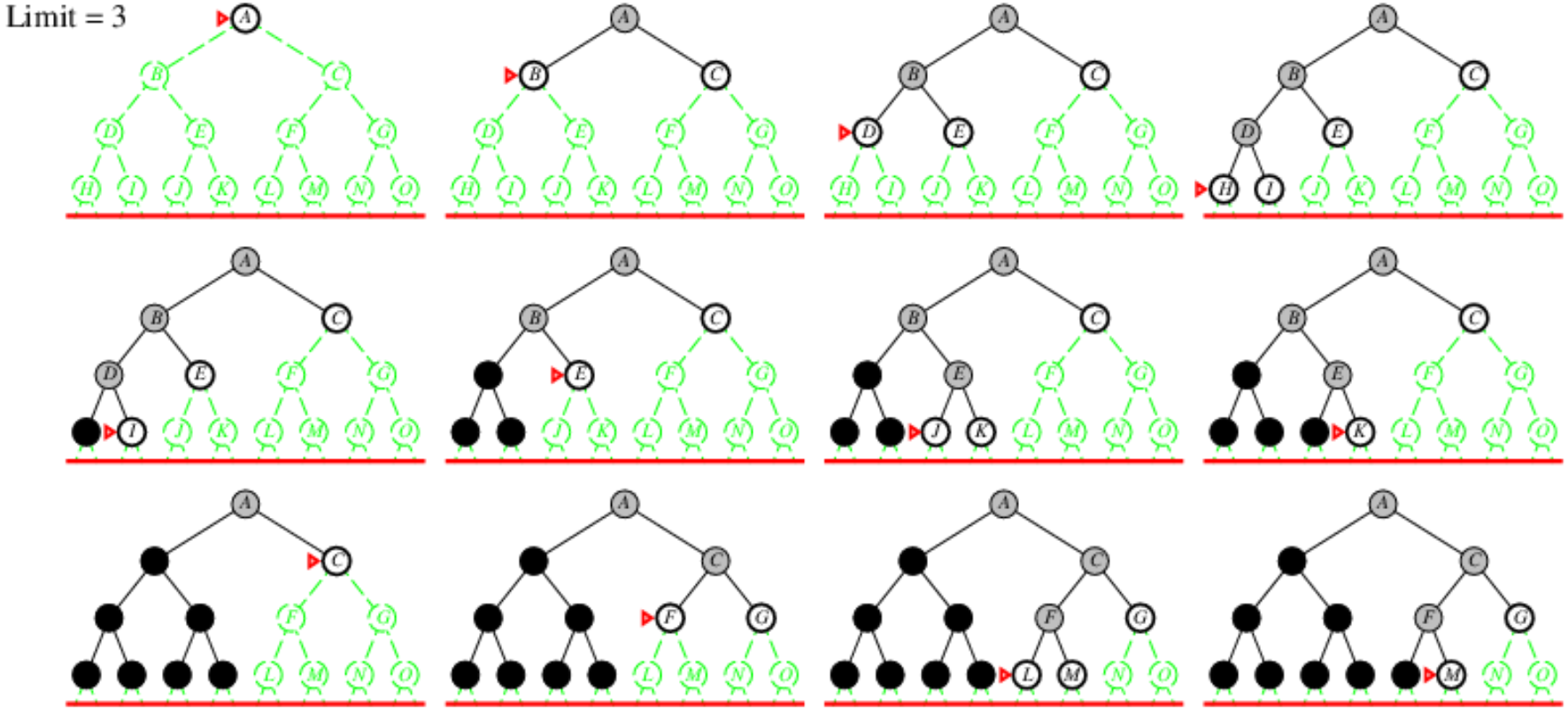
Iteratívan mélyülő mélységi keresés

Limit = 1

Iteratívan mélyülő mélységi keresés



Iteratívan mélyülő mélységi keresés



Iteratívan mélyülő mélységi keresés tulajdonságai

- Teljes?
  - igen
- Időbonyolultság?
  - $(d+1)b^0+db^1+ (d-1)b^2+ \ldots + b^d = O(b^d)$
- Tárbonyolultság?
  - O(db)
- Optimális?
  - igen, ha az élköltség 1
  - módosítható, hogy az egyenletes költségű fát fedezzen fel

Numerikus összehasonlítás: b=10, d=5

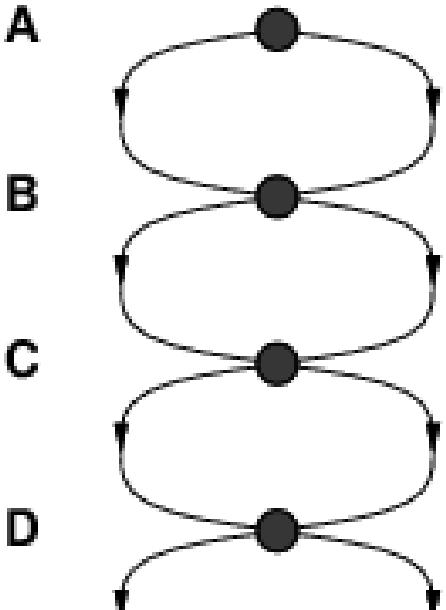
- N(IMMK) = 50+400+3000+20 000+100 000 = 123 450
- N(SzK) = 10+100+1000+10 000+100 000+999 999 = 1 111 100
- az iteratívan mélyülő gyorsabb, mert d mélységben nem terjeszti ki a csúcsokat
- a szélességi kereső módosítható, hogy a céltesztet akkor vizsgálja, amikor a csúcs generálódik

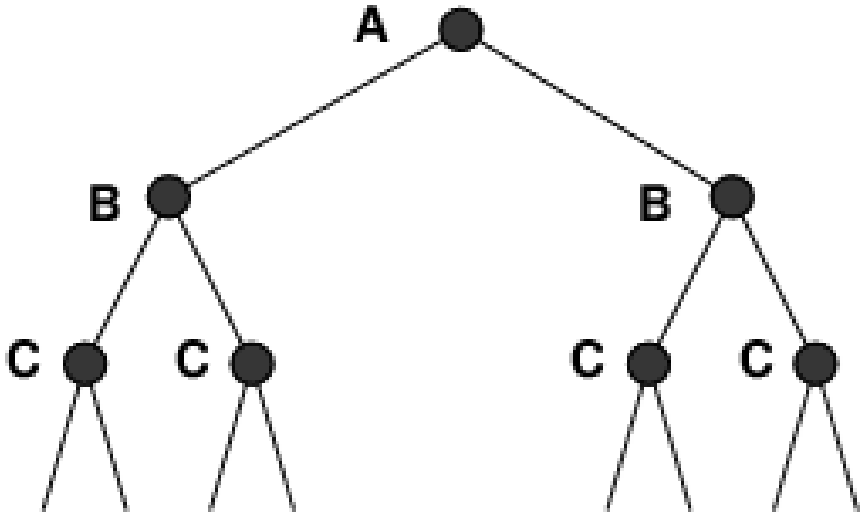
Algoritmusok összefoglalása

tulajdonság	szélességi	egyenletes költségű	mélységi	mélység-korlátozott	iteratívan mélyülő mélységi
teljes?	igen*	igen*	nem	igen, ha $l \geqq d$	igen
idő-bonyolultság	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	b^m	b^l	b^d
tár-bonyolultság	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	bm	bl	bd
optimális?	igen*	igen	nem	nem	igen*

Ismétlődő állapotok

Ha nem ismerjük fel az ismétlő állapotokat, egy lineáris probléma exponenciálissá válhat!





Gráfkeresés

```
function Graph-Search(problem, fringe): megoldás vagy „sikertelen”
    closed = ∅
    fringe = Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return „sikertelen”
        node = Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then return node
        if State[node] is not in closed then
            closed += State[node]
            fringe = InsertAll(Expand(node, problem), fringe)
    end
```

AIMA kód

```
def graph_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    frontier.append(Node(problem.initial))
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return None
```

Összefoglalás

- A probléma megfogalmazás rendszerint elszakad a valós problémától, absztrakt szinten megad egy állapotteret, melyben végrehajtható a kereső algoritmus.
- Különböző típusú problémákra különböző nem informált keresési stratégiák léteznek.
- Az iteratívan mélyülő mélységi keresés lineáris memóriát használ (akár a cache-ben is elfér), így rendszerint sokkal gyorsabban végez, mint a többi nem informált algoritmus.
- A gráfkeresés exponenciálisan hatékonyabb lehet, mint a fakeresés.