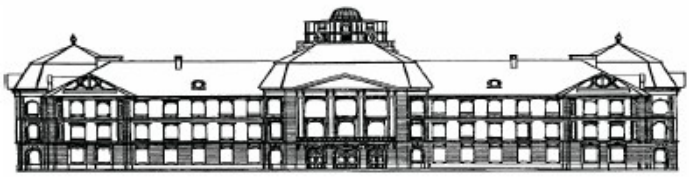


Eszterházy Károly Főiskola
Matematikai és Informatikai Intézet

# A MESTERSÉGES INTELLIGENCIA KÉRDÉSEI A KÖZÉPISKOLAI OKTATÁSBAN

*DR. KOVÁSZNAI GERGELY*
*ÉS*
*DR. KUSPER GÁBOR*

*JEGYZETE*

ÚJ SZÉCHENYI TERV

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638

MAGYARORSZÁG MEGÚJUL

A projekt az Európai Unió támogatásával, az Európai Regionális Fejlesztési Alap társfinanszírozásával valósul meg.

# Tartalomjegyzék

# 1. Bevezetés

Bizonyára felmerül mindenkiben a kérdés, hogy mi is a mesterséges intelligencia. Ilyenkor egy matematikai műveltségű ifjú kolléga rögtön válaszol: az attól függ, hogy mi a definíció. Ha azt nevezzük mesterséges intelligenciának, hogy a sakkban legyőz minket a számítógép, akkor nagyon közel állunk a mesterséges intelligencia megvalósításához. Ha az a definíció, hogy el kell vezetni egy terepjárót a sivatagon keresztül A és B pont között, akkor is jó úton jár a megvalósulás felé a mesterséges intelligencia. Viszont, ha az az elvárásunk, hogy a gép megértse, amit mondunk, attól még nagyon messze állunk.

Ez a jegyzet a mesterséges intelligenciát az első értelemben használja. Olyan „okos” algoritmusokat fogunk közölni, amelyekkel, úgynevezett gráf keresési feladatokat tudunk megoldani. Azokat a feladatokat, amelyek átírhatók gráf kereséssé (ilyen például a sakk), a számítógép meg tudja oldani.

Sajnos a számítógép nem lesz a hétköznapi értelemben okos, ha implementáljuk ezeket az algoritmusokat, legfeljebb csak arra lesz képes, hogy szisztematikusan átvizsgáljon egy gráfot megoldást keresve. Tehát a számítógépünk marad olyan földbuta, mint volt, de kihasználjuk a számítógép két jó tulajdonságát (nincs is neki több:), mégpedig:

- A számítógép gyorsan végez algebrai műveleteket (összeadás, kivonás stb. ).
- Ezeket pontosan végzi el.

Tehát azt használjuk ki, hogy az olyan feladatoknál, melyeket már egy ember nehezen tud átlátni, mint például a Rubik kocka kirakása, a problémát reprezentáló gráf a számítógép számára még relatíve kicsi, és így a gráfkereső algoritmusok által előírt lépéseket gyorsan és pontosan végrehajtva gyorsan kirakja a kockát és a pontosság miatt biztosak lehetünk benne, hogy a megoldás jó.

Ugyanakkor könnyen találhatunk olyan problémát, amelynek gráf reprezentációja olyan nagy, hogy a nagyon gyors számítógépünk se képes gyorsan megtalálni a hatalmas gráfban a megoldást. Itt jön jegyzetünk lényege, a mesterséges intelligencia emberi kreativitást igénylő oldala. Úgy reprezentálni egy problémát, hogy a gráf reprezentációja kicsi maradjon. Ez az a feladat, amelyet már középiskolában fejleszteni kell. Ehhez a következő kompetenciák fejlesztése szükséges:

- Modellalkotás a valóság absztrakciójával
- Rendszerszemlélet

Talán érdemes lenne hozzávenni a fenti listához az algoritmikus gondolkozást, ami a jegyzetben ismertetett algoritmusok megvalósításához, fejben történő végigfuttatásához szükséges. Erre egy későbbi fejezetben térünk ki.

Egy feladat megoldása mesterséges intelligencia alkalmazása esetén is a következő ábra szerint történik.

- A valós problémát modellezzük.
- A modellezett problémát megoldjuk.
- A modellben megtalált megoldás segítségével megoldjuk a valós problémát.

Minden lépést más-más tudományág segít. Az első lépésnél a valóságleírást segítő tudományok, fizika, kémia, ... vannak a segítségünkre. A második lépés már egy absztrakt fogalomrendszerrel dolgozik, ahol a logika és a matematika segítségével dolgozhatunk az absztrakt objektumokon. Végül a mérnöki tudományok, az informatika segít átültetni a modellbéli megoldást a valóságra.

Ez mind szép, de miért nem lehet a valós problémát rögtön a valóságban megoldani? Miért kell

modellezni? Erre könnyű a válasz. A valóságban általában nagyon nehéz és drága próbálgatni. Ha a jól ismert 8 királynő problémát egyenként egytonnás vas királynőkkel kellene játszani, akkor egy masszív emelő darura is szükségünk lenne és a próbálgatásra rámenne egy-két napunk és néhány száz liter dízelolaj, mire megtalálnánk a megoldást. Egy absztrakt térben egyszerűbb és olcsóbb megkeresni a megoldást. Ezért van szükség a modellezésre.

Mi garantálja, hogy az absztrakt térben megtalált megoldás működni fog a valóságban? Azaz, mi garantálja, hogy az így megépített ház nem omlik össze? Ez a kérdés nehéz. A válaszhoz nézzük meg az egyes lépéseket részletesen.

A valós problémát modellezzük:

- A probléma szempontjából lényeges részleteket felnagyítjuk, a lényegteleneket elhanyagoljuk.
- A fontos részleteket meg kell számolnunk, mérnünk.
- Fel kell ismernünk azokat a lehetséges „operátorokat”, amelyekkel megváltoztatható a valóság.

A valós probléma modellezését a mesterséges intelligencia állapottér-reprezentációnak hívja. Ezzel külön fejezet foglalkozik. Itt most az „összedől-e a ház” kérdés szempontjából foglalkozunk ezzel a kérdéssel. Sajnos itt el lehet rontani a házat, hiszen ha elhanyagolunk egy fontos részletet, pl. falvastagság, akkor összedőlhet a ház. Hogyan jelenik meg ez a probléma a középiskolában? Szerencsére általában egy szöveges feladatként, amiben ritka, hogy felesleges részlet lenne megadva. A feladat kiírója általában a másik oldalról nehezíti a feladatot, olyan részletet is észre kell vennünk a megoldáshoz, ami el van rejtve a szövegben.

Azt is fontos tudnunk, hogy a valóság mérése mindig hibával terhelt. A numerikus matematika eszközeivel megadható, hogy a kezdeti hibák milyen mértékben adódnak össze, így megmondható a megoldás hibatartalma is.

A harmadik lépés, az „operátorok” fellelése a legjelentősebb a mesterséges intelligencia szemszögéből. Operátor az, amely a valóság általunk fontosnak tartott részét változtatja, azaz az egyik jól leírható állapotból egy másikba visz. Mesterséges intelligencia szemszögéből operátor, amikor sakkban lépünk, de lehet, hogy nem operátor, ha kivágunk egy fát, hacsak a fák száma nem egy fontos részlet a probléma megoldásának szempontjából.

Látni fogjuk, hogy a modellünk, vagy más néven az állapotterünk, megadható

- a kezdőállapot,
- a célállapotok halmaza,
- a lehetséges állapotok és
- az operátorok megadásával (operátorok elő- és utófeltételével együtt).

A modellezett probléma megoldásához a következő lépéseket kell véghez vinnünk:

- A modellt megoldani képes keretrendszer kiválasztása.
- A modell megadása a keretrendszeren belül.
- A keretrendszer megoldja a problémát.

A modellünket megoldani képes keretrendszer kiválasztása jelenti annak az algoritmusnak a kiválasztását, amely képes megoldani a modellezett problémát. Ez nem feltétlenül jelenti, hogy implementálnunk kell ezt az algoritmust. Pl. a Prolog interpreter visszalépéses keresést alkalmaz. Nekünk csupán implementálni kell a modellt leíró szabályokat Prolog nyelven – és ez már a második lépés. Sajnos ezt a lépést már befolyásolja az is, hogy az állapottér-reprezentációban milyen operátorokat vettünk fel, transzformációs (egy állapotból egy állapotot készítő) vagy probléma redukciós (egy állapotból több állapotot készítő) operátorokat. Ezért az operátorok megadását tekinthetjük a keretrendszer kiválasztását követő lépésnek is. A keretrendszerek sok mindenben különbözhetnek egymástól. Lehetséges csoportosításaik:

- Véges körmentes gráfban lévő megoldást garantáltan megtaláló algoritmusok.
- Véges gráfban lévő megoldást garantáltan megtaláló algoritmusok.

- Valamilyen szempontból optimális megoldást adó gráfkereső algoritmusok.

Ha megvan a megfelelő keretrendszer, már csak implementálni kell a modellt a keretrendszeren belül. Ez általában csak a kezdőállapot, a célfeltétel és az operátorok megadását (elő- és utófeltétellel) jelenti. Ezután csak meg kell nyomnunk a gombot és a keretrendszer megoldja a problémát, mármint ha képes erre. Most tegyük fel, hogy kapunk megoldást. Először is tudnunk kell, hogy mit értünk megoldás alatt. Megoldáson a lépések (azaz operátor alkalmazások) olyan sorozatát értjük, amely kezdő állapotból eljuttat valamely célállapotba. Azaz, ha az a kezdőállapot, hogy van sok építőanyag, a célállapot pedig, hogy áll a tervrajznak megfelelő ház, akkor a megoldás azon lépések sorozata, ahogy fel kell építeni a házat.

Már csak egy kérdés maradt: össze fog-e dőlni a ház? A válasz egyértelműen nem, hogyha az előző lépésben, azaz a modellalkotásban, illetve a következő lépésnél, az absztrakt modell valóságba történő átültetése során nem hibázunk. Erre garancia, hogy a jegyzetben ismertetett algoritmusok helyesek, azaz matematikai logikai eszközökkel bebizonyítható, ha megoldást adnak, akkor ez a modellen belül helyes megoldás. Természetesen a modell implementálását elronthatjuk, mondjuk a célfeltétel megadását, de ha ezt a buktatót elkerüljük, akkor megbízhatunk a megoldásunkban. Legalábbis annyira megbízhatunk, mint amennyire a matematikai logikában megbízunk.

Az utolsó lépés az, hogy a modellben megtalált megoldás segítségével megoldjuk a valós problémát. Nincs más dolgunk, mint a modellbéli megoldás lépéseit egymás után a valóságban is végre kell hajtanunk. Itt szembesülhetünk azzal, hogy egy lépés, ami a modellben egyszerű volt (helyezd a királynőt az A1-es mezőre), az a valóságban nehéz vagy éppen lehetetlen. Ha azt találjuk, hogy lehetetlen a lépés, akkor rossz a modell. Ha nem bízunk meg a modell által nyújtott megoldásban, akkor érdemes azt kicsiben kipróbálni. Ha egyik lépést sem rontottuk el, akkor a ház állni fog, erre garancia az algoritmusok helyessége, és az, hogy a matematikai logika a valóság megfigyelésén alapszik!

# 2. A MESTERSÉGES INTELLIGENCIA TÖRTÉNETE

Az intelligencia tanulmányozása az egyik legősibb tudományos diszciplína. A filozófusok már több mint 2000 éve igyekeznek megérteni, hogy milyen mechanizmus alapján érzékelünk, tanulunk, emlékezünk és gondolkodunk. A 2000 éves filozófiai hagyományból a következtetés és a tanulás elméletei fejlődtek ki, és az a nézet, hogy az elmét egy fizikai rendszer működése hozza létre. Többek között ezen filozófiai elméletek hatására fejlődött ki a matematikából a logika, a valószínűség, a döntéshozatal és a számítások formális elmélete.

Az intelligenciával összefüggő képességek tudományos elemzését a számítógépek megjelenése az 1950-es évek elején valóságos elméleti és kísérleti tudományággá változtatta. Sokan úgy érezték, hogy ezek az „elektronikus szuperagyak" az intelligencia megvalósítása szempontjából határtalan potenciállal rendelkeznek. „Einsteinnél gyorsabb" – ez lett a tipikus újsághír. Ám az intelligens gondolkodás és viselkedés számítógépekkel való modellezése lényegesen nehezebbnek bizonyult, mint azt sokan a legelején képzelték.

*1. ábra: Az 1950-es évek kezdeti optimizmusa: „A világ legkisebb elektronikus agya"* :)

A *mesterséges intelligencia*[1] (MI) az egyik legvégső feladvánnyal foglalkozik. Hogy képes egy kicsi (akár biológiai, akár elektronikus) agy a nála sokkal nagyobb és bonyolultabb világot érzékelni, megérteni, megjósolni és manipulálni? És mi lenne, ha ilyen tulajdonságú valamit szeretnénk megkonstruálni?

Az MI az egyik legújabb tudományos terület. Formálisan 1956-ban keletkezett, amikor a nevét megalkották, ámbár abban az időben a kutatások már mintegy 5 éve folytak. Az MI eddigi történetét három nagy szakaszra osztjuk:

## 2.1. KORAI LELKESEDÉS, NAGY ELVÁRÁSOK (AZ 1960-AS ÉVEK VÉGÉIG)

Az MI korai évei bővelkedtek sikerekben, bizonyos kereteken belül. Ha figyelembe vesszük azoknak az időknek a primitív számítógépeit és programozási eszközeit és azt, hogy még néhány évvel azelőtt is csupán aritmetikai feladatok elvégzésére tartották alkalmasnak a számítógépet, megdöbbentő volt, hogy a számítógép akár csak távolból is okosnak tűnő dologra képes.

Ebben az időszakban a kutatók nagyratörő terveket fogalmaztak meg (világbajnok sakkprogram,

1 angolul: Artificial Intelligence (AI)

univerzális gépi fordítás), a kutatás fő irányának pedig az általános célú problémamegoldó módszerek kidolgozását tekintették. Allen Newell és Herbert Simon megalkottak egy általános problémamegoldó programot (*General Program Solver*, GPS), mely talán az első olyan szoftver volt, mely az emberi problémamegoldás protokolljait imitálta.

Ebben az időszakban születtek az első tételbizonyító alkalmazások. Ilyen volt Herbert Gelernter geometriai tételbizonyítója (*Geometry Theorem Prover*), mely explicit módon reprezentált axiómákra támaszkodva tételeket bizonyított.

Arthur Samuel dámajátékot játszó programot írt, melynek játékereje végül a versenyzői szintet is elérte. Samuel a programját tanulási képességgel ruházta fel. A program kezdetben kezdő szinten játszott, de néhány napnyi önmagával lejátszott játszma után nagyon erős emberi versenyeken is méltó ellenfélnek számított. Samuelnek ezzel sikerült cáfolnia azt, hogy a számítógép csak arra képes, amire utasítják, hiszen programja gyorsan megtanult nála is jobban játszani.

1958-ban John McCarthy megalkotta a *Lisp* programozási nyelvet, mely elsődleges MI-programozási nyelvvé nőtte ki magát. A Lisp a második legrégebbi programozási nyelv, amely még használatban van[2].

## 2.2. Kiábrándulás és a tudásalapú rendszerek (az 1980-as évek végéig)

Az MI korai időszakának általános célú programjai általában csak viszonylag egyszerű feladatokat oldottak meg hatékonyan, azonban szánalmasan csődöt mondtak, amikor szélesebb körben vagy netán nehezebb problémákra akarták bevetni őket. A nehézség forrása egyrészt az volt, hogy a korai programok az általuk kezelt problémákról kevés vagy szinte semmi tudással nem rendelkeztek, és csupán egyszerű szintaktikai manipulálással értek el sikereket. Egy tipikusnak mondható történet a korai gépi fordítással kapcsolatos. Az USA-ban a Szputnyik 1957-es fellövését követően meggyorsították az orosz tudományos cikkek angolra fordítását. Kezdetben úgy vélték, hogy az angol és az orosz nyelvtanra alapozó egyszerű szintaktikai transzformációk és a szóbehelyettesítés elegendő lesz a mondat pontos értelmének meghatározásához. Az anekdota szerint „A szellem készséges, de a test gyenge” híres mondat visszafordításakor a következő szöveget kapták: „Jó a vodka, de romlott a hús.” Ez világosan mutatta a tapasztalt nehézségeket és azt, hogy a fordításhoz az adott téma általános ismerete szükséges, hogy feloldhassuk a kétértelműségeket.

A másik nehézséget az jelentette, hogy sok olyan probléma, melyet az MI által kíséreltek megoldani, kezelhetetlen volt. A korai MI-programok többsége úgy dolgozott, hogy a megoldandó problémára vonatkozó alapvető tényekből kiindulva lépésszekvenciákat próbált ki, a különféle lépéskombinációkkal kísérletezve, amíg rá nem lelt a megoldásra. A korai programok azért voltak használhatók, mert az általuk kezelt világok kevés objektumot tartalmaztak. A bonyolultságelméletben az *NP-teljesség* definiálása előtt (Steven Cook, 1971; Richard Karp, 1972) általában azt gondolták, hogy ezeket a programokat nagyobb problémákra „felskálázni” csupán gyorsabb hardver és nagyobb memória kérdése. Ezt az NP-teljességre vonatkozó eredmények elméletben cáfolták meg. A korai időszakban az MI nem volt képes leküzdeni a „kombinatorikus robbanást”, melynek folyományaként az MI-kutatásokat sok helyen leállították.

Az 1960-as évek végétől ún. *szakértői rendszerek*[3] kifejlesztésére helyeződött át a hangsúly. Ezek a rendszerek az általuk kezelt tárgyterületről (szabályalapú) tudásbázissal rendelkeznek, melyen egy következtetőkomponens végez deduktív lépéseket. Ebben az időszakban komoly eredmények születtek a *rezolúciós tételbizonyítás* elméletében (J. A. Robinson, 1965), az ismeretreprezentációs technikák kidolgozásában, illetve a *heurisztikus* keresések és a

2 A FORTRAN csak egy évvel idősebb a Lispnél.

3 angolul: expert systems

*bizonytalanságkezelő* mechanizmusok területén. Az első szakértői rendszerek az orvosi diagnosztika területén születtek meg. Például a MYCIN nevű rendszer 450 szabályával elérte az emberi szakértők hatékonyságát, és a kezdő orvosoknál lényegesen jobb teljesítményt nyújtott.

Az 1970-es évek elején megszületett a *Prolog* logikai programozási nyelv, mely a rezolúciós kalkulus egy változatának számítógépes megvalósítására épül. A Prolog rendkívül elterjedt eszköz szakértői rendszerek fejlesztésében (orvosi, jogi és más területen), de természetes nyelvi elemzők is készültek ezen a nyelven. Ezen korszak nagy eredményeinek egy része a természetes nyelvi feldolgozó programokhoz kapcsolódik, melyek közül számosat használtak már adatbázis-interfészként.

## 2.3. AZ MI IPARRÁ VÁLIK (1980-TÓL)

Az első sikeres szakértői rendszer, az R1 számítógépes rendszereket segített konfigurálni, és 1986-ra a fejlesztő cégnek, a DEC-nek évi 40 millió dollár megtakarítást hozott. 1988-ban a DEC MI-csoportja már 40 szakértői rendszert állított üzembe, és több ilyen rendszeren dolgozott.

1981-ben a japánok meghirdették „*ötödik generációs* számítógép" projektjüket – egy 10 éves tervet a Prolog nyelvet gépi kódként használó, intelligens számítógépes rendszerek építésére. A japán kihívásra válaszul az USA és Európa vezető országai is hasonló célú, hosszútávú kutatásokba kezdtek. Ez a korszak hozta meg az áttörést, mellyel a MI kilépett a laboratóriumok világából, és megkezdődött a MI termékek gyakorlati felhasználása. Számos területen (pl. az orvosi diagnosztika, kémia, geológia, ipari folyamatirányítás, robotika stb.) kezdtek szakértői rendszereket alkalmazni, mely rendszerek sokszor már természetes nyelvi interfészen keresztül voltak használhatók. Mindent egybevetve az MI-iparnak az évi forgalma 1988-ra már 2 milliárd dollárra nőtt.

A szakértői rendszerek mellett új, illetve rég elfeledett technikák is felbukkantak. Ezeknek egy nagy csoportja a statisztikai MI-módszereket foglalja magában, melyek kutatása az 1980-as évek elején kapott egy nagy lökést a *neurális hálók* (újbóli) felfedezésével. Ebbe a körbe tartoznak még a beszédfelismerés és a kézírás-felismerés területén használt *rejtett Markov-modellek*. Szelíd forradalom következett be a robotika, a gépi látás, a gépi tanulás területén.

Napjainkra az MI-technológiák sokszínű képet mutatnak, mely technológiák az iparban, de egyre inkább a mindennapi szolgáltatásokban is teret hódítanak. A mindennapi életünk részévé válnak.

# 3. PROBLÉMAREPREZENTÁCIÓ

## 3.1. ÁLLAPOTTÉR-REPREZENTÁCIÓ

A kérdés először is az, hogy hogyan reprezentáljunk számítógépen egy megoldandó problémát. Miután a reprezentálási technika részleteit kidolgozzuk, már készíthetünk az ilyen típusú reprezentációkon dolgozó algoritmusokat. A továbbiakban az *állapottér-reprezentációt* fogjuk megismerni, mely egy meglehetősen általánosan használható reprezentációs technika. Ráadásul sokfajta problémamegoldó algoritmus ismert állapottér-reprezentációra, ezeknek az ismertetésébe a 3. fejezetben fogunk belemerülni.

Egy probléma reprezentáláshoz meg kell keresnünk az adott probléma világának véges sok, a probléma megoldása során fontosnak vélt tulajdonságát, jellemzőjét (pl. szín, súly, méret, ár, pozíció stb.). Például ha ezeket a jellemzőket rendre a $h_1, \ldots, h_n$ értékek jellemzik (pl. szín: fekete/fehér/piros; hőmérséklet: [-20C°, 40C°] stb.), akkor azt mondjuk, hogy a $(h_1, \ldots, h_n)$ vektorral azonosított *állapotban* van a probléma világa. Ha az $i.$ jellemző által felvehető értékek halmazát $H_i$-vel jelöljük, akkor a probléma világának állapotai elemei a $H_1 \times H_2 \times \ldots \times H_n$ halmaznak.

Miután ily módon meghatároztuk a probléma világának lehetséges állapotait, meg kell adnunk azt a speciális állapotot, mely a probléma világához tartozó jellemzők kezdőértékeit határozza meg. Ezt az állapotot *kezdőállapotnak* nevezzük.

A problémamegoldás során a kezdőállapotból indulva a probléma világának előálló állapotait rendre meg fogjuk változtatni, míg végül valamely számunkra megfelelő *célállapotba* jutunk. Akár több célállapotot is megadhatunk.

Már csak azt kell pontosan specifikálni, hogy mely állapotokat van lehetőségünk megváltoztatni, és hogy ezek megváltoztatása milyen állapotokat eredményez. Az állapotváltozásokat leíró függvényeket *operátoroknak* nevezzük. Természetesen egy operátor nem feltétlenül alkalmazható minden egyes állapotra, ezért az operátorok (mint függvények) értelmezési tartományát az *operátoralkalmazási előfeltételek* segítségével szoktuk megadni.

**1. Definíció:** *Állapottér-reprezentáció* alatt egy $\langle A, k, C, O \rangle$ formális négyest értünk, ahol:

(1) $A$ : az állapotok halmaza, $A \neq \emptyset$ .
(2) $k \in A$ : a kezdőállapot.
(3) $C \subseteq A$ : a célállapotok halmaza.
(4) $O$ : az operátorok halmaza, $O \neq \emptyset$ .
Minden $o \in O$ operátor egy $o : Dom(o) \to A$ függvény, ahol

$$Dom(o) = \{ a \mid a \notin C \ \wedge \ előfeltétel_o(a) \} \subseteq A$$

A $C$ halmaz megadása kétféleképpen történhet:

- Felsorolással (explicit módon): $C = \{ c_1, \ldots, c_m \}$
- Célfeltétel megadásával (implicit módon): $C = \{ a \mid célfeltétel(a) \}$

Az $előfeltétel_o(a)$ és a $célfeltétel(a)$ feltételek *logikai formulák* alakjában írhatók le. Mindkét formulának paramétere az $a$ állapot, az operátoralkalmazási előfeltételnek ezen kívül még az alkalmazandó $o$ operátor is.

A továbbiakban definiálnunk kell, hogy egy állapottér-reprezentált problémának mit értünk a megoldása alatt – hiszen ennek megkeresésére akarunk algoritmusokat készíteni. Egy probléma

megoldásának fogalmát a következő definíciókon keresztül lehet leírni:

**2. Definíció:** Legyen $\langle A, k, C, O \rangle$ egy állapottér-reprezentáció, és legyen $a, a' \in A$ két állapot.

Az $a$-ból az $a'$ *közvetlenül elérhető*, ha van olyan $o \in O$ operátor, hogy $előfeltétel_o(a)$ teljesül és $o(a) = a'$.

Jelölése: $a \underset{o}{\rightarrow} a'$.

**3. Definíció:** Legyen $\langle A, k, C, O \rangle$ egy állapottér-reprezentáció, és legyen $a, a' \in A$ két állapot.

Az $a$-ból az $a'$ *elérhető*, ha van olyan $a_1, a_2, \ldots, a_n \in A$ állapotsorozat, hogy

- $a_1 = a$
- $a_n = a'$
- $\forall i \in \{1,2,\ldots,n-1\}$-re: $a_i \underset{o_i}{\rightarrow} a_{i+1}$ (valamely $o_i \in O$ operátorra)

Jelölése: $a \underset{o_1, o_2, \ldots, o_{n-1}}{\rightarrow} a'$

**4. Definíció:** Az $\langle A, k, C, O \rangle$ probléma *megoldható*, ha $k \underset{o_1, \ldots, o_n}{\rightarrow} c$ valamely $c \in C$ célállapotra. Ekkor az $o_1, \ldots, o_n$ operátorsorozat a probléma egy *megoldása*.

Egyes problémák esetén akár több megoldás is létezik. Ilyenkor érdekes lehet az egyes megoldásokat *költségük* alapján összehasonlítani – és a legkevésbé költséges (legolcsóbb) megoldást kiválasztani közülük. Lehetőségünk van az operátoralkalmazásokhoz költséget rendelni: az $o$ operátor $a$ állapotra alkalmazásának költségét $költség_o(a)$-val jelöljük (feltéve persze, hogy $o$ alkalmazható $a$-ra, vagyis $előfeltétel_o(a)$ teljesül), mely egy *pozitív egész szám*.

**5. Definíció:** Legyen az $\langle A, k, C, O \rangle$ probléma esetén $k \underset{o_1, \ldots, o_n}{\rightarrow} c$ valamely $c \in C$-re. Az $o_1, \ldots, o_n$ megoldás költsége:

$$\sum_{i=1}^{n} költség(o_i, a_i) .$$

Vagyis egy megoldás költsége a megoldásban szereplő *operátoralkalmazások költségének az összege*. Sok probléma esetén az operátoralkalmazások költsége *egységnyi*, vagyis $költség(o, a) = 1$ minden $o$ operátorra és minden $a$ állapotra. Ekkor a megoldás költsége értelemszerűen nem lesz más, mint az *alkalmazott operátorok száma*.

## 3.2. ÁLLAPOTTÉR-GRÁF

Egy probléma állapottér-reprezentációjának szemléltetésére a legjobb eszköz az *állapottér-gráf*.

**6. Definíció:** Egy $\langle A, k, C, O \rangle$ probléma *állapottér-gráfja* az $\langle A, E \rangle$ gráf[4], ahol $(a, a') \in E$ és $(a, a')$ címkéje $o$ akkor és csak akkor, ha $a \underset{o}{\rightarrow} a'$.

Vagyis az állapottér-gráf csúcsai *maguk az állapotok*, illetve két csúcs között akkor és csak akkor húzunk élt, ha az egyik csúcsból (mint állapotból) a másik csúcs (mint állapot) *közvetlenül elérhető*. Az éleket a közvetlen elérhetőséget lehetővé tevő *operátorral* címkézzük.

Könnyen látható, hogy a probléma egy *megoldása* nem más, mint a $k$ csúcsból (amit *startcsúcsnak* is nevezünk) valamely $c \in C$ csúcsba (amit *célcsúcsnak* is nevezünk) vezető *út*.

4 A megszokott módon: $A$ a gráf csúcsainak halmaza, $E$ pedig a gráf éleinek halmaza.

Pontosabban: a megoldás az ezen utat alkotó élek *címkéinek* (azaz operátoroknak) a sorozata.

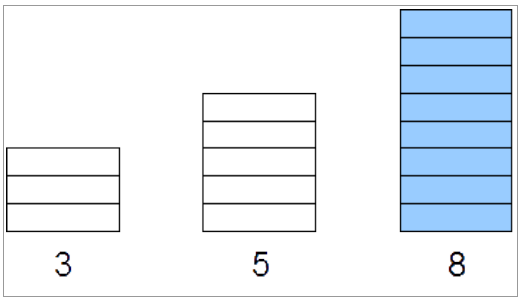
A 4. fejezetben meg fogunk ismerkedni egy jó pár megoldáskereső algoritmussal. Ezekről a keresőkről általánosságban elmondható, hogy mindegyikük az adott feladat *állapottér-gráfját járja be* valamilyen mértékben, és a gráfban keresi a megoldást reprezentáló utat.

## 3.3. PÉLDÁK

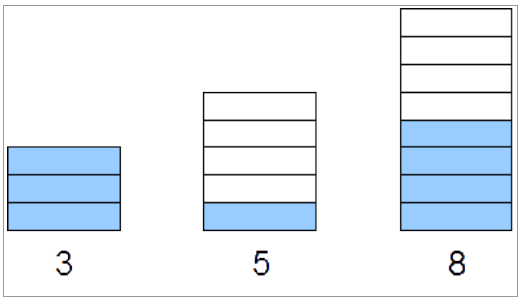
Ebben a fejezetben nevezetes problémák lehetséges állapottér-reprezentációját mutatom be.

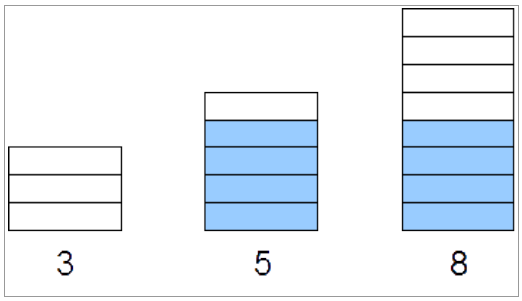
### 3.3.1. 3 KANCSÓ

Adott 3 kancsó, egyenként 3, 5 és 8 liter űrtartalmúak. A kancsókon nincs skála, azaz csak az űrtartalmuk az, amit biztosan tudunk róluk. Kezdetben a 8 literes kancsónk tele van vízzel, a másik kettő üres:




Egyik kancsóból áttölthetünk vizet egy másikba, és a cél az, hogy valamelyik kancsóba *pontosan* 4 liter vizet mérjünk ki. Hogy a másik két kancsóba a végén egyenként hány liter víz kerül, irreleváns. Íme két lehetséges célállapot:




Mivel a kancsókon nincs skála, és más eszközök nem állnak rendelkezésünkre, egy $A$ kancsóból egy $B$ kancsóba csak kétféleképpen tudunk áttöltögetni:

- Az $A$ -ból az összes vizet áttöltjük $B$ -be.
- A $B$ kancsót teletöltjük (és lehet, hogy $A$ -ban még marad víz).

Sorszámozzuk meg a kancsókat: legyen az 1-es sorszámú a legkisebb, 2-es a középső, és 3-as a legnagyobb! Általánosítsuk a feladatot akármilyen űrtartalmú kancsókra a következőképpen: vezessünk be egy 3- elemű vektort (állapottéren kívüli konstans objektumként), melyben az egyes kancsók űrtartalmát tároljuk:

$$max = (3,5,8)$$

- **Állapotok halmaza:** Az állapotokban tároljuk el, hogy melyik kancsóban hány liter víz van! Legyen tehát az állapot egy rendezett hármas, melynek az $i.$ tagja az $i$ -vel jelölt

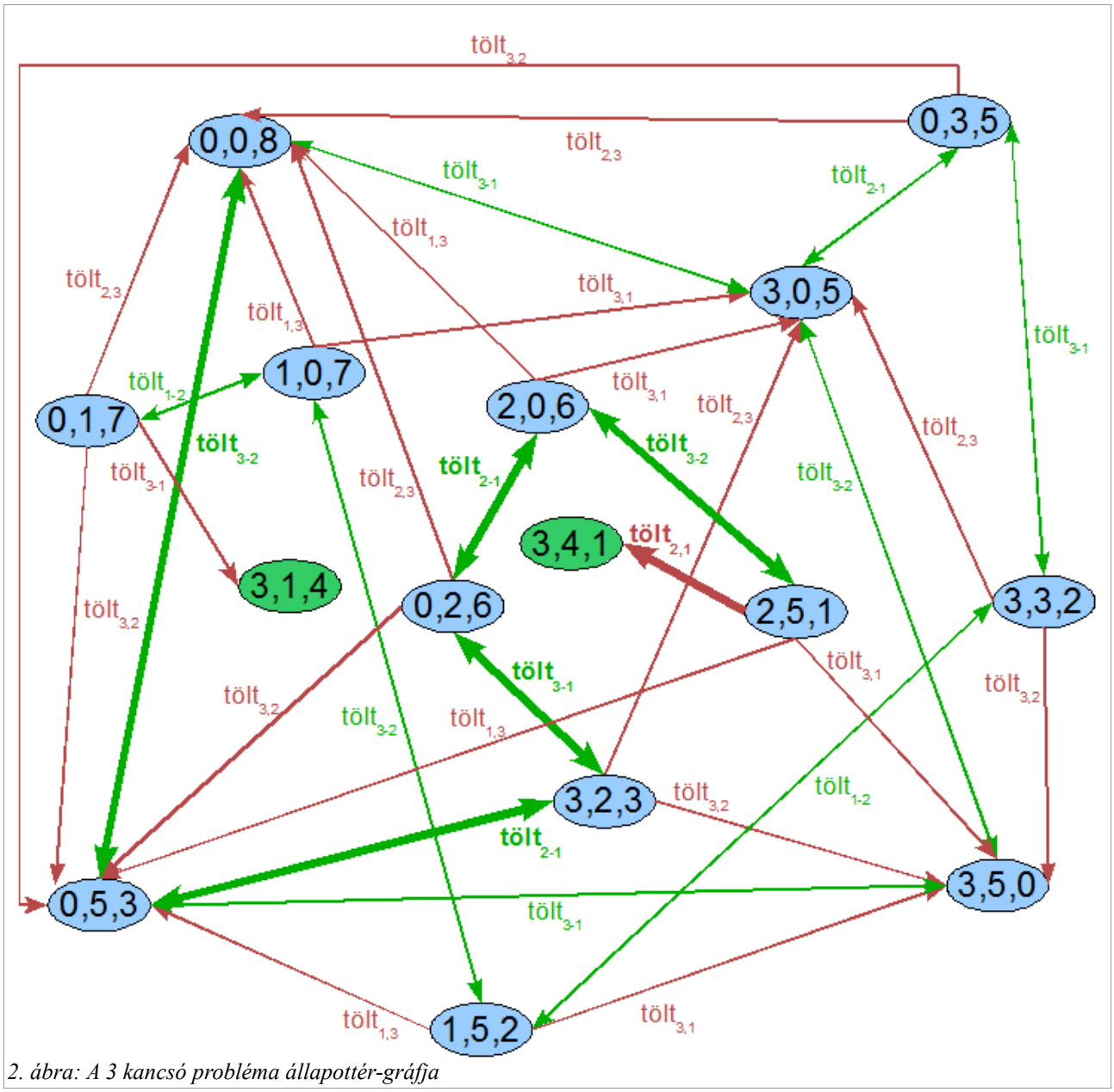
kancsóról mondja meg, hogy hány liter víz van benne.
Tehát az állapotok halmaza a következő:
$A=\{(a_1,a_2,a_3) \mid 0\le a_i\le max_i\}$
ahol minden $a_i$ egész szám.

- **Kezdőállapot:** Kezdetben a 3-as kancsó tele van, az összes többi üres. Tehát a kezdőállapot:
$$k=(0,0,max_3)$$
- **Célállapotok halmaza:** Több célállapotunk van, így célfeltétel segítségével definiáljuk a célállapotok halmazát:
$$C=\{(a_1,a_2,a_3)\in A \mid \exists i \;\; a_i=4\}$$
- **Operátorok halmaza:** Operátoraink egy kancsóból ( $i$ -vel jelöljük) egy másik kancsóba ( $j$ -vel jelöljük) való áttöltést valósítanak meg. Kikötjük még, hogy a forráskancsó ( $i$ ) és a célkancsó ( $j$ ) ne egyezzenek meg. Tehát az operátoraink halmaza:
$$O=\{tölt_{i,j} \mid i,j\in\{1,2,3\} \;\wedge\; i\neq j\}$$
- **Alkalmazási előfeltétel:** Fogalmazzuk meg, hogy egy $tölt_{i,j}$ operátor mikor alkalmazható egy $(a_1,a_2,a_3)$ állapotra! Célszerű a következő feltételeket kikötni:
  - Az $i.$ kancsó nem üres.
  - A $j.$ kancsó nincs tele.

  Tehát a $tölt_{i,j}$ operátor alkalmazási előfeltétele az $(a_1,a_2,a_3)$ állapotra:
$$a_i\neq 0 \;\wedge\; a_j\neq max_j$$
- **Alkalmazási függvény:** Adjuk meg, hogy a $tölt_{i,j}$ operátor az $(a_1,a_2,a_3)$ állapotból milyen $(a'_1,a'_2,a'_3)$ állapotot állít elő! Kérdés, hogy vajon hány liter vizet tudunk áttölteni az $i.$ kancsóból a $j$ -be. Mivel a $j.$ kancsóba aktuálisan
$$max_j-a_j$$
liter víz fér, így a
$$min(a_i \;,\; max_j-a_j)$$
minimumképzéssel tudjuk az áttölthető mennyiséget kiszámolni, melyet jelöljünk el $T$-vel! Azaz:
$tölt_{i,j}(a_1,a_2,a_3)=(a'_1,a'_2,a'_3)$, ahol
$$a'_m=\begin{cases} a_i-T & ,\text{ha } m=i \\ a_j+T & ,\text{ha } m=j \\ a_m & ,\text{egyébként}\end{cases} \qquad \text{ahol } m\in\{1,2,3\}$$

***ÁLLAPOTTÉR-GRÁF***

A fenti állapottér-reprezentáció állapottér-gráfja a 2. ábrán látható.




*2. ábra: A 3 kancsó probléma állapottér-gráfja*

A gráfban a piros élek egyirányú élek, a zöld élek kétirányúak. Természetesen a kétirányú éleket inkább lett volna helyes ténylegesen 2 db egyirányú élként megrajzolni, azonban helyhiány miatt mégis maradjunk a kétirányú éleknél! Lehet látni, hogy a kétirányú élek címkéi $tölt_{i,j1-j2}$ formájúak, vagyis eltérnek az állapottér-reprezentációban megadott $tölt_{i,j}$ formától. Ennek oka, hogy egy $tölt_{i,j1-j2}$ címke tulajdonképpen *egyszerre 2 db operátort* kódol: a $tölt_{i,j1}$ és a $tölt_{i,j2}$ operátorokat, melyek közül az egyik az egyik irányú él, a másik az ellentétes irányú él címkéje lenne.

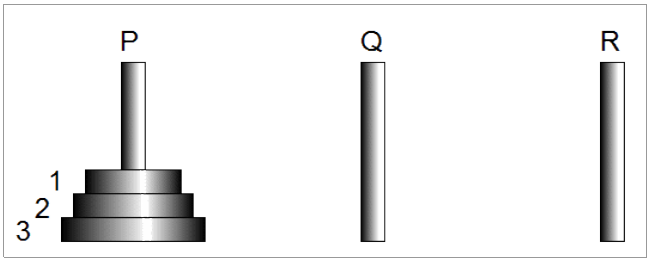
A zöld csúcsok a *célállapotok*. Az ábrán vastagon szedett élek az egyik *megoldást* írják le, és ez nem más, mint a következő operátorsorozat:

$$tölt_{3,2}\ ,\ tölt_{2,1}\ ,\ tölt_{1,3}\ ,\ tölt_{2,1}\ ,\ tölt_{3,2}\ ,\ tölt_{2,1}$$

Vegyük észre, hogy a problémának több megoldása is van. Valamint az is szembetűnő, hogy az állapottér-gráf köröket tartalmaz, melyek a megoldás(ok) megtalálását megnehezíthetik.

## 3.3.2. HANOI TORNYAI

Adott 3 különböző átmérőjű, lyukas közepű korong. Ezeket rá tudjuk ejteni 3 függőleges rúdra. Fontos, hogy ha egy korong alatt van egy másik korong, akkor annak nagyobb átmérőjűnek kell lennie. A rudakat *P, Q, R*-rel, a korongokat átmérő szerint növekvő sorrendben *1,2,3*-mal jelöljük. A korongok kezdeti helyzete a következő ábrán látható:




Egy korongot áthelyezhetünk egy másik rúdra, ha a korong

(5) legfelül helyezkedik el az aktuális rúdján, és
(6) a célrúdon az áthelyezés után is megmarad a korongok nagyság szerinti rendezése.

A cél az összes korongot áthelyezni a *R* rúdra.

A feladat *állapottér-reprezentációját* a következőképpen alkotjuk meg:

- **Állapotok halmaza:** Az állapotokban azt fogjuk tárolni, hogy a korongok mely rudakon vannak aktuálisan. Azaz az állapot egy $(a_1, a_2, a_3)$ vektor lesz, ahol $a_i$ az $i$ korong pozíciója (azaz *P, Q* vagy *R*). Azaz:

$$A = \{(a_1, a_2, a_3) \mid a_i \in \{P, Q, R\}\}$$

- **Kezdőállapot:** Kezdetben mindegyik korong a *P* rúdon van. Azaz:

$$k = (P, P, P)$$

- **Célállapotok halmaza:** A cél, hogy mind a három korongot az *R* rúdra juttassuk. Vagyis ebben a feladatban egyetlen célállapotunk van. Azaz:

$$C = \{(R, R, R)\}$$

- **Operátorok halmaza:** Az operátorok kétfajta információt fognak magukban foglalni:
  - melyik korongot helyezzük át
  - melyik rúdra.

  Azaz:

$$O = \{át_{melyiket, hova} \mid melyiket \in \{1,2,3\} \ , \ hova \in \{P, Q, R\}\}$$

- **Alkalmazási előfeltétel:** Vegyük valamely $át_{melyiket, hova}$ operátort! Vizsgáljuk meg, hogy mikor lehet alkalmazni egy $(a_1, a_2, a_3)$ állapot esetén! A következőt két feltételt kell ebbe belefoglalni:

  (1) A *melyiket* korong az $a_{melyiket}$ rúd legtetején van.

  (2) A *melyiket* korong a *hova* rúd legtetejére kerül.

  Azaz azt kell logikai formula alakjában megfogalmazni, hogy egyik *melyiket*-nél kisebb átmérőjű korong (ha egyáltalán van ilyen) sincs se az $a_{melyiket}$ rúdon, se a *hova* rúdon.

  Ehhez érdemes még hozzávenni azt a feltételt is, hogy nem akarom a korongomat ugyanarra a rúdra rakni, ahonnan éppen elvettem. Ez a feltétel nem feltétlenül szükséges, viszont gyorsítani fogja a keresést (triviális köröket vág ki az állapottér-gráfból).

Tehát az *alkalmazási előfeltétel:*

$$a_{melyiket} \neq hova \wedge \forall\, i \left(i < melyiket \;\Rightarrow\; a_i \neq a_{melyiket} \wedge a_i \neq hova\right)$$

- **Alkalmazási függvény:** Vegyük valamely $át_{melyiket,\,hova}$ operátort! Ha teljesül az alkalmazási előfeltétele az $(a_1, a_2, a_3)$ állapot esetén, akkor alkalmazhatjuk erre az állapotra. Azt kell megfogalmaznunk, hogy hogyan fog kinézni az ennek eredményeként előálló $(a'_1, a'_2, a'_3)$ állapot.

  Azt kell megfogalmaznunk, hogy a *melyiket* korong átkerül a *hova* rúdra, míg a többi korong a helyén marad. Azaz:

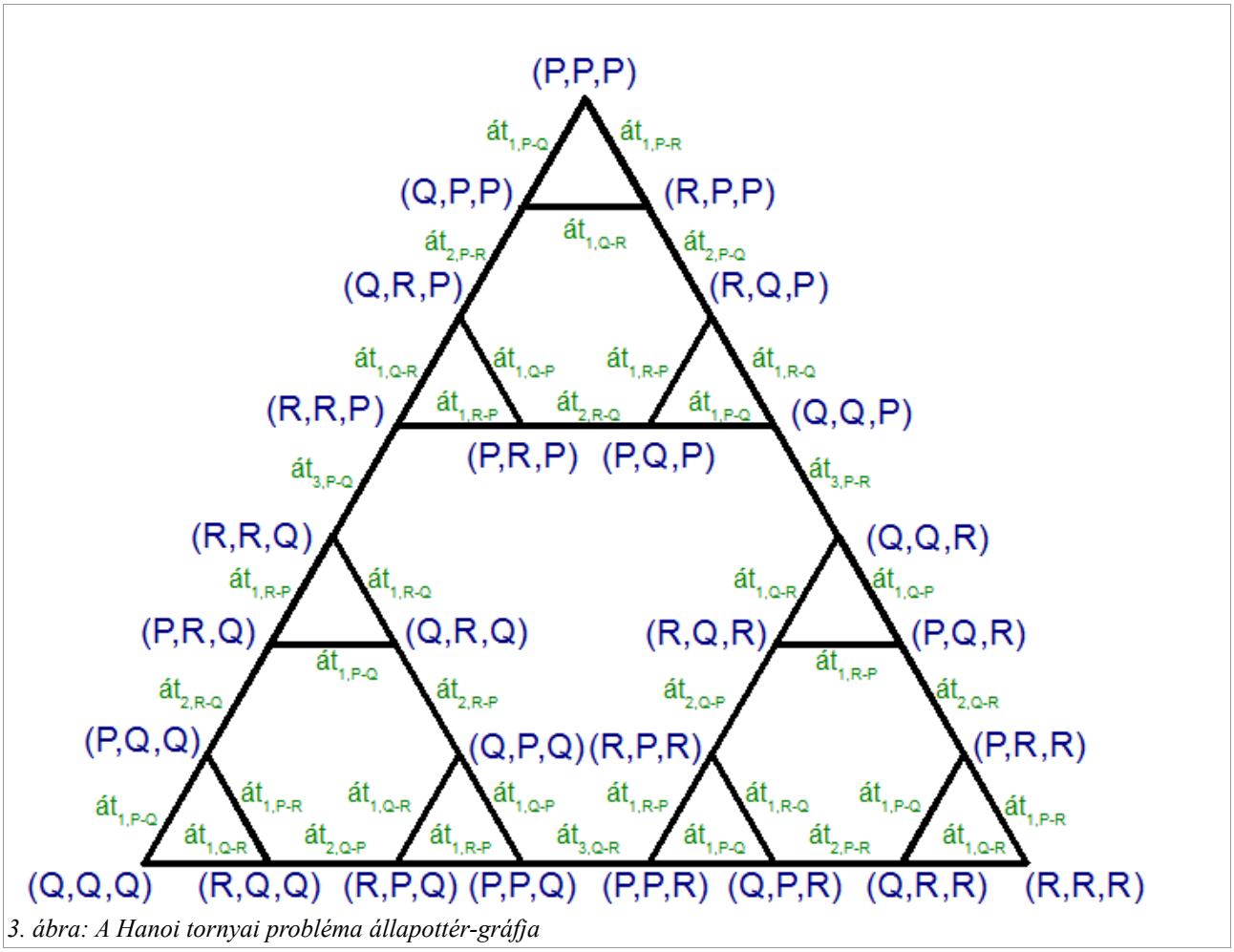
$$át_{melyiket,\,hova}(a_1, a_2, a_3) = (a'_1, a'_2, a'_3)\text{, ahol}$$

$$a'_i = \begin{cases} hova & \text{, ha } i = melyiket \\ a_i & \text{, egyébként} \end{cases} \qquad \text{ahol } i \in \{1,2,3\}$$

Fontos: az új állapotnak az *összes* elemét definiálnunk kell, nem csak azt, ami megváltozik!

### ÁLLAPOTTÉR-GRÁF

A fenti állapottér-reprezentáció állapottér-gráfja a 3. ábrán látható.



*3. ábra: A Hanoi tornyai probléma állapottér-gráfja*

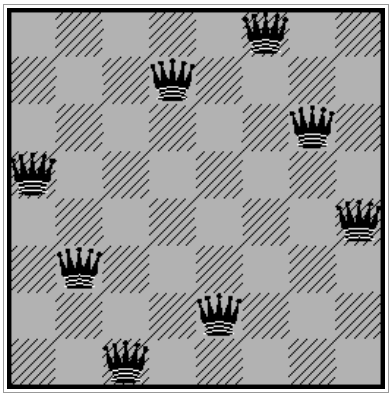
A gráfban természetesen minden él kétirányú, ezek címkéi az előző fejezetben megszokott módon értelmezhetőek: egy $át_{i,\,j1-j2}$ címke az $át_{i,\,j1}$ és az $át_{i,\,j2}$ operátorokat hivatkozza.

Az ábrából egyértelműen látszik, hogy a probléma *optimális* (azaz: legrövidebb) *megoldását* a

nagy háromszög jobb szélső oldala adja, vagyis 7 lépésből (azaz: operátorból) áll az optimális megoldás.

### 3.3.3. 8 KIRÁLYNŐ

Helyezzünk el a sakktáblán 8 db királynőt úgy, hogy egyik se üsse a másikat. Egy lehetséges megoldás:



Általánosítsuk a feladatot $N \times N$-es $(N \geqslant 1)$ sakktáblára, melyen értelemszerűen $N$ db királynőt kell elhelyeznünk. Azaz az $N$ egy állapottéren kívüli konstansként lesz megadva.

Az *állapottér-reprezentáció* alapötlete a következő: mivel a sakktábla minden sorába pontosan 1-1 db királynőt fogunk lerakni, a feladat megoldását oly módon is elvégezhetjük, hogy az egyes királynőket *soronként haladva* rakjuk fel a táblára. Azaz először az 1. sorba lerakunk egy királynőt, majd a 2. sorba egy másikat úgy, hogy az az 1. sorban levővel ne legyen ütésben. Ily módon az *i.* lépésben az *i.* sorba rakunk le egy királynőt, ellenőrizve, hogy az ne legyen az előző $i-1$ db királynővel ütésben.

- **Állapotok halmaza:** Az állapotokban tároljuk el az egyes sorokba letett királynők *soron belüli pozícióját*! Legyen tehát az állapotban egy $N$-elemű vektor, melynek az $i$. tagja megmondja, hogy az *i.* sorban hányadik oszlopban található a letett királynő. Ha az adott sorba még nem raktam le királynőt, akkor a vektorban ott 0 álljon.
  Ezen kívül az állapotokban tároljuk azt is, hogy *hányadik sorba* rakom le a *következő királynőt*.
  Tehát:
  $$A = \{(a_1, a_2, \ldots, a_N, s) \mid 0 \leqslant a_i \leqslant N, 1 \leqslant s \leqslant N+1\}$$
  Az $s$ értékeként az $N+1$ már nemlétező sorindex, melyet csak azért engedünk meg, hogy a terminálási feltételeket majd tesztelni tudjuk.
- **Kezdőállapot:** Kezdetben a tábla üres. Tehát a kezdőállapot:
  $$k = (0,0,\ldots,0,1)$$
- **Célállapotok halmaza:**
  Több célállapotunk is van. Ha az $s$ már nemlétező sorindexet tartalmaz, megoldást találtunk. Azaz a célállapotok halmaza:
  $$C = \{(a_1, \ldots, a_N, N+1) \in A\}$$
- **Operátorok halmaza:**
  Operátoraink egy királynő lerakását fogják leírni az *s.* sorba. Az operátorok csak egy bemenő adatot várnak: azt az oszlopindexet, ahová az *s.* soron belül a királynőt rakjuk. Tehát operátoraink halmaza:

$$O=\{lerak_i \mid 1\leqslant i\leqslant 8\}$$

- **Alkalmazási előfeltétel:** Fogalmazzuk meg, hogy egy $lerak_i$ operátor mikor alkalmazható egy $(a_1,\ldots,a_8,s)$ állapotra! Akkor, ha a most lerakandó királynő
  - nincs egy oszlopban semelyik korábban lerakott királynővel. Tehát azt kell vizsgálnunk, hogy $i$ értéke nem szerepel-e már az állapotban az $s.$ elem előtt. Azaz:
    $$\text{minden } m<s \text{ esetén: } a_m\neq i$$
  - átlósan nem üti semelyik korábban lerakott királynőt. Az átlós ütéseket a legkönnyebb úgy vizsgálni, hogy vesszük a két vizsgált királynő sorindexei különbségének az abszolút értékét, és összehasonlítjuk az oszlopindexeik különbségének az abszolút értékével. Ha ezek egyenlők, akkor a két királynő üti egymást. A most lerakandó királynő sorindexe $s$, oszlopindexe $i$. Azaz:
    $$\text{minden } m<s \text{ esetén: } |s-m|\neq|i-a_m|$$

  Tehát a $lerak_i$ operátoralkalmazás előfeltétele az $(a_1,\ldots,a_8,s)$ állapotra:
  $$\forall\, m\big(m<s \;\Rightarrow\; a_m\neq i \;\wedge\; |s-m|\neq|i-a_m|\big),$$
  $$\text{ahol } 1\leqslant m\leqslant 8$$

- **Alkalmazási függvény:** Adjuk meg, hogy a $lerak_i$ operátor az $(a_1,\ldots,a_8,s)$ állapotból milyen $(a'_1,\ldots,a'_8,s')$ állapotot állít elő! Csupán annyit kell változtatnunk az új állapotban, hogy
  - az állapot $s.$ elemébe beírjuk az $i$-t, és
  - az $s$ értékét eggyel megnöveljük.

  Tehát:

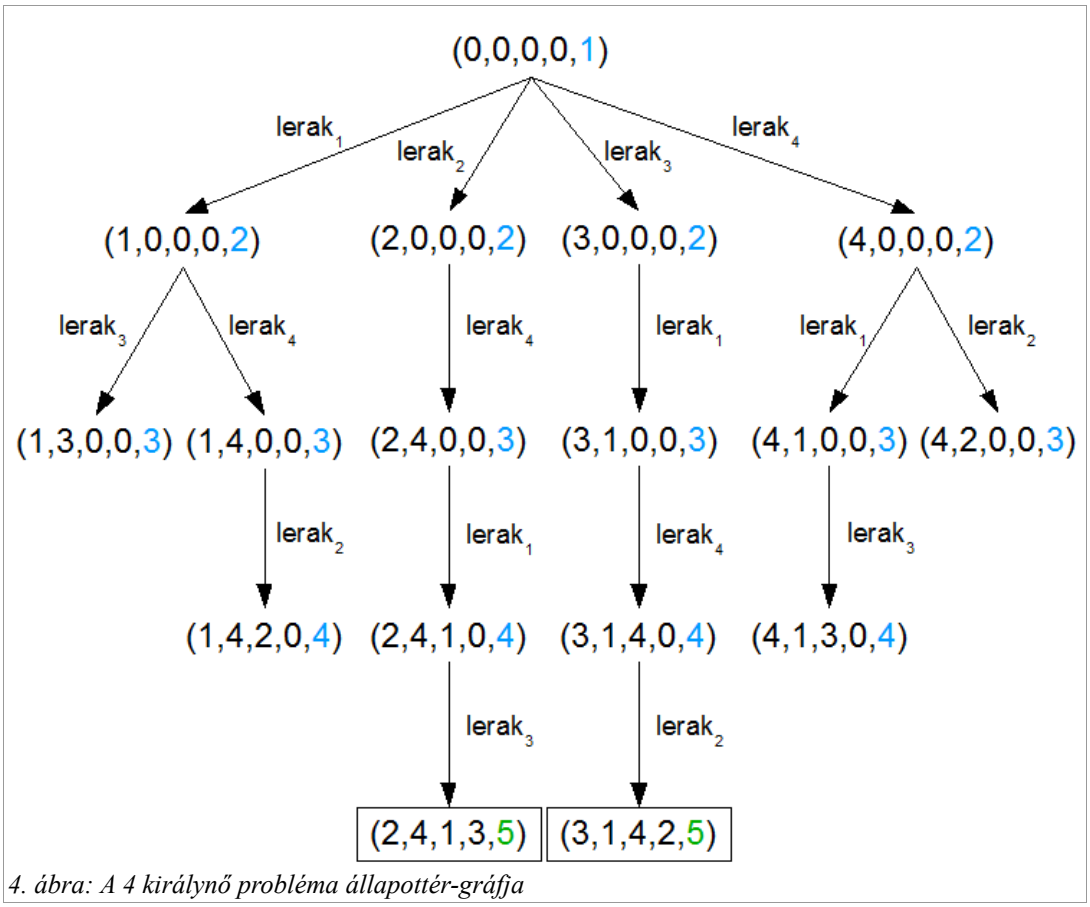
  $lerak_i(a_1,\ldots,a_8,s)=(a'_1,\ldots,a'_8,s')$, ahol:

$$a'_m = \begin{cases} i & \text{,ha } m=s \\ a_m & \text{,egyébként} \end{cases} \quad \text{,ahol } m\in\{1,2,3\}$$
$$s' = s+1$$

***ÁLLAPOTTÉR-GRÁF***

A fenti állapottér-reprezentáció állapottér-gráfja a 4. ábrán látható, $N=4$ esetre. Ebben az esetben 2 megoldása van a problémának.

Vegyük észre, hogy a feladat *minden megoldása* biztosan $N$ hosszú. Azt is fontos megjegyezni, hogy az állapottér-gráfban *nincs kör*, vagyis az állapottér-reprezentáció elemeinek ügyes megválasztásával a köröket sikerült teljesen száműzni a gráfból, aminek a megoldás keresésekor látjuk majd hasznát.




*4. ábra: A 4 királynő probléma állapottér-gráfja*

# 4. MEGOLDÁSKERESŐ RENDSZEREK

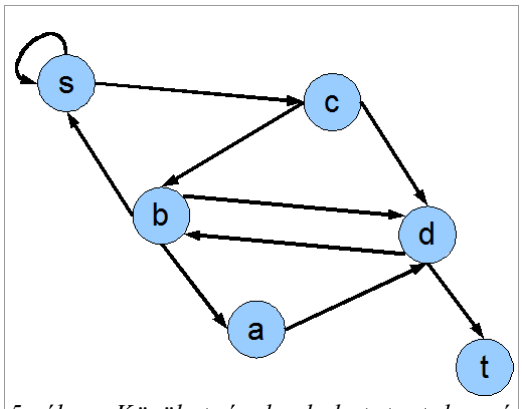
A megoldáskereső rendszerek a következő komponensekből épülnek fel:

- **Adatbázis:** az állapottér-gráf tárolt része.
  Mivel az állapottér-gráf köröket (és hurkokat) tartalmazhat, így az adatbázisban az adott gráfot *fává egyenesítve* tároljuk (lásd lentebb).
- **Műveletek:** az adatbázis módosításának eszközei,
  A műveleteknek két fajtáját szoktuk megkülönböztetni:
  - operátorokból származtatott műveletek
  - technikai műveletek
- **Vezérlő:** a keresés irányítását végzi, a következő lépésekben:
  (1) adatbázis inicializálása
  (2) adatbázis módosítandó részének kiválasztása
  (3) művelet kiválasztása és végrehajtása
  (4) terminálási feltételek vizsgálata:
    - pozitív terminálás: találtunk egy megoldást
    - negatív terminálás: megállapítjuk, hogy nincs megoldás

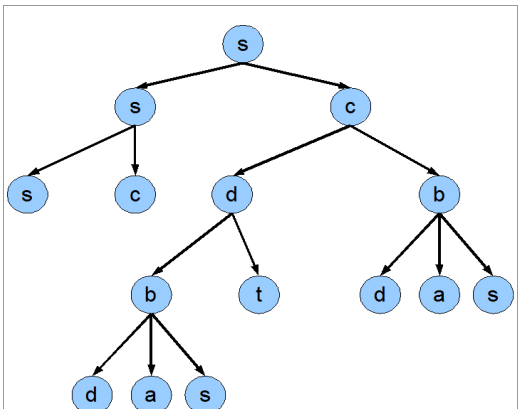
  A vezérlő az (1) és (4) közötti lépéseket ciklikusan szokta végrehajtani.

### AZ ÁLLAPOTTÉR-GRÁF FÁVÁ EGYENESÍTÉSE

Nézzük például a 5. ábrán látható gráfot. A gráf *köröket* tartalmaz, például ilyen triviális kör az $s$-ből az $s$-be mutató él, vagy például az $s, c, b, s$ útvonal, vagy a $c, d, b, s, c$ út. A köröket úgy tudjuk kivágni a gráfból, hogy a megfelelő csúcsokat *duplikáljuk*. A 6. ábrán ez látható, például az $s$-ből az $s$-be mutató élt úgy eliminátuk, hogy $s$ gyermekeként mindenhová újra beszúrtuk $s$-t. Az $s, c, b, s$ kör is megjelenik az ábrán a jobb szélső ágként. Természetesen ez a módszer végtelen fát eredményez, az ábrán ennek csak egy véges részét adtam meg.



*5. ábra: Köröket és hurkokat tartalmazó gráf*



*6. ábra: Fává egyenesített változat*

A fává egyenesítés során a fa ágain a duplikációkat muszáj lesz valamilyen módon kiszűrnünk, ha azt akarjuk, hogy a megoldáskeresés véges sok lépés után befejeződjön. E célból alkalmazzuk majd a vezérlőben (lásd lentebb) a különböző *körfigyelési technikákat*.

A megoldáskeresés végességét ugyan nem veszélyeztetik, de az adatbázisban tárolt csúcsok számát megnövelik az állapottér-gráfban szereplő *hurkok*. A 5. ábrán például hurkot alkotnak a $c, d$ és a $c, b, d$ utak, hiszen a $c$-ből ezen két útvonal *bármelyikén* eljuthatunk a $d$-be. Egy kevésbé triviális hurkot alkotnak a $c, d$ és a $c, b, a, d$ utak. A 6. ábrán meg lehet figyelni, hogy a hurkok

megléte mit eredményez a kapott fában: a csúcsok *duplikálva* kerülnek be a fába, jóllehet nem azonos ágakra (mint a körök esetén), de *különböző ágakra*. Például a $d$ csúcs három helyen is szerepel az ábrán, ez az előbb említett két hurok miatt van így. Figyeljük meg, hogy a hurkok megléte nemcsak egy-egy csúcs duplikációját okozza, hanem részfáknak a duplikációját is, pl. a $b$-ből induló $d, a, s$ csúcsokat tartalmazó részfa két helyen is szerepel az ábrán.

Mint említettem, a megoldáskeresés végességét nem veszélyeztetik a hurkok. Bizonyos problémák megoldáskeresése során azonban érdemes lesz a vezérlőben valamilyen ún. *hurokfigyelési technikát* is alkalmazni, ha az sok csúcs megspórolásával kecsegtet, hiszen ezáltal az adatbázis méretét nagymértékben csökkenthetjük, vagyis kíméljük a tárat. Ráadásul ez utóbbi a legtöbb esetben a futási idő csökkenését is maga után vonja.

#### A MEGOLDÁSKERESŐK TULAJDONSÁGAI

A további fejezetekben különböző megoldáskereső algoritmusokat fogunk megismerni. Ezek különbözni fognak egymástól az adatbázisuk összetételében, a műveleteikben és a vezérlő működésében. Ezek a különbözőségek más és más tulajdonságú keresőket fognak eredményezni, mely tulajdonságok közül a következőket minden egyes kereső esetén meg fogjuk vizsgálni:

- **Teljesség:** Vajon a kereső bármely állapottér-reprezentáció esetén véges sok lépésben megáll-e, és helyes megoldást talál-e, már ha egyáltalán létezik megoldása a problémának? Pontosabban:
  - Ha van megoldás, akkor milyen állapottér-gráf esetén találja meg a kereső?
  - Ha nincs megoldás, akkor ezt a tényt milyen állapottér-gráf esetén ismeri fel a kereső?

  Az állapottér-gráfokat többnyire *végességük* szerint fogjuk megkülönböztetni. Egy gráfot akkor tekintünk *végesnek, ha nem tartalmaz kört*.
- **Optimalitás:** Ha egy problémának több megoldása is van, akkor a kereső az *optimális*, azaz a *legkisebb költségű* megoldást állítja-e elő?

#### A MEGOLDÁSKERESŐK OSZTÁLYOZÁSA

A megoldáskeresőket különböző szempontok szerint osztályozhatjuk:

- *Visszavonható-e műveletvégzés?*
  (1) **Nem módosítható keresők:** Műveletvégzés hatása nem vonható vissza. Ez azzal jár, hogy a keresés során „zsákutcába” juthatunk, melyből nem tudunk a keresés egy korábbi állapotába visszajutni. Ezen keresők előnye az egyszerű, *kisméretű adatbázis*.
  (2) **Módosítható keresők:** A műveletvégzések hatása visszavonható. Ez azt jelenti, hogy a keresés során a kereső nem juthat „zsákutcába”. Ennek ára azonban az *összetettebb adatbázis*.
- *Mi alapján választ a vezérlő az adatbázisból?*
  (1) **Szisztematikus keresők:** Véletlenszerűen vagy valamilyen *általános szisztéma* alapján (pl. fentről le, balról jobbra). Általánosan használható keresők, ám vak, szisztematikus keresési stratégiájuk miatt nem effektívek, legtöbbször nagyméretű adatbázist eredményeznek.
  (2) **Heurisztikus keresők:** Valamilyen *becslés* felhasználásával, mely becslést a tárgyköri ismeretek alapján teszi meg a vezérlő. A heurisztika felhasználásának a lényege az adatbázis méretének csökkentése, és így a kereső effektívvé tétele. A heurisztika milyensége azonban az adott problémától függ, így nincs olyan, hogy „általános heurisztika”.

## 4.1. NEM MÓDOSÍTHATÓ KERESŐK

A nem módosítható megoldáskeresők jelentősége kisebb, tulajdonságaik miatt ritkán, csak bizonyos

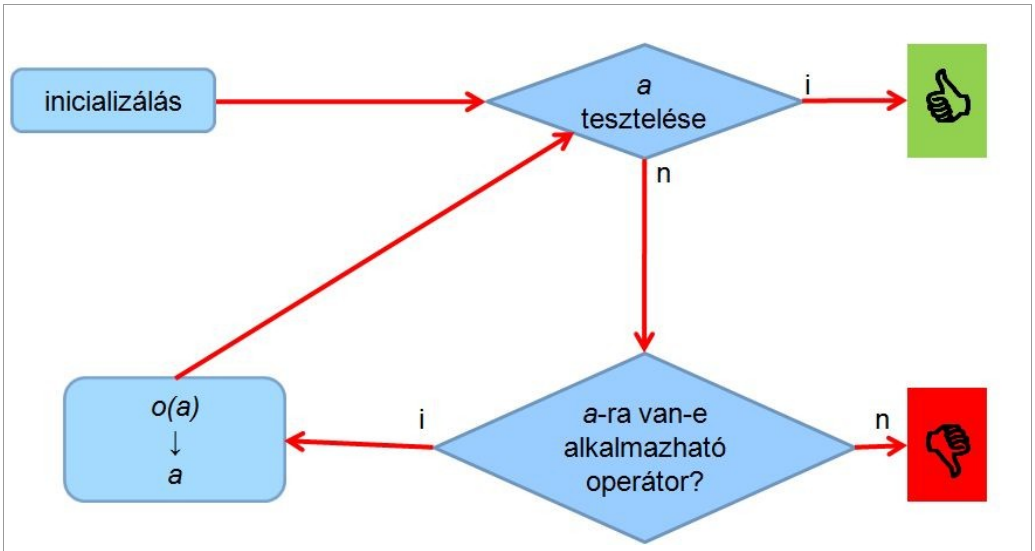
problémák esetén használják őket. Az előnyük mindenképpen az, hogy *egyszerűek*. Csak olyan problémák megoldására használják őket, ahol nem is a megoldás (mint operátorsorozat) előállítása a lényeg, hanem annak eldöntése, hogy *létezik-e megoldása* a feladatnak, és ha igen, akkor egy (valamilyen) célállapot előállítása.

A nem módosítható keresők általános felépítése:

- **Adatbázis:** egyetlen állapotból áll (*aktuális állapot*).
- **Műveletek:** az állapottér-reprezentációban megadott *operátorok*.
- **Vezérlő:** A kezdőállapotra megpróbál egy operátort alkalmazni, és az eredményül kapott állapottal felülírja a kezdőállapotot az adatbázisban. Az új állapotra is próbál operátort alkalmazni, majd ezt az állapotot is felülírja. Ez a ciklikus végrehajtás addig történik, míg az aktuális állapotról ki nem derül, hogy célállapot. Részletesen:

(1) *Inicializálás*: A kezdőállapotot elhelyezi az adatbázisban.
(2) *Ciklus*:
(a) *Tesztelés*: Ha az aktuális állapot (jelöljük *a*-val) célállapot, akkor leáll a keresés. Van megoldás.
(b) Van-e olyan operátor, mely alkalmazható *a*-ra?

- Ha nincs, akkor leáll a keresés. Nem találtunk megoldást.
- Ha van, akkor jelöljük el *o*-val. Legyen *o(a)* az aktuális állapot.



*7. ábra: A nem módosítható keresők folyamatábrája*

A nem módosítható keresők tulajdonságai:

- **Teljesség:**
  - Ha van megoldás, akkor *sem garantált* a megtalálása.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális célállapot (azaz az optimális megoldással elérhető célállapot) előállítása.

Az egyes nem módosítható keresők abban különböznek egymástól, hogy hogyan választják meg az *o* operátort az *a* állapothoz. Két megoldást említek meg:

(1) **Próba-hiba módszer:** véletlenszerűen választjuk meg *o* -t.
(2) **Hegymászó módszer:** Azt az operátort választjuk, mely becslésünk szerint legközelebb visz a (valamelyik) célállapothoz.

A nem módosítható keresők jelentőségét az adja, hogy újra lehet indítani őket. Ha az algoritmus zsákutcába fut, azaz az aktuális állapotra nincs alkalmazható operátor, akkor az algoritmust újra

indítjuk (RESTART). Egyúttal úgy egészítjük ki a feladatot, hogy kizárjuk, hogy még egyszer ugyanabba a zsákutcába fussunk bele (ezt legegyszerűbben az ide vezető operátor alkalmazási előfeltételének kiegészítésével érhetjük el). Az újraindítások számát előre rögzítjük. Belátható, hogy az újraindítások számának növelésével nő a valószínűsége, hogy az algoritmus megoldást talál, feltéve, hogy van megoldás. Ha az újraindítások száma tart a végtelenhez, akkor a megoldás megtalálásának valószínűsége tart az 1-hez.

Az újraindítást alkalmazó nem módosítható megoldást kereső algoritmusokat restartos algoritmusoknak nevezzük.

A nem módosítható keresőket szokták egy hegyes, völgyes vidékre bepottyantott labdával is szemléltetni, amely mindig lefelé gurul, de egy kicsit pattog, mielőtt a lokális minimumban megállna. Ennek az felel meg, hogy a heurisztikánk azt az operátort választja, amely valamilyen szempontból kisebb értékű állapotba visz (lefelé gurulás), ha nincs ilyen, akkor véletlenszerűen választ egy operátort (pattogás), amíg ki nem derül, hogy mindig ugyanoda gurul vissza a labda. Ez a lokális minimum lesz.

A restart ebben a példában annak felel meg, hogy ha már találtunk egy lokális minimumot, akkor újra bedobjuk a labdát egy véletlen helyen.

Restartos módszernél a megtalált legkisebb lokális minimumot fogadjuk el a globális minimum közelítésének. Ez a közelítés annál pontosabb, minél nagyobb az újraindítások száma.

A restartos nem módosítható algoritmusoknak nagy jelentőségük van például a SAT probléma megoldásában. Ezt használják az úgynevezett random walk SAT megoldó algoritmusok.

### 4.1.1. Próba-hiba módszer

Mint fentebb említettük, a próba-hiba módszer esetén az aktuális csúcsra egy véletlenszerűen kiválasztott alkalmazható operátort alkalmazunk.

- **Teljesség:**
  - Ha van megoldás, azt nem mindig találja meg.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.

Véletlen választás (egyetlen) előnye: a végtelen ciklus szinte lehetetlen.

***Ötlet:***

- Ha zsákutcába kerülünk, akkor *restart*.
- Kizárjuk, hogy még egyszer zsákutcába kerüljünk, megjegyezzük a csúcsot (bővítjük az adatbázist).

### 4.1.2. Restartos próba-hiba módszer

- **Adatbázis:** az aktuális csúcs, a megjegyzett zsákutcák, az újraindítások száma és a maximális újraindítások száma.
- **Vezérlő:**

(1) *Inicializálás:* Az aktuális csúcs legyen a startcsúcs, a megjegyzett zsákutcák listája legyen üres, az újraindítások száma 0.

(2) *Ciklus:* Az aktuális csúcsra alkalmazok egy véletlenszerűen kiválasztott alkalmazható operátort. Az így kapott új állapotot megvizsgálom, hogy benne van-e az ismert zsákutcák listájában. Ha igen, akkor ugrás a ciklus elejére. Ha nem, akkor az aktuális csúcs legyen az új állapotból készített csúcs.

(a) *Tesztelés:* Ha az aktuális csúcs terminális csúcs, akkor a megoldás a képernyőre kiírt adatokból következtethető vissza.

(b) Ha az aktuális csúcsra nincs alkalmazható operátor, azaz ha az aktuális csúcs zsákutca:

- Ha még nem értük el a maximum újraindítások számát, a megtalált zsákutcát felvesszük az adatbázisba, növeljük az újraindítások számát eggyel, az aktuális csúcs legyen a startcsúcs és ugrás a ciklus elejére.
- Ha elértük a maximális újraindítások számát, akkor kiírjuk, hogy nem találtunk megoldást.

Az algoritmus tulajdonságai:

- **Teljesség:**
  - Ha van megoldás, azt nem mindig találja meg.
  - Minél nagyobb az újraindítások száma, annál valószínűbb, hogy megtaláljuk a megoldást. Ha az újraindítások száma tart a végtelenhez, akkor annak a valószínűsége, hogy találunk megoldást, tart az egyhez.
  - Ha nincs megoldás, akkor azt felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.

A próba-hiba algoritmusnak elméleti jelentősége van. A restartos változatot véletlen sétának (random walk) nevezzük. A konjunktív normálformák kielégíthetősége legpraktikusabban ezzel az algoritmussal vizsgáltható.

## 4.1.3. Hegymászó módszer

A hegymászó módszer egy *heurisztikus kereső*. Ugyanis azt, hogy egy állapotból milyen messzire van egy (valamilyen) célállapot, egy ún. *heurisztikán* keresztül becsüljük. A heurisztika nem más, mint egy az állapotok halmazán ( $A$ ) értelmezett *függvény*, mely megmondja, hogy az adott állapotból körülbelül *milyen költségű úton* érhető el egy célállapot. Azaz:

**7. Definíció:** Az $\langle A, k, C, O \rangle$ állapottér-reprezentációhoz megadott *heurisztika* egy olyan $h: A \to \mathbb{N}$ függvény, hogy $\forall\, c \in C$-re $h(c) = 0$.

A hegymászó módszer az $a$ állapotra alkalmazható operátorok közül azt az $o$ operátort alkalmazza, melyre $h(o(a))$ *minimális*.

Nézzük meg, hogy a hegymászó módszer hogyan működik a *Hanoi tornyai* probléma esetén! Először adjunk meg egy lehetséges heurisztikát erre a problémára! Például legyen a heurisztika a korongok $R$ rúdtól való távolságainak az összege. Azaz:

$$h(a_1, a_2, a_3) = \sum_{i=1}^{3} (R - a_i)$$

ahol $R - P = 2$, $R - Q = 1$ és $R - R = 0$. Vegyük észre, hogy az $(R, R, R)$ célállapotra teljesül, hogy $h(R, R, R) = 0$.

Kezdetben az adatbázisban a kezdőállapot, azaz $(P, P, P)$ foglal helyet. Erre az állapotra az $átrak_{1,Q}$ és az $átrak_{1,R}$ operátorokat tudjuk alkalmazni. Az előző az 5 heurisztikájú $(Q, P, P)$, az utóbbi a 4 heurisztikájú $(R, P, P)$ állapotot állítja elő. Ezért az $(R, P, P)$ lesz az aktuális állapot. Hasonlóképpen, a következő lépésben $(R, Q, P)$-t rakjuk be az adatbázisba.

Következőnek két állapot közül kell választanunk: vagy az $(R, P, P)$-t, vagy a $(Q, Q, P)$-t rakjuk be az adatbázisba. A helyzet különlegessége, hogy ennek a két állapotnak egyenlő a heurisztikája, és a hegymászó módszer arról nem rendelkezik, hogy egyenlő heurisztikájú állapotok közül melyiket válasszuk. Azaz ebben a helyzetben találomra vagy véletlenszerűen választhatunk a két állapot közül. Vegyük észre, hogy ha $(R, P, P)$-t választanánk, azzal visszajutnánk az előző

(P,P,P)

(R,P,P)

(R,Q,P)

(Q,Q,P)

(Q,Q,R)

(R,Q,R)

(R,P,R)

aktuális állapothoz, ahonnan újra az $(R, Q, P)$-be jutnánk, ahonnan ismét az $(R, P, P)$-be lépnénk, és így tovább a végtelenségig. Ha viszont most $(Q, Q, P)$-t választjuk, a keresés mehet tovább egy remélhetőleg nem végtelen ágon.

Így haladva tovább hasonló szituációval találkozunk az $(R, Q, R)$ állapotban, ugyanis ebből továbbléphetünk az egyenlő heurisztikájú $(Q, Q, R)$ és $(R, P, R)$ állapotok valamelyikébe. Nyilván az előbbivel végtelen végrehajtásba futnánk.

Azt kell mondani, hogy ezzel a heurisztikával elég szerencsésnek kell lennünk, hogy a keresőnk egyáltalán leálljon. Talán egy másik, kifinomultabb heurisztikával ezt jobban lehetne szavatolni, bár egy ilyen heurisztika létezésére nincs garancia. Mindazonáltal látnunk kell, hogy a keresés „múltjának" tárolása nélkül szinte lehetetlen a végtelen végrehajtást és a „zsákutcákat" elkerülnünk.

Érdemes megjegyezni, hogy a Hanoi tornyai tipikusan olyan probléma, melyre egy nem módosítható keresőt alkalmazni nincs is értelme. Hiszen az (egyetlen) célállapot előre ismert. Ennél a problémánál ténylegesen egy *megoldás* előállítása a cél, márpedig erre egy nem módosítható kereső természeténél fogva alkalmatlan.

### 4.1.4. RESTARTOS HEGYMÁSZÓ MÓDSZER

A restartos hegymászó módszer megegyezik a hegymászó módszerrel annyi kiegészítéssel, hogy egy előre meghatározott számú újraindítást engedélyezünk. A hegymászó módszert újraindítjuk, ha az zsákutcába fut. Ha elértük a maximális újraindítási számot és zsákutcába futunk, akkor az algoritmus megáll, mert nem talált megoldást.

Fontos, hogy minden zsákutcából tanuljon az algoritmus, azaz ne tudjon kétszer ugyanabba a zsákutcába futni. Enélkül a heurisztika mindig ugyanabba a zsákutcába vezetné a hegymászót az újraindítás után, kivéve, ha a heurisztikának van véletlen része. A tanulás többféleképpen is történhet. Legegyszerűbb az állapottér-reprezentációt megváltoztatni, úgy, hogy a zsákutcába futáskor az aktuális állapotot töröljük az állapotok halmazából. Másik megoldás, hogy az adatbázist bővítjük egy tiltott állapotok listájával.

Két esetben érdemes használni:

1. ha tanul, azaz megjegyzi a felderített zsákutcákat
2. ha a heurisztika nem determinisztikus

## 4.2. VISSZALÉPÉSES KERESŐK

A módosítható megoldáskereső algoritmusok egy fajtája a visszalépéses (vagy idegen szóval: *backtrack*) kereső, melynek több változata is létezik. A backtrack keresők alapötlete: ne csak az aktuális csúcsot tároljuk az adatbázisban, hanem azokat a csúcsokat is, melyeken keresztül az aktuális csúcsba eljutottunk. Ez azt jelenti tulajdonképpen, hogy az adatbázisban az állapottér-gráfnak most már nagyobb részét fogjuk tárolni: *a startcsúcsból az aktuális csúcsba vezető utat*.

A backtrack keresők nagy előnye, hogy a keresés *nem kerülhet zsákutcába*. Ha az aktuális csúcsból nincs továbblépés a gráfban, akkor visszalépünk az aktuális csúcs szülőjébe, és abból próbálunk ezúttal más irányba lépni. Ez a speciális lépés – amit *visszalépésnek* neveznek – adta a nevét ennek a fajta keresőnek.

Logikus, hogy minden egyes az adatbázisban tárolt csúcsban az állapot mellett azt is el kell tárolni, hogy a csúcsból eddig merrefelé próbáltunk továbblépni. Vagyis: minden csúcsban *regisztrálni kell azokat az operátorokat*, melyeket a csúcsban tárolt állapotra nem próbáltunk még

alkalmazni. Abban a pillanatban, mikor egy operátort alkalmaztam az állapotra, az operátort törlöm a csúcsban tárolt regisztrációból.

### 4.2.1. ALAP BACKTRACK

- **Adatbázis:** az állapottér-gráfban a startcsúcsból az aktuális csúcsba vezető *út*.
- **Műveletek:**
  - *Operátorok*: az állapottér-reprezentációban adottak.
  - *Visszalépés*: technikai művelet, mely az adatbázisban tárolt út legalsó csúcsának a törlését jelenti.
- **Vezérlő:** Az adatbázist inicializálja, az aktuális csúcsra műveletet hajt végre, teszteli a célfeltételt, majd ez alapján eldönti, hogy leáll-e a kereséssel, vagy pedig újra megvizsgálja az aktuális csúcsot. Részletesen a vezérlő működése:

(1) *Inicializálás:* A startcsúcsot egyedüli csúcsként elhelyezi az adatbázisban.
startcsúcs= kezdőállapot + az összes operátor regisztrálva

(2) *Ciklus:*

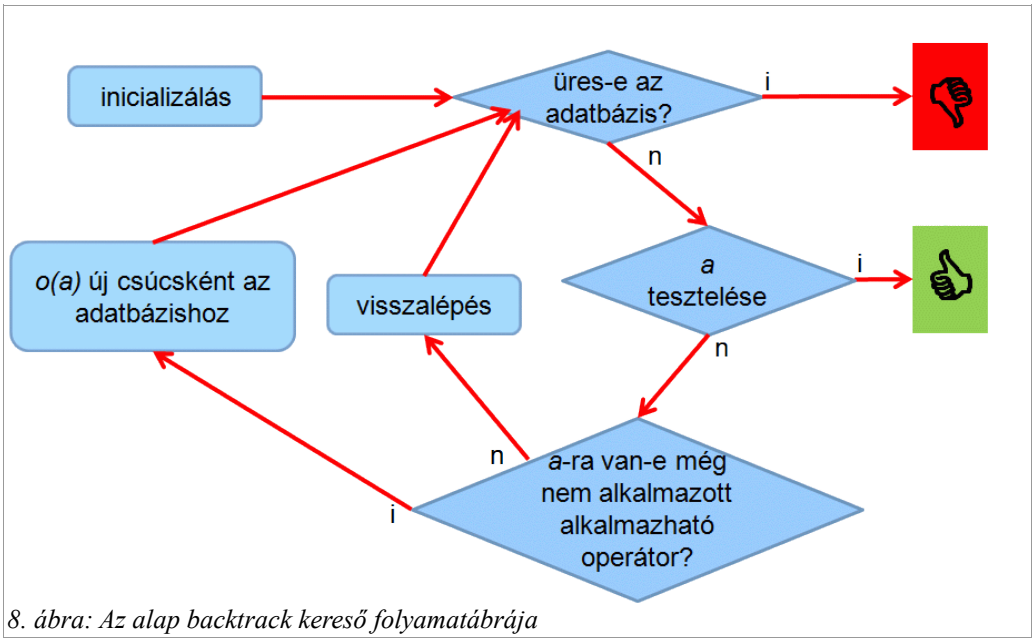
(a) Ha az adatbázis üres, leáll a keresés. Nem találtunk megoldást.

(b) Az adatbázisban tárolt út legalján elhelyezkedő (azaz az adatbázisba legkésőbb berakott) csúcsot kiválasztjuk; ezt *aktuális csúcsnak* fogjuk nevezni. Jelöljük az aktuális csúcsban tárolt állapotot $a$ -val!

(c) *Tesztelés*: Ha az $a$ célállapot, akkor leáll a keresés. A megtalált megoldás: *maga az adatbázis* (mint út).

(d) Megvizsgálja, hogy van-e olyan operátor, melyet *még nem próbáltunk alkalmazni az* $a$ *-ra*. Azaz: van-e még az aktuális csúcsban operátor regisztrálva?

- Ha van ilyen, jelöljük el $o$ -val! Töröljük $o$ -t az aktuális csúcsból. Teszteljük az $o$ alkalmazási előfeltételét az $a$ *-ra*. Ha az teljesül, akkor alkalmazzuk az $o$ -t az $a$ -ra, és az így kapott $o(a)$ állapotot beszúrjuk az adatbázisban tárolt út aljára. Az új csúcsban az $o(a)$ mellett az *összes operátort* regisztráljuk.
- Ha nincs ilyen, akkor a vezérlő *visszalép*.




*8. ábra: Az alap backtrack kereső folyamatábrája*

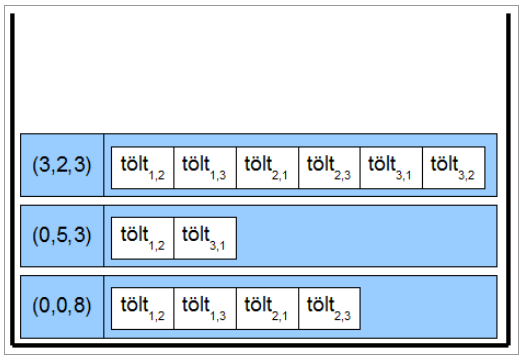
Az így kapott backtrack kereső tulajdonságai a következők:

- **Teljesség:**
  - Ha van megoldás, akkor *véges* állapottér-gráfban megtalálja azt.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.

***IMPLEMENTÁCIÓS KÉRDÉSEK***

- **Milyen adatszerkezettel valósítsuk meg az adatbázist?**
  *Veremmel.*
  A kereső műveleteinek megfeleltethetők a következő veremműveletek:
  - operátoralkalmazás ⇐ PUSH
  - visszalépés ⇐ POP
- **Milyen formában regisztráljuk az operátorokat a csúcsokban?**

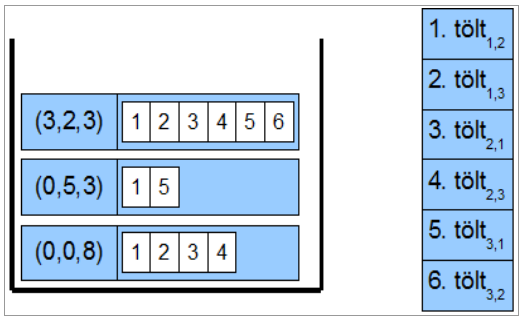
(1) Minden csúcsban *operátorok listáját* tároljuk.




Itt a veremben a 3 kancsó probléma 3 csúcsa látható. A startcsúcsra (alul) már próbáltuk alkalmazni a $tölt_{3,1}$ és $tölt_{3,2}$ operátorokat. A $tölt_{3,2}$ alkalmazásával kaptuk egyébként a 2. csúcsot, melyre már csak a $tölt_{1,2}$ és $tölt_{3,1}$ operátorok maradtak. A $tölt_{2,1}$ alkalmazásával kaptuk a 3. (legfelső, aktuális) csúcsot, melyre eddig még egy operátort sem próbáltunk alkalmazni.

Ötlet: egy-egy *új állapot* beszúrásánál az új csúcsban tárolt operátorlistába ne az összes operátort pakoljuk be, hanem csak az adott állapotra *alkalmazható* operátorokat. Némi tárat spórolunk, ám lehet, hogy feleslegesen teszteljük egyes operátorok alkalmazási előfeltételét egyes állapotokra (mivel lehet, hogy hamarabb megtaláljuk a megoldást, mintsem rájuk kerülne a sor).

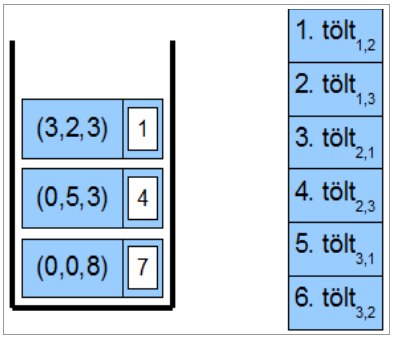
(2) Az operátorokat a csúcsokon kívül, egy *konstans tömbben* (vagy listában) tároljuk. A csúcsokban pedig csak *operátorindexeket*, azaz az adott operátornak az előbb említett tömbben elfoglalt pozícióját (indexét) tároljuk. Ennek a megoldásnak az az előnye, hogy magukat az operátorokat *csak egy helyen* tároljuk (nincs redundancia).




A 3 kancsó probléma esetén az operátorok (konstans) tömbje 6-elemű. A csúcsokban a még nem alkalmazott operátorok indexét (vagy esetleg: az operátorok referenciáit) tároljuk.

(3) Az előző megoldást továbbfejlesztjük azzal, hogy minden állapotra az operátorokat az *operátorok tömbjében elfoglalt pozíciójuk sorrendjében* alkalmazzuk. Ezzel a következőt nyerjük: a csúcsokban bőségesen elég egy-egy operátorindexet tárolni (az eddigi operátorindex-lista helyett). Ez az operátorindex jelölje ki azt az operátort, melyet *a következő alkalommal a csúcsban tárolt állapotra alkalmazni fogok.* Így tehát tudjuk, hogy az operátorok tömbjében az operátorindextől balra eső operátorokat már alkalmaztam

az állapotra, a tőle jobbra levőket pedig még nem.




Az aktuális csúcsra még nem alkalmaztunk operátort, ezért annak operátorindexe 1 lesz jelezvén, hogy a következő alkalommal a 1-es indexű operátort próbáljuk majd alkalmazni rá.

A 2. csúcsra sorban próbáltuk alkalmazni az operátorokat, legutoljára a $tölt_{2,1}$, azaz a 3. operátort alkalmaztuk. Tehát következő alkalommal a 4-iket fogjuk.

A startcsúcsra már az összes operátort megpróbáltuk alkalmazni, ezért ennek operátorindexe egy nem létező operátorra hivatkozik.

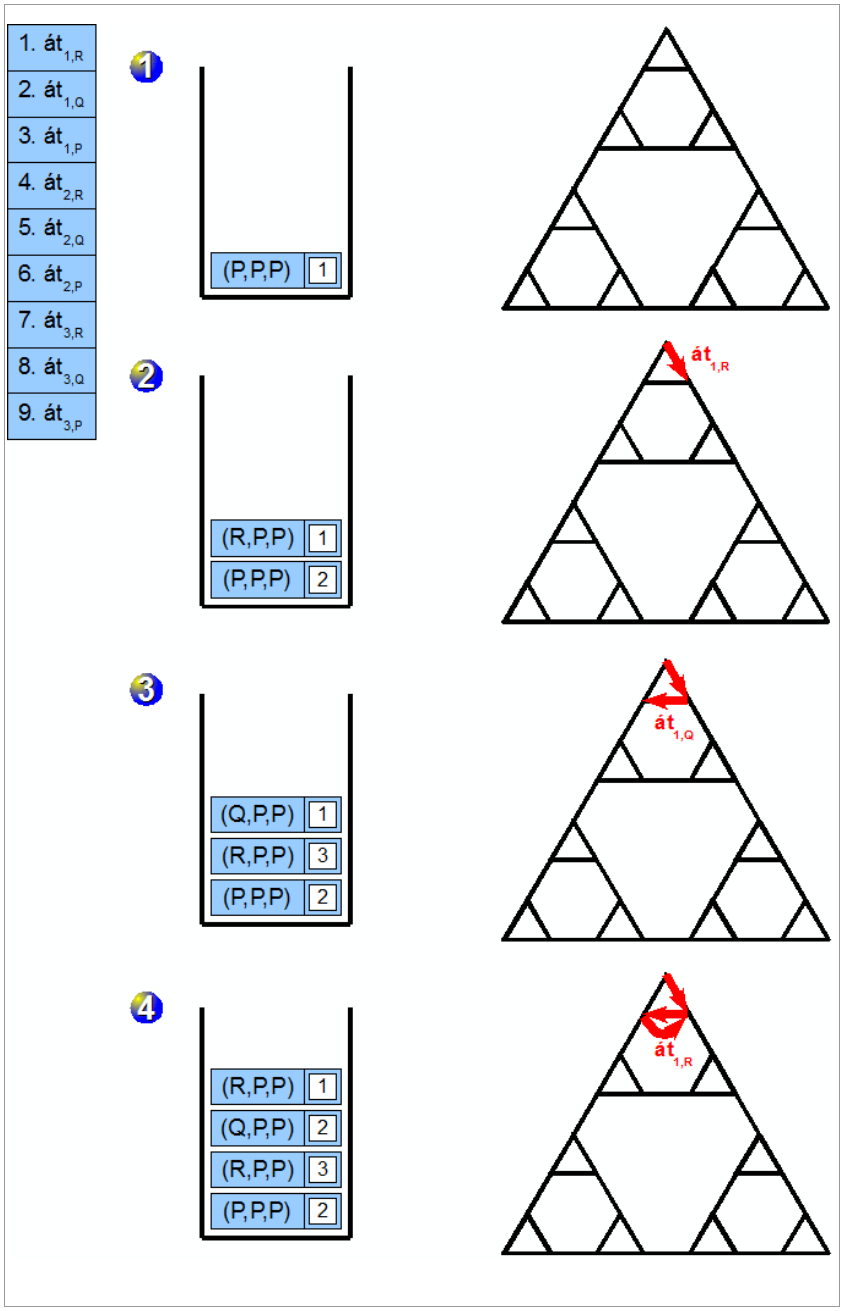
A *visszalépés* feltétele ily módon nagyon könnyen meghatározható: ha az aktuális csúcsban tárolt operátorindex nagyobb, mint az operátortömb mérete, akkor visszalépünk.

***PÉLDA:***

A *Hanoi tornyai* probléma esetén az alap backtrack kereső végtelen ciklusba kerül, ugyanis előbb-utóbb az állapottér-gráf egy körében fog megrekedni a keresés. Hogy ez hány operátoralkalmazás után következik be, az nagyban függ az attól, hogy az operátorok milyen sorrendben találhatók meg az operátorok tömbjében.

A 9. ábrán bemutatom a keresést pár lépés erejéig. Az ábra bal felső részén az operátorok tömbje látható. Lépésenként feltüntetem a kereső által használt vermet, illetve mellette az állapottér-gráfban (lásd: 3. ábra) eddig bejárt utat.

Mint látható, a keresés hátralévő részében az $(R, P, P)$ és az $(Q, P, P)$ állapotok között fogunk ide-oda lépegetni, szépen töltve fel a vermet. Mindezt pusztán ezért, mert az operátorok között egyfajta prioritást osztottunk ki, és a kereső szigorúan mindig a lehetséges legnagyobb prioritású operátort alkalmazza.




## 4.2.2. BACKTRACK ÚTHOSSZKORLÁTTAL

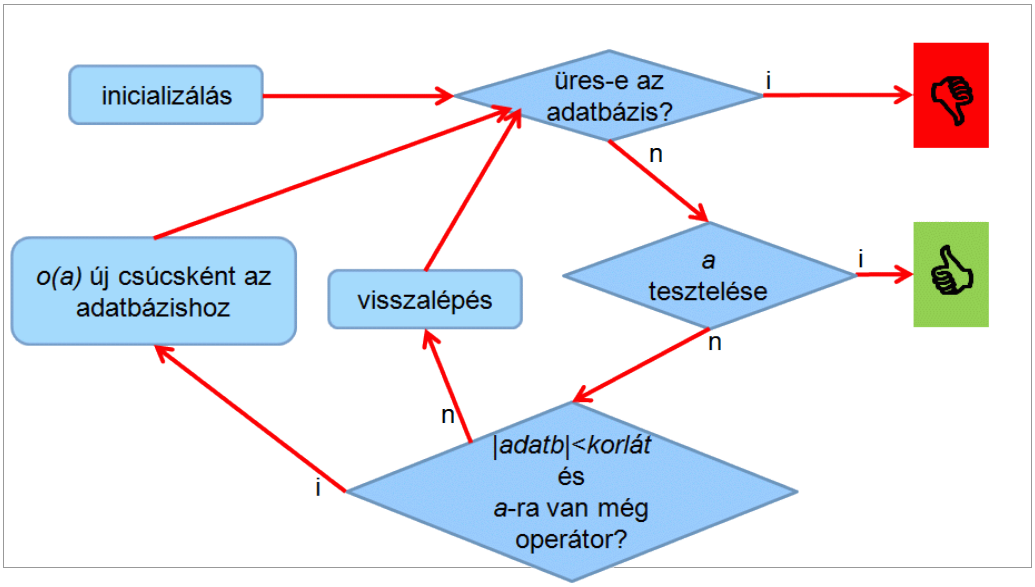
Az alap backtrackkel kapcsolatos egyik továbbfejlesztési lehetőség az algoritmus teljességének kibővítése. Vagyis megpróbáljuk kibővíteni azon állapottér-gráfok körét, melyet az algoritmus kezelni tud.

Az alap backtrack csak véges állapottér-gráf esetén garantálja azt, hogy leáll véges sok lépésben. A gráfban szereplő körök veszélyeztetik a véges végrehajtást, ezért azokat valami módon meg kell próbálnunk kivágni az állapottér-gráfból. Erre két megoldást ismerünk meg: a következő fejezetben a *körfigyeléssel* kombinált backtrack keresőt, illetve a mostani fejezetben egy egyszerűbb megoldást, mely a köröket nem abszolút mértékben vágja ki, de a körökön csak *véges sokszor* halad át.

Ezt egy egyszerű megoldással érjük el: *maximáljuk az adatbázis méretét*. Ez az állapottér-gráfban azt jelenti, hogy azt csak egy előre megadott (véges) mélységig járjuk be, implementációs szinten pedig azt, hogy a veremnek előre megadjuk a maximális méretét. Ha a keresés során

valamikor is az adatbázis ilyen értelemben „betelne”, akkor a kereső *visszalép*.

Legyen tehát előre adott a $korlát > 0$ egész szám. Az algoritmusban a visszalépési feltételt kibővítjük: ha az adatbázis mérete eléri a *korlát* -ot, akkor is visszalépést végzünk.



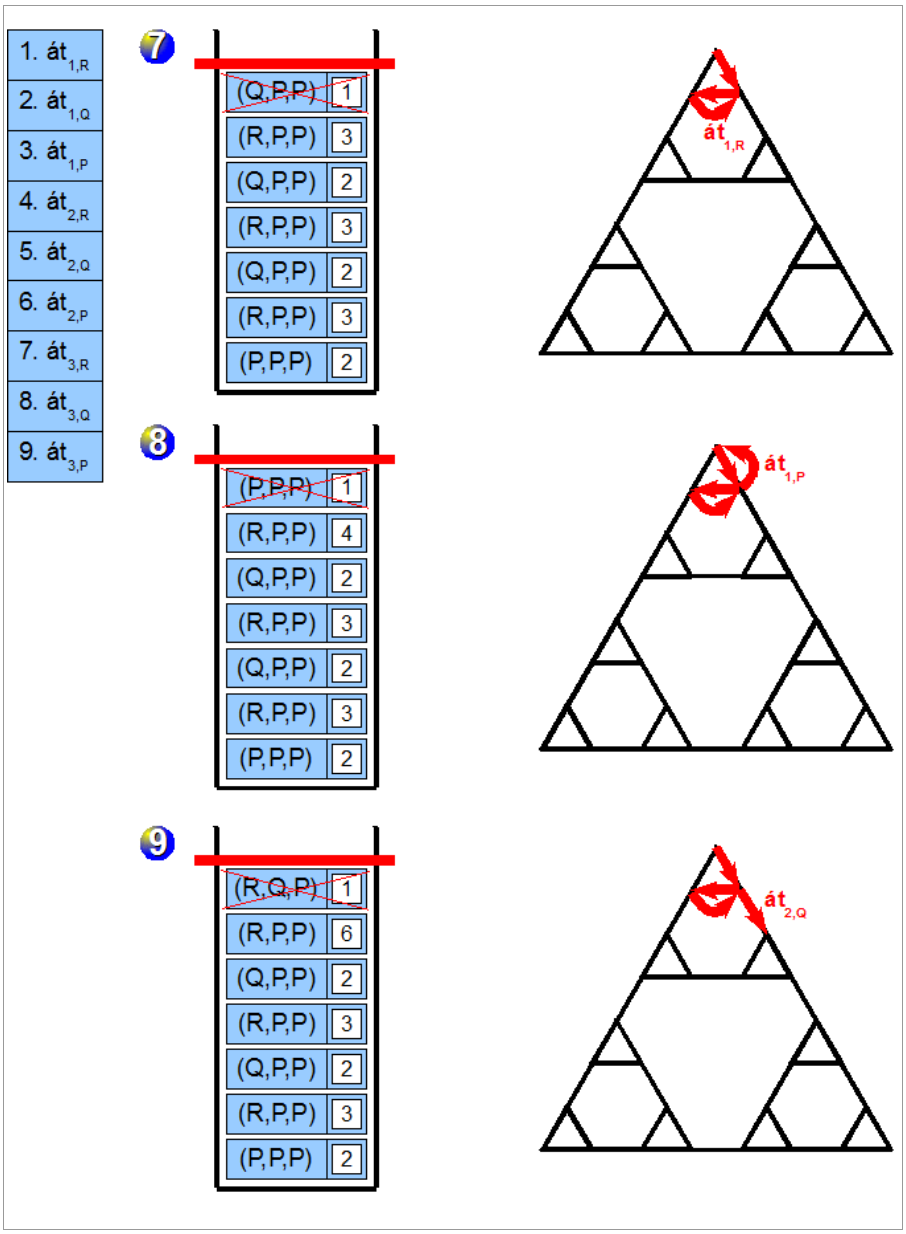

Az így kapott backtrack kereső tulajdonságai a következők:

- **Teljesség:**
  - Ha van megoldás és a *korlát* értéke nem kisebb, mint az optimális megoldás hossza, akkor *tetszőleges* állapottér-gráfban talál megoldást.
    Ha viszont a *korlát* -ot túl kicsire választjuk, akkor a kereső nem fog megoldást találni akkor sem, ha egyébként van megoldása a feladatnak. Ilyen értelemben az úthosszkorlátos backtrack *nem garantálja* a megoldást előállítását.
  - Ha nincs megoldás, akkor ezt *tetszőleges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.

***PÉLDA***

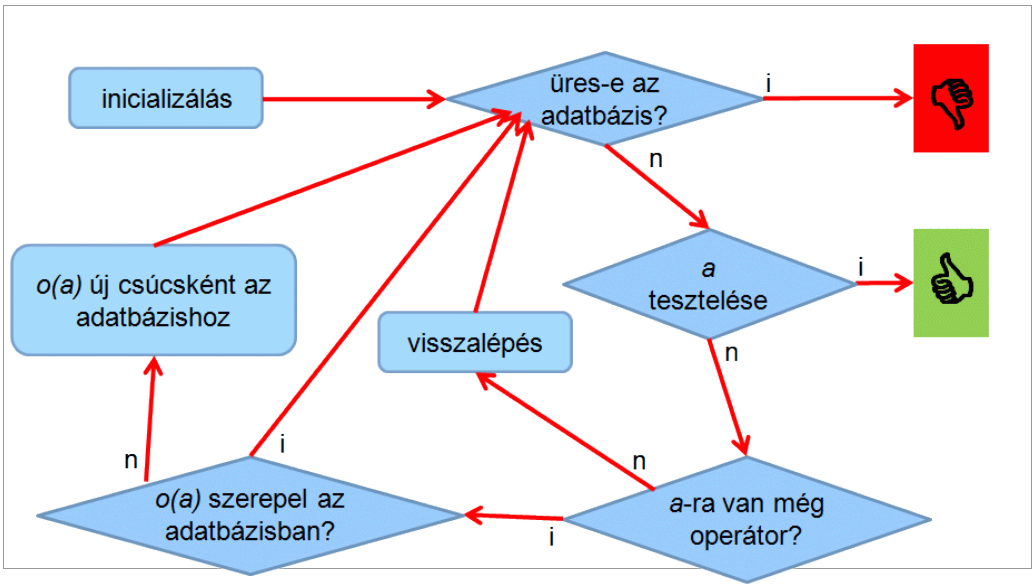
A 11. ábrán egy úthosszkorlátot határozunk meg, ez számszerűleg 7. Az úthosszkorlátot az ábrán a verem tetejére rajzolt piros vonal jelzi. A keresést ott folytatjuk, ahol a 11. ábrán abbahagytuk, azaz az $(R, P, P)$ és a $(Q, P, P)$ állapotok sorozatos duplikációival. A kereső 7. lépésénél a verem mérete eléri a korlátot, visszalépés történik, vagyis a verem tetején levő csúcsot töröljük, és az eggyel alatta levő csúcsra a következő alkalmazható operátort alkalmazzuk. Mint látható, végre kiszabadul a keresés a végtelen végrehajtásból, de azt is észre lehet venni, hogy még sok-sok visszalépés után fogunk ténylegesen elindulni a célcsúcs felé.

Vegyük észre, hogy ha az úthosszkorlátot 7-nél kisebbre választanánk, akkor a kereső nem találna megoldást!




## 4.2.3. BACKTRACK KÖRFIGYELÉSSEL

A keresés végességének biztosítására egy másik lehetőség az állapottér-gráf köreinek abszolút értelemben vett kivágása. Ezt egy plusz teszt beiktatásával érjük el: egy csúcsot *csak akkor szúrunk be* újként az adatbázisba, ha az még *nem szerepel* benne. Azaz kiiktatunk az adatbázisból mindenfajta duplikációt.




Az így kapott kereső teljesség szempontjából a lehető legjobb tulajdonságokkal bír:

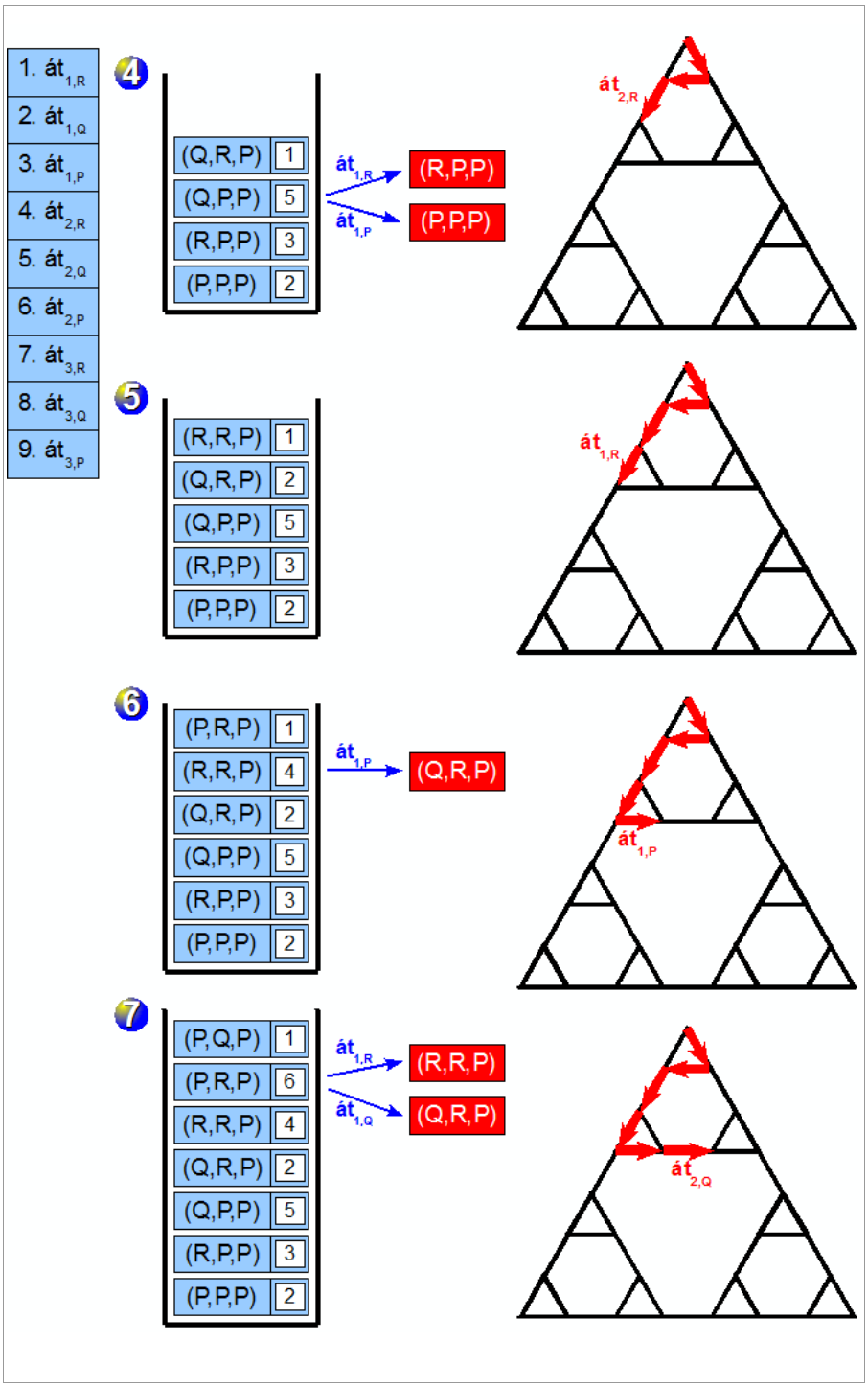
- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *tetszőleges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.

Mindezen jó tulajdonságok ára egy meglehetősen költséges plusz teszt. Fontos, hogy ezen körfigyelési tesztet csak akkor alkalmazzuk a keresőnkben, ha megbizonyosodtunk arról, hogy a feladatunk *állapottér-gráfjában van kör*. Ugyanis kissé drága mulatság minden egyes új csúcs beszúrásánál feleslegesen végigszkennelni az adatbázist!

***PÉLDA***

A 13. ábrán a körfigyeléses backtrack kereső pár lépését tudjuk nyomon követni a 9. ábra utolsó előtti konfigurációjából kezdve. A kereső $(Q, P, P)$ állapotból való továbblépési próbálkozásai között szerepel az $át_{1,R}$ és az $át_{1,P}$ operátorok alkalmazása, melyek által előállított $(R, P, P)$, illetve $(P, P, P)$ állapotok már szerepelnek a veremben, ezért oda nem rakjuk be újra őket. Így jutunk el az $át_{2,R}$ operátorig, mely által előállított $(Q, R, P)$ állapot még nem szerepel a veremben.

Ennek szellemében folytatódik a keresés, természetesen teljesen kiküszöbölve a veremből a duplikációkat. Érdemes megfigyelni, hogy hiába kerüli ki okosan a köröket a kereső, bizony elég után halad a célcsúcs felé, vagyis a végül megtalált megoldás közel sem lesz optimális.




## 4.2.4. Ág és korlát algoritmus

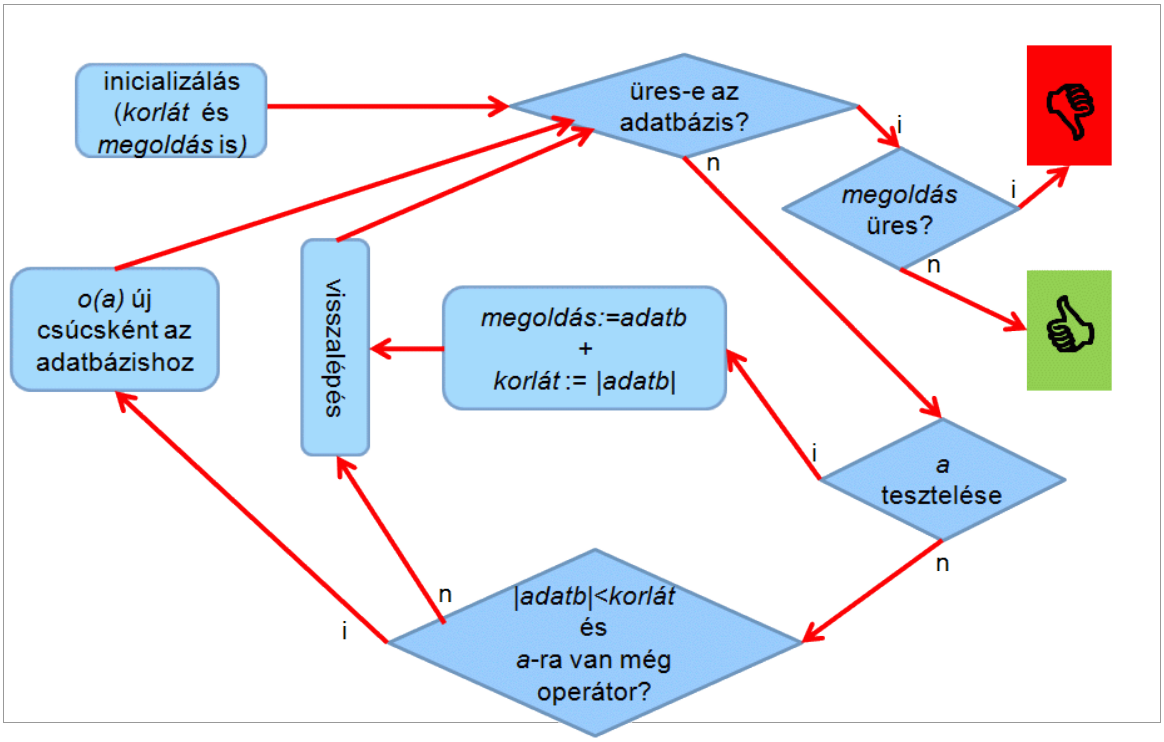
Az előző fejezetben bemutatott backtrack keresőt a következő célból próbálhatjuk meg még továbbfejleszteni: a kereső garantálja az *optimális megoldás* előállítását! Egy ilyen továbbfejlesztés elég logikusan adódik: a feladat megoldásainak univerzumában végezzen a kereső egy sima minimumkiválasztást.

A kereső tehát egy megoldás megtalálásakor nem fog terminálni, hanem az adatbázisról készít egy másolatot (erre fogja használni a *megoldás* nevű vektort), majd *visszalép*, és folytatja a keresést. Ebből adódik, hogy a keresés csak akkor ér véget, ha *kiürül az adatbázis*. Hogy ne kelljen a teljes állapottér-gráfot bejárnunk ehhez, használjuk az *úthosszkorlátos backtrack* keresőnek egy olyan

változatát, melyben a *korlát* értéke *dinamikusan változik*. Egy megoldás megtalálásakor (és a *megoldás* vektorban való letárolásakor) a *korlát* új értéke legyen a megtalált megoldás hossza! Azaz a megtalált megoldás alatti mélységben már nem járjuk be az állapottér-gráfot.

Triviális, hogy az így kapott backtrack kereső – melyet *ág és korlát algoritmusnak* neveznek – az optimális megoldást állítja elő (már ha létezik az adott feladatnak megoldása).

A keresés elején a felhasznált *megoldás* és *korlát* változókat inicializálni kell. A *megoldás* kezdetben legyen üres vektor, a *korlát* pedig egy olyan nagy szám, mely az optimális megoldás hosszánál mindenképpen nagyobb. A programozók a *korlát* kezdeti értékeként a legnagyobb ábrázolható egész számot szokták választani. Az ág és korlát algoritmust általában körfigyeléssel is kombinálják.




Az így kapott ág és korlát algoritmus tulajdonságai:

- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *tetszőleges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *garantált* az optimális megoldás előállítása.

## 4.3. KERESŐFÁVAL KERESŐK

A módosítható megoldáskeresők egy másik nagy csoportját a *keresőfával keresők* alkotják. Alapvető különbség a backtrack keresőkhöz képest, hogy az adatbázisban az állapottér-gráfnak nem csak egy ágát, hanem egy összetettebb részét tároljuk (egy fa formájában). Tulajdonképpen a keresést *egyszerre több ágon* végezzük, így valószínűleg hamarabb találunk megoldást, és talán az optimális megoldást. Ennek a módszernek nyilvánvaló hátránya a nagyobb tárigény.

A következő fejezetben egy általános keresőfával kereső leírását adom meg. Ez a kereső még nem egy konkrét kereső, hanem csak a később tárgyalandó (konkrét) keresőfával keresők – szélességi kereső, mélységi kereső, optimális kereső, best-first kereső és A-algoritmus – közös komponenseit foglalja magában.

## 4.3.1. Általános keresőfával kereső

- **Adatbázis:** Az állapottér-gráf egy fává egyenesített része. Az adatbázisban tárolt fa csúcsait két csoportra osztjuk:
  - *Zárt csúcsok:* a már korábban kiterjesztett csúcsok (lásd a műveleteknél).
  - *Nyílt csúcsok:* a még ki nem terjesztett csúcsok.
- **Művelet:** *kiterjesztés*.
  Csak nyílt csúcsot terjeszthetünk ki. Egy csúcs kiterjesztése a következőt jelenti: a csúcsra az *összes rá alkalmazható operátort* alkalmazzuk. Tulajdonképpen a csúcsnak az összes (állapottér-gráfbeli) gyermekét előállítjuk.
- **Vezérlő:** Az adatbázist inicializálja, az általa kiválasztott nyílt csúcsot kiterjeszti, teszteli a célfeltételt, majd ez alapján eldönti, hogy leáll-e a kereséssel, vagy pedig újabb csúcsot terjeszt ki. Részletesen a vezérlő működése:

(1) *Inicializálás:* A startcsúcsot egyedüli csúcsként elhelyezi az adatbázisban, mégpedig nyílt csúcsként.

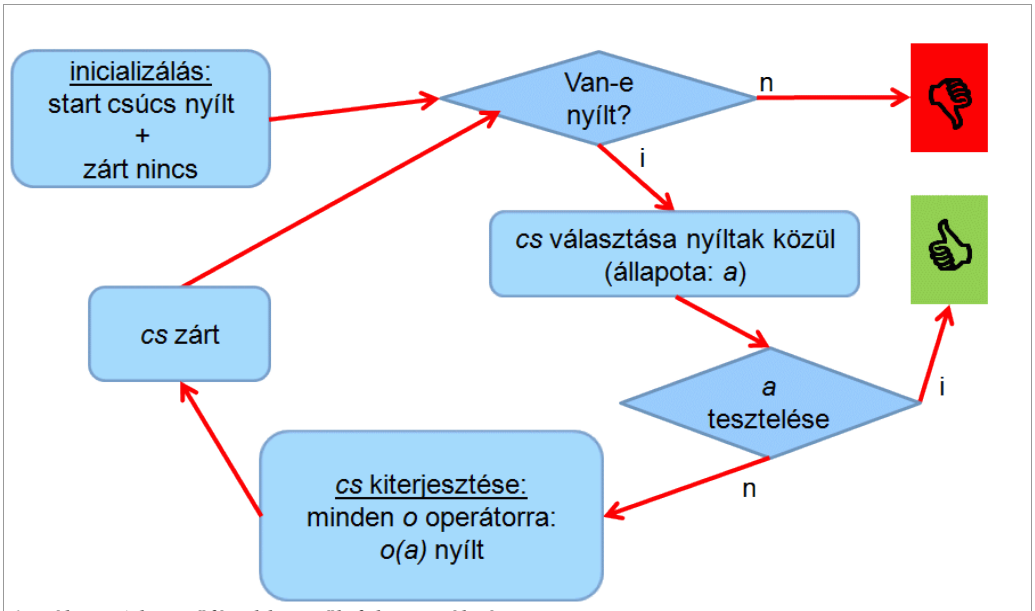
(2) *Ciklus:*

(a) Ha az adatbázisban nincs nyílt csúcs, leáll a keresés. Nem találtunk megoldást.

(b) A nyílt csúcsok közül kiválaszt egy csúcsot; ezt *aktuális csúcsnak* fogjuk nevezni. Ezt a csúcsot jelöljük *cs* -vel!

(c) Ha *cs* célcsúcs, akkor leáll a keresés. A megtalált megoldás: az adatbázisban a startcsúcsból a *cs* -be vezető út.

(d) A *cs* -t kiterjeszti, az így előálló új állapotokat *cs* gyermekeként nyílt csúcsként az adatbázisba szúrja, majd *cs* -t zárttá minősíti.




*15. ábra: A keresőfával keresők folyamatábrája*

A fent leírt általános keresőfával kereső vezérlőjében csupán három nem teljesen rögzített elem van:

- (2)(a) pontban: Ha több nyílt csúcs is van az adatbázisban, akkor ezek közül melyiket válasszuk kiterjesztésre?
- (2)(c) pontban: A célfeltételt az aktuális csúcson teszteljük. Azaz egy csúcs hiába kerül be az adatbázisba, rajta a célfeltétel teljesülésének vizsgálatával várnunk kell addig, míg az adott csúcsot ki nem választjuk kiterjesztésre. Egyes keresők esetén a keresés gyorsítása érdekében a *tesztelés előrehozható*, ami azt jelenti, hogy a (2)(d)-ben az előállított új állapotokra azon nyomban (még az adatbázisba való beszúrásuk előtt) tesztelhetjük a célfeltételt.
- (2)(d) pontban: Használjunk-e valamilyen körfigyelési technikát, és ha igen, milyet? Azaz ha a

kiterjesztés eredményeként előálló valamely állapot már szerepel az adatbázisban, beszúrjuk-e újra ezt az adatbázisba?

Kétfajta körfigyelési technikát lehet használni:

- A teljes adatbázist végigszkenneljük az előállított állapotot keresve. Ekkor nem csak körfigyelést, de *hurokfigyelést* is végzünk, hiszen az adatbázisban egyáltalán nem lesz két azonos állapot. Nyilvánvalóan a duplikációk teljes kizárása gyorsítja a keresést, azonban a teljes adatbázis állandó jellegű végigszkennelése meglehetősen költséges eljárás.
- Csak az aktuális csúcsba vezető ágat szkenneljük végig. Ekkor csak körfigyelést végzünk, a hurkokat nem zárjuk ki az adatbázisból. Kevésbé költséges eljárás, mint az előző, de az adatbázisban előfordulhatnak duplikált állapotok.

Az, hogy kör-, illetve hurokfigyelést beépítünk-e a keresőnkbe, a megoldandó probléma jellegétől függ. Ha a probléma állapottér-gráfja egyáltalán nem tartalmaz kört, akkor nyilván nincs körfigyelésre szükség. Ellenkező esetben mindenképpen be kell építenünk valamilyen körfigyelési eljárást. Ha a gráf kevés hurkot tartalmaz, akkor elegendő a kevésbé költséges eljárást alkalmaznunk, ha viszont olyan sok hurkot tartalmaz, hogy a duplikációk a keresést lényegesen lelassítják, akkor célszerűbb az adatbázis teljes végigszkennelését választani (hiszen ez az körfigyelési eljárás egyúttal hurokfigyelést is végez).

Hogy a (2)(a) pontban *melyik nyílt csúcsot választjuk kiterjesztésre*, megegyezés kérdése. Az egyes konkrét (már nem általános) keresőfával keresők *ezen a ponton térnek el* egymástól. A következő fejezetekben a legnevezetesebb ilyen keresőket vesszük sorra, megvizsgálva, hogy az adott keresőket milyen jellegű problémák megoldására érdemes használni.

***IMPLEMENTÁCIÓS KÉRDÉSEK***

- **Milyen adatszerkezettel valósítsuk meg az adatbázis csúcsait?**
  Minden csúcsban le kéne tárolni a csúcs *gyermekeire mutatókat*. Ez minden csúcsban egy *csúcslista* alkalmazását kívánja meg. Sokkal gazdaságosabb egy csúcsban a csúcs *szülőjére mutatót* tárolni, mivel míg gyermeke több is lehet egy csúcsnak, szülője csak egy lehet (kivéve a gyökércsúcsot, mert annak nincs szülője).
- **Hogyan tartsuk nyilván a nyílt és zárt csúcsokat?**
  Egyik lehetőség, hogy magukban a csúcsokban tároljuk azt az információt, hogy az adott csúcs nyílt vagy zárt. Ez a megoldás azt vonná maga után, hogy a kiterjesztendő nyílt csúcsot mindig meg kéne keresnünk a fában. Ez egyrészt túlságosan költséges dolog, másrészt ha az előző pont alapján csak szülőre mutatót tárolunk a csúcsban (és gyermekekre mutatókat nem), akkor a fa bejárása fentről lefelé lehetetlen.
  Ezért célszerű lenne a csúcsokat még külön egy-egy listában is letárolni. Alkalmaznánk a *nyílt csúcsoknak egy listáját*, és a *zárt csúcsoknak is egy listáját*. A kiterjesztendő csúcsot a nyíltak listájából vennénk (mondjuk a lista elejéről), majd a kiterjesztés után átpakolnánk a zártak listájába. A kiterjesztés során előálló új csúcsokat a nyíltak listájához fűznénk hozzá.
- **Hogyan alkalmazzunk körfigyelést?**
  Ha a körfigyelési technikák közül csak az *aktuális ág végigszkennelését* alkalmazzuk, akkor könnyű dolgunk van: a szülőkre mutatók miatt ez könnyen kivitelezhető.
  Ha a *teljes adatbázis végigszkennelését* választjuk (azaz hurokfigyelést is végzünk), akkor már nehezebb dolgunk van, mivel a teljes fát be kell járni, ami a szülőkre mutatók miatt lehetetlen. Ha viszont a csúcsokat a fent írt két listában is letároljuk, akkor ez a fajta körfigyelés is könnyen elvégezhető: a nyílt csúcsok listáját is és a zárt csúcsok listáját is végig kell szkennelnünk.

## 4.3.2. SZISZTEMATIKUS KERESŐFÁVAL KERESŐK

A 21. oldalon a keresőknek egy olyan csoportosítását láttuk, mely szisztematikus keresőket és

heurisztikus keresőket különböztet meg. Ebben a fejezetben a keresőfával keresők szisztematikus változataival ismerkedünk meg.

## *Szélességi kereső*

A szélességi kereső a nyílt csúcsok közül mindig a *legkisebb mélységben* levő csúcsot terjeszti ki (ha több ilyen is van, akkor ezek közül bármelyiket választhatja). Ily módon az adatbázisban tárolt fa egyes szintjeit (azaz azonos mélységben levő csúcsait) *széltében* végig előállítjuk, és csak ezután lépünk a következő szintre. Innen a kereső neve.

Az egzakt leírás érdekében az adatbázisban tárolt csúcsok mindegyikéhez egy-egy ún. *mélységi számot* rendelünk. A szélességi kereső tehát a *legkisebb mélységi számú* nyílt csúcsot választja kiterjesztésre.

**8. Definíció:** Egy keresőfabeli csúcs *mélységi száma* a következőképpen definiált:

- $g(s)=0$, ahol $s$ a startcsúcs.
- $g(m)=g(n)+1$, ahol az $n$ csúcsnak az $m$ gyermeke.

Könnyen belátható, hogy a szélességi kereső ha talál megoldást, akkor *optimális megoldást* talál. Ennek persze ára van: a fa minden egyes szintjét széltében le kell generálni, ami egyes problémák esetén rengeteg csúcsot jelent. Ez a gyakorlatban különösen akkor okoz gondot, ha a megoldandó problémának hosszú megoldásai vannak; ezek megtalálása roppant időigényes tud lenni.

- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *garantált* az optimális megoldás előállítása.
- **Tesztelés:** előrehozható.

Bár a szélességi kereső körfigyelési technika alkalmazása nélkül is véges sok lépésben megoldást talál (már ha egyáltalán van megoldás), bizonyos problémák esetén célszerű beépíteni valamelyik plusz tesztet a 4.3.1. fejezetben leírtak közül. Természetesen azon problémák esetén éri ez meg, melyek állapottér-gráfjában gyakoriak a körök (és a hurkok), ugyanis lényegesen le tudjuk csökkenteni az adatbázisba kerülő csúcsok számát. Arról nem is beszélve, hogy ha nincs megoldás, akkor ezt is véges sok lépésben felismerjük.

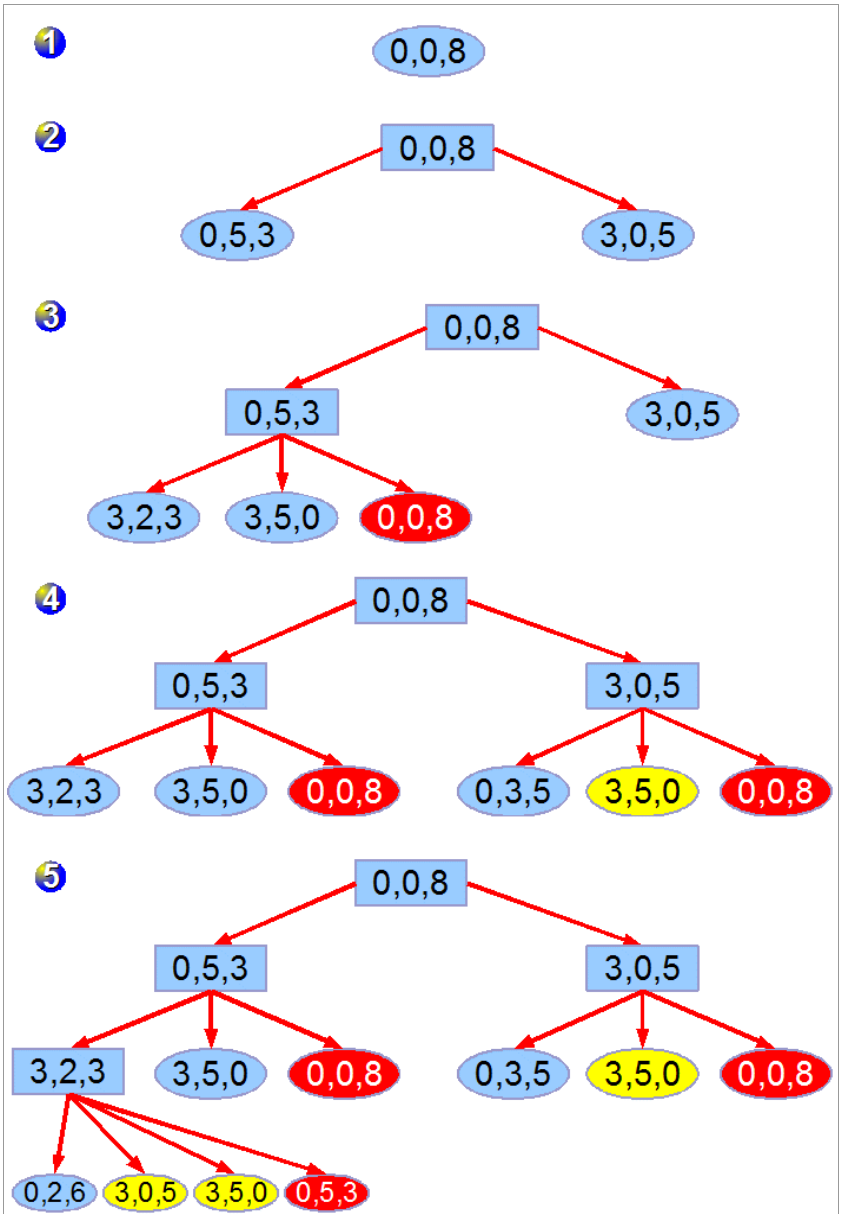
### *Implementációs kérdések*

- **Hogyan válasszuk ki a legkisebb mélységi számú nyílt csúcsot?**
  Az egyik lehetőség, hogy minden csúcsban tároljuk az adott csúcs mélységi számát, és minden kiterjesztés előtt megkeressük a nyílt csúcsok listájában a legkisebb ilyen értékű csúcsot.
  Ehhez egy további lehetőség, hogy a nyílt csúcsokat mélységi számuk szerint *rendezve* tároljuk a listában. A rendezettség biztosításának az a legolcsóbb módja, hogy az új csúcsokat mindig *rendezve szúrjuk be* a listába (mélységi szám szerint).
  Könnyű észrevenni azonban, hogy az új csúcsok mindig a lista végére fognak így kerülni. Azaz a nyílt csúcsok listája olyan adatszerkezetként fog funkcionálni, melybe az elemek *hátul kerülnek be és elől távoznak* (kiterjesztéskor). Azaz a nyílt csúcsok tárolására a legcélszerűbb egy *sort* (queue) alkalmazni.

### *Példa*

A 16. ábrán a szélességi kereső által felépített keresőfa látható a 3 kancsó probléma esetén, egy pár lépés erejéig. Érdemes végigkövetni a 2. ábrán, hogy hogyan épül fel ez a keresőfa az állapottér-gráf alapján. A keresőfában a nyílt csúcsokat ellipszisbe, a zárt csúcsokat téglalapba zárva ábrázolom. Az itt látható keresőfát kör- és hurokfigyelés nélküli szélességi kereső építi fel. A

pirossal színezett csúcsokat hagyná el az a körfigyelési technika, mely csak az *ágakon szereplő duplikációkat* szűri ki. A sárgával színezett csúcsok azok, melyeket (a pirosakon kívül) már a *teljes kör- és hurokfigyeléssel* (azaz az adatbázis végigszkennelésével) is kidobnánk. Látható, hogy ez az utóbbi körfigyelési technika mennyire lecsökkenti az adatbázis méretét, legalábbis a 3 kancsó probléma esetén.



*16. ábra: Szélességi kereső a 3 kancsó probléma esetén*

## MÉLYSÉGI KERESŐ

A mélységi kereső a *legnagyobb mélységi számú* nyílt csúcsot terjeszti ki (ha több ilyen is van, akkor ezek közül bármelyiket választhatja). Ennek az az eredménye, hogy a fa szintjeit nem kell széltében előállítanunk, akár a nagy mélységben megbúvó célcsúcsokat is gyorsan megtalálhatjuk. Ezen gyorsaságnak persze ára van: a megtalált megoldás egyáltalán *nem biztos, hogy optimális*.

A mélységi kereső a szélességi keresőhöz viszonyítva általában gyorsabban megoldást talál; ez persze függ az állapottér-gráf milyenségétől. Ha a gráf elég sűrűn tartalmaz célcsúcsokat, esetleg nagy mélységben, akkor a mélységi kereső jobb választás, ha ritkán és kis mélységben, akkor viszont a szélességi. Persze ha a célunk az, hogy optimális megoldást keressünk, a mélységi kereső (általánosságban) szóba sem jöhet.

- **Teljesség:**
  - Ha van megoldás, akkor *véges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.
- **Tesztelés:** előrehozható.

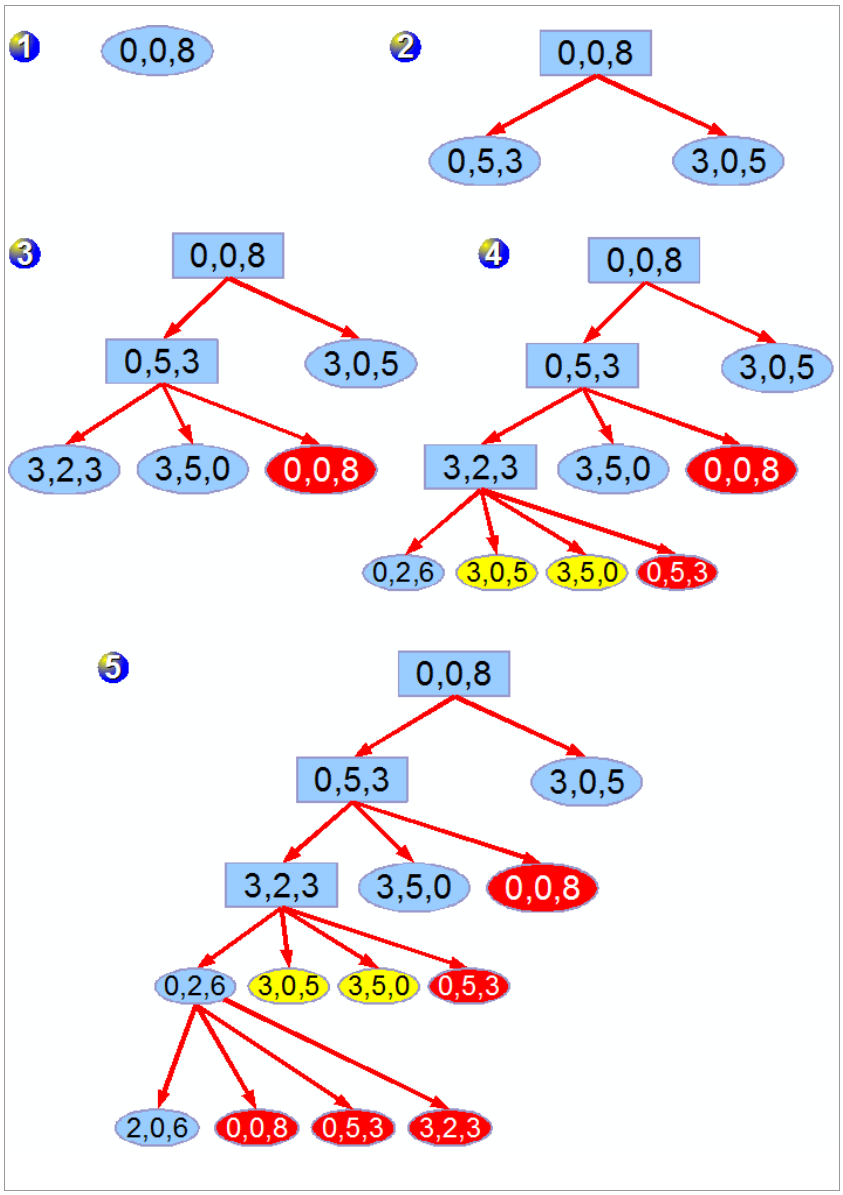
Feltűnő a hasonlóság a mélységi kereső és a *backtrack* kereső között. Mindkettő „mélységében" tárja fel az állapottér-gráfot, mindkettő véges gráf esetén teljes (persze ha nem alkalmazunk valamilyen körfigyelési technikát), és a kettő közül egyik sem optimális megoldás keresésére való. A kérdés: milyen plusz szolgáltatást nyújt a mélységi kereső a backtrackhez képest? Mit nyerünk azzal, hogy nem csak a startcsúcsból az aktuális csúcsba vezető utat tároljuk az adatbázisban, hanem az *összes* eddig előállított csúcsot? A válasz egyszerű: *hurokfigyelést* tudunk végezni. Vagyis a mélységi keresőt a 4.3.1. fejezetben leírt kör- és hurokfigyelési technikával ötvözve (vagyis az adatbázisnak az új csúcsok beszúrása előtti végigszkennelésével) el tudjuk azt érni, hogy a keresés során egy állapotba legfeljebb csak egyszer „fussunk bele". Ilyenfajta vizsgálat backtrack kereső esetén elképzelhetetlen volt.

#### *Implementációs kérdések*

- **Hogyan válasszuk ki a legnagyobb mélységi számú nyílt csúcsot?**
  Míg a szélességi keresőnél a nyílt csúcsokat egy sorban, addig mélységi kereső esetén egy *veremben* (stack) célszerű tárolni.

#### *Példa*

A 17. ábrán a mélységi kereső által épített keresőfát mutatom be a 3 kancsó probléma esetén. Mivel ennek a problémának az állapottér-gráfja nem véges (vannak benne körök), így mindenképpen szükség van valamilyen körfigyelési technika alkalmazására. Pirossal a körfigyelés során eltávolított csúcsokat, sárgával a hurokfigyelés során eltávolítottakat jelölöm.




## *Optimális kereső*

Az optimális kereső azon problémák esetén használható kereső, melyeknél az operátoralkalmazásokhoz valamilyen *költség* van rendelve. Frissítsük fel a költségekre vonatkozó fogalmakat és jelöléseket a 11. oldalról: az $o$ operátor $a$ állapotra való alkalmazásának költségét $költség_o(a)$ -val jelöljük, illetve egy megoldás költsége alatt a megoldást alkotó operátoralkalmazások költségeinek összegét értjük.

A *szélességi és a mélységi* kereső olyan keresők voltak, melyeknél az operátoralkalmazásokhoz nem rendeltünk költséget. Természetesen nem minden probléma ilyen, ezért van szükség az optimális keresőre. Hogy leírjuk az optimális kereső keresési stratégiáját, vezessük be a következő fogalmat (a 37. oldalon definiált mélységi szám mintájára):

**9. Definíció:** Egy keresőfabeli *csúcs költsége* a következőképpen definiált:

- $g(s)=0$ , ahol $s$ a startcsúcs.
- $g(m)=g(n)+költség_o(n)$ , ahol $n$ -ből az $o$ operátor alkalmazásával nyertük az $m$ -et.

Az *optimális kereső* a nyílt csúcsok közül mindig a *legkisebb költségűt* terjeszti ki.

Vegyük észre, hogy a szélességi és a mélységi keresők esetén használt *mélységi szám*

tulajdonképpen speciális csúcsköltség, mégpedig azon esetre vonatkoztatva, mikor minden $o$ operátor és minden $n$ állapot esetén $költség_o(n)=1$. Azaz a következőt lehet megállapítani: *a szélességi kereső olyan speciális optimális kereső, melynél minden operátoralkalmazás költsége egységnyi*.

Az optimális kereső tulajdonságai:

- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *garantált* az optimális megoldás előállítása.
- **Tesztelés:** nem hozható előre, mivel a tesztelés előrehozásával a kereső nem feltétlenül az optimális megoldást állítaná elő.

Míg szélességi és mélységi keresők esetén a *körfigyelés* nagyon egyszerűen megoldható volt (ha egy csúcs már szerepelt az adatbázisban, nem vettük fel újra), addig az optimális kereső esetén valamivel bonyolódik. Vizsgáljuk meg azt a szituációt, mikor az $m$ csúcsot előállítjuk az $n$ csúcs gyermekeként, és be szeretnénk szúrni az adatbázisba! Tegyük fel, hogy az $m$ már szerepel az adatbázisban! Ekkor két eset lehetséges aszerint, hogy az adatbázisban levő $m$ csúcs költsége (amit $g(m)$-mel jelölünk) milyen relációban áll az új csúcs költségével (ami nem más, mint $g(n)+költség_o(n)$):

- $g(m)\le g(n)+költség_o(n)$: Az adatbázisban az új csúcs egy már optimálisabb (pontosabban: nem nagyobb költségű) úton le van tárolva. Ekkor az új csúcsot *nem vesszük fel* az adatbázisba.
- $g(m)> g(n)+költség_o(n)$: A $m$-et most sikerült optimálisabb úton előállítani. Ekkor *cseréljük le* az adatbázisban tárolt $m$-et az új csúcsra!

Az utóbbi művelet könnyen elvégezhető: az $m$-et át kell láncolni a fában az $n$ alá, illetve az $m$-hez rendelt költséget (g-értéket) frissíteni kell $g(n)+költség_o(n)$-re.

Akkor okozhat gondot egy ilyen átláncolás, ha az $m$-nek a fában már vannak gyermekei, hiszen ekkor az $m$-ből egy részfa indul, amiben a csúcsok g-értékét szintén frissíteni kell (mivel mindannyiuk g-értéke a $g(m)$-ből származtatott). Más szóval akkor van probléma, ha az $m$ zárt. Ezt nevezzük *zárt csúcsok problémájának*. A zárt csúcsok problémájára több megoldást is ki lehet dolgozni, de szerencsére erre az optimális kereső esetében nem lesz szükség, mivel be lehet látni, hogy az optimális kereső esetén a zárt csúcsok problémája *nem fordulhat elő*. Ezt a következő állítás szavatolja:

**10. Állítás:** Ha az optimális kereső adatbázisában az $m$ csúcs zárt, akkor $g(m)$ optimális.

*Bizonyítás:* Ha $m$ zárt, akkor hamarabb terjesztettük ki, mint az aktuálisan kiterjesztett $n$-et. Azaz:

$$g(m)\le g(n)$$
$$\text{ß}$$
$$g(m)< g(n)+költség_o(n)$$

Azaz az $m$-hez feltárt bármilyen új út költségesebb, mint az adatbázisban levő.

***IMPLEMENTÁCIÓS KÉRDÉSEK***

- **Hogyan válasszuk ki a legkisebb költségű nyílt csúcsot?**
  Optimális kereső esetén a nyílt csúcsok tárolására sajnos se sor, se verem nem alkalmazható. Sajnos ezúttal ténylegesen csak az a megoldás járható, hogy a csúcsokban l*etároljuk a költségüket*, és a nyílt csúcsok listája eszerint a költség szerint lesz rendezett. Mint azt

korábban írtam, a rendezettséget az új csúcsok *rendezett beszúrásával* érdemes biztosítani.

## 4.3.3. Heurisztikus keresőfával keresők

Az optimális kereső és a szélességi kereső (mely speciális optimális kereső) nagyon jó tulajdonságokkal bírnak, melyek közül az optimális megoldás előállításának szavatolása a leglényegesebb. De ennek elég súlyos ára van, és ez nem más, mint a hosszú futási idő. Ez utóbbit pedig az adatbázis nagy méretűre való hízlalása okozza. Az adatbázisban tárolt keresőfa nagyon sok csúcsból állhat. Például ha a fában egy csúcsnak max. $d$ gyermeke van és az optimális megoldás hossza $n$, akkor a keresőfa legfeljebb

$$1+d+d^2+d^3+\ldots+d^n = \frac{d^{n+1}-1}{d-1}$$

csúcsot tartalmaz. Azaz a keresőfa csúcsainak száma a megoldás hosszának *exponenciális* függvénye.

Az optimális kereső ún. szisztematikus kereső, azaz valamiféle „vak” keresési stratégiával bír, és ez okozza a sok csúcs legenerálásának szükségességét. Hogy ezt elkerüljük, megpróbáljuk a keresőinket valamiféle „előrelátással” felruházni, saját emberi intuíciónkat a keresőkbe beledrótozni, és ennek eszköze lesz a *heurisztika*. A heurisztika nem más, mint egy *becslés*, mely egy csúcsra nézve megmondja, hogy a csúcs melyik gyermeke felé induljunk tovább a keresésben. Vegyük észre, hogy ténylegesen csak becslésről beszélhetünk, hiszen a gyermekekből induló részfákat a kereső még nem generálta le, azaz nem lehetünk biztosak, hogy tényleg jó irányba (azaz valamelyik célcsúcs felé) indulunk-e a fában.

Mindenesetre nekünk az lenne a legjobb, ha olyan heurisztikánk lenne, mely minden csúcsban pontosan meg tudja mondani, hogy merre (melyik gyermeke felé) van a legközelebbi célcsúcs. Az ilyen heurisztikát *perfekt heurisztikának* nevezzük, és egy ilyen heurisztika legfeljebb

$$1+d+d+\ldots+d = 1+n\cdot d$$

csúcsot generál le, azaz ilyen heurisztika esetén a csúcsok száma már csak *lineáris* függvénye a megoldás hosszának. Persze perfekt heurisztika *nem létezik*, hiszen ha lenne, akkor már előre ismernénk a megoldást, akkor meg minek megkeresni?!

A *heurisztika* definíciója nagyon egyszerű, melyet a helymászó módszer kapcsán a 4.1.3. fejezetben már megadtam, és ennek a lényege, hogy a heurisztika mint függvény minden állapothoz egy-egy természetes számot rendel. Ezzel a számmal azt becsüljük, hogy az adott állapotból mekkora összköltségű operátoralkalmazásokkal jutunk el valamely célállapothoz. A keresőfában gondolkodva: egy csúcs heurisztikája megadja, hogy hozzávetőlegesen *mekkora költségű úton* tudunk eljutni valamelyik célcsúcsba. Olyan állapottér-gráf esetén pedig, ahol az élek egységnyi költségűek (pl. a szélességi keresőt alkalmaztuk ilyen gráfokban), a csúcs heurisztikája azt becsüli meg, hogy *hány élnyi távolságra* van valamelyik célcsúcs. Mindenesetre a heurisztikák alkalmazásával a keresőfa méretét reméljük csökkenteni.

### *Best-first kereső*

A best-first kereső olyan keresőfával kereső, mely a *legkisebb heurisztikájú* nyílt csúcsot választja kiterjesztésre. Az így kapott kereső előnye az egyszerűsége, ugyanakkor a fontos tulajdonságok tekintetében nagyon alulmarad az optimális (és a szélességi) keresőkkel vett összehasonlításban.

- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
    *Kivétel*: ha minden állapot heurisztikája egyenlő (azaz egységnyi).

- Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.

- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.
- **Tesztelés:** előrehozható, hiszen az optimális megoldás előállításáról úgysem beszélhetünk.

#### *IMPLEMENTÁCIÓS KÉRDÉSEK*

- **Hogyan válasszuk ki a legkisebb heurisztikájú nyílt csúcsot?**
  Nyilvánvalóan a nyílt csúcsok listájának a csúcsok heurisztikája szerint rendezettnek kell lennie.

## *A-ALGORITMUS*

A best-first kereső kapcsán láttuk azt, hogy a heurisztika alkalmazása az optimális kereső (és a szélességi kereső) szinte összes jó tulajdonságát elrontotta. Ugyanakkor a heurisztika alkalmazására szükség van az optimális kereső effektivitási problémáinak megoldásához. Az optimális kereső nem volt effektív, mivel a keresés során csak a „múltat" vette figyelembe. A best-first kereső ezzel szemben csak a „jövőbe tekintett", azaz nem „tanult" a keresés múltjából, és ezért nem feltétlenül az ésszerű irányba haladt.

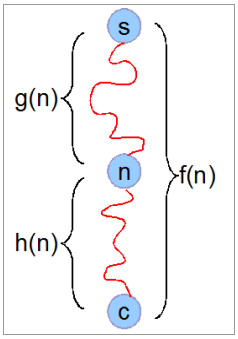
Az ötlet: ötvözzük az optimális és a best-first keresőket! Az így kapott keresőt *A-algoritmusnak* nevezzük. Hogy leírjuk az A-algoritmus keresési stratégiáját, vezessük be a következő fogalmat:

**11. Definíció:** Egy keresőfabeli $n$ *csúcs összköltségét* $f(n)$-nel jelöljük, és a következőképpen definiáljuk:

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

Az *A-algoritmus* mindig a *legkisebb összköltségű* (f-értékű) nyílt csúcsot választja kiterjesztésre.

Egy $n$ csúcs f-értéke tulajdonképpen nem más, mint a startcsúcsból az $n$-en keresztül valamely célcsúcsba vezető út (azaz az $n$-en átvezető valamely megoldás) becsült költsége.




Vegyük észre, hogy *az optimális kereső egy olyan speciális A-algoritmus, ahol a heurisztika minden csúcs esetén nulla*. Ez persze azt is jelenti, hogy a szélességi kereső is speciális A-algoritmus: egységnyi költségű operátoralkalmazások és azonosan nulla heurisztika.

Az A-algoritmus esetén van azonban egy nagyon kényes pont: a *körfigyelés*. Ugyanaz mondható el, mint az optimális kereső esetében (lásd a . fejezetet): ha a beszúrandó csúcs már szerepel az adatbázisban és most kisebb költséggel állítottuk elő, cseréljük le az adatbázisban szereplő csúcsot az új csúcsra (frissítsük a szülőjét és a g-értékét)! Míg azonban az optimális kereső esetén nem fordulhatott elő a *zárt csúcsok problémája*, az A-algoritmus esetén bizony *előfordulhat*. Azaz megtörténhet, hogy a lecserélendő csúcs ($m$) zárt, vagyis az adatbázisban vannak leszármazottjai, ami annak kényszerét jelenti, hogy az ő g-értéküket is frissíteni kell. Erre a lehetséges megoldások:

(1) Járjuk be az adatbázisban az $m$-ből induló részfát, és frissítsük a benne található csúcsot g-értékét! Mivel az implementációs megfontolások miatt csak szülőre mutatókat használunk, egy ilyen bejárás *nem kivitelezhető*.

(2) Bízzuk az értékek frissítését az A-algoritmusra! Ezt úgy tudjuk kikényszeríteni, hogy az $m$-et *visszaminősítjük nyílt csúccsá*. Ez azt jelenti, hogy az $m$ összes leszármazottja (azaz az $m$-ből induló részfa összes csúcsa) egyszer újra elő lesz állítva, mégpedig a jelenlegi g-értéküknél kisebb költséggel, vagyis egyszer az ő g-értéküket is frissíteni fogja az A-algoritmus.

(3) Előzzük meg a zárt csúcsok problémájának előfordulását! Az optimális keresőnél ez a probléma egyáltalán nem fordult elő, így – mivel az optimális kereső speciális A-algoritmus – felmerül kérdésként, hogy *milyennek kéne a heurisztikának lennie*, hogy a zárt csúcsok problémája egyáltalán ne alakulhasson ki?

A (3) lehetőséget majd a . fejezetben vizsgáljuk meg. Most egyelőre alkalmazzuk a (2) megoldást! Azaz megengedjük az algoritmus számára, hogy zárt csúcsokat bizonyos esetekben visszaminősítsen nyílttá. Ez azonban előhozza annak a veszélyét, hogy az algoritmus soha nem áll meg, mivel egy csúcsot ki tudja, hányszor (akár végtelen alkalommal is) oda-vissza minősíthetünk. Szerencsére azonban be lehet bizonyítani a következő állítást:

**12. Állítás:** A keresőfa bármely csúcsa csak véges sokszor lesz visszaminősítve nyílttá.

*Bizonyítás:* Definíció szerint tudjuk (11. oldal), hogy minden operátoralkalmazás költsége pozitív. Jelöljük a legkisebb ilyen költséget $\delta$-val! Könnyen belátható, hogy egy csúcs nyílttá való visszaminősítése során a csúcs g-értéke legalább $\delta$-val csökken. Az is triviális, hogy minden csúcs g-értékének van egy alsó korlátja: a csúcsba jutás optimális költsége (a startcsúcsból). Mindezek maguk után vonják a bizonyítandó állítás igazságát.

Ezek után vizsgáljuk meg az A-algoritmus tulajdonságait! Az A-algoritmus teljességéről az előző állítás alapján a következő mondható el:

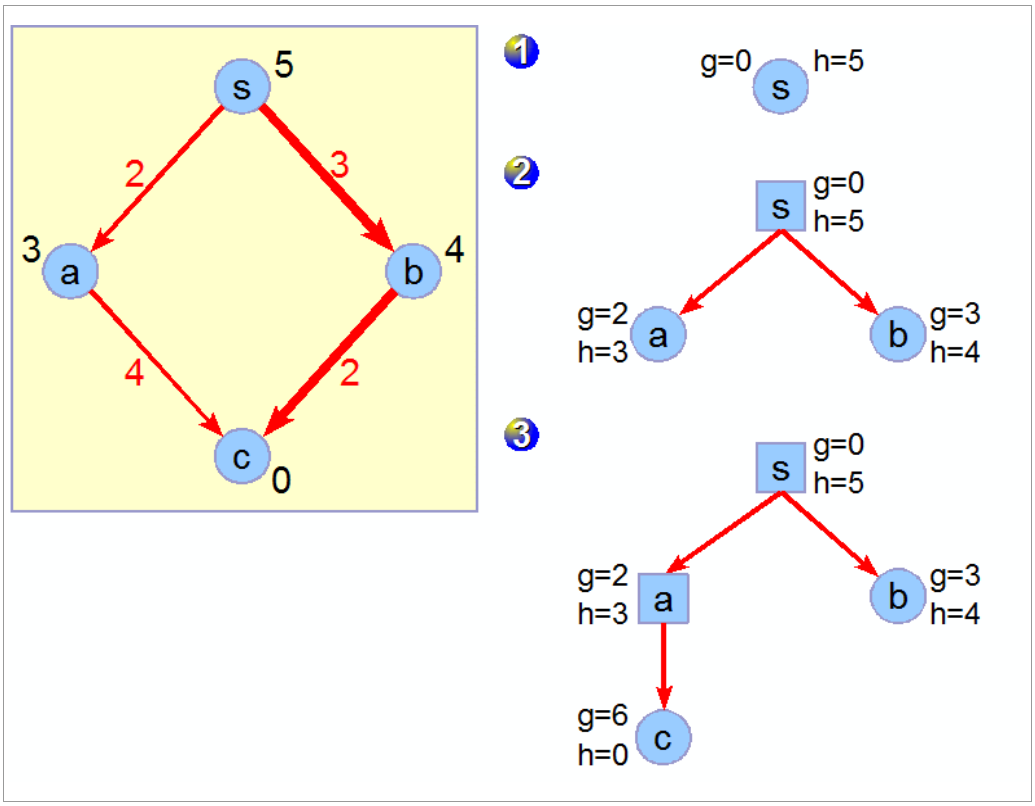
### Teljesség:

- Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
- Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.

Vajon mi a helyzet az előállított megoldás *optimalitásával*? Vajon az A-algoritmus garantáltan az optimális megoldást állítja elő? A 18. ábrán látható ellenpélda megmutatja, hogy nem. Az ábrán a bal oldalon egy állapottér-gráf látható, melyben a csúcsok mellé írtam azok heurisztikáját: az $s$ startcsúcs mellé például az $5$-öt, vagy a $c$ célcsúcs mellé a $0$-át (nagyon helyesen, hiszen minden célállapotnak $0$ kell legyen a heurisztikája). Az élekre az adott él (mint operátoralkalmazás) költségét írtam. Az ábrán vastagon kihúztam az optimális megoldást.

Az ábra jobb oldalán lépésről lépésre végigkövetem az A-algoritmus működését ezen az állapottér-gráfon. Minden csúcs mellé odaírtam azok g- és h-értékét (költségét és heurisztikáját). A 2. lépésben látható, hogy az $a$ nyílt csúcsot fogja a kereső kiterjeszteni, mivel ennek f-értéke $2+3=5$, ami kisebb mint a másik nyílt csúcs $3+4=7$ f-értéke.

A 3. lépésben pedig be is fejeződik a keresés, hiszen a kereső a $6+0=6$ f-értékű $c$ csúcsot választja kiterjesztésre, ami pedig célcsúcs. Vegyük észre, hogy így a kereső egy $6$ költségű megoldást talált meg, ami nem az optimális megoldás!




Vagyis az A-algoritmus optimalitásáról (és a tesztelés előrehozásáról) a következő mondható el:

- **Optimalitás:** *nem garantált* az optimális megoldás előállítása.
- **Tesztelés:** előrehozható, hiszen az optimális megoldás előállítása sem garantált.

### *Implementációs kérdések*

- **Hogyan válasszuk ki a legkisebb összköltségű nyílt csúcsot?**
  Az optimális keresőhöz hasonlóan a csúcsokban l*etároljuk a költségüket*, és ebből származtatjuk az f-értéküket. A nyílt csúcsok listáját f-érték szerint lesz rendezzük.
- **Hogyan kezeljük a zárt csúcsok problémáját?**

  A 43. oldalon leírt lehetőségek közül érdemes a (2)-t választani. Mikor egy csúcsot a meglévőnél jobb költséggel szúrunk be az adatbázisba, érdemes a régi (zárt) csúcsot törölni, majd ezután beszúrni az új (nyílt) csúcsot.

### *Példa*

A 19. és 20. ábrákon az A-algoritmus által a Hanoi tornyai problémára felépített keresőfa látható, lépésenként, egészen a megoldás megtalálásáig. A következő heurisztikát választottam:

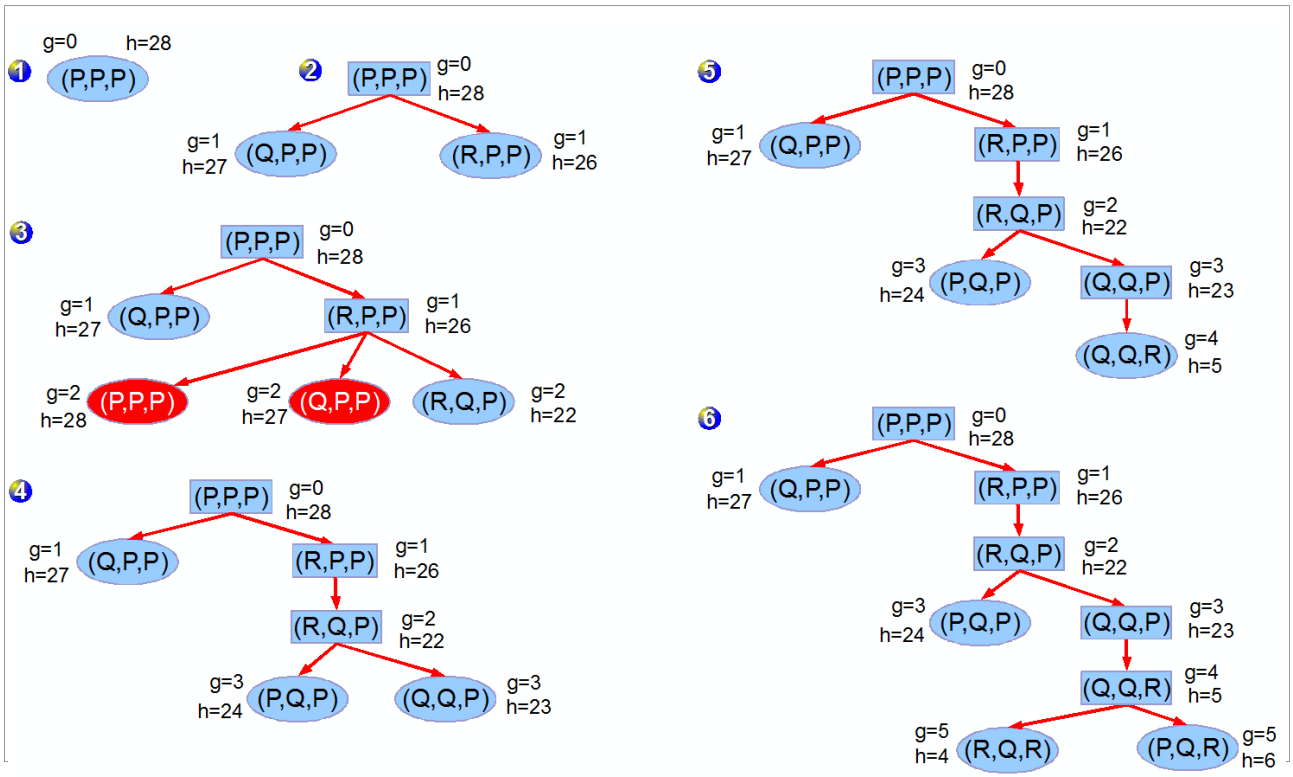
$$h(a_1, a_2, a_3) = 28 - index(a_1) - 4 \cdot index(a_2) - 9 \cdot index(a_3)$$

ahol

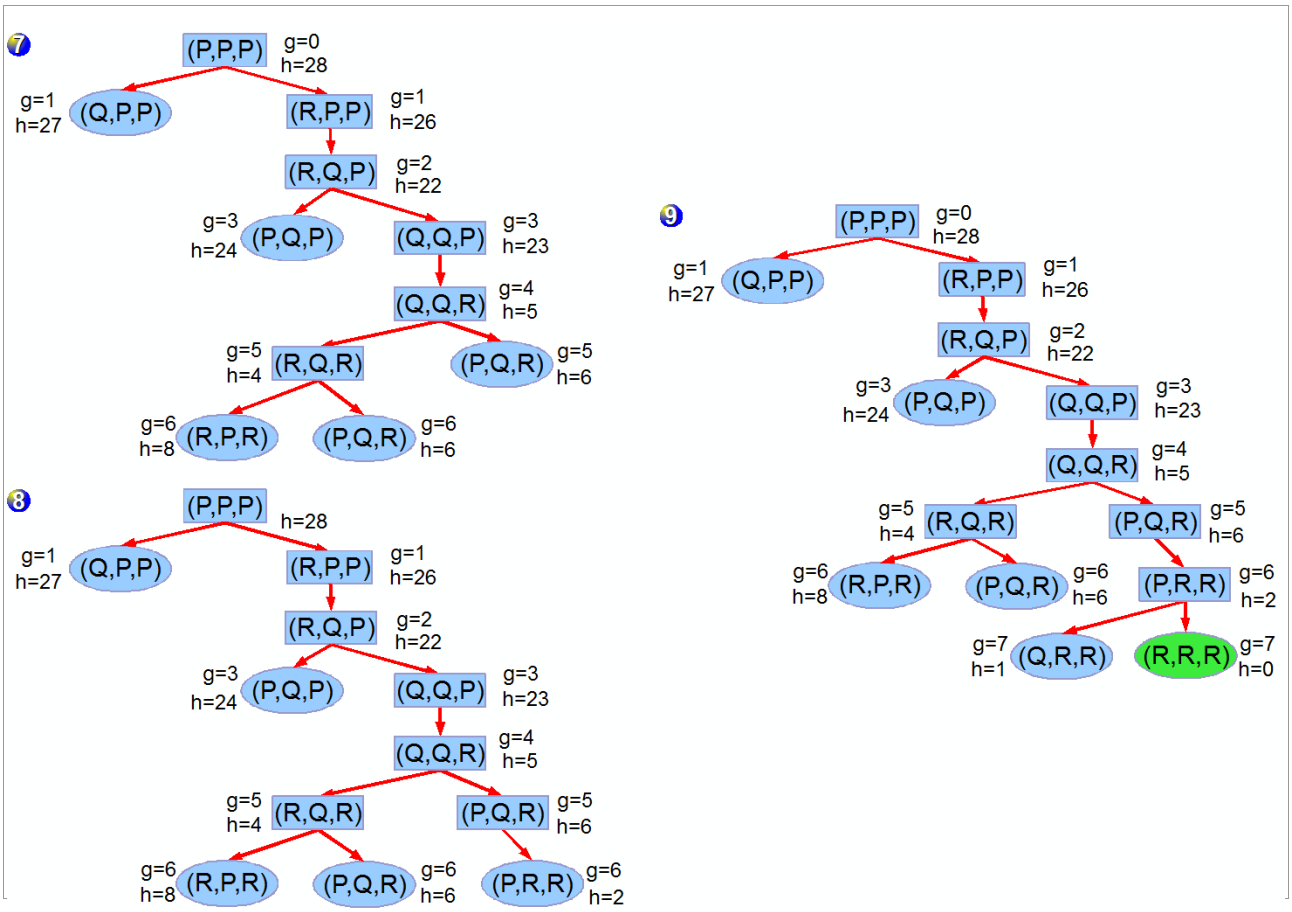
$$index(a_i) = \begin{cases} 0 & , \text{ha } a_i = P \\ 1 & , \text{ha } a_i = Q \\ 2 & , \text{ha } a_i = R \end{cases}$$

Vagyis a korongokhoz sorszámokat rendelek: a $P$-hez $0$-t, a $Q$-hoz $1$-et, az $R$-hez $2$-t.

Látható, hogy a heurisztikában nagyobb korongokat nagyobb értékekkel súlyozom, hiszen a cél a minél nagyobb korongok átmozgatása az $R$ rúdra. Az is látható, hogy a $28$-ból való kivonás a célból történik, hogy az $(R, R, R)$ célállapotban a heurisztika $0$ legyen. Megjegyzem, hogy a Hanoi tornyai állapottér-reprezentációjában célszerűbb lett volna a korongokat eleve numerikus értékekkel (0,1 és 2) jelölni.




A 3. lépésben pirossal feltüntettem két olyan csúcsot, melyeket az A-algoritmus alapértelmezés szerint nem vesz hozzá az adatbázishoz, hiszen azok f-értéke nem kisebb, mint az adatbázisban már megtalálható ugyanolyan tartalmú csúcsok f-értéke. A további lépésekben a hasonlóképpen eldobott csúcsokat fel sem tüntetem.




A következő konzekvenciákat lehet levonni:

- A kereső az optimális megoldást találta meg. Kérdés, hogy ez véletlenül alakult-e így, vagy esetleg a választott heurisztika valamiképpen kényszerítette ki ezt a keresőből? Lásd a következő fejezetet!
- A keresés során nem fordult elő a zárt csúcsok problémája. Azaz nem volt olyan eset, mikor egy újonnan előállított csúcsot zárt csúcsként és nagyobb f-értékkel találtam meg az adatbázisban.
- A keresést nagyon célirányossá tette a heurisztika használata. A keresés során tulajdonképpen egyetlenegyszer indultunk csak el nem az optimális irányba: a 6. lépésben, amikor az $(R,Q,R)$ csúcsot terjesztettük ki. De 1 lépés múlva már vissza is tért a kereső az optimális útvonalra.

## A*-ALGORITMUS

Az A*-algoritmus olyan A-algoritmusfajta, mely *garantálja az optimális megoldás előállítását*. Biztosan tudjuk, hogy például az optimális kereső ilyen kereső, és az is nyilvánvaló, hogy az azonosan nulla heurisztikának köszönheti ezen pozitív tulajdonságát. Kérdés: milyen tulajdonságú (de azért nem azonosan nulla) heurisztikával tudnánk az optimális megoldás előállítását garantálni?

Hogy ezt pontosan leírjuk, be kell vezetnünk a következő jelöléseket, a keresőfa minden $n$ csúcsára:

- $h^*(n)$ : az $n$ -ből valamely célcsúcsba jutás *optimális* költsége.
- $g^*(n)$ : a startcsúcsból $n$ -be jutás *optimális* költsége.
- $f^*(n)=g^*(n)+h^*(n)$ : értelemszerűen a startcsúcsból $n$ -en keresztül valamely célcsúcsba jutás optimális költsége.

**13. Definíció:** Egy $h$ heurisztika *alulról becslő heurisztika*, ha minden $a$ állapotra teljesül a következő:

$$h(a) \le h^*(a) .$$

Az *A*-algoritmus* olyan A-algoritmus, melynek a *heurisztikája alulról becslő*. A továbbiakban bebizonyítom, hogy az A*-algoritmus garantáltan az optimális megoldást állítja elő. Ehhez fel fogunk használni két segédtételt (lemmát) is.

**14. Lemma:** Ha van megoldás, akkor az A*-algoritmusra bármely időpillanatban igaz a következő: *az optimális megoldásnak van eleme a nyílt csúcsok között*.

*Bizonyítás:* A bizonyítás teljes indukcióval történik. Jelöljük az *optimális megoldásban* szereplő csúcsokat a következőképpen:

$$(s=) n_1 \ , \ n_2 \ , \ \dots \ , \ n_r (=c)$$

- Az 1. kiterjesztés előtt: az $s$ startcsúcs nyílt csúcs.
- **Indukciós feltevés:** Tegyük fel, hogy az aktuális kiterjesztés előtt van az optimális megoldásnak eleme a nyílt csúcsok között, legyen ezek között a legkisebb indexű az $n_i$.
- **Indukciós lépés:** Nézzük meg, mi a helyzet a következő kiterjesztés előtt! Két lehetőség van:
  - Ha nem az $n_i$-t terjesztettük most ki, akkor az $n_i$ nyílt maradt.
  - Ha az $n_i$-t terjesztettük most ki, akkor az $n_{i+1}$ nyíltként bekerült az adatbázisba.

**15. Lemma:** Az A*-algoritmus által kiterjesztésre kiválasztott bármely $n$ csúcsra:

$$f(n) \le f^*(s)$$

*Bizonyítás:* Az előző lemma alapján az optimális megoldásnak mindig van eleme a nyílt csúcsok között. Jelöljük a megoldás első ilyen csúcsát $n_i$-vel. Mivel $n_i$ már az optimális úton tárolódik az adatbázisban, így:

$$g(n_i) = g^*(n_i)$$

Jelöljük $n$-nel a kiterjesztésre kiválasztott csúcsot! Mivel az A*-algoritmus mindig a legkisebb f-értékű csúcsot választja kiterjesztésre, így tudjuk, hogy $f(n)$ nem nagyobb bármely nyílt csúcs f-értékénél, és így $f(n_i)$-nél is:

$$f(n) \le f(n_i)$$

Ezek alapján:

$$f(n) \le f(n_i) \ = \ \underbrace{g(n_i)}_{=g^*(n_i)} + \underbrace{h(n_i)}_{\le h^*(n_i)} \ \le \ g^*(n_i) + h^*(n_i) \ = \ f^*(n_i) \ = \ f^*(s)$$

A $h(n_i) \le h^*(n_i)$ összefüggés az *alulról becslő* heurisztikából fakad. Az $f^*(n_i) = f^*(s)$ összefüggés pedig abból, hogy tudjuk: $n_i$ eleme az *optimális megoldásnak*, és ezért a rajta áthaladó megoldások optimális költsége nem más, mint az optimális megoldás költsége, azaz $f^*(s)$.

**16. Tétel:** Az *A*-algoritmus* garantálja az *optimális megoldás* előállítását.

*Bizonyítás:* A megoldás előállításának pillanatában a $c$ célcsúcsot választja kiterjesztésre a kereső. Az előző lemma alapján:

$$f(c) \le f^*(s)$$

Felhasználva azt a tény, hogy $c$ *célcsúcs*:

$$f(c) = g(c) + \underbrace{h(c)}_{=0} \ = \ g(c) \ \le \ f^*(s)$$

Azaz az adatbázisban a startcsúcsból a $c$-be vezető út költsége ($g(c)$) nem nagyobb, mint az optimális megoldás költsége. Nyilvánvalóan $g(c)$ nem lehet kisebb az optimális megoldás költségénél, ezért:

$$g(c)=f^*(s)\,.$$

Foglaljuk össze az A*-algoritmus tulajdonságait!

- **Teljesség:**
  - Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
  - Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.
- **Optimalitás:** *garantált* az optimális megoldás előállítása.
- **Tesztelés:** nem hozható előre.

## *MONOTON A-ALGORITMUS*

Az előző fejezetben megvizsgáltuk, hogy az A-algoritmus milyen heurisztikával garantálja az optimális megoldás előállítását. Most vizsgáljuk meg, hogy milyen heurisztikával lehetne megelőzni a *zárt csúcsok problémáját.* Az optimális kereső (mint speciális A-algoritmus) a maga azonosan nulla heurisztikájával megelőzte ezt a problémát, de vajon általánosságban mi mondható el ezen a téren az A-algoritmus heurisztikáiról?

**17. Definíció:** Egy $h$ heurisztika *monoton heurisztika*, ha minden $a$ állapotra igaz: ha az $o$ operátor $a$ -ra való alkalmazásával az $a'$ állapotot kapjuk, akkor

$$h(a)\ \le\ h(a')+költség_o(a)\,.$$

A monoton heurisztikával rendelkező A-algoritmust *monoton A-algoritmusnak* nevezzük, és bebizonyítható, hogy nála nem fordulhat elő a zárt csúcsok problémája. Ez a következő tétel egyenes következménye:

**18. Tétel:** A monoton A-algoritmus által kiterjesztésre kiválasztott bármely $n$ csúcsra:

$$g(n)=g^*(n)\,.$$

*Bizonyítás:* A bizonyítás indirekt.

**Indirekt feltevés:** $g(n)>g^*(n)$. Azaz feltesszük, hogy a kereső még nem találta meg a startcsúcsból az $n$ -be vezető optimális utat.

Jelöljük a startcsúcsból az $n$ -be vezető *optimális úton* szereplő csúcsokat a következőképpen:

$$(s=)\,n_1\ ,\ n_2\ ,\ \dots\ ,\ n_r(=n)$$

Legyen $n_i$ ezen csúcsok közül az *első nyílt csúcs*. A következő összefüggést lehet felírni:

$$f(n_i)=g(n_i)+h(n_i)\ =\ g^*(n_i)+h(n_i)$$

Itt kihasználtuk, hogy az $n_i$ már biztosan az optimális úton szerepel az adatbázisban; azaz $g(n_i)=g^*(n_i)$. Folytassuk a fenti összefüggést!

$$f(n_i)=g(n_i)+h(n_i)\ =\ g^*(n_i)+h(n_i)\ \le$$
$$\le\ g^*(n_i)+h(n_{i+1})+költség_{o_i}(n_i)\ =\ g^*(n_{i+1})+h(n_{i+1})$$

Itt egyrészt kihasználtuk, hogy a heurisztika monoton, azaz hogy $h(n_i)\le h(n_{i+1})+költség_{o_i}(n_i)$. Másrészt felhasználtuk a $g^*(n_{i+1})=g^*(n_i)+költség_{o_i}(n_i)$ összefüggést. Folytassuk tovább a fenti egyenlőtlenséget hasonlóképpen!

$$
\begin{aligned}
f(n_i) = g(n_i) + h(n_i) \;&=\; g^*(n_i) + h(n_i) \;\le \\
\le\; g^*(n_i) + h(n_{i+1}) + \text{költség}_{o_i}(n_i) \;&=\; g^*(n_{i+1}) + h(n_{i+1}) \;\le \\
\le\; g^*(n_{i+1}) + h(n_{i+2}) + \text{költség}_{o_{i+1}}(n_{i+1}) \;&=\; g^*(n_{i+2}) + h(n_{i+2}) \;\le \\
&\vdots \\
\le\; g^*(n_r) + h(n_r)
\end{aligned}
$$

Hol is tartunk tehát most?

$$f(n_i) \;\le\; g^*(n) + h(n) \;<\; g(n) + h(n) = f(n)$$

Itt felhasználtuk az indirekt feltevésünket, azaz: $g^*(n) < g(n)$. Ezzel pedig *ellentmondásra* jutunk, hiszen azt kapjuk, hogy az $n_i$ f-értéke kisebb a $n$ f-értékénél, vagyis ezen két nyílt csúcs közül nem az $n$-t, hanem az $n_i$-t kellett volna kiterjesztésre kiválasztanunk!

Az előző tétel következménye az, hogy a monoton A-algoritmus esetén bőségesen elég egy olyan körfigyelési technika, mely az új csúcsot *csak a nyílt csúcsok között* keresi. A monoton A-algoritmus teljességéről mindezek alapján a következő mondható el:

### Teljesség:

- Ha van megoldás, akkor *tetszőleges* állapottér-gráfban talál megoldást.
- Ha nincs megoldás, akkor ezt *véges* állapottér-gráf esetén felismeri.

A monoton A-algoritmus tulajdonságainak meghatározásakor másik nagy kérdés, hogy az algoritmus az *optimális megoldást* állítja-e elő? Szerencsére be lehet látni, hogy a *monoton A-algoritmus egy speciális A*-algoritmus*, és ezért az előző kérdésre „igen” a válasz. Ezt az alábbi tétel segítségével bizonyítom.

**19. Tétel:** Ha egy heurisztika monoton, akkor alulról becslő is.

*Bizonyítás:* Tetszőleges $n$ csúcsra be kell tehát látni, hogy a $h$ monoton heurisztika esetén teljesül a következő:

$$h(n) \le h^*(n)$$

Két lehetőség van. Az egyik, hogy $n$-ből nem érhető el egy célcsúcs sem. Ekkor $h^*(n)$ definíció szerint $\infty$, azaz a bizonyítandó reláció mindenképpen teljesül.

A másik lehetőség, hogy $n$-ből elérhető valamelyik célcsúcs. Vegyük az $n$-ből valamely célcsúcsba vezető utak közül az optimálisat, és jelöljük az ezen szereplő csúcsokat a következőképpen:

$$(n =) n_1 \;,\; n_2 \;,\; \ldots \;,\; n_r (= c)$$

A $h$ monotonitásából a következő összefüggéseket lehet felírni:

$$
\begin{aligned}
h(n_1) \;&\le\; h(n_2) + \text{költség}_{o_1}(n_1) \\
h(n_2) \;&\le\; h(n_3) + \text{költség}_{o_2}(n_2) \\
&\vdots \\
h(n_{r-1}) \;&\le\; h(n_r) + \text{költség}_{o_{r-1}}(n_{r-1})
\end{aligned}
$$

Adjuk össze ezeknek az egyenlőtlenségeknek a bal és a jobb oldalait! Ezt kapjuk:

$$\sum_{i=1}^{r-1} h(n_i) \;\le\; \sum_{i=2}^{r} h(n_i) \;+\; \sum_{i=1}^{r-1} \text{költség}_{o_i}(n_i)$$

Ha kivonjuk a jobb oldal első szummáját mindkét oldalból:

$$h(n_1) - h(n_r) \;\le\; \sum_{i=1}^{r-1} \text{költség}_{o_i}(n_i) \;=\; h^*(n)$$

Itt kihasználtuk, hogy $h^*(n)$ nem más, mint a $c$-be (azaz $n_r$-be) vezető út (mely optimális!) költsége. Sőt, még azt is tudjuk, hogy mivel $n_r$ célcsúcs, így $h(n_r)=0$. Ebből már következik:

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

A monoton A-algoritmus optimalitásáról és a tesztelés előrehozásáról tehát a következő mondható el:

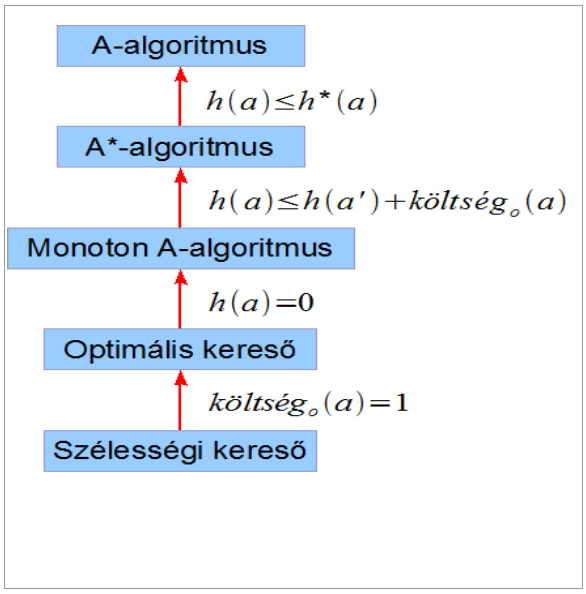
- **Optimalitás:** *garantált* az optimális megoldás előállítása.
- **Tesztelés:** nem hozható előre.

## Kapcsolat az egyes A-algoritmusfajták között

A következő összefüggéseket ismertük meg eddig:

(1) A *szélességi kereső* olyan optimális kereső, melyben az operátoralkalmazások költsége egységnyi.
(2) Az *optimális kereső* olyan A-algoritmus, melyben a heurisztika azonosan nulla.
(3) Az *A*-algoritmus* olyan A-algoritmus, melyben a heurisztika alulról becslő. Az *optimális kereső* A*-algoritmus is, hiszen az azonosan nulla heurisztika alulról becslő.
(4) A *monoton A-algoritmus* olyan A-algoritmus, melyben a heurisztika monoton. Az *optimális kereső* monoton A-algoritmus is, hiszen az azonosan nulla heurisztika monoton.

Ezek alapján a 21. ábrán látható tartalmazási viszonyok állnak fenn a különböző A-algoritmusfajták között. Ahogy az ábrán felülről lefelé haladunk, úgy kapunk egyre speciálisabb algoritmusokat és egyre szűkebb algoritmusosztályokat.



# 5. KÉTSZEMÉLYES JÁTÉKOK

A játékok – néha már riasztó mértékben – a civilizáció kezdete óta foglalkoztatják az emberek intellektuális képességeit. Ez valószínűleg azért alakult így, mert a játék felfogható a világ egy olyan idealizált modelljének, melyben ellenséges játékosok versengenek egymással, és – lássuk be – a versengés az élet minden területén meglévő jelenség.

A mesterséges intelligencia kutatások sok első eredménye a játékokhoz köthető. Természetesen azon játékok kutatása nagy kihívás, melyeknél a játékosoknak (akár az emberi, akár a gépi játékosnak) ellenőrizhető befolyásuk van a játék kimenetelére. Az ilyen játékokat *stratégiai játékoknak* nevezzük; ilyenek például a sakk, a dáma vagy akár a póker. Már a számítástechnika kialakulása előtt is léteztek játékgépek, ilyen volt például a spanyol Quevedo sakkgépe, mely a sakk végjátékára specializálódott (a gép királlyal és bástyával az emberi játékos királya ellen), és képes volt bármilyen kiinduló állásból mattot adni (ha a gép lépett először). Hasonló képességekkel bírt az 1940-ben megépített Nimotron, mely a Nim játékot volt képes mindig megnyerni.

Mindezek a kezdeti próbálkozások elszigeteltek maradtak az 1940-es évek közepéig, az első programozható digitális számítógépek kifejlesztéséig. 1944-ben jelent meg a témában alapműnek számító „Theory of Games and Economic Behavior” című könyv Neumann és Morgenstern tollából, mely átfogó elméleti elemzését adja a játékstratégiáknak. Már ebben a könyvben hangsúlyosan szerepel a *minimax algoritmus*, mely a számítógépes játékprogramok egyik alapvető algoritmusává vált a későbbiekben.

1951-ben Alan Turing írta meg az első valódi számítógépes programot, mely képes volt egy teljes sakkjátszmát végigjátszani. Valójában Turing programja sohasem futott számítógépen, kézi szimulációval tesztelték egy nagyon gyenge emberi játékos ellen, aki legyőzte a programot. A sakk egy jellegzetesen nagyméretű játéktérrel rendelkező játék, ez okozza többek között a sakkprogramok készítésének nehézségét. A minimax algoritmust éppen ezért (a játéktér csökkentése érdekében) bizonyos szempontok szerint próbálták meg az idők folyamán továbbfejleszteni, melyek közül a McCarthy által 1956-ban kidolgozott *alfabéta vágás* érdemel még nagy figyelmet.

Mikor egy játékot próbálunk számítógépre átültetni, valamilyen módon meg kell adnunk a következőket:

- a játék lehetséges állásait
- a játékosok számát
- a szabályos lépéseket
- a kezdőállást
- azt, hogy mikor ér véget a játék, és ki (mennyit) nyer
- azt, hogy a játékosok milyen információkkal rendelkeznek a játék során
- azt, hogy van-e a véletlennek szerepe a játékban

A játékokat ezen ismérvek alapján különböző osztályokba sorolhatjuk:

(1) Játékosok száma szerint:
   - *Kétszemélyes játékok*
   - *Háromszemélyes játékok*
   - stb.

(2) Véletlen szerepe szerint:
   - *Determinisztikus játékok*: a véletlen nem játszik szerepet.
   - *Sztochasztikus játékok*: a véletlennek szerepe van.

(3) A játék végessége szerint:
   - *Véges játékok*: az állásokban véges sok lépési lehetősége van minden játékosnak, és a játék véges sok lépés után véget ér.

(4) Az információ mennyisége szerint:
- *Teljes információjú játékok*: a játékosok a játék során felmerülő összes információval rendelkeznek. Például a legtöbb kártyajáték nem ilyen, mivel ezekben takart lapok is van.

(5) Nyereségek és veszteségek szerint:
- *Zérusösszegű játékok*: a játékosok nyereségeinek és veszteségeinek összege 0.

A továbbiakban kétszemélyes, véges, determinisztikus, teljes információjú, zérusösszegű játékokkal foglalkozunk.

## 5.1. ÁLLAPOTTÉR-REPREZENTÁCIÓ

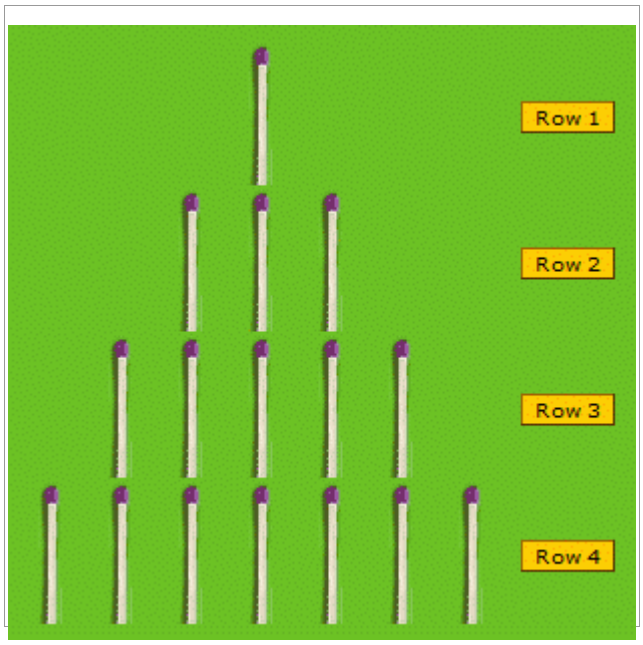
A játékok reprezentálására is használhatunk állapottér-reprezentációt. Egy játék állapottér-reprezentációja formailag ugyanúgy épül fel, mint azt a 3.1. fejezetben megadtam. Csak pár apró tartalmi különbség adódik:

(1) Az *állapotok* $(a, p)$ alakúak, ahol az $a$ a játék egy állása, a $p$ pedig a következőnek lépő játékos. A játékosokat $\mathtt{A}$-val és $\mathtt{B}$-vel fogjuk jelölni, azaz $p \in \{\mathtt{A}, \mathtt{B}\}$.
(2) Minden egyes *célállapot* esetén azt is pontosan meg kell adni, hogy az adott célállapotban *ki nyer/veszít*, vagy hogy esetleg *döntetlen-e* az eredmény.
(3) Minden egyes $o$ operátorhoz (változatlan módon) tartozik egy alkalmazási előfeltétel és egy alkalmazási függvény. Az alkalmazási függvény megadja, hogy ha az $o$-t alkalmazzuk az $(a, p)$ állapotra, milyen $(a', p')$ állapotot kapunk. Fontos tehát a $p'$-t is definiálni, azaz a $p$ játékos *után lépő* játékost!

## 5.2. PÉLDÁK

### 5.2.1. NIM

Adott $n$ db kupac, melyek mindegyikében gyufaszálak találhatók. A következőnek lépő játékos kiválaszt egy kupacot, melyből elvehet akárhány gyufát (legalább 1-et, és persze legfeljebb annyit, amennyi a kupacban van). A játékosok felváltva lépnek. Az veszít, aki az utolsó gyufaszálat veszi el.




- **Állapotok halmaza:** Az állapotokban tároljuk el, hogy az egyes kupacokban hány gyufaszál van! Legyen tehát az állapot egy rendezett $n$-es, ahol az $n>0$ egy állapottéren kívüli konstans, mely a kupacok számát jelöli! Legyen még előre megadva egy $max>0$ szám, mely felülről korlátozza az egy kupacban található gyufák számát!
  Természetesen az állapotban tárolni kell a következőnek lépő játékos jelét is.
  Tehát az állapotok halmaza a következő:
  $$A=\{(a_1,a_2,\ldots,a_n,p) \mid \forall\, i \;\; 0\le a_i\le max \;\;,\;\; p\in\{\mathtt{A},\mathtt{B}\}\}$$
  ahol minden $a_i$ egész szám.
- **Kezdőállapot:** A kezdőállapot bármelyik értelmes állapot lehet, azaz csak ennyit kötünk ki:
  $$k\in A$$
- **Célállapotok halmaza:** Akkor ér véget a játék, ha minden kupacból elfogynak a gyufák. Ilyenkor az a játékos veszít, aki az utolsó gyufát elvette, vagyis a következőnek lépő játékos nyer. Azaz:
  $$C=\{(0,\ldots,0,p)\}\subseteq A \;\;,\;\; p \text{ nyer}$$
- **Operátorok halmaza:** Operátoraink egy kupacból elvesznek valahány darab gyufaszálat. Tehát az operátoraink halmaza:
  $$O=\{elvesz_{kupac,db} \mid 1\le kupac\le n \;\;,\;\; 1\le db\le max\}$$
- **Alkalmazási előfeltétel:** Fogalmazzuk meg, hogy egy $elvesz_{kupac,db}$ operátor mikor alkalmazható egy $(a_1,\ldots,a_n,p)$ állapotra! A következő feltételeket kell formalizálni:
  - A $kupac.$ kupac nem üres.
  - A $db$ értéke legfeljebb annyi, mint ahány gyufa a $kupac.$ kupacban van.

  Tehát az $elvesz_{kupac,db}$ operátor alkalmazási előfeltétele az $(a_1,\ldots,a_n,p)$ állapotra:
  $$a_{kupac}>0 \;\wedge\; db\le a_{kupac}$$
- **Alkalmazási függvény:** Adjuk meg, hogy az $elvesz_{kupac,db}$ operátor az $(a_1,\ldots,a_n,p)$ állapotból milyen $(a'_1,\ldots,a'_n,p')$ állapotot állít elő! Azaz:
  $elvesz_{kupac,db}(a_1,\ldots,a_n,p)=(a'_1,\ldots,a'_n,p')$, ahol
  $$a'_i=\begin{cases} a_i-db & \text{, ha } i=kupac \\ a_i & \text{, egyébként} \end{cases} \qquad \text{ahol } 1\le i\le n$$

$$p' = \begin{cases} \mathtt{A} & , \text{ha } p = \mathtt{B} \\ \mathtt{B} & , \text{ha } p = \mathtt{A} \end{cases}$$

***MEGJEGYZÉSEK***

A játékosokat érdemesebb numerikus értékekkel jelölni, például erre egy gyakori megoldás az $1$ és a $-1$ értékek használata. Ennek egyrészt az operátorok alkalmazási függvényében vesszük hasznát, hiszen például a $p'$ fenti definícióját ilyen könnyen le lehet írni:

$$p' = -p$$

Másrészt nagyon hasznos lesz a játékosok eme jelölése a *negamax algoritmus* esetén (5.5. fejezet).

## 5.2.2. TIC-TAC-TOE

Ezt a játékot 3x3-as amőbaként ismerjük. A 3x3-as táblára a két játékos felváltva rakja le a saját jeleit. Az győz, aki 3 saját jelet rak egy sorba vagy egy oszlopba, vagy esetleg átlósan. A játék kimenetele döntetlen is lehet: ha a tábla betelik, és senkinek nem gyűlt ki a 3 jele egymás mellett.

- **Állapotok halmaza:** Az állapotokban le kell tárolnunk a teljes 3x3-as táblát, illetve mellette természetesen a következőnek lépő játékos jelét is. A 3x3-as mátrix egy cellája akkor $0$, ha oda még egyik játékos sem rakta le a jelét.
  Tehát az állapotok halmaza a következő:
  $$A = \left\{ (T_{(3\times 3)}, p) \;\middle|\; \forall\, i, j \;\; T_{i,j} \in \{0, \mathtt{A}, \mathtt{B}\} \;\;,\;\; p \in \{\mathtt{A}, \mathtt{B}\} \right\}$$
  ahol minden $a_i$ egész szám.
- **Kezdőállapot:** A kezdőállapotban a tábla teljesen üres, illetve kezdjen mondjuk az $\mathtt{A}$-es játékos!
  $$k = \left( \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \mathtt{A} \right)$$
- **Célállapotok halmaza:** Két okból érhet véget a játék:
  - Valamelyik játékosnak kigyűlik egymás mellett a 3 jel.
  - Betelik a tábla.

  Azaz: $C = C_1 \cup C_2$, ahol
  $$C_1 = \left\{ (T, p) \;\middle|\; \begin{array}{c} \exists i \;\; T_{i,1} = T_{i,2} = T_{i,3} = p' \\ \vee \\ \exists i \;\; T_{1,i} = T_{2,i} = T_{3,i} = p' \\ \vee \\ T_{1,1} = T_{2,2} = T_{3,3} = p' \\ \vee \\ T_{1,3} = T_{2,2} = T_{3,1} = p' \end{array} \right\} \;, p' \text{ nyer}$$
  ($p'$ definícióját lásd az operátorok alkalmazási függvényében)

  és
  $$C_2 = \left\{ (T, p) \notin C_1 \;\middle|\; \neg \exists i, j \;\; T_{i,j} = 0 \right\} \;, \text{döntetlen}$$
  A $C_1$ halmazban a nyerő célállapotokat adom meg. Mivel legutoljára a $p$ ellenfele lépett, természetesen elegendő csak a $p'$ jelekből álló hármasokat keresni a táblán. A négy sorból álló feltételben (sorrendben) az egy sorban, egy oszlopban, a főátlóban, illetve a mellékátlóban szereplő hármasokat írom le.

A $C_2$ halmazban adom meg a döntetlen célállapotokat. Akkor végződik döntetlennel a játék, ha betelik a tábla (és nem gyűlt ki senkinek három jel egymás mellett).

- **Operátorok halmaza:** Operátoraink a tábla egy cellájába lerakják az éppen lépő játékos jelét. Tehát az operátoraink halmaza:

$$O=\{lerak_{x,y} \mid 1\le x, y\le 3\}$$

- **Alkalmazási előfeltétel:** Fogalmazzuk meg, hogy egy $lerak_{x,y}$ operátor mikor alkalmazható egy $(T, p)$ állapotra! Természetesen akkor, ha a $T$ $(x, y)$ cellája üres.
Tehát a $lerak_{x,y}$ operátor alkalmazási előfeltétele a $(T, p)$ állapotra:

$$T_{x,y}=0$$

- **Alkalmazási függvény:** Adjuk meg, hogy a $lerak_{x,y}$ operátor a $(T, p)$ állapotból milyen $(T', p')$ állapotot állít elő! Azaz:
$lerak_{x,y}(T, p)=(T', p')$, ahol

$$T'_{i,j}=\begin{cases} p & \text{, ha } i=x \wedge j=y \\ T_{i,j} & \text{, egyébként} \end{cases} \qquad \text{ahol } 1\le i, j\le 3$$

$$p'=\begin{cases} \texttt{A} & \text{, ha } p=\texttt{B} \\ \texttt{B} & \text{, ha } p=\texttt{A} \end{cases}$$

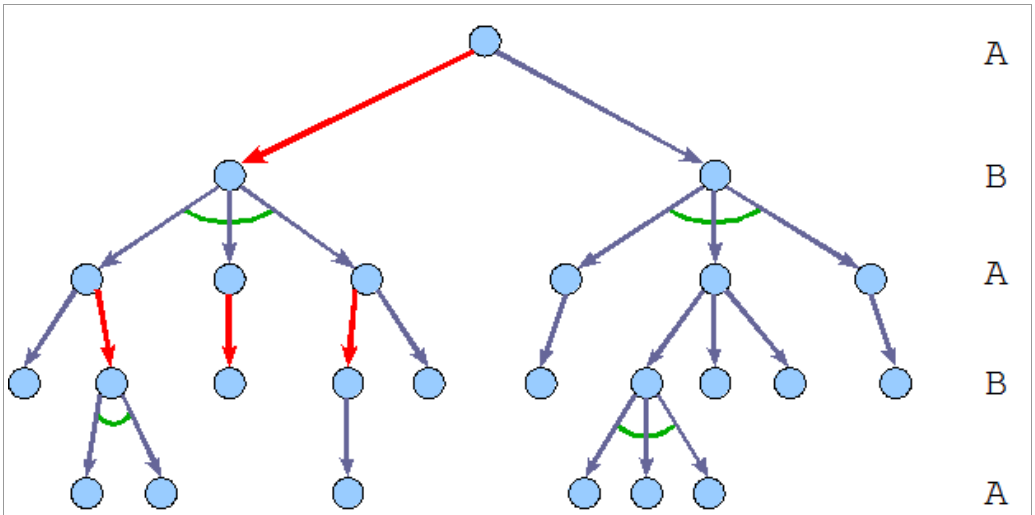
## 5.3. JÁTÉKFA ÉS STRATÉGIA

A kétszemélyes játékok állapottér-reprezentációja alapján a 3.2. fejezetben leírt módon *állapottér-gráfot* tudunk előállítani. Mi az előbbiekben rögzített módon csak *véges játékokkal* foglalkozunk, azaz ennek a gráfnak *véges gráfnak* kell lennie. Ez maga után vonja, hogy a gráfban nem lehetnek körök. A köröket egyébként a játékszabályok korlátozásai általában eliminálják. Az állapottér-gráfot fává szoktuk alakítani, és így kapjuk meg az ún. *játékfát.*[5] A játék egy *játszmáját* a fa egy ága reprezentálja.

Ahhoz, hogy egy játékprogram automatikusan el tudja dönteni a játékos következő lépését, egy *stratégiára* van szükség. A stratégia nem más, mint egyfajta előírás: egy-egy állapotban melyik operátort alkalmazza a gép. Érdekes kérdés lenne annak eldöntése, hogy egy játék esetén van-e és melyik játékosnak van *nyerő stratégiája?* Azaz olyan stratégia, melynek az előírásai szerint alkalmazva az operátorotokat az adott játékos mindenképpen nyer (az ellenfél lépéseitől függetlenül).

Ehhez azonban ki kéne nyomoznunk, hogy a játékfában hogyan is jelenik meg egy adott ( `A` vagy a `B` játékhoz tartozó) stratégia. Ha a $p\in\{\texttt{A},\texttt{B}\}$ játékos lehetséges stratégiáit szeretnénk megjeleníteni, akkor a játékfát *ÉS/VAGY-gráffá* alakítjuk a következő módon: az összes olyan csúcsból induló élt összekötjük egy-egy ívvel, melyben a $p$ *ellenfele* lép. A 23. ábrán egy az `A` játékos szerinti ÉS/VAGY-gráf látható. Azon csúcsokból induló éleket kötöttem össze egy-egy ívvel, melyekben a `B` játékos lép. Az így kapott összekötött éleket *ÉS-élkötegeknek* nevezzük, és segítségükkel azt fejezzük ki, hogy a $p$ játékos nincs befolyással ellenfele döntéseire, azaz $p$ stratégiájának az ellenfél bármely lehetséges lépésére fel kell készülnie. Az ábrán pirossal kiemelt élek lehetnének `A` stratégiájának a részei, hiszen a játék során végig egyértelműen meghatározzák `A` lépéseit.

5 A fává alakítás tulajdonképpen a hurkok eliminálását jelenti. Azaz egyes csúcsoknak (és a belőlük induló részfáknak) másolatait kell a fába beszúrni.



*23. ábra: Játékfa mint ÉS/VAGY-gráf*

Most formalizáljuk egzakt módon, mit is értünk ÉS/VAGY-gráf és stratégia alatt!

**20. Definíció:** Az *ÉS/VAGY-gráf* egy $\langle V, E \rangle$ pár, ahol

- $V \neq \emptyset$ a csúcsok halmaza, és
- $E$ a *hiperélek* halmaza, ahol

$$E \subseteq \{(n, M) \in V \times P(V) \mid M \neq \emptyset\}$$

Az $(n, M) \in E$ hiperél kétfajta lehet:

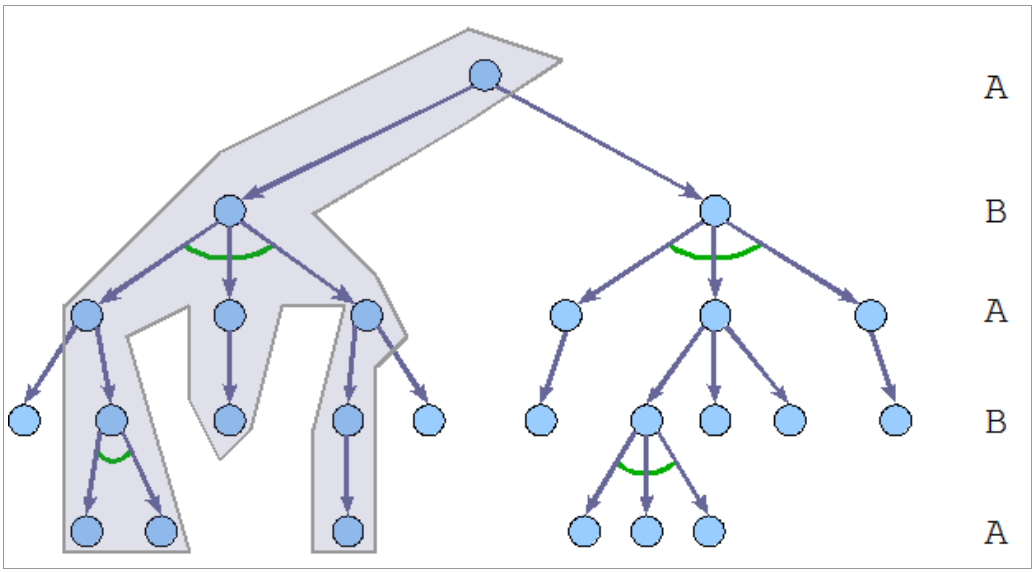
- *VAGY-él*, ha $|M| = 1$.
- *ÉS-élköteg*, ha $|M| > 1$.

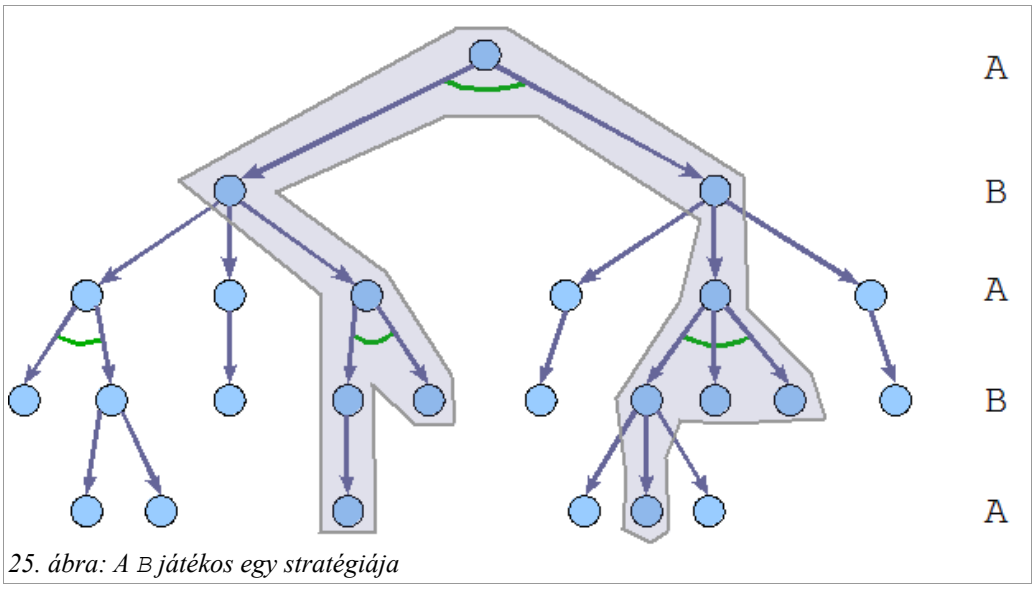
A *hiperél* az eddig általunk ismert él-fogalom általánosítása: a hiperél egy csúcsból akárhány csúcsba húzható; ezért definiáltam $M$-et csúcsok halmazaként[6]. A megszokott él-fogalomnak tulajdonképpen a VAGY-él felel meg, mivel ez egy olyan hiperél, mely egy csúcsból csak egyetlenegy csúcsba vezet.

*Hiperútnak* nevezzük az eddigi út-fogalom megfelelő általánosítását: az $n$ csúcsból az $M$ csúcshalmazba vezető hiperút az $n$ és az $M$ közötti hiperélek sorozata.

Ezek alapján már könnyű formalizálni a $p$ játékos *stratégiáját*: a $p$ szerinti ÉS/VAGY-gráfban a startcsúcsból egy $M \subseteq C$ csúcshalmazba vezető hiperút (azaz $M$ minden eleme célcsúcs). A 24. ábrán az `A` játékosnak, a 25 ábrán a `B` játékosnak látható egy-egy lehetséges stratégiája, a fában bekeretezve.

6 $P(V)$ a $V$ halmaz hatványhalmazát (azaz $V$ részhalmazainak a halmazát) jelöli.




*25. ábra: A `B` játékos egy stratégiája*

### 5.3.1. Nyerő stragégia

Egyik csábító cél, hogy egy játék kapcsán nyerő stratégiát keressünk. Vajon létezik ilyen? És ha igen, akkor melyik játékos számára?

Először is tisztázni kell, mit nevezünk a $p$ játékos *nyerő stratégiának*: $p$-nek olyan stratégiáját, mely (mint hiperút) csupa olyan célállapotba vezet, melyben $p$ nyer.

**21. Tétel:** Minden olyan (általunk vizsgált) játék esetén, melyben nem lehet döntetlen az eredmény, valamelyik játékosnak *van nyerő stratégiája*.

*Bizonyítás:* Legeneráljuk a játékfát, majd felcímkézzük a leveleit `A`-val vagy `B`-vel attól függően, hogy ki nyer az adott csúcsban.

Ezek után lentről felfelé címkézzük a csúcsokat a fában, a következő módon: ha az adott csúcsban $p$ lép, akkor

- $p$ címkét kap, ha van $p$ címkéjű gyermeke;

- az ellenfél címkéjét kapja, ha nincs $p$ címkéjű gyermeke (azaz a csúcs összes gyermeke az ellenfél jelével címkézett).

Miután felcímkéztük az összes csúcsot, a startcsúcs (azaz a gyökér) címkéje mutatja a nyerő stratégiájú játékost.

Azon játékok esetén, melyek döntetlennel is végződhetnek, *nem vesztő stratégiát* lehet garantálni az egyik játékosra nézve.

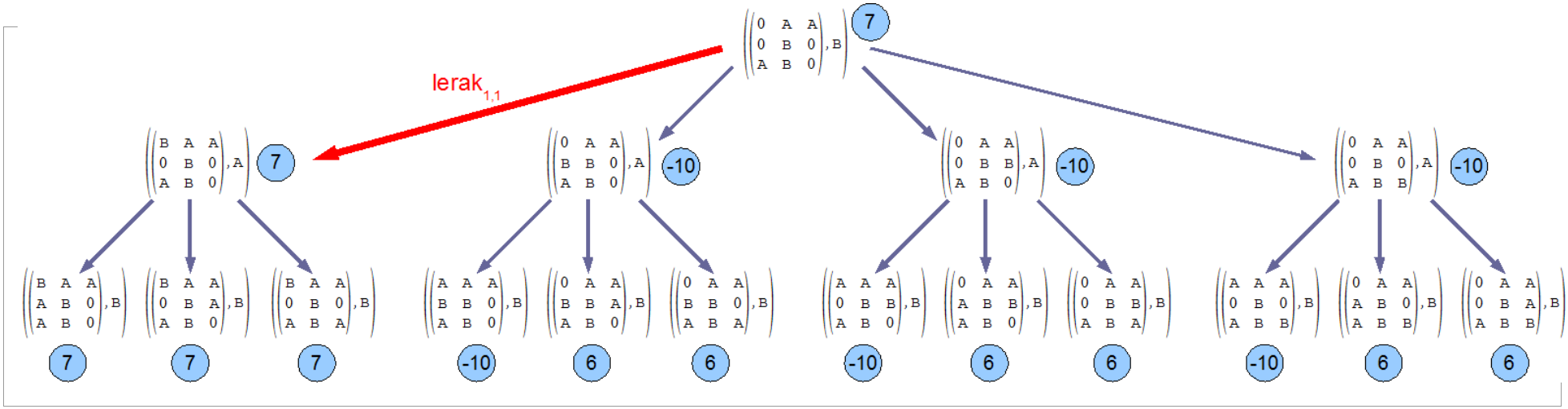
## 5.4. MINIMAX ALGORITMUS

Mivel a játékfa óriási méretű lehet, teljesen irreális kívánság, hogy a játékprogramunk teljes terjedelmében legenerálja azt (és nyerő stratégiát keressen benne). Legfeljebb azt tehetjük meg, hogy a számítógépi játékos minden lépése előtt felépítjük a játékfának *egy részét*. Ennek a részfának

- a gyökere legyen a játék aktuális állapota, és
- a mélysége legyen limitálva egy előre megadott *maxMélység* értékkel.

A fa ily módon történő legenerálása tulajdonképpen annak a valóságban is alkalmazott „trükknek" felel meg, mikor a játékosok pár lépés erejéig előregondolkoznak, minden lehetséges lépéskombinációt fejben végigjátszva. A *maxMélység* annak a meghatározása, hogy hány lépés erejéig „gondolkodjon" a játékprogram előre. A *maxMélység* -et érdemes a játékprogramunk indulásakor bekérni, mint a *nehézségi szintet*.

A 26. ábrán látható példán a Tic-tac-toe játék $\left(\begin{pmatrix} 0 & \mathtt{A} & \mathtt{A} \\ 0 & \mathtt{B} & 0 \\ \mathtt{A} & \mathtt{B} & 0 \end{pmatrix}, \mathtt{B}\right)$ állapotából kiindulva építettem fel a részfát, $maxMélység = 2$ esetén.




Az így felépített fa levélelemei kétfajtájúak lehetnek. Egyesek közülük *célcsúcsok*, ezek hasznosságáról pontos fogalmaink vannak: ha ebben a levélben az adott játékos nyer, akkor ez egy „nagyon jó” célállapot számára, ha veszít, akkor pedig „nagyon rossz”.
A többi levél *nem célcsúcs*; ezeket a mélység lekorlátozása miatt már nem terjesztettük ki (a 26. ábrán a legtöbb levélelem ilyen). Ezért a levélelemek „jóságát” valamilyen becslés alkalmazásával próbáljuk meghatározni; azaz szükségünk lesz egy *heurisztikára*.

### ***HEURISZTIKA***

Milyen legyen a választott heurisztika? A heurisztika 24. oldalon megadott definíciója kétszemélyes játékok esetén felülvizsgálatra szorul:

- A heurisztika *tetszőleges egész* érték lehessen. Minél jobb az adott játékosnak, annál nagyobb (pozitív) érték, minél rosszabb a játékosnak, annál kisebb (negatív) érték.
- Célállapot esetén csak akkor legyen $0$ a heurisztika, ha a célállapot *döntetlent* eredményez. Emlékeztetőül: a 24. oldalon minden célállapothoz $0$ heurisztikát rendeltünk. Kétszemélyes játékok esetén ha az adott játékos *nyer* a célállapotban, akkor a heurisztika valamilyen nagyon nagy szám legyen, ha viszont *veszít*, akkor legyen nagyon kicsi szám.
- Mivel két játékosunk is van, így 2 db heurisztikára is szükségünk van. A $h_{\mathtt{A}}$ az $\mathtt{A}$ játékos heurisztikája, a $h_{\mathtt{B}}$ a $\mathtt{B}$-jé.

Egy adott játék esetén a heurisztikát mi választjuk meg. A Tic-tac-toe esetén a $h_p$ heurisztika értéke (ahol $p \in \{\mathtt{A}, \mathtt{B}\}$) legyen például a következő:

(1) $10$ , ha olyan célállapotban vagyunk, melyben $p$ nyer.
(2) $-10$ , ha olyan célállapotban vagyunk, melyben $p$ ellenfele nyer.
(3) $0$ , ha döntetlen célállapotban vagyunk.
(4) Egyébként azon sorok, oszlopok, átlók száma, melyeket $p$ „blokkol” (azaz van ott jele).

***A MINIMAX ALGORITMUS MŰKÖDÉSE***

Jelöljük a fa gyökerében (azaz az aktuálisan) lépő játékost *aktP* -vel! A fa összes $(a, p)$ csúcsához hozzárendelünk egy-egy *súlyt* a következő módon:

(1) Ha $(a, p)$ levél, akkor súlya legyen $h_{aktP}(a, p)$ .

(2) Ha $(a, p)$ nem levél, az azt jelenti, hogy vannak gyermekei. A gyerekek súlyait használjuk arra, hogy a szülőjük súlyát kiszámoljuk. Ezt a következőképpen tesszük:

- Ha $p = aktP$ , akkor a gyermekek súlyának a *maximumát* rendeljük az $(a, p)$ -hez.
- Ha $p \neq aktP$ , akkor a gyermekek súlyának a *minimumát* rendeljük az $(a, p)$ -hez.

A maximum/minimum felhasználásának oka triviális: *aktP* mindig a számára legkedvezőbb, azaz a lehető legnagyobb heurisztikájú állapot felé fog lépni. Az *aktP* ellenfele épp ellenkezőleg.

A 26. ábrán a csúcsok mellé írt értékek a csúcsok súlyai, mégpedig a gyökérben lépő $aktP = \mathtt{B}$ játékos heurisztikája szerint.

Miután a fa minden csúcsához valamilyen súlyt rendeltünk, következhet az a lépés, melynek érdekében mindezt elvégeztük. Meghatározzuk, hogy a fa gyökerére (azaz a játék aktuális állapotára) *melyik operátort* érdemes alkalmaznunk a leginkább. Magyarul: a játékprogramnak azt az operátort kell alkalmaznia (vagy ha az emberi játékos lép aktuálisan, akkor azt az operátort kell tanácsolnia), mely a *fa gyökeréből* a *legnagyobb súlyú gyermekbe* vezet (mint él). A 26. ábrán a $lerak_{1,1}$ operátort alkalmazza (vagy tanácsolja) a játékprogram.

## 5.5. NEGAMAX ALGORITMUS

A minimax algoritmus alkalmazásának egyik nehézsége, hogy egy játékhoz *kétfajta heurisztikát* is ki kell találni. Márpedig könnyű észrevenni, hogy a két heurisztika között erős összefüggés van. Ez az összefüggés egyszerű szavakkal így írható le: a játék egy állapota az egyik játékosnak minél jobb, a másiknak annál rosszabb. Azaz az egyik játékos heurisztikája nyugodtan lehet a másik játékos heurisztikájának *ellentettje* (negáltja), vagyis $-1$ -szerese.

Tehát bőségesen elég lesz 1 db heurisztika. Az $(a, p)$ állapot heurisztikája a $p$ *szerinti heurisztika* lesz.

A minimax algoritmust ezen a nyomon elindulva két ponton is egyszerűbbé tudjuk tenni:

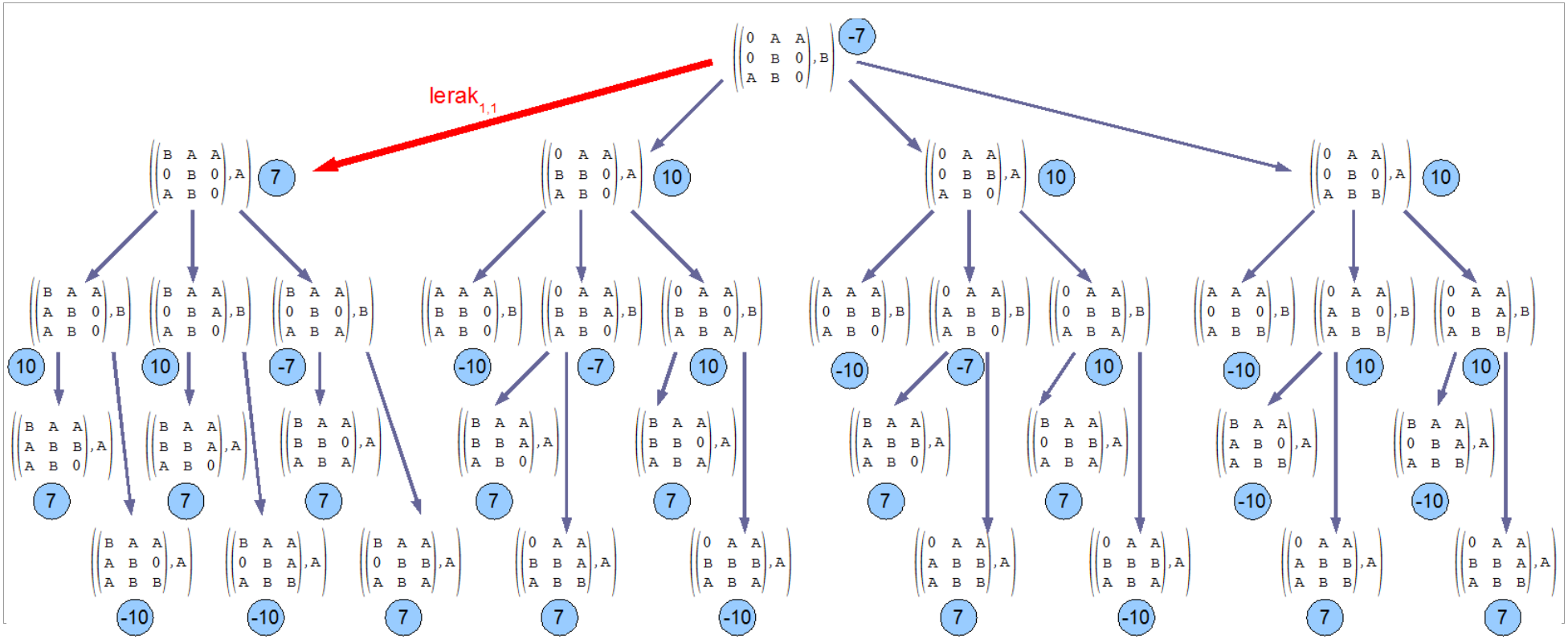
(1) A legenerált fa levélcsúcsaihoz azok heurisztikáját rendeljük, mely során nem kell figyelembe vennünk, hogy melyik játékos lép a fa gyökerében.

(2) A fa minden nem levél $(a, p)$ csúcsa súlyának meghatározásakor minden esetben *maximumot* fogunk számolni. A trükk abban áll, hogy az $(a, p)$ csúcs $(a', p')$ gyermekének súlyát mi módon transzformáljuk ehhez:

- Ha $p \neq p'$, akkor $(a', p')$ súlyának a $-1$ -szeresét használjuk fel $(a, p)$ súlyának meghatározásakor.
- Ha $p = p'$, akkor $(a', p')$ súlyát (változatlanul) használjuk fel $(a, p)$ súlyának meghatározásakor.

A 27. ábrán a Tic-tac-toe játék esetén a negamax algoritmus által legenerált fa látható, ezúttal

*maxMélység* = 3 esetén. Érdemes összevetni a 26. ábrával. Ezeken az ábrákon az is látható, hogy az aktuálisan lépő `B` játékos okosan lép, hiszen az egyetlen olyan irányba viszi a játék menetét, amerre esélye van a nyerésre. Vagyis egész jól választottuk meg a heurisztikát.

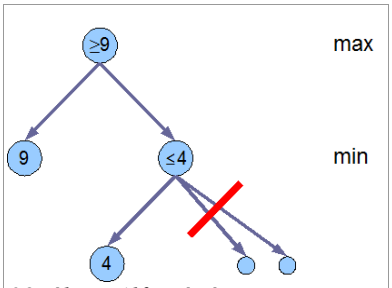



## 5.6. ALFABÉTA-VÁGÁS

Ha nagyra választjuk a *maxMélység* értékét, a legenerált fa (és vele együtt a minimax algoritmus időigénye) nagyon nagyméretű lehet. Ezért érdemes optimalizálni az algoritmuson, mégpedig abból a megfigyelésből kiindulva, hogy a fa egyes részei sokszor feleslegesen lesznek előállítva. Az ilyen részek eliminálását célozza meg az *alfabéta-vágás*.

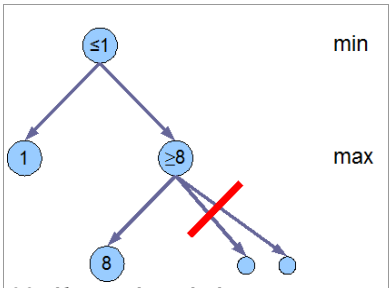
Az alfabéta-vágással egy-egy csúcs *kiterjesztését szakíthatjuk félbe*. Abban az esetben tesszük ezt, mikor már a kiterjesztés befejezése előtt kiderül, hogy az éppen kiterjesztés alatt álló csúcsnak nem lesz beleszólása a szülője súlyának alakulásába. Nézzük a két lehetséges szituációt:

(1) **Alfa-vágás**: Az aktuális csúcsban minimumot számolunk, a szülőjében maximumot. Az aktuális csúcs kiterjesztését félbeszakítjuk, ha a csúcsban eddig kiszámolt minimum kisebbé válik, mint a szülőben eddig kiszámolt maximum. A 28. ábrán látható egy ilyen szituáció: az aktuálisan kiterjesztendő csúcs a $\leq 4$ feliratot viseli, hiszen már egy $4$ súlyú gyermekét előállítottuk, vagyis az aktuális csúcs súlya max. $4$ lesz. Mivel hasonló okból a szülőjének a súlya min. $9$ lesz, így az aktuális csúcs többi gyermekét már szükségtelen előállítanunk, hiszen az aktuális csúcs súlya innentől már teljesen irreleváns.



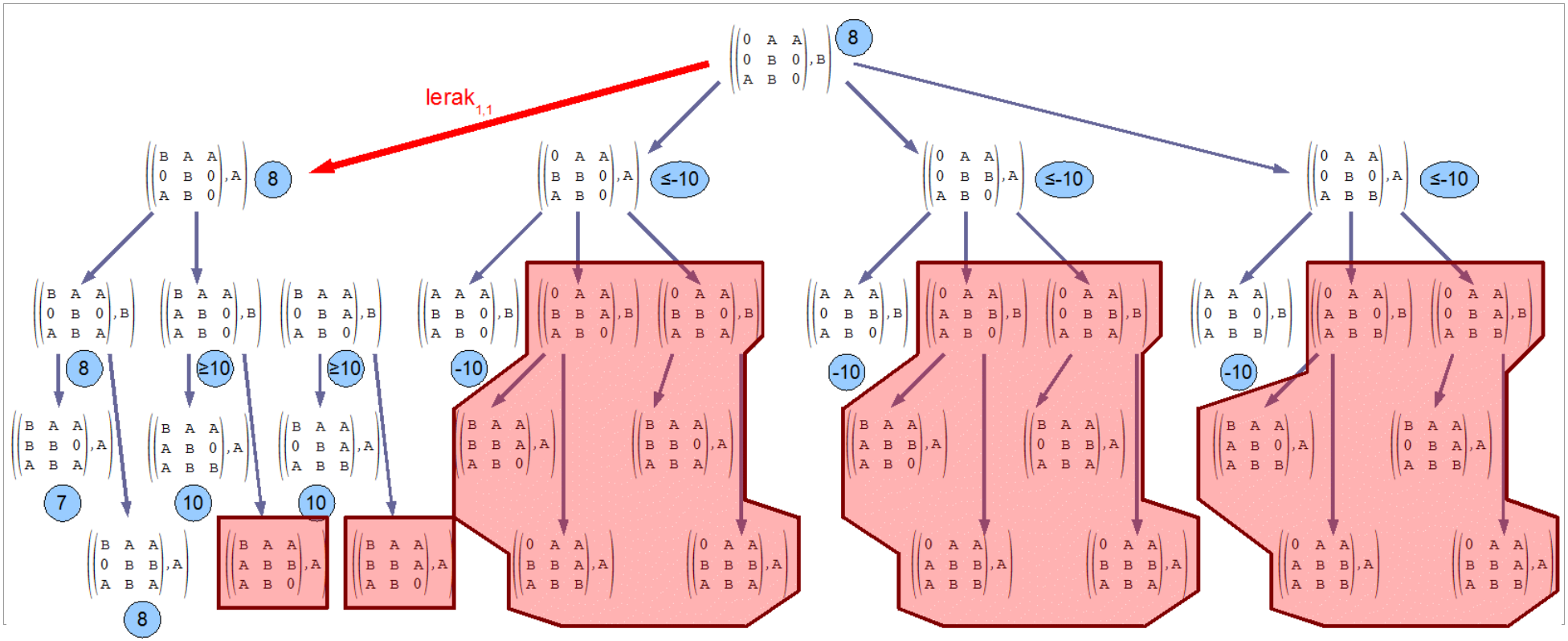
*28. ábra: Alfa-vágás*

(2) **Béta-vágás**: Ugyanaz, mint az alfa-vágás, csak most az aktuálisan kiterjesztett csúcsban maximumot, annak szülőjében pedig minimumot képzünk. A 29. ábrán látunk egy ilyen szituációt.



*29. ábra: Béta-vágás*

A 30. ábrán a Tic-tac-toe minimax által legenerált fája látható $maxMélység = 3$ esetén. Piros keretbe foglaltam a fának azon részeit, melyeket alfabéta-vágást alkalmazva le sem kell generálnia az algoritmusnak. Meg lehet számolni, hogy ez 20 db csúccsal kevesebbet jelent, és így összesen 15 db csúcs előállítására volt szükség. Azaz a fa csúcsainak 57%-át sikerült kivágnunk.


lerak1,1
8
≤-10
≤-10
≤-10
8
≥10
≥10
-10
-10
-10
7
10
10
8

# 6. AZ MI ALKALMAZÁSA AZ OKTATÁSBAN

A mesterséges intelligencia tárgy fő témáját a megoldáskereső algoritmusok alkotják, amelyek egy gráfban a start csúcstól egy célcsúcsba vezető utat keresnek. Ezen algoritmusok szemléltetése a táblán nehézkes, mert egy-egy csúcs állapota az idővel változik. Például a mélységi keresésnél egy csúcs lehet még fel nem derített, nyílt vagy zárt csúcs. Mivel a táblán nincs mód a gráfot többször felrajzolni, a szemléltetés csak úgy oldható meg, hogy ugyanabban az ábrában írjuk át a csúcs státusát. Így a hallgatóság könnyen elveszti az időrendi sorrendet. Erre a problémára keresünk többek közt választ.

A javasolt megoldás animált ábrák használata, ahol látható az algoritmus és a gráf. Az ábra jobb oldalán látható, amint az algoritmus lépésről lépésre végrehajtódik. Az aktuális sort kiszínezzük. Ha valamelyik utasítás megváltoztatja valamely csúcs állapotát, akkor az ábra bal oldalán megváltozik a gráf megfelelő csúcsa. Általában más színű lesz. Természetesen minden szín jelentése jelölve van.

Tapasztalatok szerint ezen segédeszköz használata segíti, mélyebbé teszi a kereső algoritmusok megértését.

Megfigyeltük, hogy a hallgatók visszariadnak a kereső algoritmusoktól, túl bonyolultnak, a képességeiket meghaladónak érzik azokat. Ezért nagyon lényeges, hogy ezeket az algoritmusokat megfoghatóvá, testközelivé tegyük. Erre remek módszer az algoritmusok vizualizálása, ezért elektronikus jegyzetünk több animációt is tartalmaz.

## 6.1. A PROBLÉMA

A probléma kétrétű, egyfelől pszichológiai, másfelől technikai.

A kereső algoritmusok megvalósításánál szembesülnek a hallgatók először azzal a problémával, hogy a programozási ismereteiket komplexen kell alkalmazni. Először az állapottér-reprezentációt kell elkészíteni az ahhoz tartozó adatszerkezetekkel (általában egy Állapot és egy Csúcs osztály segítségével), aztán ki kell választani a megfelelő algoritmust, amely sok esetben rekurzív.

Mivel a megoldás hossza előre meg nem határozható, dinamikus adatszerkezet kell a tárolásához, amely leggyakrabban egy lista (esetleg verem vagy sor). Tehát összefoglalva, rutinszerűen kell használni:

- az osztályokat,
- a rekurziót és
- a dinamikus adatszerkezeteket.

Ezek a követelmények sok hallgatót elrémisztenek magától a tárgytól, még mielőtt meglátnák annak szépségét. Ha a hallgató úgy érzi, hogy a tárgy túl bonyolult, a képességeit meghaladó, akkor annak megértésétől eleve elzárkózik. Ez a pszichológiai probléma.

A másik probléma inkább technikai. A legjelentősebb megoldáskereső algoritmusok szemléltetése a táblán nehézkes, mert egy-egy csúcs állapota az idővel változik. Például a mélységi keresésnél egy csúcs lehet még fel nem derített, nyílt majd zárt. Mivel a táblán nincs mód a gráfot többször felrajzolni, a szemléltetés csak úgy oldható meg, hogy ugyanabban az ábrában írjuk át a csúcs státusát. Így a hallgatóság könnyen elveszti az időrendi sorrendet. Mindkét problémára megoldást találunk a multimédia segítségül hívásával.

Megoldási javaslat

A multimédia használata az oktatásban nem újkeletű. Széles körben elfogadott nézet, hogy bármely tárgyból a tanultak jobban rögzülnek, ha a tanításhoz multimédiás oktatási segédeszközöket használnak, mint például:

- **színes ábrák,**
- **videók, animációk,**
- **interaktív programok.**

Ennek a jelenségnek az az oka, hogy a multimédia testközelbe hozza, megfoghatóvá teszi az elméletben elmondottakat.

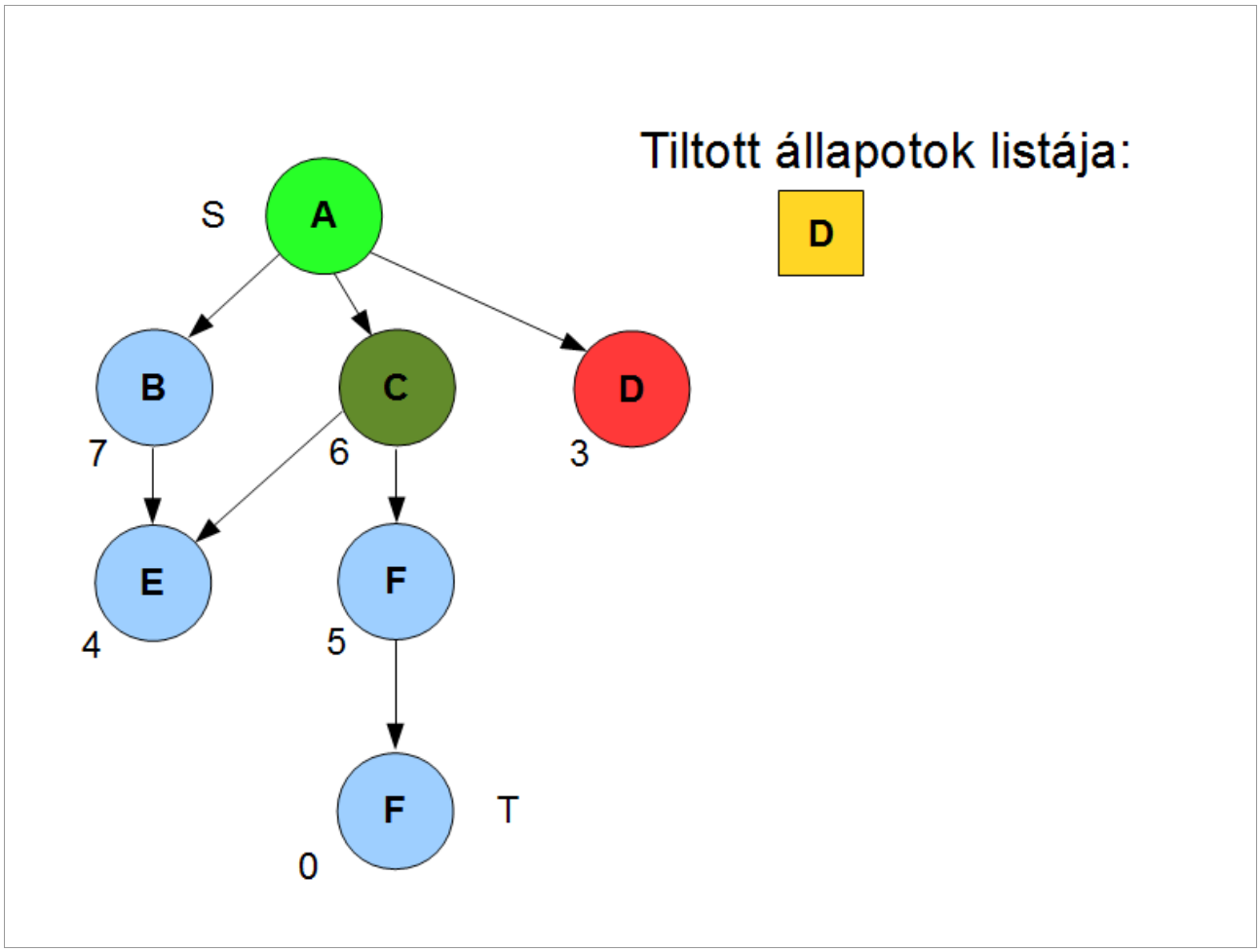
Ugyanakkor a multimédia használatának hátrányai is vannak. Használatához olyan eszközök kellenek, amelyek nem minden tanteremben adottak, elromolhatnak, időbe telik az elindításuk és hasonló problémák.

A mesterséges intelligencia gyakorlati részének oktatása általában számítógépes laborban történik, ezért a multimédia feltételei általában adottak.

A megjelenítendő információ változik az egyes megoldáskereső algoritmusok esetén, ezért szükséges, hogy mindegyikhez meghatározzuk, milyen információknak szükséges megjelenni az animáción, a gráf részeként vagy azt kiegészítve, illetve hogy melyiknél milyen adatszerkezetet/adatszerkezeteket használunk adatbázisként, ha úgy tetszik a gráf tárolására. A következőkben a különböző keresőtípusok megjelenése közti különbségeket fogjuk tárgyalni.

### 6.1.1. Nem módosítható keresők

A nem módosítható keresőket olyan speciális esetekben használjuk, mikor nem a megoldásig elvezető operátorsorozat előállítása a célunk, hanem csupán arra keressük a választ, hogy egyáltalán létezik-e megoldása a feladatnak és ha igen, akkor ezt a célállapotot állítsuk elő. A nem módosítható keresők egyik legismertebb és leginkább az adott feladatnak megfelelően testreszabható változata a Hegymászó módszer. Ez egy heurisztikus kereső, ami annyit jelent, hogy egy bizonyos állapothoz általunk megadott becslés alapján választ operátort az algoritmust. Ez azokban az esetekben nagyon szerencsés, mikor képesek vagyunk hatékony heurisztika megadására. Ellenkező esetben az egyszerűbb Próba-Hiba módszert is alkalmazhatjuk, amely véletlenszerűen választ operátort. A dolgozatban a Hegymászó módszeren szemléltetem az animáció felépítését, annak is a kibővített, úgynevezett Restartos Hegymászó módszer változatán. Ebben a változatban nyilvántartjuk azon állapotoknak a listáját, melyekből nem sikerült folytatni a keresést és ennek a listának a maximális elemszámát is meg kell adjuk. Mikor nem tudjuk folytatni a keresést, az aktuális csúcs a tiltott csúcsok listájába kerül és a keresés újból indul.




A fenti ábrán a kereső abban az állapotban látható, mikor már egy restarton túl, a D csúcsot a tiltott csúcsok listájára felvettük. A kereső jelenlegi, aktuális csúcsa a C csúcs. A következő lépést innen fogja megtenni és ekkor az E csúcs felé vezető operátort fogja választani. Ezt onnan tudhatjuk, hogy az E csúcsban lévő állapot heurisztikája kisebb. Az egyes csúcsokban lévő állapotok heurisztikája a csúcs melletti számmal van feltüntetve.

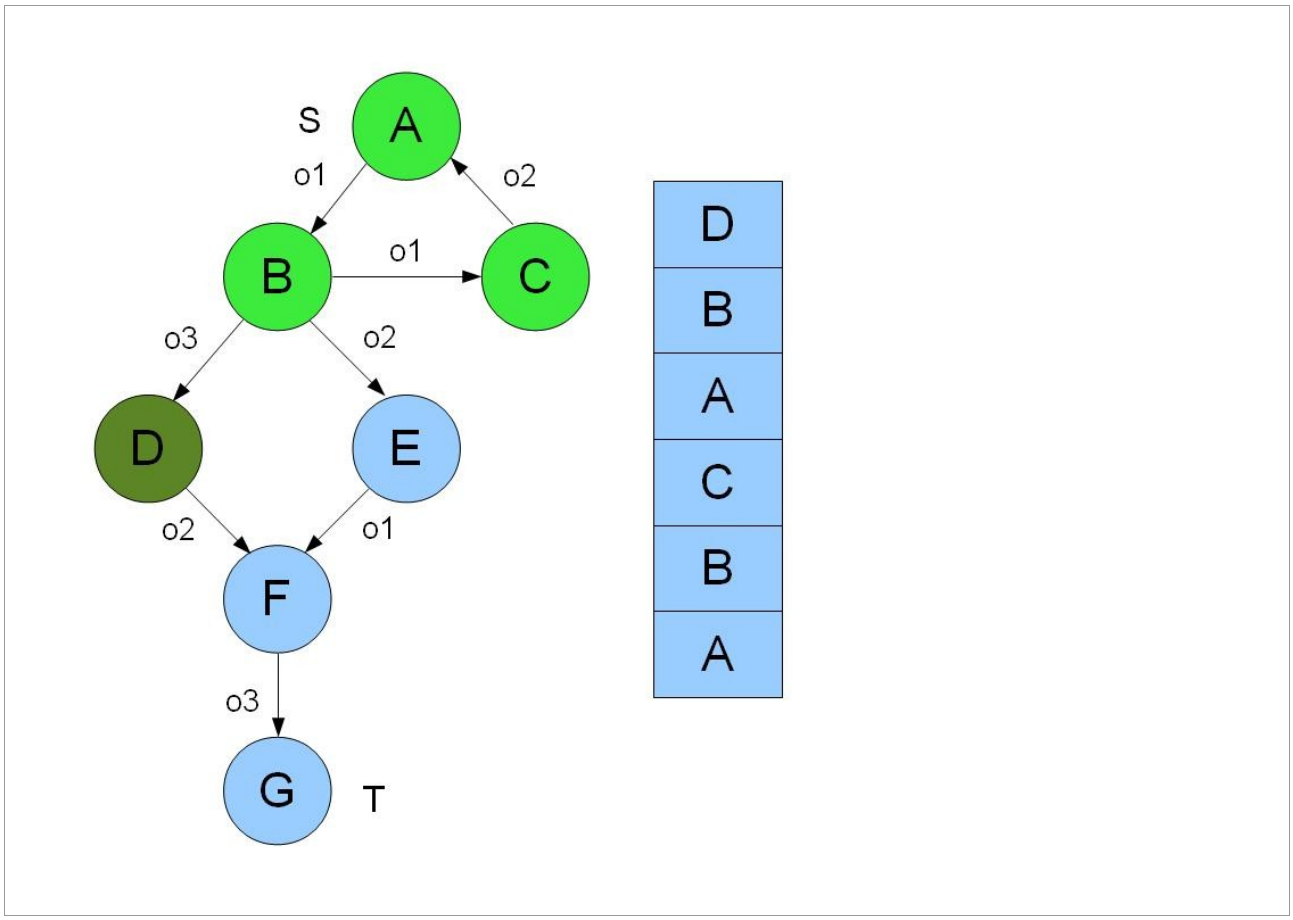
Jelmagyarázat:

| **Fogalmak** | **Jelölések** |
|---|---|
| Még nem látogatott csúcs | kék |
| Aktuális csúcs | sötétzöld |
| Bejárt csúcs | zöld |
| Tiltott csúcs/állapot | piros |
| Heurisztika | csúcs mellett lévő szám |
| Kezdő csúcs | A csúcs mellett S szerepel |
| Terminális csúcs | A csúcs mellett T szerepel |
| Tiltott állapotok listája | Felirat alatti sárga listában |

A restartok számát külön nem jelöljük, azt, ha szükséges, az újraindításkor jelenítjük meg.

## 6.1.2. Visszalépéses keresők (Backtrack)

A módosítható megoldáskereső algoritmusok egyik fajtája a visszalépéses kereső, melynél az aktuális csúcson kívül kibővítjük az adatbázisunkat, az aktuális csúcsba vezető csúcsok listájával. Ezzel lehetőségünk nyílik arra, hogy amennyiben az algoritmus zsákutcába fut a gráfban, visszalépünk és másik irányba folytatjuk a keresést. Ez a visszalépés adja a kereső nevét. Amennyiben terminális csúcsot találunk, arra is lehetőségünk nyílik a kibővített adatbázis által, hogy megadjuk azt az operátorsorozatot, amelyet a kezdő állapotra folyamatosan alkalmazva megkapjuk a célállapotot. Az ábrán a visszalépéses kereső, úthosszkorlátos változata látható. Ebben a változatban előre megadott méretű adatbázissal dolgozunk. Ezt akkor alkalmazhatjuk, ha jól meg tudjuk becsülni a probléma optimális megoldásának lépésszámát.




Az ábrán szerepló kereső abban az állapotban látható, mikor a csúcsok tárolására használt adatbázis betelik. Az adatbázis az ábrán lentről felfelé töltődik fel elemekkel. Ez jól szemlélteti a verem adatszerkezet működését is, amit az adatbázis megvalósítására szokás használni backtrack keresők esetén. Így könnyen leolvasható az ábráról, hogy mivel az aktuális csúcs a D és ez alatt a B csúcs található, a B csúcsba fogunk visszalépni a következő lépésnél.

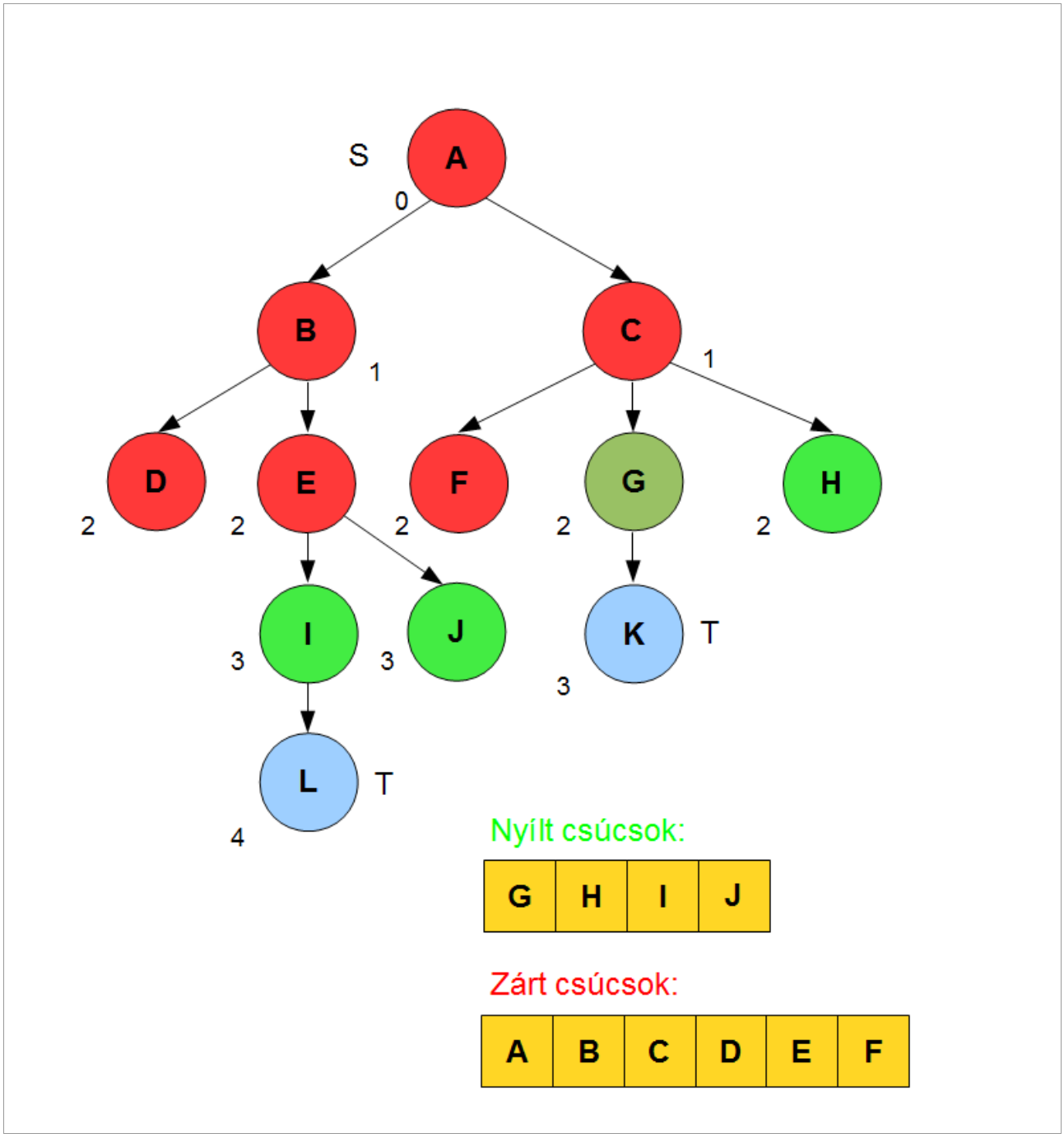
Jelmagyarázat:

| **Fogalmak** | **Jelölések** |
|---|---|
| Bejáratlan csúcs | kék |
| Aktuális csúcs | sötétzöld |
| Bejárt csúcs | zöld |

| | |
|---|---|
| Operátor, mellyel továbblépünk | nyíl mellett lévő szám |
| Kezdő csúcs | A csúcs mellett S szerepel |
| Terminális csúcs | A csúcs mellett T szerepel |
| Adatbázist reprezentáló verem | A gráftól jobbra található oszlop |

### 6.1.3. Keresőfával keresők

Ezek alkotják a módosítható megoldáskeresők másik nagy csoportját. Az adatbázis további kiterjesztésével próbáljuk elérni azt, hogy gyorsabban találjunk megoldást és ha lehet, akkor a legkevesebb lépésből állót. Míg a visszalépéses keresők esetében csak egy utat tárolunk mindig, a keresőfával keresők esetén több utat tárolunk egy időben, melyet egy faként lehet reprezentálni. A keresést a különböző utakon, felváltva folytatjuk. A különböző algoritmusok abban különböznek, hogy miként választják meg azt, hol kell folytatni a keresést. A fában való továbbhaladást kiterjesztésnek nevezzük, ez alapján megkülönböztetünk nyílt és zárt csúcsokat. A zárt csúcsok azok, melyeket már kiterjesztettünk, a többit nyílt csúcsnak tekintjük. Másik új fogalom a mélységi szám. Egy adott csúcs mélységi száma azt adja meg, hogy azt a start-csúcsból hány lépésből tudjuk elérni.




Az ábrán a szélességi kereső látható, amely a legkisebb mélységi számú csúcsot választja kiterjesztésre. Amennyiben több csúcsnak is ez a mélysége, véletlenszerűen vagy valamilyen megegyezés szerint, például balról jobbra haladva választja ki a kiterjesztendő csúcsot. Az ábrán szereplő kereső éppen a G jelű csúcs kiterjesztésénél tart. A következő lépésben a G csúcs már zárt lesz és a K csúcs meg fog jelenni a nyílt csúcsok listájában, valamint a színe is zöldre fog váltani.

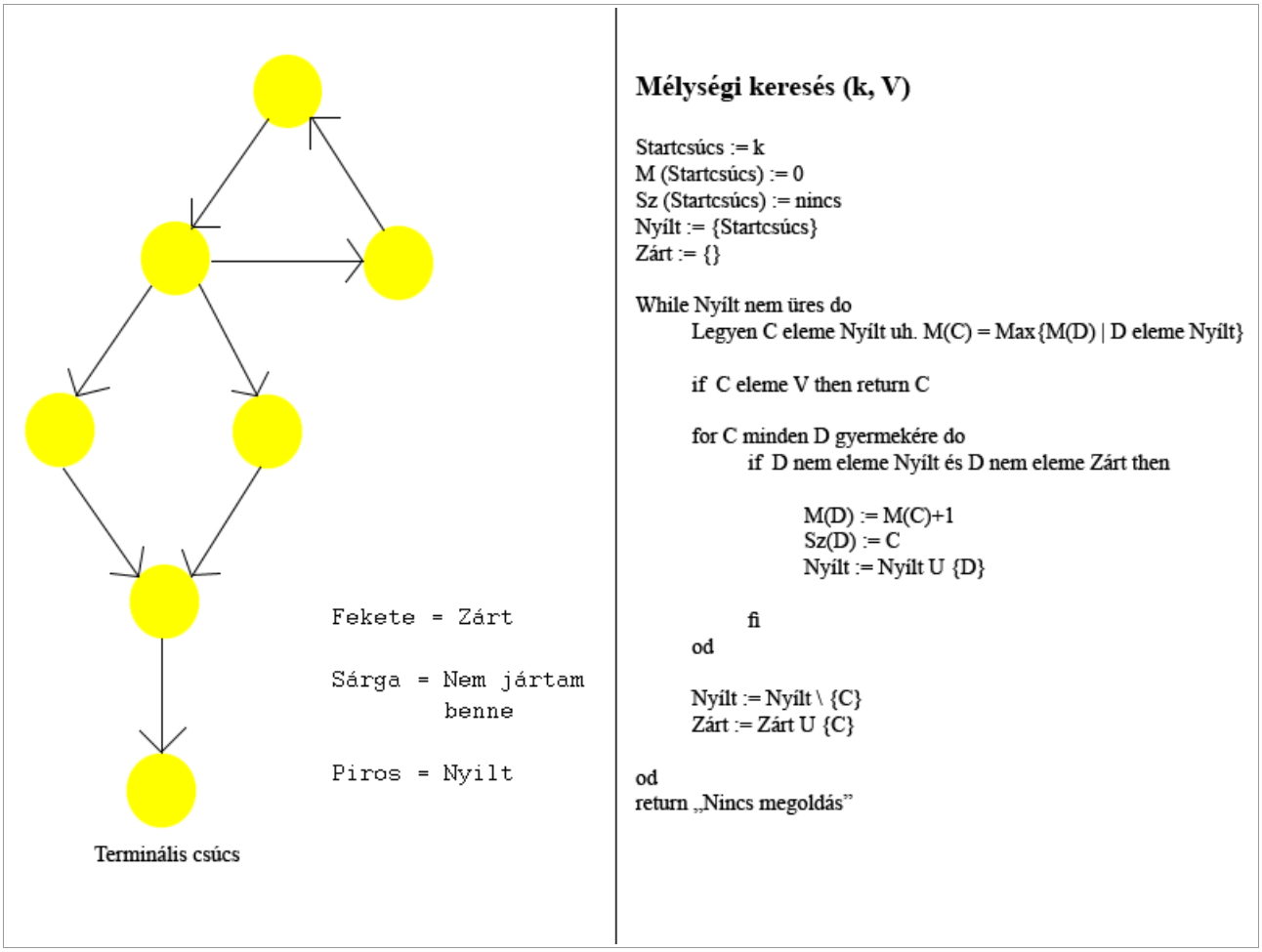
Jelmagyarázat:

| **Fogalmak** | **Jelölések** |
|---|---|
| Bejáratlan csúcs | kék |
| Kiterjesztés alatt lévő csúcs | sötétzöld |

| | |
|---|---|
| Nyílt csúcs | zöld |
| Zárt csúcs | piros |
| Mélységi szám | csúcs mellett lévő szám |
| Kezdő csúcs | A csúcs mellett S szerepel |
| Terminális csúcs | A csúcs mellett T szerepel |
| Nyílt/Zárt csúcsok listája | Az adott felirat alatti lista |

### 6.1.4. MÉLYSÉGI KERESÉS

Itt a mélységi keresést mutatjuk be röviden.




Látható, hogy az ábra két fő részre oszlik. Az animáció bal oldalán egy gráf látható, amiben a megoldást keressük. A jobb oldalon maga a mélységi keresés algoritmusa figyelhető meg. Az ábra jobb oldalán látható, amint az algoritmus lépésről lépésre végrehajtódik. Az aktuális sor piros. Ha valamelyik utasítás megváltoztatja valamely csúcs állapotát, akkor az ábra bal oldalán megváltozik a gráf megfelelő csúcsa. A sárga csúcsok a még fel nem derített csúcsokat jelölik, a feketék a zártak, a pirosak a nyílt csúcsok. A startcsúcs a gráf legfelső csúcsa, a cél (vagy terminális) csúcs a legalsó. A csúcsokban lévő számok a csúcs mélységi számát mutatják.

Itt kell még megemlíteni azt a fontos tényt is, hogy a képen vagy az animáción látható gráf

kialakítása nem teljesen véletlenszerű, hanem nagyon is tudatosan felépített formájú. Célja az, hogy olyan speciális eseteket is bemutasson az ábra, melyek a mesterséges intelligencia némely algoritmusánál problémásak szoktak lenni. Ilyenek a hurok és a kör.

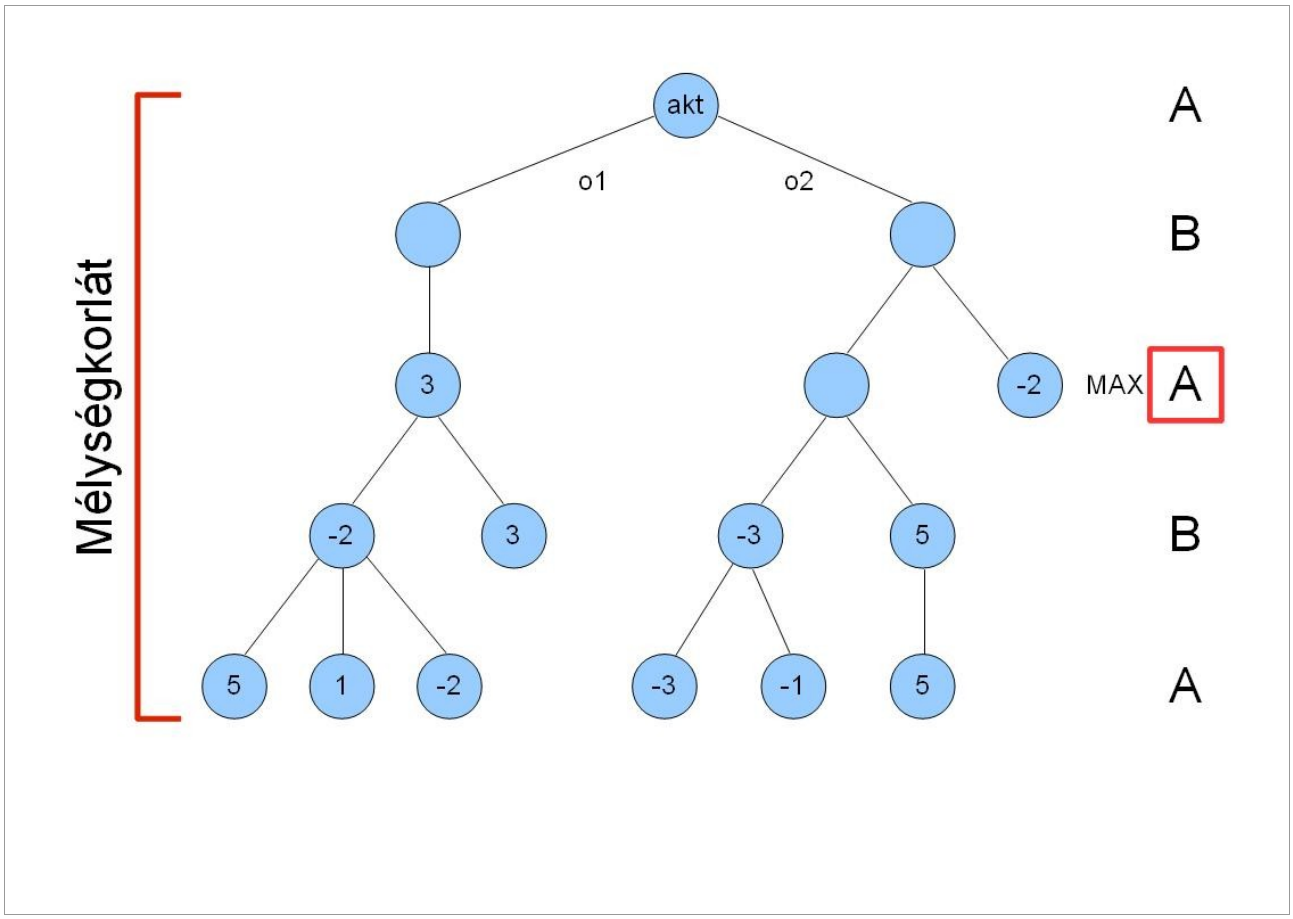
Hogy megértsük az animációt, röviden tekintsük át a mélységi keresést. A mélységi keresés a nyílt csúcsok közül (ezek a pirosak) azt terjeszti ki, aminek a mélysége (a csúcsban szereplő szám) a legnagyobb. Ha több ilyen csúcs is van, akkor ezek közül véletlenül választ. A kiterjesztésre kiválasztott csúcsot bekarikázzuk. A kiterjesztés hatására a csúcs még fel nem tárt (sárga) fiai nyíltak (pirosak) lesznek, a kiterjesztett csúcs pedig zárt (fekete). A keresésnek akkor van vége, ha vagy elfogynak a nyílt csúcsok (ekkor nem talált megoldást), vagy ha egy feltárt csúcs célcsúcs (ekkor megoldást talált).

A fenti magyarázatból is látható, hogy olyan elvont fogalmakat, mint nyílt csúcsok, konkrétan fogalmazhatok meg: piros csúcsok. Az alábbi táblázat a fogalmakat és az animáció jelöléseit párosítja össze:

| **Fogalmak** | **Jelölések** |
|---|---|
| fel nem tárt csúcsok | sárga |
| nyílt csúcsok | piros |
| zárt csúcsok | fekete |
| mélységi szám | csúcsban lévő szám |
| kiterjesztésre kiválasztott csúcs | bekarikázott csúcs |
| kiterjesztés | sárga leszármazottak pirossá válnak, a bekarikázott feketévé |

### 6.1.5. KÉTSZEMÉLYES JÁTÉKPROGRAMOK

Kétszemélyes stratégiai játékot, mint például a sakk, dáma vagy az ötödölő, játékfával tudunk szemléltetni. Ezt a játék állapottér-gráfjából tudjuk megalkotni. Kétszemélyes játékok esetében a fa egy-egy ága a különböző játékmeneteket reprezentálják. Ahhoz, hogy az egyes állapotokban tudjunk dönteni, hogy merre lépjünk tovább, vagyis melyik operátort alkalmazzuk, tudnunk kell, hogy melyik játékos léphet az egyes körökben. Ez alapján tudjuk felépíteni az És/Vagy gráfot, mely az útválasztásban nyújt segítséget.




Az ábrán a Minimax algoritmus látható. Az algoritmus úgy működik, hogy az aktuális állapotból előre meghatározott mélységig legeneráljuk a játékfát. Mikor ezzel megvagyunk, a fa levélelemeiben, azaz az egyes ágak utolsóként legenerált csúcsán megpróbálunk becslést adni, hogy mennyire szerencsés számunkra az adott állapot. Ezután a fában felfelé, az aktuális csúcs felé haladva, annak függvénynében, hogy mi vagy az ellenfelünk teszi az előző lépést, a csúcsba a gyermek elemek maximuma vagy minimuma kerül. Hogy mikor mit kell választani, azt az animáció jobb oldalán jelöljük a keret mellett.

Jelmagyarázat:

| **Fogalmak** | **Jelölések** |
|---|---|
| Aktuális csúcs | "akt" feliratú csúcs |
| Választható operátorok | Nyilak melletti felirat |
| Mélységkorlát | Ábra bal szélén található |
| Játékos | Ábra jobb oldalán "A" és "B" felirattal jelölve |
| Minimum/maximum választás | Ábra jobb oldalán a keret mellett jelölve |

## 6.2. ELŐNYÖK, HÁTRÁNYOK

Tapasztalatok szerint az animáció jelöléseit használva sokkal intuitívabb a magyarázat. Az algoritmus megfoghatóvá, láthatóvá válik, így kevesebben érzik azt, hogy nem érdemes figyelni, mert túl nehéz a tárgy. Így a pszichológiai problémára megoldást találtunk.

A pszichológiában különbségcsökkentésnek nevezik, amikor nehezen megoldható problémát az ember egyszerűbb problémákra vezet vissza, amit már meg tud oldani. Ezzel magyarázzuk, hogy az animációk használata csökkenti a nehéz anyagtól való idegenkedést, hiszen az animáció megértése sokkal egyszerűbb.

Az animációval a technikai probléma is megszűnik, hiszen nem kell táblát használnunk a gráf felrajzolásához. Az animációt mindenki megtekintheti a saját gépén, vagy projektor segítségével kivetíthető, és ha a diák elsőre nem tudta követni, az automatikusan újraindul.

Hátrány, hogy az animáció megtekintéséhez számítógépre van szükség, de legalábbis egy projektorra. Ha nincs áram, ezek a segédeszközök nem működnek. Ahhoz, hogy letöltsék, internet elérés kell.

A másik hátrányos jelenség az lehet, hogy a hallgatóság a fogalmakra azok jelölésével hivatkozik a visszakérdezés során is. Ez már kicsit nagyobb probléma, de amennyiben valaki eljutott oda, hogy megértette az algoritmusok működését, már nem kell olyan sok energiát fektetnie abba, hogy a korrekt fogalmakat is használni tudja. Sőt a visszakérdezés során a tanár akár közvetett módon segíthet is a jelölések segítségével, amikor a diák elakad gondolatmenetében.

# 7. Összegzés

Összefoglalásként nézzünk egy rövid heurisztikát, mit lehet csinálni, ha egy problémát elkezdtünk megoldani az itt tanult módszerekkel, mégse találja a programunk a megoldást emberi időn belül.

Jegyzetünkben bemutattuk a gráfkereső algoritmusok legismertebb változatait, megmutattuk, hogyan lehet ezeket felhasználni olyan problémák megoldására, amelyek a hétköznapi értelemben nehezek, az ember nem tudja egyből megoldani. Ugyanakkor azt is látnunk kell, hogy ezek az algoritmusok igazán a szisztematikus próbálgatáshoz állnak közel, leszámítva egy-két triviális optimalizációt, mint a körfigyelés.

Szisztematikus próbálgatás, ennyi az egész? Igen, ennyi az egész! Ki kell próbálnunk rengeteg lehetőséget, úgy, hogy minden lehetséges esetet le kell fednünk. Ez a szisztematikusság. Persze az sem mindegy, hogy milyen gyorsan próbáljuk ki a lehetőségeket.

Egy bizonyos szintig teljesen mindegy, hogy milyen sorrendben próbálgatjuk az operátorokat, mert viszonylag kicsi az állapottér, amit a gyors számítógépünk gyorsan átnéz. De könnyen belefuthatunk olyan problémába, amelynek az állapottere nagyon nagy.

Ilyenkor még mindig trükközhetünk azzal, hogy az operátorokat vagy magukat az algoritmusokat próbáljuk optimalizálni, például implementáljuk a backtrack hatékonyabb rokonát a backjumping-ot. Ettől is hatásosabb, ha lusta adatszerkezeteket használunk. Ez az jelenti, hogy az algoritmus lépései közül, amit lehet, azt halogatjuk egészen addig, amíg csak lehet. Az algoritmusok ilyen változatát lusta algoritmusoknak, az ezeket támogató adatszerkezeteket pedig lusta adatszerkezeteknek nevezzük.

Ez segít, de ez nem a mesterséges intelligencia válasza a problémára. A mesterséges intelligencia megoldása a probléma-reprezentáció helyes megválasztása. Egy problémát lehet úgy is reprezentálni, hogy nagyon nagy az állapottere, meg úgy is, hogy viszonylag kicsi.

Honnan tudom, hogy az állapottér kicsi vagy nagy? Általában, minél kevesebb operátorom van, annál kisebb az állapottér, tehát törekedni kell arra, hogy minél kevesebb operátorunk legyen.

Nézzünk egy egyszerű problémát: A 3 szerzetes és 3 kannibál probléma esetén lehet az az operátor, hogy a csónak átvisz néhány szerzetest és kannibált a folyó másik oldalára, de külön operátor lehet az is, hogy beülünk és kiszállunk. Ebben a második esetben sokkal nagyobb lesz az állapottér.

Sajnos egy nehézségi szint felett már az sem segít, ha kicsi állapotteret próbálunk elkészíteni, a feladat megoldása így is sok-sok óra lehet. Ilyenkor jön a kettes számú csodafegyver, a heurisztikák alkalmazása.

Mit is mondtunk? A gráfkeresés csak szisztematikus próbálgatás. A heurisztika mondja meg, hogy melyik operátort érdemes kipróbálni. Egy jó heurisztika nagyságrendekkel gyorsíthatja fel a keresést.

A jegyzetben viszonylag keveset foglalkoztunk heurisztikákkal, igazán csak a Minimax algoritmus kapcsán. Ez nem véletlen. Ha oda kerül a kedves olvasó, hogy játékprogramokhoz kell mesterséges intelligenciát készítenie, ami megmondja, mit lépjen a gép, akkor nagyon könnyű dolga lesz, kivéve a heurisztika megírását. Ez lesz a legnehezebb és ezen múlik majd, mennyire lesz „okos” a játék.

Ha a kettes számú csodafegyver sem segít, akkor még mindig marad egy esélyünk. Lemondunk a szisztematikusságról és bízunk a próbálgatásban. Igen, a végső csodafegyver a hegymászó módszer, méghozzá a restartos hegymászó. Ki hitte volna?

A restartos hegymászó algoritmus egyrészt használ heurisztikát, másrészt nagyon egyszerű, így gyors. Ez az algoritmus nem képes felismerni, ha nincs megoldás, de gyakran ez nem is érdekes, csak az, hogy ha van megoldás, akkor azt gyorsan megtaláljuk. Ha jó a heurisztika, akkor erre van

is esélyünk. Ha nem találjuk meg gyorsan a megoldást, akkor előre megadott idő után feladhatjuk a keresést.

Restartos hegymászó algoritmust használunk az egyik legjelentősebb mesterséges intelligencia probléma, a SAT probléma megoldására is. Azt a változatát, amit ott használunk, Random Walk-nak nevezzük. A SAT problémát részletesen a Számításelmélet jegyzetben tárgyaljuk.

Ebből a rövid eszmefuttatásból is láthatja kedves olvasónk, hogy ez a jegyzet csak megkarcolta a mesterséges intelligencia felszínét, ugyanakkor az itt leírtak nagyon jó alapot nyújtanak a további tanulmányokhoz.

# 8. Példaprogramok

Ebben a fejezetben több összefüggő osztályt ismertetünk. Az osztályok két algoritmust, a visszalépéses keresést (backtrack) és a mélységi keresést valósítanak meg. Ezek az osztályok néhány év tapasztalatát sűrítik magukba, sokszor kipróbáltuk, teszteltük, tehát valószínűleg működnek. Mégis, néhány helyen hozzá kell nyúlni a forráskódhoz, ha az ember szeretné felhasználni egy másik gráfkeresési probléma megoldására. Szerencsére ehhez csak egy saját Állapot osztályt kell írni, aminek az AbsztraktÁllapot osztályból kell származni. Ezen túl sem a Csúcs sem a BackTrack stb. osztályokat nem kell átírni, csak a fő programot.

A fejezet következő részében ismertetjük az egyes osztályokat, külön kihangsúlyozva, mi az, amit át kell írni és mi az, ami biztosan maradhat jelenlegi formájában.

## 8.1. Az AbsztraktÁllapot osztály

Az AbsztrakÁllapot osztály minden Állapot osztály őse. Megfelel az állapottér reprezentációnak. Előírja, hogy agy Állapot osztálynak tartalmaznia kell egy ÁllapotE függvényt, ami megfelel az állapot-e predikátumnak, egy CélÁllapotE függvényt, ami megfelel a célállapot-e predikátumnak. Az operátorok halmaza első ránézésre egy kicsit furán van megadva. Operátorok halmaza nincs, de tudom, hogy hány eleme lenne. Ezt adja vissza az OperátorokSzáma függvény. Ezt is a gyermek osztálynak kell kifejtenie. Az operátorokat sem kell megadnom, de tudom, hogy az úgynevezett szuper operátor segítségével elérhetem őket. Erre szolgál SzuperOperátor(int i) metódus.

A többi metódust nem muszáj felülírni. A Clone metódus valószínűleg nem kell egy gyermek osztályban sem felülírni. Csak akkor kellene, ha az Állapot belső mezői tömbök lennének. Ilyenkor a tömböket is klónozni kell. Ezt nevezzük mély klónozásnak.
Az Equals metódust akkor kell felülírni, ha körfigyeléses backtrack-et szeretnénk használni, vagy körfigyeléses mélységi keresést (mélységi keresésnél a körfigyelés alapértelmezett). Ha az Equals-t felülírjuk, akkor illik felülírni a GetHashCode nevű metódust is. Ez nem kötelező, de egyébként csúnya figyelmeztetéseket ad a fordító.

### 8.1.1. Forráskód

```
/// <summary>
/// Minden állapot osztály őse.
/// </summary>
abstract class AbsztraktÁllapot : ICloneable
{
   // Megvizsgálja, hogy a belső állapot állapot-e.
   // Ha igen, akkor igazat ad vissza, egyébként hamisat.
   public abstract bool ÁllapotE();
   // Megvizsgálja, hogy a belső állapot célállapot-e.
   // Ha igen, akkor igazat ad vissza, egyébként hamisat.
   public abstract bool CélÁllapotE();
   // Visszaadja az alapoperátorok számát.
   public abstract int OperátorokSzáma();
   // A szuper operátoron keresztül lehet elérni az összes operátort.
   // Igazat ad vissza, ha az i.dik alap operátor alkalmazható a belső állapotra.
   // for ciklusból kell hívni 0-tól kezdve az alap operátorok számig. Pl. így:
   // for (int i = 0; i < állapot.GetNumberOfOps(); i++)
   // {
   //     AbsztraktÁllapot klón=(AbsztraktÁllapot)állapot.Clone();
   //     if (klón.SzuperOperátor(i))
```

```
    //     {
    //         Console.WriteLine("Az {0} állapotra az {1}.dik " +
    //             "operátor alkalmazható", állapot, i);
    //     }
    // }
    public abstract bool SzuperOperátor(int i);
    // Klónoz. Azért van rá szükség, mert némelyik operátor hatását vissza kell vonnunk.
    // A legegyszerűbb, hogy az állapotot leklónozom. Arra hívom az operátort.
    // Ha gond van, akkor visszatérek az eredeti állapothoz.
    // Ha nincs gond, akkor a klón lesz az állapot, amiből folytatom a keresést.
    // Ez sekély klónozást alkalmaz. Ha elég a sekély klónozás, akkor nem kell felülírni a gyermek osztályban.
    // Ha mély klónozás kell, akkor mindenképp felülírandó.
    public virtual object Clone() { return MemberwiseClone(); }
    // Csak akkor kell felülírni, ha emlékezetes backtracket akarunk használni, vagy mélységi keresést.
    // Egyébként maradhat ez az alap implementáció.
    // Programozás technikailag ez egy kampó metódus, amit az OCP megszegése nélkül írhatok felül.
    public override bool Equals(Object a) { return false; }
    // Ha két példány egyenlő, akkor a hasítókódjuk is egyenlő.
    // Ezért, ha valaki felülírja az Equals metódust, ezt is illik felülírni.
    public override int GetHashCode() { return base.GetHashCode(); }
}
```

## 8.2. Hogyan kell elkészíteni saját operátoraimat?

Ebben a fejezetben a VakÁllapot osztályt tárgyaljuk. Ennek az osztálynak nincs funkciója, csak bemutatja általánosan, hogy hogyan kell alapoperátorokat és paraméteres operátorokat létrehozni és ezeket bekötni a szuper operátorban. Alapoperátornak nevezzük azokat az operátorokat, amiknek nincs paramétere, vagy már minden paramétere fix. Ezeket kell felsorolni egy switch szerkezetben a szuper operátorban. Egy paraméteres operátor segítségével több alapoperátor is megadható a paraméterek különböző értékeivel.
A példából jól látható, hogy minden operátor igaz/hamis visszatérési értékkel bír. Igaz, ha alkalmazható az Állapot példány jelenlegi belső állapotára, egyébként hamis. Az operátorokat sokféleképpen meg lehetne valósítani programozás-technikailag. Talán az itt választott megoldás követi legjobban az OOP egységbezárásának elvét, ami azt mondja ki, hogy a példány belső állapotát csak a példány metódusai tudják módosítani. Mesterséges intelligenciából tudjuk, hogy ezeket a metódusokat operátornak hívják.
Mivel az alap operátorokat a szuper operátoron keresztül kell elérni, ezért ezek láthatósági szintje lehet privát. Csak a szuper operátornak kell publikusnak lennie.
Minden operátornak van előfeltétele. Utófeltétele is van, de az mindig megegyezik az ÁllapotE predikátummal. Egy új operátornak mindig azzal kell kezdődnie, hogy meghívja az előfeltételét. Az előfeltétel ugyanazokat a paramétereket kell, hogy kapja, mint maga az operátor. Ha az előfeltétel nem igaz, akkor eleve nem alkalmazható az operátor. Ha igaz, akkor végre kell hajtani az állapot átmenetet. Ehhez az állapot mezőit kell a feladatnak megfelelően módosítani. Ezután ellenőrizni kell, hogy nem léptünk-e ki az állapottérből az ÁllapotE predikátummal. Ha ez igazat ad, vagyis az állapottéren belül maradtunk, akkor készen vagyunk, alkalmazható az operátor. Egyébként nem alkalmazható és ráadásul az állapot átmenetet is vissza kell csinálni.

### 8.2.1. Forráskód

```
/// <summary>
/// A VakÁllapot csak a szemléltetés kedvért van itt.
/// Megmutatja, hogy kell az operátorokat megírni és bekötni a szuper operátorba.
/// </summary>
abstract class VakÁllapot : AbsztraktÁllapot
{
    // Itt kell megadni azokat a mezőket, amelyek tartalmazzák a belső állapotot.
```

```
// Az operátorok belső állapot átmenetet hajtanak végre.

// Először az alapoperátorokat kell megírni.
// Minden operátornak van előfeltétele.
// Minden operátor utófeltétele megegyezik az ÁllapotE predikátummal.
// Az operátor igazat ad vissza, ha alkalmazható, hamisat, ha nem alkalmazható.
// Egy operátor alkalmazható, ha a belső állapotra igaz
// az előfeltétele és az állapotátmenet után igaz az utófeltétele.
// Ez az első alapoperátor.
private bool op1()
{
   // Ha az előfeltétel hamis, akkor az operátor nem alkalmazható.
   if (!preOp1()) return false;
   // állapot átmenet
   //
   // TODO: Itt kell kiegészíteni a kódot!
   //
   // Utófeltétel vizsgálata, ha igaz, akkor alkalmazható az operátor.
   if (ÁllapotE()) return true;
   // Egyébként vissza kell vonni a belső állapot átmenetet,
   //
   // TODO: Itt kell kiegészíteni a kódot!
   //
   // és vissza kell adni, hogy nem alkalmazható az operátor.
   return false;
}
// Az első alapoperátor előfeltétele. Az előfeltétel neve általában ez: pre+operátor neve.
// Ez segíti a kód megértését, de nyugodtan eltérhetünk ettől.
private bool preOp1()
{
   // Ha igaz az előfeltétel, akkor igazat ad vissza.
   return true;
}
// Egy másik operátor.
private bool op2()
{
   if (!preOp2()) return false;
   // Állapot átmenet:
   // TODO: Itt kell kiegészíteni a kódot!
   if (ÁllapotE()) return true;
   // Egyébként vissza kell vonni a belső állapot átmenetet:
   // TODO: Itt kell kiegészíteni a kódot!
   return false;
}
private bool preOp2()
{
   // Ha igaz az előfeltétel, akkor igazat ad vissza.
   return true;
}
// Nézzük, mi a helyzet, ha az operátornak van paramétere.
// Ilyenkor egy operátor több alapoperátornak felel meg.
private bool op3(int i)
{
   // Az előfeltételt ugyanazokkal a pereméterekkel kell hívni, mint magát az operátort.
   if (!preOp3(i)) return false;
   // Állapot átmenet:
   // TODO: Itt kell kiegészíteni a kódot!
   if (ÁllapotE()) return true;
   // egyébként vissza kell vonni a belső állapot átmenetet
   // TODO: Itt kell kiegészíteni a kódot!
   return false;
}
// Az előfeltétel paraméterlistája megegyezik az operátor paraméterlistájával.
```

```
    private bool preOp3(int i)
    {
        // Ha igaz az előfeltétel, akkor igazat ad vissza. Az előfeltétel függ a paraméterektől.
        return true;
    }
    // Ez a szuper operátor. Ezen keresztül lehet hívni az alapoperátorokat.
    // Az i paraméterrel mondjuk meg, hanyadik operátort akarjuk hívni.
    // Általában egy for ciklusból hívjuk, ami 0-tól az OperátorokSzáma()-ig fut.
    public override bool SzuperOperátor(int i)
    {
        switch (i)
        {
            // Itt kell felsorolnom az összes alapoperátort.
            // Ha egy új operátort veszek fel, akkor ide is fel kell venni.
            case 0: return op1();
            case 1: return op2();
            // A paraméteres operátorok több alap operátornak megfelelnek, ezért itt több sor is tartozik hozzájuk.
            // Hogy hány sor, az feladat függő.
            case 3: return op3(0);
            case 4: return op3(1);
            case 5: return op3(2);
            default: return false;
        }
    }
    // Visszaadja az alap operátorok számát.
    public override int OperátorokSzáma()
    {
        // Az utolsó case számát kell itt visszaadni.
        // Ha bővítjük az operátorok számát, ezt a számot is növelni kell.
        return 5;
    }
}
```

## 8.3. Egy példa Állapot osztály: ÉhesHuszárÁllapot

Ebben a fejezetben egy jól megírt állapottér reprezentációt láthatunk. Láthatjuk, hogy miden Állapot osztálynak az AbsztraktÁllapot osztályból kell származnia. Láthatjuk, hogyan kell megvalósítani az egyes metódusokat. Mivel egy saját feladat megoldásához csak saját Állapot osztályt kell írni, ezért érdemes ebből vagy a következő fejezetben ismertetett példából kiindulni.

### 8.3.1. Forráskód

```
/// <summary>
/// Ez a “éhes huszár” probléma állapottér reprezentációja.
/// A huszárnak az állomás helyétől, a bal felső sarokból,
/// el kell jutnia a kantinba, ami a jobb alsó sarokban van.
/// A táblát egy (N+4)*(N+4) mátrixszal ábrázolom.
/// A külső 2 széles rész margó, a belső rész a tábla.
/// A margó használatával sokkal könnyebb megírni az ÁllapotE predikátumot.
/// A 0 jelentése üres. Az 1 jelentése, itt van a ló.
/// 3*3-mas tábla esetén a kezdő állapot:
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,1,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
///  0,0,0,0,0,0,0
/// A fenti reprezentációból látszik, hogy elég csak a ló helyét nyilvántartani,
```

```
/// mert a táblán csak a ló van. Így a kezdő állpot (bal felső sarokból indulunk):
///  x = 2
///  y = 2
/// A célállapot (jobb alsó sarokba megyek):
///  x = N+1
///  y = N+1
/// Operátorok:
///  A lehetséges 8 ló lépés.
/// </summary>
class ÉhesHuszárÁllapot : AbsztraktÁllapot
{
   // Alapértelmezetten egy 3*3-as sakktáblán fut.
   static int N = 3;
   // A belső állapotot leíró mezők.
   int x, y;
   // Beállítja a kezdő állapotra a belső állapotot.
   public ÉhesHuszárÁllapot()
   {
      x = 2; // A bal felső sarokból indulunk, ami a margó
      y = 2; // miatt a (2,2) koordinátán van.
   }
   // Beállítja a kezdő állapotra a belső állapotot.
   // Itt lehet megadni a tábla méretét is.
   public ÉhesHuszárÁllapot(int n)
   {
      x = 2;
      y = 2;
      N = n;
   }
   public override bool CélÁllapotE()
   {
      // A jobb alsó sarok a margó miatt a (N+1,N+1) helyen van.
      return x == N + 1 && y == N + 1;
   }
   public override bool ÁllapotE()
   {
      // a ló nem a margon van
      return x >= 2 && y >= 2 && x <= N + 1 && y <= N + 1;
   }
   private bool preLóLépés(int x, int y)
   {
      // jó lólépés-e, ha nem akkor return false
      if (!(x * y == 2 || x * y == -2)) return false;
      return true;
   }
   /* Ez az operátorom, igaz ad vissza, ha alakalmazható,
    * egyébként hamisat.
    * Paraméterek:
    * x: x irányú elmozdulás
    * y: y irányú elmozdulás
    * Az előfeltétel ellenőrzi, hogy az elmozdulás lólépés-e.
    * Az utófeltétel ellenőrzi, hogy a táblán maradtunk-e.
    * Példa:
    * lóLépés(1,-2) jelentése felfelé 2-öt jobbra 1-et.
    */
   private bool lóLépés(int x, int y)
   {
      if (!preLóLépés(x, y)) return false;
      // Ha az előfeltétel igaz, akkor megcsinálom az
      // állapot átmenetet.
      this.x += x;
      this.y += y;
      // Az utófeltétel mindig megegyezik az ÁllapotE-vel.
```

```
        if (ÁllapotE()) return true;
        // Ha az utófeltétel nem igaz, akkor vissza kell csinálni.
        this.x -= x;
        this.y -= y;
        return false;
    }
    public override bool SzuperOperátor(int i)
    {
        switch (i)
        {   // itt sorolom fel a lehetséges 8 lólépést
            case 0: return lóLépés(1, 2);
            case 1: return lóLépés(1, -2);
            case 2: return lóLépés(-1, 2);
            case 3: return lóLépés(-1, -2);
            case 4: return lóLépés(2, 1);
            case 5: return lóLépés(2, -1);
            case 6: return lóLépés(-2, 1);
            case 7: return lóLépés(-2, -1);
            default: return false;
        }
    }
    public override int OperátorokSzáma() { return 8; }
    // A kiíratásnál kivonom x-ből és y-ból a margó szélességét.
    public override string ToString() { return (x-2) + " : " + (y-2); }
    public override bool Equals(Object a)
    {
        ÉhesHuszárÁllapot aa = (ÉhesHuszárÁllapot)a;
        return aa.x == x && aa.y == y;
    }
    // Ha két példány egyenlő, akkor a hasítókódjuk is egyenlő.
    public override int GetHashCode()
    {
        return x.GetHashCode() + y.GetHashCode();
    }
}
```

## 8.4. MÉG EGY PÉLDA ÁLLAPOT OSZTÁLY

Ebben a fejezetben a jól ismert 3 szerzetes és 3 kannibál probléma állapottér reprezentációt láthatjuk. Mint minden Állapot osztálynak, ennek is az AbsztraktÁllapot osztályból kell származnia. Mivel egy saját feladat megoldásához csak saját Állapot osztályt kell írni, ezért érdemes ebből vagy az előző fejezetben ismertetett példából kiindulni.

### 8.4.1. A 3 SZERZETES ÉS 3 KANNIBÁL PÉLDA FORRÁSKÓDJA

```
/// <summary>
/// A "3 szerzetes és 3 kannibál" probléma állapottér reprezentációja.
/// Illetve általánosítása akárhány szerzetesre és kannibálra.
/// Probléma: 3 szerzet és 3 kannibál van a folyó bal partján.
/// Át kell juttatni az összes embert a másik partra.
/// Ehhez rendelkezésre áll egy két személyes csónak.
/// Egy ember is elég az átjutáshoz, de kettőnél több ember nem fér el.
/// Ha valaki átmegy a másik oldalra, akkor ki is kell szállni, nem maradhat a csónakban.
/// A gond ott van, hogy ha valamelyik parton több kannibál van,
/// mint szerzetes, akkor a kannibálok megeszik a szerzeteseket.
/// Kezdő állapot:
///  3 szerzetes a bal oldalon.
///  3 kannibál a bal oldalon.
///  A csónak a bal parton van.
```

```
///  0 szerzetes a jobb oldalon.
///  0 kannibál a jobb oldalon.
/// Ezt az állapotot ezzel a rendezett 5-össel írjuk le:
///  (3,3,'B',0,0)
/// A célállapot:
///  (0,0,'J',3,3)
/// Operátor:
///  Op(int sz, int k):
///  sz darab szerzetes átmegy a másik oldalra és
///  k darab kannibál átmegy a másik oldalra.
/// Lehetséges paraméterezése:
///  Op(1,0): 1 szerzetes átmegy a másik oldalra.
///  Op(2,0): 2 szerzetes átmegy a másik oldalra.
///  Op(1,1): 1 szerzetes és 1 kannibál átmegy a másik oldalra.
///  Op(0,1): 1 kannibál átmegy a másik oldalra.
///  Op(0,2): 2 kanibál átmegy a másik oldalra.
/// </summary>
class SzerzetesekÉsKannibálokÁllapot : AbsztraktÁllapot
{
   int sz;   // ennyi szerzetes van összesen
   int k;    // ennyi kannibál van összesen
   int szb;  // szerzetesek száma a bal oldalon
   int kb;   // kannibálok száma a bal oldalon
   char cs;  // Hol a csónak? Értéke vagy 'B' vagy 'J'.
   int szj;  // szerzetesek száma a jobb oldalon
   int kj;   // kannibálok száma a jobb oldalon
   public SzerzetesekÉsKannibálokÁllapot(int sz, int k) // beállítja a kezdő állapotot
   {
      this.sz = sz;
      this.k = k;
      szb = sz;
      kb = k;
      cs = 'B';
      szj = 0;
      kj = 0;
   }
   public override bool ÁllapotE()
   {   // igaz, ha a szerzetesek biztonságban vannak
      return ((szb >= kb) || (szb == 0)) &&
         ((szj >= kj) || (szj == 0));
   }
   public override bool CélÁllapotE()
   {
      // igaz, ha mindenki átért a jobb oldalra
      return szj == sz && kj == k;
   }
   private bool preOp(int sz, int k)
   {
      // A csónak 2 személyes, legalább egy ember kell az evezéshez.
      // Ezt végül is felesleges vizsgálni, mert a szuper operátor csak ennek megfelelően hívja.
      if (sz + k > 2 || sz + k < 0 || sz < 0 || k < 0) return false;
      // Csak akkor lehet átvinni sz szerzetest és
      // k kannibált, ha a csónak oldalán van is legalább ennyi.
      if (cs == 'B')
         return szb >= sz && kb >= k;
      else
         return szj >= sz && kj >= k;
   }
   // Átvisz a másik oldalra sz darab szerzetes és k darab kannibált.
   private bool op(int sz, int k)
   {
      if (!preOp(sz, k)) return false;
      SzerzetesekÉsKannibálokÁllapot mentes = (SzerzetesekÉsKannibálokÁllapot)Clone();
```

```
        if (cs == 'B')
        {
            szb -= sz;
            kb -= k;
            cs = 'J';
            szj += sz;
            kj += k;
        }
        else
        {
            szb += sz;
            kb += k;
            cs = 'B';
            szj -= sz;
            kj -= k;
        }
        if (ÁllapotE()) return true;
        szb = mentes.szb;
        kb = mentes.kb;
        cs = mentes.cs;
        szj = mentes.szj;
        kj = mentes.kj;
        return false;
    }
    public override int OperátorokSzáma() { return 5; }
    public override bool SzuperOperátor(int i)
    {
        switch (i)
        {
            case 0: return op(0, 1);
            case 1: return op(0, 2);
            case 2: return op(1, 1);
            case 3: return op(1, 0);
            case 4: return op(2, 0);
            default: return false;
        }
    }
    public override string ToString()
    {
        return szb + "," + kb + "," + cs + "," + szj + "," + kj;
    }
    public override bool Equals(Object a)
    {
        SzerzetesekÉsKannibálokÁllapot aa = (SzerzetesekÉsKannibálokÁllapot)a;
        // szj és kj számítható, ezért nem kell vizsgálni
        return aa.szb == szb && aa.kb == kb && aa.cs == cs;
    }
    // Ha két példány egyenlő, akkor a hasítókódjuk is egyenlő.
    public override int GetHashCode()
    {
        return szb.GetHashCode() + kb.GetHashCode() + cs.GetHashCode();
    }
}
```

## 8.5. A Csúcs osztály

Ebben a fejezetben a csúcs osztályt ismertetjük. Ehhez szerencsére nem kell hozzányúlni, ha egy saját feladatot akarunk megoldani backtrack-kel vagy mélységi kereséssel. Abban az esetben, ha nem mélységre épülő algoritmust, mondjuk az optimális keresőt akarjuk implementálni, akkor sajnos módosítanunk kell, hogy az állapot tartalmazza a költség függvény értékét is.

Ebben a formájában a csúcs tartalmaz egy állapotot, a mélységét és a szülőjét. Támogatja a kiterjesztést, illetve a szuper operátor segítségével egyesével is használhatjuk az operátorokat.
Mivel a csúcs ismeri a szülőjét, ezért egy út nyilvántartásához elegendő az út utolsó csúcsát nyilvántartani. Ez azt jelenti, hogy a megoldáshoz elegendő visszaadni a terminális csúcsot, amibe a megoldás vezet.

### 8.5.1. Forráskód

```
/// <summary>
/// A csúcs tartalmaz egy állapotot, a csúcs mélységét, és a csúcs szülőjét.
/// Így egy csúcs egy egész utat reprezentál a start csúcsig.
/// </summary>
class Csúcs
{
   // A csúcs tartalmaz egy állapotot, a mélységét és a szülőjét
   AbsztraktÁllapot állapot;
   int mélység;
   Csúcs szülő; // A szülőkön felfelé haladva a start csúcsig jutok.
   // Konstruktor:
   // A belső állapotot beállítja a start csúcsra.
   // A hívó felelőssége, hogy a kezdő állapottal hívja meg.
   // A start csúcs mélysége 0, szülője nincs.
   public Csúcs(AbsztraktÁllapot kezdőÁllapot)
   {
      állapot = kezdőÁllapot;
      mélység = 0;
      szülő = null;
   }
   // Egy új gyermek csúcsot készít.
   // Erre még meg kell hívni egy alkalmazható operátor is, csak azután lesz kész.
   public Csúcs(Csúcs szülő)
   {
      állapot = (AbsztraktÁllapot)szülő.állapot.Clone();
      mélység = szülő.mélység + 1;
      this.szülő = szülő;
   }
   public Csúcs GetSzülő() { return szülő; }
   public int GetMélység() { return mélység; }
   public bool TerminálisCsúcsE() { return állapot.CélÁllapotE(); }
   public int OperátorokSzáma() { return állapot.OperátorokSzáma(); }
   public bool SzuperOperátor(int i) { return állapot.SzuperOperátor(i); }
   public override bool Equals(Object obj)
   {
      Csúcs cs = (Csúcs)obj;
      return állapot.Equals(cs.állapot);
   }
   public override int GetHashCode() { return állapot.GetHashCode(); }
   public override String ToString() { return állapot.ToString(); }
   // Alkalmazza az összes alkalmazható operátort.
   // Visszaadja az így előálló új csúcsokat.
   public List<Csúcs> Kiterjesztes()
   {
      List<Csúcs> újCsúcsok = new List<Csúcs>();
      for (int i = 0; i < OperátorokSzáma(); i++)
      {
         // Új gyermek csúcsot készítek.
         Csúcs újCsúcs = new Csúcs(this);
         // Kiprobálom az i.dik alapoperátort. Alkalmazható?
         if (újCsúcs.SzuperOperátor(i))
         {
            // Ha igen, hozzáadom az újakhoz.
```

```
            újCsúcsok.Add(újCsúcs);
          }
        }
        return újCsúcsok;
    }
}
```

## 8.6. A GráfKereső osztály

Ebben a fejezetben a gráfkereső algoritmusok közös ősét mutatjuk be. Ezt az osztályt valószínűleg nem kell módosítani még akkor sem, ha megírunk egy új gráfkereső algoritmust. Minden gráfkeresésnek ebből az osztályból kell származnia.

Egy absztrakt metódusa van, a Keresés. Ezt kell a gyermek osztályokban megvalósítani. Ez a backtrack esetén maga a backtrack, mélységinél a mélységi keresés lesz. A Keresés egy csúcsot ad vissza. Ennek egy terminális csúcsnak kell lennie, ha van megoldás, egyébként null-nak. A null visszatérési érték jelzi, hogy nincs megoldás.
A megoldást ki lehet íratni a megoldásKiírása metódussal. Ez csak egy kis segédlet, nem muszáj használni. Ha mégis használni akarjuk, akkor a főprogramban láthatunk példát a hívására.
Minden gráf kereső a start csúcsból indítja a keresést, ezért előírunk egy konstruktort is. Ezt minden gyermek osztálynak meg kell hívnia.

### 8.6.1. Forráskód

```
/// <summary>
/// Minden gráfkereső algoritmus őse.
/// A gráfkeresőknek csak a Keresés metódust kell megvalósítaniuk.
/// Ez visszaad egy terminális csúcsot, ha talált megoldást, egyébként null értékkel tér vissza.
/// A terminális csúcsból a szülő referenciákon felfelé haladva áll elő a megoldás.
/// </summary>
abstract class GráfKereső
{
    private Csúcs startCsúcs; // A start csúcs csúcs.
    // Minden gráfkereső a start csúcsból kezd el keresni.
    public GráfKereső(Csúcs startCsúcs)
    {
        this.startCsúcs = startCsúcs;
    }
    // Jobb, ha a start csúcs privát, de a gyermek osztályok lekérhetik.
    protected Csúcs GetStartCsúcs() { return startCsúcs; }
    /// Ha van megoldás, azaz van olyan út az állapottér gráfban,
    /// ami a start csúcsból egy terminális csúcsba vezet,
    /// akkor visszaad egy megoldást, egyébként null.
    /// A megoldást egy terminális csúcsként adja vissza.
    /// Ezen csúcs szülő referenciáin felfelé haladva adódik a megoldás fordított sorrendben.
    public abstract Csúcs Keresés();
    /// <summary>
    /// Kiíratja a megoldást egy terminális csúcs alapján.
    /// Feltételezi, hogy a terminális csúcs szülő referenciáján felfelé haladva eljutunk a start csúcshoz.
    /// A csúcsok sorrendjét megfordítja, hogy helyesen tudja kiírni a megoldást.
    /// Ha a csúcs null, akkor kiírja, hogy nincs megoldás.
    /// </summary>
    /// <param name="egyTerminálisCsúcs">
    /// A megoldást képviselő terminális csúcs vagy null.
    /// </param>
    public void megoldásKiírása(Csúcs egyTerminálisCsúcs)
    {
        if (egyTerminálisCsúcs == null)
        {
```

```
            Console.WriteLine("Nincs megoldás");
            return;
        }
        // Meg kell fordítani a csúcsok sorrendjét.
        Stack<Csúcs> megoldás = new Stack<Csúcs>();
        Csúcs aktCsúcs = egyTerminálisCsúcs;
        while (aktCsúcs != null)
        {
            megoldás.Push(aktCsúcs);
            aktCsúcs = aktCsúcs.GetSzülő();
        }
        // Megfordítottuk, lehet kiírni.
        foreach(Csúcs akt in megoldás) Console.WriteLine(akt);
    }
}
```

# 8.7. A BackTrack osztály

Ebben a fejezetben a backtrack gráf keresés egy rekurzív megoldását mutatjuk be. A különböző konstruktorokkal lehet mélységi korlátot, emlékezetet vagy ezek kombinációját használó backtracket is létrehozni.

Érdemes megpróbálni a lenti megvalósítást átírni ciklusra. Ez remek ujjgyakorlat és ha nem sikerül, akkor még mindig ott van az eredeti a beadandó program elkészítéséhez. Ebben az esetben hozzá se kell nyúlnunk ehhez a kódhoz. Rengetegszer kipróbált, letesztelt a kód.

## 8.7.1. Forráskód

```
/// <summary>
/// A backtrack gráfkereső algoritmust megvalósító osztály.
/// A három alap backtrack algoritmust egyben tartalmazza. Ezek
/// - az alap backtrack
/// - mélységi korlátos backtrack
/// - emlékezetes backtrack
/// Az ág-korlátos backtrack nincs megvalósítva.
/// </summary>
class BackTrack : GráfKereső
{
    int korlát; // Ha nem nulla, akkor mélységi korlátos kereső.
    bool emlékezetes; // Ha igaz, emlékezetes kereső.
    public BackTrack(Csúcs startCsúcs, int korlát, bool emlékezetes) : base(startCsúcs)
    {
        this.korlát = korlát;
        this.emlékezetes = emlékezetes;
    }
    // nincs mélységi korlát, se emlékezet
    public BackTrack(Csúcs startCsúcs) : this(startCsúcs, 0, false) { }
    // mélységi korlátos kereső
    public BackTrack(Csúcs startCsúcs, int korlát) : this(startCsúcs, korlát, false) { }
    // emlékezetes kereső
    public BackTrack(Csúcs startCsúcs, bool emlékezetes) : this(startCsúcs, 0, emlékezetes) { }
    // A keresés a start csúcsból indul.
    // Egy terminális csúcsot ad vissza. A start csúcsból el lehet jutni ebbe a terminális csúcsba.
    // Ha nincs ilyen, akkor null értéket ad vissza.
    public override Csúcs Keresés() { return Keresés(GetStartCsúcs()); }
    // A kereső algoritmus rekurzív megvalósítása.
    // Mivel rekurzív, ezért a visszalépésnek a "return null" felel meg.
    private Csúcs Keresés(Csúcs aktCsúcs)
    {
        int mélység = aktCsúcs.GetMélység();
```

```
        // mélységi korlát vizsgálata
        if (korlát > 0 && mélység >= korlát) return null;
        // emlékezet használata kör kiszűréséhez
        Csúcs aktSzülő = null;
        if (emlékezetes) aktSzülő = aktCsúcs.GetSzülő();
        while (aktSzülő != null)
        {
           // Ellenőrzöm, hogy jártam-e ebben az állapotban. Ha igen, akkor visszalépés.
           if (aktCsúcs.Equals(aktSzülő)) return null;
           // Visszafelé haladás a szülői láncon.
           aktSzülő = aktSzülő.GetSzülő();
        }
        if (aktCsúcs.TerminálisCsúcsE())
        {
           // Megvan a megoldás, vissza kell adni a terminális csúcsot.
           return aktCsúcs;
        }
        // Itt hívogatom az alapoperátorokat a szuper operátoron
        // keresztül. Ha valamelyik alkalmazható, akkor új csúcsot
        // készítek, és meghívom önmagamat rekurzívan.
        for (int i = 0; i < aktCsúcs.OperátorokSzáma(); i++)
        {
           // Elkészítem az új gyermek csúcsot.
           // Ez csak akkor lesz kész, ha alkalmazok rá egy alkalmazható operátort is.
           Csúcs újCsúcs = new Csúcs(aktCsúcs);
           // Kipróbálom az i.dik alapoperátort. Alkalmazható?
           if (újCsúcs.SzuperOperátor(i))
           {
              // Ha igen, rekurzívan meghívni önmagam az új csúcsra.
              // Ha nem null értéket ad vissza, akkor megvan a megoldás.
              // Ha null értéket, akkor ki kell próbálni a következő alapoperátort.
              Csúcs terminális = Keresés(újCsúcs);
              if (terminális != null)
              {
                 // Visszaadom a megoldást képviselő terminális csúcsot.
                 return terminális;
              }
              // Az else ágon kellene visszavonni az operátort.
              // Erre akkor van szükség, ha az új gyermeket létrehozásában nem lenne klónozást.
              // Mivel klónoztam, ezért ez a rész üres.
           }
        }
        // Ha kipróbáltam az összes operátort és egyik se vezetett megoldásra, akkor visszalépés.
        // A visszalépés hatására eggyel feljebb a következő alapoperátor kerül sorra.
        return null;
     }
  }
```

## 8.8. A MélységiKereső osztály

Ebben a fejezetben a mélységi keresést ismertetjük két verzióban. Az első verzió használ körfigyelést, mint ahogy az eredeti mélységi keresés is használ. A második verzió nem, így sokkal gyorsabb. A második verziót csak akkor szabad használni, ha az állapottér gráfban nincs kör, mivel egyébként végtelen ciklusba eshet a kereső.

Mindkét verzió feltételezi, hogy a start csúcs még nem terminális. Ha ez nem feltételezhető, akkor a kiterjesztésre kiválasztott csúcsot is tesztelni kell, hogy terminális csúcs-e. Egyébként elegendő az új csúcsokat tesztelni, mint ahogy a lenti kód is teszi.
Jól ismert, hogy a mélységi keresést veremmel, a szélességi keresést sorral érdemes megvalósítani. Ennek megfelelően a nyílt csúcsokat veremben tároljuk. Így a verem tetején mindig a legnagyobb

mélységű nyílt csúcs lesz. Ez nagyban gyorsítja a keresést, hiszen nem kell keresgetni, melyik a legnagyobb mélységű nyílt csúcs.

Javasoljuk, hogy a lenti kód alapján az olvasó írja meg a szélességi keresést. Ez nem okozhat különösebb gondot. Kicsit nehezebb feladat ezek alapján megírni az optimális keresést. Ehhez a Csúcs osztályt is ki kell egészíteni a költséggel, illetve verem helyett rendezett listát (SortedList) érdemes használni, hogy könnyen megtalálható legyen a legkisebb költségű nyílt csúcs. Az optimális keresésnél külön oda kell figyelni, hogy egy csúcsba több út is vezethet és csak a legkisebb költségűt kell nyilvántartani. Még izgalmasabb az A algoritmus, ahol van visszaminősítés is.

## 8.8.1. Forráskód

```
/// <summary>
/// Mélységi keresést megvalósító gráfkereső osztály.
/// Ez a megvalósítása a mélységi keresésnek felteszi, hogy a start csúcs nem terminális csúcs.
/// A nyílt csúcsokat veremben tárolja.
/// </summary>
class MélységiKeresés : GráfKereső
{
   // Mélységi keresésnél érdemes a nyílt csúcsokat veremben tárolni,
   // mert így mindig a legnagyobb mélységű csúcs lesz a verem tetején.
   // Így nem kell külön keresni a legnagyobb mélységű nyílt csúcsot, amit ki kell terjeszteni.
   Stack<Csúcs> Nyilt; // Nílt csúcsok halmaza.
   List<Csúcs> Zárt; // Zárt csúcsok halmaza.
   bool körFigyelés; // Ha hamis, végtelen ciklusba eshet.
   public MélységiKeresés(Csúcs startCsúcs, bool körFigyelés) :
      base (startCsúcs)
   {
      Nyilt = new Stack<Csúcs>();
      Nyilt.Push(startCsúcs);    // kezdetben csak a start csúcs nyílt
      Zárt = new List<Csúcs>();  // kezdetben a zárt csúcsok halmaza üres
      this.körFigyelés = körFigyelés;
   }
   // A körfigyelés alapértelmezett értéke igaz.
   public MélységiKeresés(Csúcs startCsúcs) : this(startCsúcs, true) { }
   // A megoldás keresés itt indul.
   public override Csúcs Keresés()
   {
      // Ha nem kell körfigyelés, akkor sokkal gyorsabb az algoritmus.
      if (körFigyelés) return TerminálisCsúcsKeresés();
      return TerminálisCsúcsKeresésGyorsan();
   }
   private Csúcs TerminálisCsúcsKeresés()
   {
      // Amíg a nyílt csúcsok halmaza nem nem üres.
      while (Nyilt.Count != 0)
      {
         // Ez a legnagyobb mélységű nyílt csúcs.
         Csúcs C = Nyilt.Pop();
         // Ezt kiterjesztem.
         List<Csúcs> újCsucsok = C.Kiterjesztes();
         foreach (Csúcs D in újCsucsok)
         {
            // Ha megtaláltam a terminális csúcsot, akkor kész vagyok.
            if (D.TerminálisCsúcsE()) return D;
            // Csak azokat az új csúcsokat veszem fel a nyíltak közé,
            // amik nem szerepeltek még sem a zárt, sem a nyílt csúcsok halmazában.
            // A Contains a Csúcs osztályban megírt Equals metódust hívja.
            if (!Zárt.Contains(D) && !Nyilt.Contains(D)) Nyilt.Push(D);
```

```
            }
            // A kiterjesztett csúcsot átminősítem zárttá.
            Zárt.Add(C);
        }
        return null;
    }
    // Ezt csak akkor szabad használni, ha biztos, hogy az állapottér gráfban nincs kör!
    // Különben valószínűleg végtelen ciklust okoz.
    private Csúcs TerminálisCsúcsKeresésGyorsan()
    {
        while (Nyilt.Count != 0)
        {
            Csúcs C = Nyilt.Pop();
            List<Csúcs> ujCsucsok = C.Kiterjesztes();
            foreach (Csúcs D in ujCsucsok)
            {
                if (D.TerminálisCsúcsE()) return D;
                // Ha nincs kör, akkor felesleges megnézni, hogy D volt-e már nyíltak vagy a zártak közt.
                Nyilt.Push(D);
            }
            // Ha nincs kör, akkor felesleges C-t zárttá minősíteni.
        }
        return null;
    }
}
```

## 8.9. A FŐPROGRAM

Ebben a fejezetben láthatjuk, hogyan kell összecsiszolni az egyes részeket. Lényeges, hogy a start csúcs a kezdő állapotot kapja meg. A gráfkereső konstruktorába pedig a start csúcsot kell átadni. Ez mind a főprogram felelőssége.

Érdekes megnézni, hogy ugyanarra a start csúcsra más és más keresőket hívhatok és egy kicsit más eredményt is kapok. Érdemes eljátszani a konstansokkal is. Megfigyelhető, hogy míg a 10 mélységi korlát elegendő az éhes huszár példánál, addig a 3 szerzetes 3 kannibálnál már nem elég. Érdekes azt is megnézni, mekkora feladatnál kezd nehéz lenni a probléma. Az is szembe ötölhet, hogy nagyobb problémáknál a mélységi keresés kifuthat a memóriából, amikor a backtrack-nek még bőven elég a memória. Érdemes megnézni azt is, hogy ha a szuper operátorban átrendezem az alapoperátorok sorrendjét, akkor az nagyban befolyásolja a futási időt. Ha valaki kicsit bátrabb, a szuper operátort átírhatja úgy, hogy használjon heurisztikát.

Látható, hogy bármelyik gráfkereső algoritmust használjuk, akkor is a Keresés metódust kell meghívni. Az általa visszaadott csúcsra a megoldásKiírása metódust érdemes hívni.

### 8.9.1. FORRÁSKÓD

```
class Program
{
    static void Main(string[] args)
    {
        Csúcs startCsúcs;
        GráfKereső kereső;
        Console.WriteLine("Az éhes huszár problémát megoldjuk 4x4-es táblán.");
        startCsúcs = new Csúcs(new ÉhesHuszárÁllapot(4));

        Console.WriteLine("A kereső egy 10 mélységi korlátos és emlékezetes backtrack.");
        kereső = new BackTrack(startCsúcs, 10, true);
        kereső.megoldásKiírása(kereső.Keresés());
```

```
        Console.WriteLine("A kereső egy mélységi keresés körfigyeléssel.");
        kereső = new MélységiKeresés(startCsúcs, true);
        kereső.megoldásKiírása(kereső.Keresés());

        Console.WriteLine("A 3 szerzetes 3 kanibál problémát oldjuk meg.");
        startCsúcs = new Csúcs(new SzerzetesekÉsKannibálokÁllapot(3,3));

        Console.WriteLine("A kereső egy 15 mélységi korlátos és emlékezetes backtrack.");
        kereső = new BackTrack(startCsúcs, 15, true);
        kereső.megoldásKiírása(kereső.Keresés());

        Console.WriteLine("A kereső egy mélységi keresés körfigyeléssel.");
        kereső = new MélységiKeresés(startCsúcs, true);
        kereső.megoldásKiírása(kereső.Keresés());

        Console.ReadLine();
    }
}
```

# 9. Játékelméleti példaprogramok

Ebben a fejezetben a kétszemélyes játékokhoz kötődő algoritmusok közül a Negamax módszer használatát mutatjuk be egy teljes példán belül, ahol kétfajta játékban, a Tic Tac Toe és a Fejben 21 játékban lehet játszani a gép ellen. A gép lépéseit a Negamax módszerrel számolja ki. Be lehet állítani, hogy mennyire legyen intelligens a gép. Az intelligenciát természetesen csak szimuláljuk. Igazság szerint nem az intelligenciát, hanem az előretekintés lépésszámát. Ha ezt növeljük, akkor a gép több információ alapján tudja kiszámolni, hogy mit lépje, így jobb lépést választ és ezáltal intelligensebbnek tűnik.

Sajnos egy-két osztályt az előző fejezetből muszáj volt átírni. Ezeket egészben közöljük. Így ennek a fejezetnek és az előző fejezetnek van redundáns közös része. Ugyanakkor ez a fejezet az előző nélkül megállja a helyét, ami mindenképp előnyös.

## 9.1. Az AbsztraktÁllapot osztály

Az új AbsztraktÁllapot osztálynak csak egy új metódusa van, a GetHerisztika(). Ezt gyakran úgy programozzák le, hogy a heurisztika tudja, hogy melyik játékos (az éppen lépő, vagy az ellenfele) számára kell kiszámolni, hogy mennyire jó, illetve rossz az adott állás. Ez lenne a Minimax módszer alapja. Mivel ezt a fajta paraméteres heurisztikát nem fejlesztettük le, ezért csak a Negamax módszert tudtuk leprogramozni.

A Negamax módszer esetén az egyik játékosnak éppen annyira jó egy adott lépés, mint amennyire az ellenfél játékosnak jó, azaz az első érték (-1)-szerese a második érték. Így az AbsztraktÁllapot osztálynak nem kell tudnia, hogy ki lép, azt elegendő, ha a GetHerisztika() hívója tudja.

### 9.1.1. Forráskód

```
// Az AbsztraktÁllpot osztályt ki kellett bővíteni a GetHeurisztika metódussal.
// Egyébként megegyezik az előző fejezetben tárgyalt változattal.
abstract class AbsztraktÁllapot : ICloneable
{
   public abstract bool ÁllapotE();
   public abstract bool CélÁllapotE();
   public abstract int OperátorokSzáma();
   public abstract bool SzuperOperátor(int i);
   public virtual object Clone() { return MemberwiseClone(); }
   public override bool Equals(Object a) { return false; }
   public override int GetHashCode() { return base.GetHashCode(); }
   // Ez a metódus adja vissza, mennyire jó az adott állapot
   // Ez csak egy hook, felül kell írni, ha
   // kétszemélyes játékot vagy best-first algoritmus alkalmazunk,
   // vagy bármilyen más algoritmust, ami heurisztikán alapszik.
   // Más esetben, pl. backtrack esetén, nem kell felülírni.
   public virtual int GetHeurisztika() { return 0; }
}
```

## 9.2. A Tic Tac Toe játék állapot osztálya

A Tic Tac Toe játék annyiban érdekes, hogy nincs nyerő stratégiája, hiszen vannak olyan játékállás, ami döntetlennek felel meg. Ennek megfelelően a gép nem mindig nyer. Mindenesetre, ha elrontjuk

a saját első lépésünket és nem sarokba teszünk, akkor a gép nyerni fog, feltéve, ha elég intelligensre van állítva. Látni fogjuk, hogy ezzel a játékkal szemben a fejben 21-ben semmi esélyünk nem lesz a gép ellen, mert ott van nyerő stratégia.

Ha saját játékot szeretnénk fejleszteni, akkor ebből az osztályból, mint az AbsztraktÁllapot osztály gyermekéből kiindulhatunk. A többi osztályhoz nem is nagyon kell hozzányúlni. Legfeljebb a megjelenítést kell szépíteni.

Ennek az osztálynak is, mint minden konkrét állapot osztálynak a kétszemélyes játékok területén, a legfontosabb része a heurisztikát számító GetHeurisztika() függvény. Ettől függ, mennyire jó lépést fog javasolni a programunk. A másik tényező, amitől ez függ, hogy milyen mélységű játékfát generálunk le.

### 9.2.1. Forráskód

```
// A Tic Tac Toe játék állapot osztálya.
class TicTacToeÁllapot : AbsztraktÁllapot
{
   private static int N = 3; // 3 szor 3-mas tábla
   // NxN-es char mátrix
   // ' ': üres, 'X':első játékos jele, 'O': második játékos jele
   private char[,] tábla;
   private int countX, countO; // X-ek és O-k száma
   private bool nyert; // nyertes állapt-e
   private int üresekSzáma;
   public TicTacToeÁllapot()
   {
      tábla = new char[N, N];
      for (int i = 0; i < N; i++)
      {
         for (int j = 0; j < N; j++){ tábla[i, j] = ' ';}
      }
      countX = 0; // kezdetben egy X sincs
      countO = 0; // kezdetben egy O sincs
      nyert = false;
      üresekSzáma = N * N;
   }
   public override bool ÁllapotE() { return true; }
   public override bool CélÁllapotE() { return nyert || üresekSzáma == 0; }
   private bool preRak(int x, int y)
   {
      return x >= 0 && x < N && y >= 0 && y < N && tábla[x, y] == ' ';
   }
   private bool rak(int x, int y)
   {
      if (!preRak(x, y)) return false;
      char c;
      if (countX > countO) c = 'O'; else c = 'X';
      tábla[x, y] = c;
      bool régiNyert = nyert;
      nyert = // nem elég általános, csak N = 3-ra jó!
         (tábla[0, y] == c && tábla[1, y] == c && tábla[2, y] == c) ||
         (tábla[x, 0] == c && tábla[x, 1] == c && tábla[x, 2] == c);
      nyert = nyert || (x == y &&
         tábla[0, 0] == c && tábla[1, 1] == c && tábla[2, 2] == c);
      nyert = nyert || (x + y == N - 1 &&
         tábla[0, 2] == c && tábla[1, 1] == c && tábla[2, 0] == c);
      üresekSzáma--;
      if (c == 'X') countX++; else countO++;
      if (ÁllapotE()) return true;
```

```
    tábla[x, y] = ' '; // visszavonás
    nyert = régiNyert;
    üresekSzáma++;
    if (c == 'X') countX--; else countO--;
    return false;
}
public override int OperátorokSzáma() { return 9; }
public override bool SzuperOperátor(int i)
{
    switch (i)
    {
        case 0: return rak(0, 0);
        case 1: return rak(0, 1);
        case 2: return rak(0, 2);
        case 3: return rak(1, 0);
        case 4: return rak(1, 1);
        case 5: return rak(1, 2);
        case 6: return rak(2, 0);
        case 7: return rak(2, 1);
        case 8: return rak(2, 2);
        default: return false;
    }
}
// Ezt most felül kell írni, mert tömb típusú mezőnk is van.
// Egy szűk területre kell koncentrálni.
public override object Clone()
{
    TicTacToeÁllapot új = new TicTacToeÁllapot();
    új.tábla = (char[,])tábla.Clone();
    új.countX = countX;
    új.countO = countO;
    új.nyert = nyert;
    új.üresekSzáma = üresekSzáma;
    return új;
}
public override bool Equals(Object a)
{
    TicTacToeÁllapot másik = (TicTacToeÁllapot)a;
    return tábla.Equals(másik.tábla);
}
public override int GetHashCode() { return tábla.GetHashCode(); }
// Ez a metódus adja vissza, mennyire jó az adott állapot.
public override int GetHeurisztika()
{
    if (nyert) return 100 * (3 * N + 1);
    // szabad sorok, oszlopok, és átlok száma
    // szabad a sor, ha csak egy fajta szimbolum van benne
    return szabad('X');
}
// Ez egy kicsit általánosra sikerült, hiszen csak
// szabad('X') formában fogjuk hívni, habár hívható lenne
// szabad('0') formában is. Ez akkor lesz hasznos, ha a
// GetHeurisztika() függvényt át akarjuk írni.
private int szabad(char c)
{
    int count = 0;
    for (int i = 0; i < N; i++)
    {
        int sorX = 0, sorO = 0;
        int oszlopX = 0, oszlopO = 0;
        for (int j = 0; j < N; j++)
        {
            if (tábla[i, j] == 'X') sorX++;
```

```
            if (tábla[i, j] == 'O') sorO++;
            if (tábla[j, i] == 'X') oszlopX++;
            if (tábla[j, i] == 'O') oszlopO++;
        }
        if (c == 'X' && sorX > 0 && sorO == 0) count += sorX;
        if (c == 'O' && sorO > 0 && sorX == 0) count += sorO;
        if (c == 'X' && oszlopX > 0 && oszlopO == 0) count += oszlopX;
        if (c == 'O' && oszlopO > 0 && oszlopX == 0) count += oszlopO;
      }
      int átló1X = 0, átló1O = 0;
      int átló2X = 0, átló2O = 0;
      for (int i = 0; i < N; i++)
      {
        if (tábla[i, i] == 'X') átló1X++;
        if (tábla[i, i] == 'O') átló1O++;
        if (tábla[N - 1 - i, i] == 'X') átló2X++;
        if (tábla[N - 1 - i, i] == 'O') átló2O++;
      }
      if (c == 'X' && átló1X > 0 && átló1O == 0) count += átló1X;
      if (c == 'O' && átló1O > 0 && átló1X == 0) count += átló1O;
      if (c == 'X' && átló2X > 0 && átló2O == 0) count += átló2X;
      if (c == 'O' && átló2O > 0 && átló2X == 0) count += átló2O;
      return count;
    }
    public override string ToString()
    {
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < N; i++)
      {
        sb.Append('\n');
        for (int j = 0; j < N; j++)
        {
          sb.Append(tábla[i, j]);
          sb.Append(',');
        }
        sb.Remove(sb.Length - 1, 1);
      }
      return sb.ToString();
    }
}
```

## 9.3. A Fejben 21 játék állapot osztálya

A játék szabályai a következőek: A kezdő játékos egyet, kettőt vagy hármat mondhat. Ehhez a következő játékos egyet, kettőt vagy hármat adhat hozzá. Ez addig folytatódik, amíg valaki huszonegyet nem mond, és aki ezt mondja, az nyer.

A Fejben 21 játék azért nagyon érdekes, mert ennek a játéknak van nyerő stratégiája. Ha a kezdő játékos rendre a következő számokat mondja, akkor biztosan nyer: 1, 5, 9, 13, 17, 21.

Ha a kezdő játékos játékos egyet mond, akkor a következő játékos kettőt, hármat, vagy négyet mondhat. Bármit is mond, a kezdő játékos mondhat ötöt. Így a kezdő játékos mondhatja fenti számsorozatot és biztosan nyer. A lenti forráskódban úgy van kialakítva a heurisztika, hogy a gép minimális előretekintéssel is képes legyen megtalálni a fenti nyerő stratégiát.

### 9.3.1. Forráskód

```
// A fejben 21 játék állapot osztálya.
// Ezt sikerült általánosan megírni az N és a K mezők segítségével.
```

```
class Fejben21Állapot : AbsztraktÁllapot
{
   private static int N = 21; // fejben 21, 21-ig kell menni
   private static int K = 3; // max K-t lehet hozzáadni a számhoz
   private int szám;
   public Fejben21Állapot() { szám = 0; } // az első játékos 0-tól indul
   public override bool ÁllapotE() { return szám <= N; }
   public override bool CélÁllapotE() { return szám == N; }
   // Ennek a játéknak egyszerű nyerő stratégiája van.
   // A kezdő játékos nyer, ha 1-et mond, majd rendre:
   // 5, 9, 13, 17, 21.
   // Ha valaki 1-gyel kezd, akkor az ellenfél mondhat 2, 3, vagy 4-et.
   // Bármelyiket is mondja az ellenfél, a kezdő játékos mondhat 5-öt.
   // Így az 1, 5, 9, 13, 17, 21 sor tartható.
   // Hogy a rendszer megtalálja ezt a sorozatot, ezért a sorozat elemeire
   // 100-at ad a heurisztika, minden más értékre csak 1-et.
   public override int GetHeurisztika() { return szám % (K + 1) == 1 ? 100 : 1; }
   private bool preLép(int i) { return i >= 0 && i < K; }
   private bool lép(int i)
   {
      if (!preLép(i)) return false;
      i++; // ha i=0, akkor 1-et kell hozzáadni, stb..
      szám += i;
      if (ÁllapotE()) return true;
      szám -= i;
      return false;
   }
   // Itt szerencsére nem kellett belső switch-t írni.
   // Ez inkább kivételes eset.
   public override bool SzuperOperátor(int i) { return lép(i); }
   public override int OperátorokSzáma() { return K; }
   public override string ToString() { return szám.ToString(); }
}
```

## 9.4. A Csúcs és a JátékCsúcs osztály

Az alábbi két osztály segítségével építjük fel a játék fát. Az itt közölt Csúcs osztály majdnem teljesen megegyezik az előző fejezetben közölt csúcs osztállyal. A fő különbség abban rejlik, hogy ebben a változatban a csúcsban megjegyezzük, a szülőből melyik operátorral jutottunk ide. Ez azért fontos információ, mert amikor majd a Negamax módszer lépést ajánl, akkor egy csúcsot ad vissza, ahova szerinte érdemes lépni. A visszaadott csúcsból lekérdezve, hogy melyik operátorral jutottunk ide, máris tudjuk melyik operátort kell alkalmazni.

A JátékCsúcs tovább bővíti a Csúcs lehetőségeit egy értékkel, amit úgy neveztünk el, hogy mennyire jó. Ez csúcsban tárolt állapot heurisztikájával egyezik meg, illetve annak mínusz egyszerese. Ez attól függ, hogy kinek a szemszögéből építjük a játékfát.

### 9.4.1. Forráskód

```
// Az előző fejezetben ismertetett Csúcs osztályt bővítettük
// egy-két metódussal, amely a két személyes játékok megvalósításához kell.
class Csúcs
{
   AbsztraktÁllapot állapot;
   int mélység;
   Csúcs szülő;
   // A szülőkön túl a gyermekeket is tartalmazza a Csúcs osztály.
   List<Csúcs> gyermekek = new List<Csúcs>();
```

```
// Ez a mező tartalmazza, hogy melyik operátor segítségével jutottunk ebbe a csúcsba a szülő csúcsból.
// Ennek segítségével tudom megmondani, melyik az ajánlott lépés NegaMaxMódszer esetén.
int melyikOperátorralJutottamIde = -1; // ha -1, akkor még nincs beállítva
public Csúcs(AbsztraktÁllapot kezdőÁllapot)
{
   állapot = kezdőÁllapot;
   mélység = 0;
   szülő = null;
}
public Csúcs(Csúcs szülő)
{
   állapot = (AbsztraktÁllapot)szülő.állapot.Clone();
   mélység = szülő.mélység + 1;
   this.szülő = szülő;
}
// Erre a metódusra azért van szükség, hogy a kiterjesztés működjön a JátékCsúcsra is.
protected virtual Csúcs createGyermekCsúcs(Csúcs szülő){ return new Csúcs(szülő); }
public Csúcs GetSzülő() { return szülő; }
public int GetMélység() { return mélység; }
public bool TerminálisCsúcsE() { return állapot.CélÁllapotE(); }
public int OperátorokSzáma() { return állapot.OperátorokSzáma(); }
public bool SzuperOperátor(int i)
{
   // megjegyzem, melyik operátorral jutottam ebbe az állapotba
   melyikOperátorralJutottamIde = i;
   return állapot.SzuperOperátor(i);
}
public override bool Equals(Object obj)
{
   Csúcs cs = (Csúcs)obj;
   return állapot.Equals(cs.állapot);
}
public override int GetHashCode() { return állapot.GetHashCode(); }
public override String ToString() { return állapot.ToString(); }
// Alkalmazza az összes alkalmazható operátort.
// Visszaadja az így előálló új csúcsokat.
public List<Csúcs> Kiterjesztés()
{
   gyermekek = new List<Csúcs>();
   for (int i = 0; i < OperátorokSzáma(); i++)
   {
      // Új gyermek csúcsot készítek.
      // Ezzel a sorral nem működik a Kiterjesztés a JátékCsúcsban.
      // --- Csúcs újCsúcs = new Csúcs(this); ---
      // Ezért ezt használjuk:
      Csúcs újCsúcs = createGyermekCsúcs(this);
      // Kipróbálom az i.-dik alapoperátort. Alkalmazható?
      if (újCsúcs.SzuperOperátor(i))
      {
         // Ha igen, hozzáadom az újakhoz.
         gyermekek.Add(újCsúcs);
      }
   }
   return gyermekek;
}
// Visszaadja a csúcs heurisztikáját.
// Ha saját heurisztikát akarunk írni, akkor azt a saját állapot osztályunkba kell megírni.
public int GetHeurisztika() { return állapot.GetHeurisztika(); }
// Visszaadja melyik operátorral jutottunk ide.
// Ezzel az int értékkel kell majd meghívni a SzuperOperátor-t.
public int GetMelyikOperátorralJutottamIde() { return melyikOperátorralJutottamIde; }
// Nyomkövetéshez hasznos.
public void Kiir()
```

```
    {
        Console.WriteLine(this);
        foreach (Csúcs gyermek in gyermekek){ gyermek.Kiir(); }
    }
}
// A játék csúcs a csúcs osztály kibővítése egy heurisztika értékkel.
// Ezt a fajta csúcsot fel lehet használni a best first algoritmushoz is.
class JátékCsúcs : Csúcs
{
    int mennyireJó = -1; // mennyire jó, ha -1, akkor még nincs beállítva
    // Konstruktor:
    // A belső állapotot beállítja a start csúcsra.
    // A hívó felelősége, hogy a kezdő állapottal hívja meg.
    // A start csúcs mélysége 0, szülője nincs.
    public JátékCsúcs(AbsztraktÁllapot kezdőÁllapot) :
        base(kezdőÁllapot) { }
    // Egy új gyermek csúcsot készít.
    // Erre még meg kell hívni egy alkalmazható operátor is, csak azután lesz kész.
    public JátékCsúcs(Csúcs szülő) : base(szülő) { }
    // Erre a metódusra azért van szükség, hogy a kiterjesztés
    // működjön a JátékCsúcsra is.
    protected override Csúcs createGyermekCsúcs(Csúcs szülő)
    {
        return new JátékCsúcs(szülő);
    }
    // Visszaadja a csúcshoz tartozó heurisztikát.
    // Ez a csúcsban lévő állapot heurisztikája
    // megszorozva a paraméterben megkapott szor értékkel.
    // NegaMax esetén a szor általában 1, ha az a játékos lép,
    // akinek jó lépést keresünk, -1, ha az ellenfél lép.
    // Mivel minden állapothoz csak egy heurisztika van, ami nem
    // veszi figyelembe, hogy ki lép, ezért a MiniMax-nak is
    // úgy kell használni ezt a metódust, mint a NegaMax-nak.
    public int GetMennyireJó(int szor)
    {
        if (mennyireJó == -1) mennyireJó = GetHeurisztika() * szor;
        return mennyireJó;
    }
}
```

## 9.5. A Stratégia és a NegaMaxMódszer osztály

A Stratégia osztály minden lépés ajánló algoritmus őse. Egyetlen metódust tartalmaz, a MitLépjek metódust. Ez egy játékcsúcsot kap paraméterében. Ebből kiindulva megmondja, hova érdemes továbblépni. Ezt egy játékcsúcs formájában adja vissza. Ebből a csúcsból lekérdezhető, melyik operátorral jutottunk ide. Ez az ajánlott operátor.

A MitLépjek metódus null-t ad vissza, ha a paraméterében megkapott játékcsúcs zsákutca. Ez egyébként a játékokra nem jellemző, hiszen kevés olyan játék van, amiben van olyan eset, hogy már nem lehet lépni, mégsem nyert senki és nem is döntetlen.

Egyelőre csak egy lépés ajánló algoritmust ismertetünk. Ez a Negamax módszer. Ez lényegében megegyezik a Minimax módszerrel, eltekintve attól, hogy a Negamax esetén elvárás, hogy egy játék állás, azaz egy állapot, ugyanannyira legyen jó az egyik játékosnak, mint amilyen rossza a másiknak.

A Minimax és a Negamax esetén is meg kell adni, hogy hány lépéssel tekintsünk előre. Miután legeneráltuk a megfelelő  mélységű fát, azután a levélek értéket kapnak a heurisztika szerint. Ha a levél elembe az ellenfél lépet, akkor a heurisztikáját megszorozzuk mínusz eggyel. Ezután már az algoritmus megegyezik a Minimax módszernél megtanultakkal.

## 9.5.1. Forráskód

```
// Ebből kell leszármaztatni a lépés ajánló algoritmusokat, mint a NegaMax módszer.
abstract class Stratégia
{
   // Ha a start csúcs zsákutca, akkor null-t ad vissza.
   // Egyébként azt a csúcsot, amibe a stratégia szerint érdemes lépni.
   public abstract JátékCsúcs MitLépjek(JátékCsúcs start);
}
// Egy lépést ajánl valamely játékosnak a NegaMax módszer alapján.
// Ehhez előre tekint és a heurisztika meghatározása után megkeresi a legkedvezőbb utat a játék fában.
// Ha a játékfa levelei terminális csúcsok, akkor a legkedvezőbb út a nyerő stratégia lesz.
class NegaMaxMódszer : Stratégia
{
   int maxMélység; // Ennyi lépésre tekintünk előre.
   // Minél több lépésre tekintünk előre, annál intelligensebbnek tűnik a gép, hiszen jobb lépést választ.
   // Ezt a TicTacToe esetén figyelhetjük meg.
   // Ugyanakkor minél több lépést generálunk, annál lassabb lesz az algoritmus.
   // Ennek egy megoldása az Alfabéta-vágás, de ezt nem programoztuk le.
   public NegaMaxMódszer(int intelligencia){ maxMélység = intelligencia; }
   // Egy játékcsúcsot ad vissza.
   // Ennek a GetMelyikOperátorralJutottamIde() függvénye mondja meg,
   // melyik lépést ajánlja a NegaMax módszer.
   // Ha zsákutcában van, akkor null-t ad vissza.
   public override JátékCsúcs MitLépjek(JátékCsúcs start)
   {
      Csúcs levél = MaxLépés(start, start.GetMélység() + maxMélység);
      if (levél == start) return null;
      while (levél.GetSzülő() != start) { levél = levél.GetSzülő(); }
      //levél.Kiir(); // Nyomkövetés esetén hasznos segítség
      return (JátékCsúcs)levél;
   }
   // Feltételezi, hogy a start csúcsban a kérdező játékos kérdezi, mit lépjen.
   // A kérdező játékos legjobb lépését, tehát a legnagyobb heurisztikájú
   // csúcs felé vezető lépést választja.
   // A gyermek csúcsok heurisztikáját NegaMax módszerrel számoljuk.
   private JátékCsúcs MaxLépés(JátékCsúcs start, int maxMélység)
   {
      JátékCsúcs akt = start;
      if (akt.GetMélység() == maxMélység) { return akt; }
      if (akt.TerminálisCsúcsE()) { return akt; }
      List<Csúcs> gyermekek = null;
      gyermekek = akt.Kiterjesztés();
      if (gyermekek.Count == 0) { return akt; }
      JátékCsúcs elsőGyermek = (JátékCsúcs)gyermekek[0];
      JátékCsúcs leg = MinLépés(elsőGyermek, maxMélység);
      int h = leg.GetMennyireJó(+1);
      for (int i = 1; i < gyermekek.Count; i++)
      {
         JátékCsúcs gyermek = (JátékCsúcs)gyermekek[i];
         JátékCsúcs legE = MinLépés(gyermek, maxMélység);
         int hE = legE.GetMennyireJó(+1);
         if (hE > h) { h = hE; leg = legE; }
      }
      return leg;
   }
   // Felételezi, hogy a start csúcsban a a kérdező játékos ellenfele lép.
   // Az ellenfél játékos legjobb lépését választja, tehát azt,
   // ami a kérdező játékosnak a legrosszabb.
   // Egybe lehetne vonni a MaxLépéssel, hiszen csak 5 helyen más.
   // Ezek a sorokat megjelöltük.
   private JátékCsúcs MinLépés(JátékCsúcs start, int maxMélység)
```

```
    {
        JátékCsúcs akt = start;
        if (akt.GetMélység() == maxMélység) { return akt; }
        if (akt.TerminálisCsúcsE()) { return akt; }
        List<Csúcs> gyermekek = null;
        gyermekek = akt.Kiterjesztés();
        if (gyermekek.Count == 0) { return akt; }
        JátékCsúcs elsőGyermek = (JátékCsúcs)gyermekek[0];
        JátékCsúcs leg = MaxLépés(elsőGyermek, maxMélység); //más
        int h = leg.GetMennyireJó(-1);  // más
        for (int i = 1; i < gyermekek.Count; i++)
        {
            JátékCsúcs gyermek = (JátékCsúcs)gyermekek[i];
            JátékCsúcs legE = MaxLépés(gyermek, maxMélység); //más
            int hE = legE.GetMennyireJó(-1); //más
            if (hE < h) { h = hE; leg = legE; } //más
        }
        return leg;
    }
}
```

## 9.6. A JÁTÉK OSZTÁLY ÉS A FŐPROGRAM

Az itt ismertetett Játék osztály levezényel egy játékot. A játékot a gép kezdi, aztán bekéri az ember lépését. Itt nagy gond, hogy a Játék osztálynak nincs információja a választható lépésekről, így csak azok sorszámát írja ki. Így például a Fejben 21 játékban a 0.-dik operátor az jelenti, hogy a játékos egyet ad hozzá a jelenlegi számhoz.

Ezt úgy lehetne kiküszöbölni, ha bevezetnénk egy külön operátor osztályt, aminek ToString metódusa ki tudja írni az operátor nevét.

A Játék osztály másik hátránya, hogy ez egy konzolos megoldás. Ha valaki egy szép grafikus felületet szeretne, akkor ezt az osztályt át kell írnia.

A Játék osztályban lévő kód eredetileg a főprogram része volt. Onnan azért emeltük ki, hogy megmutassuk, elég általános, hogy TictacToe és Fejben21 játékot is játszhassunk vele.

A főprogram lényegében csak a Játék start metódusát hívja. A program első verziójában volt egy Játékos osztály is, ami lehetővé tette, hogy több mint két személyes játékokat is leprogramozzunk. Ha erre lenne szükség, akkor javasoljuk a Játékos osztály bevezetését.

### 9.6.1. FORRÁSKÓD

```
// Egy játékot vezényel le.
// Ez egybevonható a főprogramból, hiszen abból emeltük ki.
class Játék
{
    AbsztraktÁllapot startÁllapot;
    Stratégia strat;
    public Játék(AbsztraktÁllapot startÁllapot, Stratégia strat)
    {
        this.startÁllapot = startÁllapot;
        this.strat = strat;
    }
    public void start()
    {
        JátékCsúcs akt = new JátékCsúcs(startÁllapot);
        Console.WriteLine("Játszhat a gép ellen!");
        while (!akt.TerminálisCsúcsE())
```

```
        {
            // Ellenfél lépése.
            Console.WriteLine("Jelenlegi állás: {0}", akt);
            akt = strat.MitLépjek(akt);
            int i = akt.GetMelyikOperátorralJutottamIde();
            Console.WriteLine("A gép ezt az operátort választotta: {0}", i);
            // Nyertes állás?
            if (akt.TerminálisCsúcsE()) break;
            // Saját lépésem.
            bool b = false;
            while (!b)
            {
                Console.WriteLine("Jelenlegi állás: {0}", akt);
                Console.WriteLine("Melyik operátort választja? (0,..,{0}): ", akt.OperátorokSzáma() - 1);
                int k = 0;
                try
                {
                    k = int.Parse(Console.ReadLine());
                }
                catch (Exception e)
                {
                    Console.WriteLine("Hiba: {0}", e.ToString());
                    Console.WriteLine("Érvénytelen lépés. Újra!");
                    continue;
                }
                b = akt.SzuperOperátor(k);
                if (!b) Console.WriteLine("Ez az operátor nem alkalmazható. Újra!");
            }
        }
        Console.WriteLine("Jelenlegi állás: {0}", akt);
        Console.WriteLine("Aki utoljára lépett, az nyert, vagy döntetlen.");
    }
}
// Főprogram.
class Program
{
    static void Main(string[] args)
    {
        Stratégia strat = new NegaMaxMódszer(5); // 5 mélységbe tekint előre
        AbsztraktÁllapot startFejben21Állapot = new Fejben21Állapot();
        Játék fejben21 = new Játék(startFejben21Állapot, strat);
        Console.WriteLine("A Fejben 21 játék kezdetét veszi!");
        fejben21.start();
        Console.ReadLine();
        AbsztraktÁllapot startTTTÁllapot = new TicTacToeÁllapot();
        Játék tictactoe = new Játék(startTTTÁllapot, strat);
        Console.WriteLine("A Tic Tac Toe játék kezdődik!");
        tictactoe.start();
        Console.ReadLine();
    }
}
```

## 9.7. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A jegyzet egy következő verziójában tervezzük a két példaprogramos fejezet egybevonását és néhány tervezési minta alkalmazását. Például az ebben a fejezetben használt AbsztraktÁllapot osztály díszítő minta segítségével levezethető az előző fejezet hasonló nevű osztályából. Ugyanez igaz az itteni Csúcs és az előző Csúcs osztályokra is.

# IRODALOMJEGYZÉK


(1) Stuart J. Russell, Peter Norvig: Mesterséges intelligencia modern megközelítésben
Panem – Prentice Hall, 2000

(2) Pásztorné Varga Katalin, Várterész Magda: A matematikai logika alkalmazásszemléletű tárgyalása
Panem, 2003

(3) Dr. Várterész Magda: Mesterséges intelligencia
http://www.inf.unideb.hu/~varteres/mi/tartalom.htm