SZAKDOLGOZAT

Nagy Gábor

Debrecen

2008

Debreceni Egyetem

Informatikai Kar

Megoldáskereső Módszerek

Témavezető:
Dr. Halász Gábor
Egyetemi docens

Készítette:
Nagy Gábor
Programtervező Informatikus

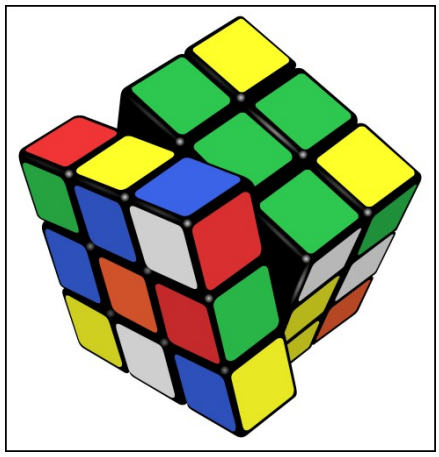
Debrecen
2008

Tartalomjegyzék

1. Bevezetés

Számos módszer látott már napvilágot a Rubik-kocka megjelenése óta annak megoldására. Ezek közül a szakdolgozathoz írt program készítésekor egy emberek által is felfoghatót választottam, melyet kezdetben a gyorsasági versenyeken is előszeretettel használtak hatékonyságára való tekintettel. Ezt a módszert a későbbiekben ismertetni fogom, de előbb nézzük meg, hogyan is keletkezett a kocka és milyen problémát állít elénk annak megoldása.

1.1. A Rubik-kocka rövid története



1. ábra: Rubik-kocka eltekert állapotban

A kockát Rubik Ernő szobrász, építészmérnök és egyetemi tanár találta fel 1974-ben, a kocka szabadalmának beadása 1975-ben történt. Eredetileg a 2x2x2-es kockát szerette volna megalkotni, méghozzá térbeli mozgások szemléltetésére, azonban a gumigyűrűs és mágneses megoldásai is kudarcot vallottak. A gyűrűk nem bizonyultak elég tartósnak, a mágnesek pedig nem tartották össze kellőképpen a szerkezetet. Ekkor tervezte meg a 3x3x3-as kockát, melynek elemeit úgy faragta ki, hogy azok alakjuknál fogva egymást tartsák össze és egyúttal minden oldala forgatható legyen az oldal középpontja körül. Később az oldalakat beszínezte az azóta is leggyakrabban használt sárga, kék, piros, zöld, narancs és fehér színekkel, hogy ezáltal jobban nyomon lehessen követni az elemek mozgását. A színösszeállítás jól megfontolt, hiszen a párhuzamos oldalak színei a sárga komponensben különböznek, így a színek könnyen megkülönböztethetőek. Csak ekkor fedezte fel, hogy nemcsak oktatási célokra használható a kocka, hanem nagyszerű fejtörő is, és ezáltal eladható. A sors fintora, hogy bár oktatási segédeszközként sohasem fogadták el, azonban minden idők legtöbb példányban eladott játéka lett világszerte. Fénykorát a nyolcvanas években élte, de a mai napig is nagy népszerűségnek örvend. Ezt mi sem mutatja jobban, mint a minden évben megrendezett világversenyek, azonban nemcsak a játék szerelmesei foglalkoznak a kockával, hanem komoly matematikusok is kihívást látnak benne, mint ahogy ezt később látni fogjuk. [1] [2]

1.2. Rubik-kocka számokban

A klasszikus 3x3x3-as Rubik-kockának 6 középső eleme, 12 éle és 8 sarka van. A középső elemek helyzete nem változik meg soha, csak egy helyben forognak. A nyolc sarokelem 8!--féle elrendezésben foglalhat helyet. Ezek közül hét bármelyik irányba nézhet, míg a nyolcadik irányát az előző hét kocka határozza meg, így kapjuk a 3^7-féle lehetőséget. Az élek 12!/2-féle elrendezésben állhatnak, mert a sarkok helyzete részben meghatározza az élek helyzetét. Végül az élek közül tizenegy bármelyik irányban állhat, míg a tizenkettedik irányát az előző tizenegy határozza meg, s eszerint az élek 2^{11}–féleképpen állhatnak a helyükön. Ezekből adódóan a kockának $8!\cdot3^7\cdot12!\cdot2^{10}$-féle állapota létezik, amely megközelítőleg $4.33\cdot10^{19}$, egészen pontosan 43 252 003 274 489 856 000. Ez kimondva negyvenháromtrillió--kétszázötvenkétbilliárd-hárombillió-kétszázhetvennégymilliárd--négyszáznyolcvankilencmillió-nyolcszázötvenhatezer, ha szemléltetésképpen ennyi 57 milliméteres Rubik-kockát egymás után rakunk, akkor nagyjából 261 fényév hosszú sort kapunk. Fontos viszont, hogy ez a szám a kirakott kockából csak tekerésekkel, szétszedés nélkül elérhető állapotok száma. Továbbá létezik a kockának olyan változata is, melynél a hat középső elemen elhelyezett jelzés miatt azok irányára is figyelni kell, s így a lehetséges permutációk száma $4.33 \cdot 10^{19} \cdot 4^6/2$. [1] [2]

1.3. A kocka algoritmusai

A Rubik-kocka megoldására alkalmas algoritmusok matematikai értelemben vett algoritmusok, azaz jól definiált lépések sorozata, melyekkel egy adott kezdőállapotból jól meghatározott állapotokon keresztül elérhetünk egy kívánt állapotot. Itt a kívánt állapot nemcsak célállapot lehet, hanem egy kedvező állapot is, mely közelebb juttat minket a végső cél eléréséhez. Minden módszernek megvannak a maga algoritmusai, és tudni lehet azok hatását a kockára, valamint azt is, hogy azokat mikor érdemes alkalmazni, hogy általuk közelebb kerüljünk a megoldáshoz. A legtöbb ilyen algoritmust úgy tervezték, hogy a kockának csak egy adott részét változtassák meg anélkül, hogy az addig már kirakott részeket elrontanák. Ilyen például két szomszédos él irányának megfordítása. Vannak azonban olyan algoritmusok is, melyek nem csak a kívánt részt változtatják meg, hanem vannak bizonyos mellékhatásai is. Például két szomszédos él cseréje permutálja a közös oldal sarkait is. Ezek általában egyszerűbb algoritmusok, mint a mellékhatás nélküli társaik és a megoldás kezdeti

fázisában alkalmazhatóak, amikor a mellékhatások még nem számítanak. Ahogy haladunk a megoldás felé, az algoritmusok egyre bonyolultabbá válnak, hiszen egyre nagyobb részre kell vigyázniuk. Általánosan igaz, hogy minél kevesebb algoritmust kell memorizálni egy módszer elsajátításához, annál kevésbé hatékony a módszer. A rétegenként haladó módszerhez (layer by layer method) nagyjából 15-20 algoritmust kell memorizálni és átlagosan 100-120 tekeréssel oldhatjuk meg vele a kockát. A bonyolultabb módszerekhez, mint Jessica Fridrich vagy Lars Petrus módszeréhez akár több mint 100 ilyen algoritmus ismerete szükséges, cserébe viszont hatékonyabban működnek. Isten módszerének nevezik azt a ma még nem létező módszert, mely képes megoldani a kockát, az ahhoz szükséges minimális számú tekerés segítségével. A kocka feltalálása óta kutatják, hogy maximum hány lépésből oldható meg a kocka bármelyik állapota. Két matematikus, Daniel Kunkle és Gene Cooperman 2007-ben bizonyították, hogy bármely állapot megoldható maximum 26 tekerés segítségével, amennyiben a 180^0-os tekerést egy tekerésnek számítjuk. Ehhez egy szuperszámítógépet vettek igénybe, amely hatalmas számítási kapacitásával is 63 órán keresztül futott. A Northeastern University kutatási módszereihez hasonlóan indult ki a Stanford University matematikusa, Tomas Rokicki is: a kocka lehetséges forgatási lépéseit halmazokba kötötte össze, majd megvizsgálta, hogy azok közül melyek azonosak. A 2 milliárd halmaz egyenként 20 milliárd lépése közül a lehetséges konfigurációk java megegyezik, így azokat el is távolította a forgatási lehetőségek közül. Rokicki a számolásokhoz egy 1.6 gigahertzes processzorral és 8 gigabájt memóriával felszerelt PC-t használt, amely a huszonöt lépéses variációkat 1500 órán keresztül számolta. A matematikus itt nem állt meg, ugyanis jelenleg már azon dolgozik, hogy a maximális forgatások számát miként lehet huszonnégyre vagy kevesebbre csökkenteni, ehhez előzetes becslései szerint több hónapon keresztül számoltatnia kell számítógépét. Ha ezt megoldotta, következő célja annak bebizonyítása, hogy huszonhárom lépésből is ki lehet rakni a Rubik-kockát. Az Isten számának is nevezett értéket 20-ra becsülik, de ennek bizonyításához szükséges eszközök még nem állnak rendelkezésre és valószínűleg nem is fognak egyhamar. Viszont biztató jel, hogy senki sem talált még olyan elrendezést, melynek megoldásához több mint 20 lépésre lenne szükség. [1] [2]

2. Tárgyalás

A továbbiakban szót ejtünk a mesterséges intelligencia azon elemeiről, melyek ismerete feltétlenül szükséges egy probléma kereséssel történő megoldásához, valamint megnézzük, hogy hogyan is valósul meg a keresés folyamata, milyen keresési stratégiákat alkalmaztak és alkalmaznak napjainkban. A szakdolgozathoz megírt programról is szó esik a dolgozat végén, kitérve az abban alkalmazott keresési stratégiára, a kocka megoldásához használt layer by layer metódusra, az arra épülő kiértékelő függvényre, és a grafikai rész megvalósítására is.

2.1. Ágensek

A világ minden olyan objektuma nevezhető ágensnek, mely szenzoraival érzékeli az őt körülvevő környezetet, s azt beavatkozásra alkalmas eszközeivel meg is tudja változtatni. Ebből a szempontból ágensnek tekinthető egy ember, egy bogár, vagy akár egy robot. A különbség csupán annyi, hogy míg az embernek a füle, szeme és bőre van, addig a robot mesterséges érzékelőkkel rendelkezik, mint például nyomásérzékelőkkel és kamerákkal. Azonban egy ágensnek nem kell kézzel foghatónak lennie. Egy programot is lehet ágensnek tekinteni, melynek környezete bitekből áll, valamint érzékelésén és beavatkozásán ezen bitek kiolvasását és manipulációját értjük. Az ágenseket több csoportba sorolhatjuk, ez a dolgozat ezek közül a célorientált ágensekkel, azon belül is a problémamegoldó ágensekkel foglalkozik. [3]

2.1.1. Célorientált ágensek

Nagyban segíti az ágenst a döntéshozatalban, ha kezdettől fogva egy cél lebeg a szeme előtt, s mindenkor úgy hozza meg a döntéseit, hogy beavatkozása után ehhez a célhoz minél közelebb kerüljön. Ez akkor a legegyszerűbb, ha egyetlen beavatkozással elérheti célját. Ez az eset azonban nagyon ritka, sokszor ugyanis lépésről lépésre kell haladnia, minden új állapotnál eldöntve, hogy mit is tegyen. A keresés és a tervkészítés a mesterséges intelligencia azon területei, melyek az ágens céljait elérő cselekvéssorozat megtalálásával foglalkoznak. [3]

2.1.2. Problémamegoldó ágensek

Egy ilyen ágens megalkotásakor tehát az első feladatunk az ágens által megoldani kívánt probléma pontos megfogalmazása, annak leírása, valamint a problémára megoldást jelentő cél kitűzése. A probléma megfogalmazásához többféle reprezentációs eszköz is rendelkezésünkre áll. Leírhatjuk az elsőrendű logika, probléma-redukció, vagy állapottér reprezentáció segítségével. Ha az ágens ismeri a problémát, akkor képes a rendelkezésére álló cselekvések egy olyan sorozatát összeállítani, mellyel célját eléri. Ezt a folyamatot keresésnek nevezzük, a keresés végtermékét pedig megoldásnak. Mivel a szakdolgozathoz írt programban állapottér reprezentáció segítségével írtam le a problémát, nézzük meg annak elemeit. [3]

2.2. Állapottér-reprezentáció

Legyen p egy probléma. Vegyük sorra p világának azon tulajdonságait, amelyek a probléma megoldásának szempontjából fontosak, a többi figyelmen kívül hagyható. Feltételezhető hogy legalább egy ilyet találunk, és véges sok ilyen tulajdonság létezik. Ezek a jellemzők mind p világának valamilyen tulajdonságát írják le (például ledöntött bábúk száma), s együttesen határozzák meg a világnak egy állapotát. Ha a világnak n db ilyen jellemzője létezik, akkor a világ egy állapotát egy ($j_1, j_2, ..., j_n$) érték n-essel írhatjuk le.

Legyen H_i az i-edik jellemző által felvehető értékek halmaza, ekkor p állapotai a $H_1 \times H_2 \times ... \times H_n$ halmazból kerülnek ki, s ezt a halmazt jelöljük A-val és nevezzük állapottérnek. Előfordulhat azonban, hogy ennek a Descartes szorzatnak az A halmaz csak egy részhalmaza, hiszen vannak bizonyos jellemzők, amelyek nem léphetnek fel egyszerre. A kényszerfeltétel az, ami kijelöli a Descartes szorzat azon részét, ami beletartozik az állapottérbe. Ha a egy valódi állapot, akkor a kényszerfeltétel(a) igaz.

$$A = \{ a \mid a \in H_1 \times ... \times H_n \text{ és kényszerfeltétel(a)} \}$$

Azt a kitüntetett szerepű állapotot, amelyben a probléma megoldásának kezdésekor vagyunk, kezdőállapotnak hívjuk. A célállapotok halmazát C-vel jelöljük, egy elemét c-vel. Az ágens célja egy, vagy az összes ilyen célállapot megtalálása is lehet. Ez a C halmaz megadható explicit módon felsorolással, vagy célfeltétel megadásával:

$$C = \{ a \mid a \in A \text{ és célfeltétel(a)} \}$$

A célfeltétel(a) igaz, amennyiben az a állapot célállapot.

Ahhoz azonban, hogy el is érhessünk egy ilyen állapotot, szükség van állapottérből állapottérbe leképező függvényekre, melyekkel megvalósulhat az ágens beavatkozása, azaz az állapotok megváltoztatása. Egy ilyen tulajdonságú o: A→A függvényt operátornak nevezünk, az operátorok halmazát O-val jelöljük. Mivel nem minden állapotra alkalmazható minden operátor, ezért meg kell adni azok értelmezési tartományát egy operátor-alkalmazási előfeltétel segítségével:

$$dom(o) = \{ a \mid a \in A, \text{előfeltétel}_o(a) \}$$

Az előfeltétel$_o$(a) igaz, amennyiben o alkalmazható, azaz o(a) = a' ∈ A.

A p probléma állapottér-reprezentációját megadtuk, ha felírtuk a p = < A, k, C, O > négyest.

A Rubik-kocka állapottér reprezentációja:

A kocka egy állapotát 54 szám határozza meg, ezek a számok a [0,5] intervallumból kerülhetnek ki, s a számok egy-egy színt szimbolizálnak.

$$H = \{ (0, 0, \ldots, 0), (1, 0, \ldots, 0), \ldots, (5, 5, \ldots, 5) \}$$

A≠H, mert H nem minden eleme valódi állapot.

$$A = \{ a \mid a \in H_1 \times \ldots \times H_n \text{ és kényszerfeltétel}_{Rubik}(a) \}$$

A kényszerfeltétel$_{Rubik}$(a) igaz, ha minden színből pontosan 9 szerepel a kockán, létezik az összes megfelelő szín-összeállítású él és sarkok, valamint a 8 sarokból és 12 élből irányuk szerint megkülönböztetve páros számú azonos szerepel, ellenkező esetben a kocka megoldhatatlan.

Kezdőállapot az A halmaz bármelyik eleme lehet. A C célállapotok halmaza egyelemű, a kocka célállapotban van, ha minden oldal minden lapja egyforma színű. Az O operátorok halmaza egyetlen egyparaméterű operátort tartalmaz. Ennek a Teker(szám) operátornak a paramétere egy 1 és 12 közötti szám, a 12 lehetséges tekerési iránynak megfelelően. A Rubik--kocka esetében operátor-alkalmazási előfeltételt nem kell vizsgálni, hiszen a kocka bármely állapotára alkalmazható valamennyi operátor. Az így leírt < A, kezdő, C, O> a probléma egy lehetséges állapottér reprezentációja. [4]

2.3. Keresés

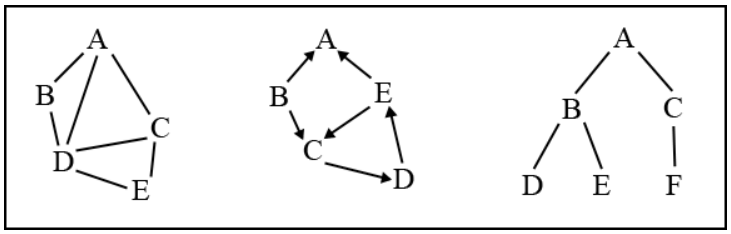
Miután definiáltuk a problémát és az elérni kívánt célt is, hozzáfoghatunk a probléma megoldását jelentő cselekvéssorozat létrehozásához. Ehhez az állapottérben történő keresésre van szükség. Mivel a legtöbb probléma elég bonyolult ahhoz, hogy ne tudjuk annak egész állapotterét letárolni, ezért olyan adatszerkezetekre van szükség, melyekben mindig csak az állapottér addig bejárt részét tároljuk, és minden körben bővítjük azt.

2.3.1. Gráfok és fák

A későbbiekben tárgyalt keresési technikák nagyon általánosak és szinte bármilyen probléma esetében alkalmazhatóak. Így tehát szükség van valamilyen absztrakt módszerre, amellyel úgy reprezentálhatjuk a problémákat, hogy az általános célú algoritmusokat anélkül használhassuk, hogy minden egyes új feladatnál új módszert kellene kidolgoznunk. Erre szolgálnak a gráfok.

A gráfok csomópontokból vagy csúcsokból és irányított vagy irányítatlan élekből állnak. Egy él mindig két csomópont között található, amennyiben nem hurokélről van szó. A hurokél ugyanabban a csomópontban végződik, melyből kiindul. A gyermek- vagy utódcsomópont alatt olyan csomópontot értünk, amely az aktuális csomópontból egyetlen él mentén elérhető. Az út élek egymáshoz csatlakozó sorozata, amely egy csomópontot legfeljebb egyszer tartalmaz. A gráf kört tartalmaz, ha van olyan útja, melynek kiindulási és végpontja ugyanaz a csomópont. A gráfban lehet továbbá húr, ha van olyan csomópontja, amely több úton is elérhető.

A gráfok egy speciális fajtája a fa, mely hurok és körmentes, valamint általában van egy kitüntetett gyökércsomópontja. A fának minden éle irányított, s amennyiben egy gráf egyértelműen fa, annál jelezni ezt nem szokás. A gyermekkel nem rendelkező csomópontokat levélelemeknek nevezzük. Ha egy A csomópontból egy B csomópont nem egy él mentén érhető el, akkor A B-nek az őse és B A-nak a leszármazottja. Általában a keresési problémákat fákkal reprezentáljuk, de vannak olyan problémák is, melyekkel csak gráfjuk megvágása után tehetjük ezt meg. [5]



4. ábra: egy gráf, egy irányított gráf és egy fa

2.3.2. Cselekvéssorozatok létrehozása

A keresés folyamatát úgy tekintjük, mintha egy keresési fát építenénk, amit ráillesztenénk az állapottérre. Ez a kört tartalmazó gráfok esetében is igaz, ekkor viszont a keresőnek a szóban forgó éleket körfigyeléssel el kell metszenie, hogy a gráfot fává alakítsa. A keresési fa gyökere a kiinduló állapotnak megfelelő keresési csomópont. A fa levélcsomópontjai vagy még nem lettek kiterjesztve, vagy kiterjesztéskor nem találtunk az állapotára alkalmazható operátort és így a kiterjesztés az állapotok egy üres halmazát eredményezte. A keresési algoritmus minden körben a még ki nem terjesztett csomópontjai közül választ ki egyet kiterjesztésre. Kiterjesztés alatt azt a folyamatot értjük, melynek során az aktuális állapotra az összes operátor-alkalmazási előfeltételnek megfelelő operátort alkalmazzuk, ezzel generálva az állapotoknak egy új halmazát. Ha ezen csomópontok között nincs célállapot, akkor újabb kiterjesztésre van szükség. Ezt addig folytatjuk, míg megoldásra nem lelünk, vagy míg el nem fogynak a kiterjeszthető csomópontok. A keresés során meghatározó, hogy a kiterjesztés közben keletkező állapotok melyikét választjuk meg aktuális állapotnak, azaz a lehetséges utak közül melyiket választjuk. Ezt a keresési stratégiánk fogja meghatározni. [3]

2.3.3 Keresési fák adatszerkezete

A keresési fákat tehát csomópontokkal valósítjuk meg. A csomópont egy adatnyilvántartásra használt adatszerkezet, amit egy adott problémapéldány egy adott algoritmus által generált keresési fájának leírására használunk. Fontos, hogy két különböző csomópont tartalmazhatja a világnak ugyanazt a konfigurációját leíró állapotát is, amennyiben ez a csomópont több út mentén is elérhető.

Egy csomópontnak sokféle összetevője lehet, de a legáltalánosabb adattagjai a következők:

- Az állapottér egy állapota, melyet a csomópont reprezentál
- Azon csomópontra mutató mutató, melynek kiterjesztése során ezt a csomópontot generáltuk, azaz a szülőcsomópontra mutató mutató
- A csomópontot generáló operátor
- A csomópont mélysége, ami megadja a gyökércsomóponttól való távolságot
- A gyökércsomóponttól eddig a csomópontig számított útköltség

Ezen adattagok meghatározását mindig a kiterjesztést végző függvény végzi el. De nem csak ezekre az információkra lehet szükség, hanem nyilván kell tartani a kiterjesztésre váró csomópontokat is. Ennek legkézenfekvőbb módja egy sor használata, mert a keresési stratégiától függően az új csomópontokat beszúrhatjuk a sor elejére, végére, vagy akár valamilyen szempont szerint rendezhetjük az egész sort. Így az egyszerűbb stratégiáknál nem kell minden körben végignézni az összes kiterjesztésre váró csomópontot, hogy kiválasszunk egyet közülük. Bizonyos esetekben a körfigyelés miatt szükség lehet a már kiterjesztett csomópontok tárolására is. [3]

2.4. Keresési stratégiák

Mint azt már említettem, a keresési stratégia határozza meg, hogy a kiterjesztésre váró csomópontok közül melyiket terjesszük ki legközelebb. A stratégiákat a következő szempontok szerint értékelhetjük:

- Teljesség (completeness): ha létezik megoldás, akkor azt a kereső garantáltan megtalálja.
- Optimalitás (optimality): ha több különböző megoldás létezik, akkor a kereső megtalálja azok közül a legjobb minőségűt (legrövidebbet, legolcsóbbat)
- Időigény (time complexity): mennyi ideig tart a megoldás keresése, valamint minden esetben belátható időn belül lefut-e a keresés?
- Tárigény (space complexity): a kereséshez szükséges memória mennyisége

Mivel a kereső algoritmusok általánosak és úgy íródtak, hogy változtatás nélkül oldják meg a különböző problémákat, ezért a reverzíbilis operátorokkal rendelkező problémák miatt a körfigyelés szinte mindegyiknek a részét képezi. A körfigyelés feladata elkerülni a már korábban egy másik úton megtalált és kifejtett állapotok ismételt kifejtéséből adódó időpocsékolást. Ezeket az ismétlődő állapotokat az alábbi módokon kerülhetjük el:

- Csak azt figyeljük, hogy az aktuális csomópont állapotára ne alkalmazzuk az őt generáló operátorral ellentétes hatású operátort, s így nem hozunk létre az aktuális csomópont szülőcsomópontjának állapotával megegyező állapotú csomópontot.
- A kiterjesztés során ne alkalmazzunk olyan operátort az aktuális csomópont állapotára, melynek hatására a keletkező csomópont állapota megegyezik ezen csomópont valamelyik ősének állapotával.
- A kiterjesztés során ne generáljunk olyan csomópontot, melynek állapota korábban már előfordult. Ehhez viszont az összes eddig vizsgált és generált csomópontot el kell tárolnunk.

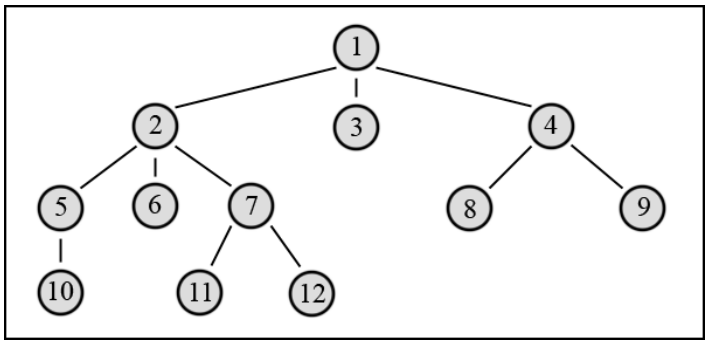
Ezeknek a vizsgálatoknak azonban nemcsak jótékony hatása, hanem tár- és időigénye is van. Minél több kört tartalmaz az adott problémapéldány, annál jobban megéri esetében figyelni a köröket. Viszont az sem elhanyagolható, hogy a körfigyelés hatására a keresés nem futhat végtelen ciklusba. Ennek oka, hogy a körfigyelés a végtelen keresési fákat véges méretűre vágja, azaz a kereső a keresési fának csak az állapottér gráfját kifeszítő részét fogja generálni. [3] [5]

2.4.1. Nem informált keresők

A keresési algoritmusokat megkülönböztetjük abból a szempontból, hogy el vannak-e látva valamilyen megoldás megtalálását segítő tudással. Azokat a keresőket, melyek az aktuális állapotról csak annyit tudnak, hogy célállapot-e, nem informált (noninformed search) vagy vak (blind search) keresési stratégiáknak nevezzük. Ha a kereső további következtetéseket tud levonni egy-egy állapotról, akkor informált (informed search) vagy heurisztikus (heuristic search) keresési stratégiákról beszélünk. Vegyük most sorra a legismertebb nem informált keresési stratégiákat. [3] [5]

2.4.1.1. Szélességi kereső

A szélességi kereső (breadth-first search) a legegyszerűbb informálatlan keresési stratégiák közé tartozik. Először a gyökércsomópontot terjeszti ki, majd innentől kezdve a kiterjesztések során generált új csomópontokat a várakozási sor végére szúrja be. Ennek hatására szintenként halad, azaz először mindig a kisebb mélységűeket választja ki. Például, ha éppen egy ötmélységű csomópontot választott ki, akkor biztos, hogy ötnél kisebb mélységű csomópont már nem várakozik. A stratégia tehát meglehetősen szisztematikus, ahogy az a 3. ábrán is látszik. Feltesszük, hogy az ábrán látható fán a 12-es csomópont tartalmazza a célállapotot, azaz egy legrosszabb esetet látunk.



3. ábra: célteszt és kiterjesztés sorrendje szélességi keresés során

Amennyiben a feladatnak létezik megoldása, úgy a szélességi kereső garantáltan meg is találja azt, több megoldás esetén pedig először biztosan a legrövidebb megoldást találja meg. Ez előnyt jelent a mélységi keresővel szemben az olyan feladatoknál, ahol a legrövidebb megoldás megkeresése a cél, hiszen a szélességi kereső az első célállapot elérése után befejezheti a keresést. A szélességi kereső tehát teljes, és ha minden operátor alkalmazási költsége egyenlő, akkor optimális is.

Hátránya, hogy bonyolultabb feladatoknál a számítási idő és a tárigény is nagyon megnő. Ha feltesszük, hogy minden állapotra b db operátor alkalmazható, azaz minden állapot kiterjesztésekor b db új állapotot kapunk, akkor b elágazási tényező (branching factor) és m mélységű legsekélyebb megoldás mellett a legrosszabb esetben $1 + b + b^2 + b^3 + \ldots + b^m$ db csomópontot kell kiterjeszteni, hogy megtaláljuk az első megoldást. Ehhez hozzátartozik az is, hogy a legkedvezőbb esetben ez a szám jóval kisebb. A 4. ábra feltételezi, hogy másodpercenként 1000 csomópont céltesztjét és kiterjesztését lehet elvégezni, illetve egy csomópont letárolásához 100 bájtra van szükség. Számos feladvány jellegű probléma felel meg ezeknek a feltételezéseknek, ha azokat modern személyi számítógépen vagy

munkaállomáson futtatjuk. Az ábra alapján látszik, hogy a szélességi keresés esetén a tárigény nagyobb problémát jelent az időigénynél. A legtöbb ember türelmesen ki tud várni 31 órát, hogy egy 8-as mélységű keresés lefusson, de már messze nincs annyi embernek a kereséshez szükséges 11 Gb-nyi memóriája.

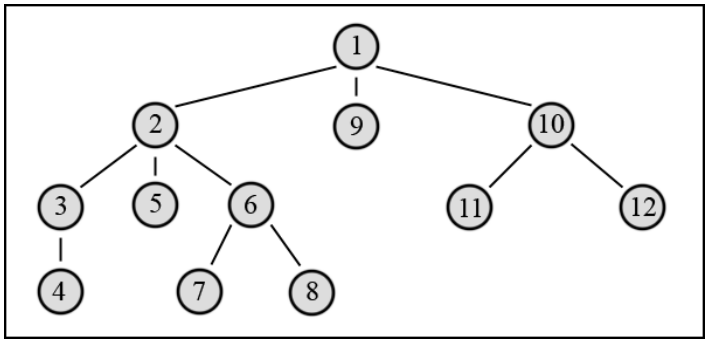
Mélység	Csomópontok száma	Időigény	Tárigény
0	1	0,001 másodperc	100 bájt
2	111	0,1 másodperc	11 Kb
4	11111	11 másodperc	1 Mb
6	10^6	18 perc	111 Mb
8	10^8	31 óra	11 Gb
10	10^{10}	128 nap	1 Tb
12	10^{12}	35 év	111 Tb
14	10^{14}	1500 év	11111 Tb

4. ábra: a szélességi keresés idő- és tárigénye

Könnyen belátható az ábra alapján, hogy ezzel a stratégiával az exponenciális komplexitású keresési problémák közül csak a legkisebb problémapéldányokat vagyunk képesek megoldani. [3] [5]

2.4.1.2. Mélységi kereső

A mélységi kereső (depth-first search) a generálás során létrejövő új csomópontokat a kiterjesztésre váró csomópontok sorának az elejére szúrja be, s így a várakozó csomópontok közül mindig a legmélyebb csomópontok egyikét választja ki kiterjesztésre. Egy ilyen kereső csak akkor terjeszt ki az előző csomópontnál kisebb mélységű csomópontot, ha az előzőnek nincs gyermeke, és nem célállapotot tartalmaz. Ezt visszalépésnek (backtracking) nevezzük. Az 5. ábrán látható, hogyan is működik pontosan egy mélységi kereső. Itt is feltesszük, hogy a 12-es csomópont tartalmazza a célállapotot, így ez is egy legrosszabb eset.



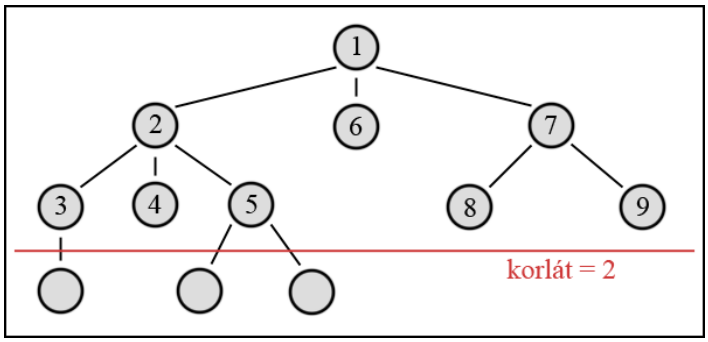
5. ábra: célteszt és kiterjesztés sorrendje mélységi keresés során

A mélységi kereső előnye szélességi társával szemben, hogy míg annak az összes bejárt csomópontot tárolnia kell, addig ennek elég egyszerre csak egy gyökértől levélelemig tartó utat és az út csomópontjai melletti kiterjesztésre váró csomópontokat tárolni, hogy a megoldás a célállapotot tartalmazó csomópontból visszavezethető legyen. Azt mondhatjuk, hogy a szélességi kereső hatalmas tárigényéhez képest a mélységinek b elágazási tényező és m mélységű keresőfa mellett csupán b·m db csomópont méretének megfelelő mennyiségű tárra van szüksége. A mélységi keresés időigénye $O(b^m)$, ami megegyezik a szélességi keresés legrosszabb esetben vett időigényével. A kedvezőbb algoritmus kiválasztásához több szempontot is figyelembe kell venni:

- Fontos a memóriahasználat? A mélységi keresés általában sokkal kevesebb memóriát igényel.
- A legrövidebb megoldást keressük? Ha így van, akkor a szélességi kereső jobb lehet, hiszen mindig a legrövidebb megoldást találja meg legelőször. A mélységi kereső optimális megoldást csak az összes megoldás megkeresésével tud biztosítani, ehhez azonban a teljes keresési fát be kell járnia.
- Gyorsan szeretnénk megtalálni a megoldást? Ebben az esetben kissé bonyolultabb a választás, ugyanis a mélységi keresés gyorsabb, ha sok út vezet a célhoz, de ezek viszonylag hosszúak. A szélességi keresés akkor gyorsabb, ha az állapottér nagy és mély, de van a gyökérelemhez közeli sekély megoldás.
- Végtelen a keresési fa? Mert ha igen, akkor a mélységi kereső körfigyelés és mélységkorlát nélkül nagyon könnyen végtelen ciklusba futhat, és soha nem ér véget a keresés. Ebből kifolyólag a mélységi keresés nem teljes. [3] [5]

2.4.1.3. Mélységkorlátozott keresés

A mélységkorlátozott keresés (depth-limited search) szinte megegyezik a mélységi kereséssel, de annak csapdáját hivatott kikerülni azáltal, hogy egy bizonyos mélység után nem terjeszti ki a csomópontokat, ezáltal kényszerítve ki a visszalépést. A mélységkorlátozott keresés működését a 6. ábra szemlélteti. Ha feltesszük, hogy a 9-es csomópont tartalmazza a célállapotot, akkor a jól megválasztott korlát = 2 mellett ez is egy legrosszabb eset.

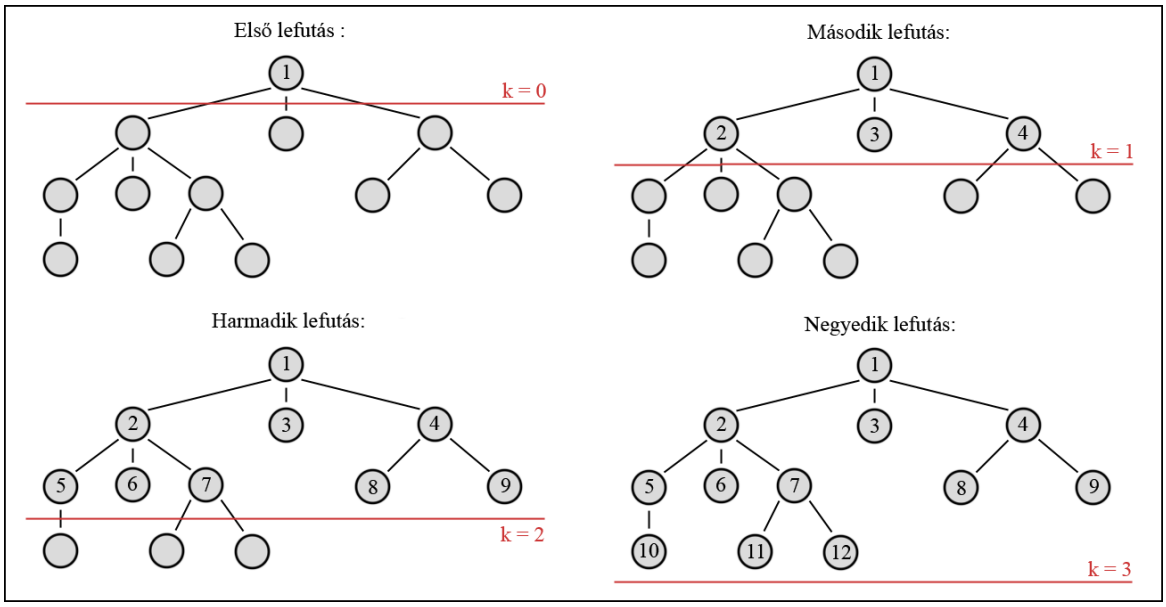


6. ábra: célteszt és kiterjesztés sorrendje mélységkorlátozott keresés során

De nem csak a végtelen ciklus ellen nyújt védelmet, hiszen ha tudjuk egy problémáról, hogy 50-es mélység előtt biztosan van megoldás, akkor a korlát hatására a kereső nem fog feleslegesen időt tölteni a fában a szóban forgó mélységnél mélyebben. Ha jól választjuk meg a mélységkorlátot, akkor a kereső teljes, de ez sem garantálja, hogy az első megoldás optimális. Rossz korlát esetén a kereső nem teljes, ráadásul leghosszabb megoldás keresésekor megfoszthatjuk magunkat a lehetőségtől, hogy optimális megoldásra találjunk, ha az mélyebben van a korlátként megadott mélységnél. A mélységi keresőhöz hasonlóan időigénye $O(b^k)$, tárigénye $O(b \cdot k)$, ahol k a mélységkorlát. [3] [5]

2.4.1.4. Iteratívan mélyülő keresés

Nyilván való, hogy mélységkorláttal történő keresésnél kulcsfontosságú egy jó korlát megválasztása. De mit tehetünk akkor, ha a keresés előtt nem tudjuk elég pontosan megbecsülni a megfelelő értéket? Erre a kérdésre ad választ az iteratívan mélyülő keresés (iterative deepening search). A stratégia lényege, hogy ötvözi a szélességi és mélységi keresés előnyös tulajdonságait, egyúttal leveszi a vállunkról a jó korlát megbecslésének a terhét. Mindezt úgy teszi, hogy végig próbálja az összes lehetséges mélységkorlátot, míg megoldást nem talál. Az első mélységkorlát a 0, a második az 1, majd 2, és így tovább. A kiterjesztés a szélességi keresésnek megfelelően történik, azzal a különbséggel, hogy itt egy csomópontot többször is kiterjeszthet, s így a keresés teljes és optimális, de a mélységi kereséshez hasonlóan memóriaigénye viszonylag kicsi. Működése a 7. ábrán látható, ahol csak úgy, mint eddig, a 12-es csomópont tartalmazza a célállapotot.



7. ábra: célteszt és kiterjesztés sorrendje iteratívan mélyülő keresés során

Ez a stratégia első hallásra nagyon pazarlónak tűnhet, hiszen ha például a 100-as korlátig eljut, akkor az 50-es mélységű csomópontokat 50-szer is kiterjesztheti. Jobban belegondolva viszont észrevehetjük, hogy az exponenciális komplexitású problémák keresési fájában majdnem az összes elem a legmélyebb részeken található. Ez az oka annak, hogy a többszöri kiterjesztéssel járó többletmunka elenyésző. Idézzük vissza, hogy k mélységig, b elágazási tényezővel rendelkező keresési fában a mélységkorlátozott keresésnél a kiterjesztések száma

$$1 + b + b^2 + b^3 + \ldots + b^k$$

volt, míg az iteratívan mélyülő keresésnél ez

$$(k + 1)1 + (k)b + (k - 1)b^2 + \ldots + 2b^{k-1} + 1b^k.$$

Konkretizálva például b = 10 és k = 5 esetén ezek a számok:

$$1 + 10 + 100 + 1000 + 10\,000 + 100\,000 = 111\,111,$$

és

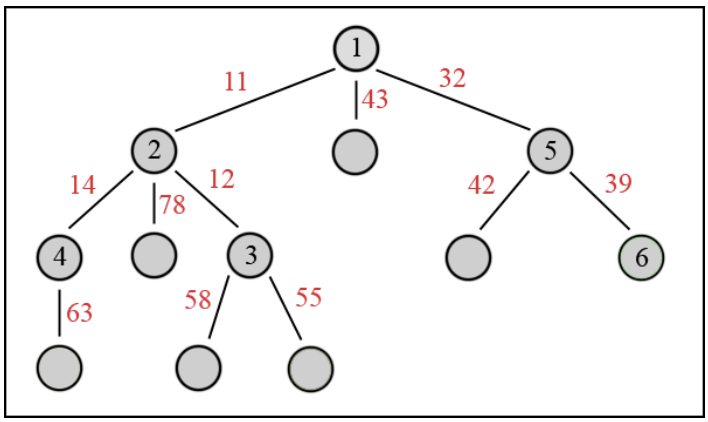
$$6 + 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,456.$$

Mindent egybevetve b = 10 elágazási tényező mellett az iteratívan mélyülő keresés 1-es mélységtől k mélységig csak körülbelül 10%-kal terjeszt ki több csomópontot, mint az

egyszerű szélességi keresés vagy a k mélységkorláttal rendelkező mélységkorlátozott keresés. Ha ugyanezeket az értékeket megnézzük b = 2 esetében is, akkor észrevehetjük, hogy minél nagyobb az elágazási tényező, annál kevesebb lesz a többletmunka. Az iteratívan mélyülő keresés időigénye megmaradt $O(b^k)$, de tárigénye csak $O(b \cdot k)$. Összegezve a tapasztaltakat elmondható, hogy amennyiben a keresési fa nagy és nem tudjuk jól megbecsülni a megoldások mélységét, akkor az iteratívan mélyülő keresés ideális választás. [3]

2.4.1.5. Egyenletes költségű keresés

Korábban már szó esett róla, hogy a szélességi kereső csak akkor optimális, ha minden operátor költsége azonos, ezért általános útköltség függvény használatakor nem biztos, hogy az általa először megtalált legsekélyebb célállapot optimális megoldás lesz. Az egyenletes költségű keresés (uniform cost search) annyiban módosítja a szélességi keresést, hogy a legkisebb mélységű csomópont helyett mindig a legkisebb útköltségű csomópontot választja ki kiterjesztésre. Ennek hatására az első sikeres célteszt esetén biztosan a legolcsóbb megoldást találtuk meg, mert ha lett volna kisebb költségű út, akkor a kereső azt választotta volna hamarabb, ahogy az a 8. ábrán is látható.



8. ábra: célteszt és kiterjesztés sorrendje egyenletes költségű keresés során

Tekintve a 8. ábrán látható példát, ezt hétköznapi nyelven úgy lehetne megfogalmazni, hogy mivel az összes út közül mindig a legkisebb költségűt „folytatta”, ezért a 6-os, célállapotot tartalmazó csomópont céltesztje után másik csomópontot kiterjesztve nem találhatunk kisebb útköltségű és célállapotot tartalmazó csomópontot, hiszen az a másik csomópont már így is költségesebb. Ezért nem számít az sem, ha az a bizonyos másik is célállapotot tartalmaz, arról nem is beszélve, hogy abból kiindulva keressünk egy másik

célállapotot. A piros számok a megfelelő élekhez tartozó útköltségeket jelölik. Lehetnének például városok közötti távolságok km-ben megadva, és segítségükkel keressük a legrövidebb utat A és B között. A gyökércsomópont állapota írná le azt, hogy „A-ban vagyunk”, a 6-os (6.-jára vizsgált) csomópont célállapota pedig azt, hogy „B-ben vagyunk”.

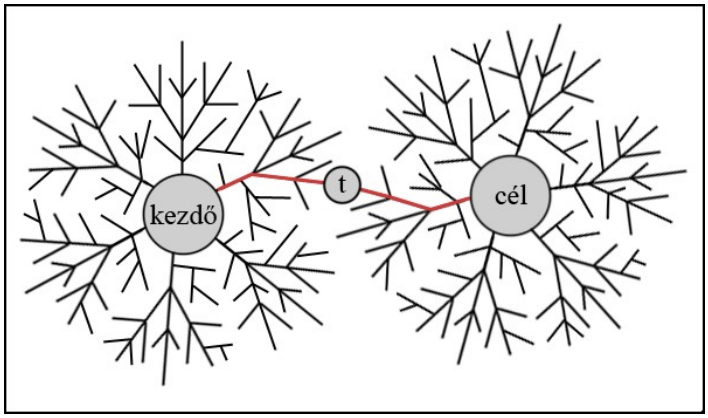
Az egyenletes költségű keresés tehát a legolcsóbb megoldást találja meg, de ehhez be kell tartanunk egy egyszerű feltételt: az útköltség egy út bejárása során sohasem csökkenhet. Más szavakkal minden n csomópontra igaz, hogy:

$$g(\text{KÖVETŐ}(n)) \geq g(n).$$

Ez azt jelenti, hogy egy csomópont útköltsége nem a szülőcsomópontból való odajutás költségével egyenlő, hanem a gyökércsomóponttól odáig vezető összköltséggel. Negatív útköltség használata nem ajánlott, de ha valamilyen okból mégis muszáj ilyet alkalmaznunk, akkor optimális megoldás kereséséhez az egész fát be kell járni, mert előre nem tudhatjuk, hogy egy hosszabb út vajon „lerövidül-e” annyira, hogy az legyen a legrövidebb. [3]

2.4.1.6. Kétirányú keresés

A kétirányú keresés (bidirectional search) azt az ötletet valósítja meg, miszerint gazdaságos lehet, ha egy időben ugyanazon a problémán két kereső kooperál. Ily módon a keresés akkor eredményes, ha a két keresés összetalálkozik valahol a keresési térben. Ehhez a két keresőnek tisztában kell lennie azzal, hogy az éppen általa generált csomópontot a másik kereső feldolgozta-e már, s amennyiben ez megtörtént, akkor a találkozási pontot jelentő csomóponttól a célig és a gyökérig vezető fél utakból előállítható a megoldás.



9. ábra: a két kereső találkozik a t csomópontban, így előáll egy megoldás

Az ábrán szemléltetett kétirányú kereső két szélességi keresőt használ, azonban lehetőségünk van más keresési stratégiák használatára is. Mindig az adott problémapéldányhoz legjobban illő keresőket érdemes választani, sok esetben viszont nehéz lehet megállapítani, hogy melyik a leghatékonyabb párosítás.

Fontos kérdés, hogy hogyan lehet megvalósítani a célcsomópontból hátrafelé haladó keresést, ami az elődcsomópontok generálását jelenti. Ha egy probléma esetében az összes operátor reverzíbilis, akkor az előd- és utódcsomópontok halmaza megegyezik, amikor viszont ez a szerencsés helyzet nem áll fent, akkor az elődcsomópontok meghatározása meglehetősen nehéz feladat.

Amikor a feladatnak egyetlen célállapota van, akkor nem kérdéses, hogy a második kereső honnan induljon. De mit tehetünk, ha sok célállapot létezik? Ha a célállapotok explicit módon fel vannak sorolva, akkor az elődcsomópontok generálását nem csak egyetlen csomópont állapotára alkalmazhatjuk, hanem állapotok egy halmazára is. Sok esetben viszont a problémának olyan sok megoldása van, hogy azokat explicit módon felsorolni nem érdemes, ehelyett egy célfeltétellel jelöljük ki az állapottérnek a célállapotok halmazát alkotó részét (pl. sakk). Előfordulhat az is, hogy nem ismerjük a célállapotokat, s ezért kell feltétellel megadnunk azokat (pl. a 8 királynő probléma). Ezek alapján elmondható, hogy a kétirányú keresőt csak a kevés és jól ismert célállapottal rendelkező problémák esetében érdemes használni.

A stratégia komplexitása $O(b^{m/2})$, hiszen m mélység esetén a kereső mindkét oldalán m/2 mélységig kell, hogy eljusson, s így

$$1 + b + b^2 + b^3 + \ldots + b^m$$

helyett csak

$$2(1 + b + b^2 + b^3 + \ldots + b^{m/2})$$

csomópontot kell generálnia. Ez b = 10 és m = 4 mellett 11 111 db helyett 222 db csomópontot jelent. Ahhoz, hogy a két kereső összetalálkozása megvalósulhasson, legalább az egyiknek el kell tárolnia az általa generált összes csomópontot, tehát a kétirányú keresés tárigénye $O(b^{m/2})$. [3]

2.4.2. Informált keresési stratégiák

Ahol a keresési tér túl nagy ahhoz, hogy az összes csomópontot végigjárjuk a keresés során, ott egy olyan állapotokat kiértékelő függvényre van szükség, amely megbecsüli, hogy melyek a legígéretesebb csomópontok. Általában elmondható, hogy a függvény egy célállapot kiértékelésekor nullát ad vissza, s egy állapotot minél messzebbinek ítél meg a célállapottól, a visszatérési értéke annál nagyobb. Használatával a kereső a nem túl hasznosnak ígérkező csomópontokat elkerülve csak azokra a csomópontokra koncentrál, melyek feltehetőleg közelebb visznek a célhoz. Mint azt fentebb is említettem, azokat a keresőket hívjuk informáltnak vagy más néven heurisztikusnak, amelyek plusz információval rendelkeznek a csomópontokban lévő állapotokról. A plusz információ alatt ezt az információt értjük. A heurisztika szó a „megtalál" vagy „felfedez" jelentésű görög heuriskein szóból származik, és olyan „ökölszabályként" gondolnak rá, amely segítségével a terület szakértői kimerítő keresés nélkül is jó megoldásokat tudnak előállítani. A keresési algoritmusok esetében egy olyan függvényt jelent, amely a megoldás költségét becsli. Vegyük most sorra a legismertebb heurisztikus stratégiákat. [3] [5]

2.4.2.1. Iteratívan javító keresők

Az élet során számos példát láthattunk már olyan problémára, amelynél nem a cselekvéssorozat meghatározása az elsődleges cél, hanem egyedül a megoldásra vagyunk kíváncsiak. Ilyen a 8 királynő probléma is, amikor úgy kell elhelyeznünk 8 királynőt egy sakktáblán, hogy azok ne üthessék egymást egy lépésből. A probléma szempontjából lényegtelen, hogy milyen sorrendben rakjuk le őket, csak azokat a felállásokat keressük, amelyek megfelelnek ennek a követelménynek. Az ilyen jellegű problémákat célszerű úgy megoldani, hogy egy teljes konfigurációból kiindulva (pl. amikor az összes királynő a táblán van, álljanak azok bárhogyan is) próbálunk olyan változtatásokat végrehajtani, melyek hatására javul a konfiguráció minősége. A legkönnyebben úgy képzelhetjük ezt el, mintha az állapottér egy tájkép lenne, ahol minden egyes állapot magasságát a kiértékelő függvény által azon az állapoton felvett érték határozná meg, és a kereső ennek a tájnak a legmagasabb pontjára akarna feljutni. Erre ad lehetőséget az iteratívan javító keresők (iterative improvement search) két legismertebb változata, a hegymászó és a szimulált lehűtéses módszer. [3]

2.4.2.1.1. Hegymászó keresés

A hegymászó keresés az egyik legegyszerűbb informált keresési módszer, mindösszesen egy ciklusból áll, amely az aktuális csomópontból generálja annak gyermekeit és a legjobbat (a legmagasabban fekvőt) kiválasztva halad tovább. Amennyiben az aktuális csomópontnál csak rosszabb kiértékelésű csomópontokat generált, a keresés véget ér. A nevét is innen kapta, hiszen rosszabb csomópontot sohasem választva mindig felfelé törekszik, s ha már nem juthat föntebb vagy maradhat ugyanazon a magasságon, akkor megáll. Amennyiben több legjobb csomópont is létrejött, akkor véletlenszerűen választ ki közülük egyet kiterjesztésre. A keresőnek ehhez nem kell keresési fát nyilvántartania, és a csomópontban is csak az állapotot és az ahhoz tartozó kiértékelést kell tárolnia. A kereső minden esetben elér egy olyan pontot, ahonnan már nem tud továbbhaladni. Az algoritmussal kapcsolatban azonban több probléma is felmerül:

- Ha a kereső az állapottérben olyan pontra téved, amely minden szomszédos pontnál magasabban fekszik, de nem az állapottér legmagasabb csúcsa, akkor egy lokális maximum fogságába esett. Ily módon a kereső a céltól meglehetősen távol is befejezheti működését.
- Az állapottérnek egy területe lehet fennsík, amely annyit tesz, hogy ott a kiértékelési függvény gyakorlatilag lapos. Az ilyen területeken a hegymászó kereső bolyonghat.
- Előfordulhat, hogy a kereső a hegygerincre viszonylag könnyen feljut, de az lassan emelkedik a csúcs felé. Ha ekkor nincsenek olyan operátorok, melyek a gerinccel párhuzamosan vezetnek felfelé, akkor a kereső elkezdhet oszcillálni a gerinc két oldala között.

Ezen buktatók kikerülése végett alkalmazható a véletlen újraindítású hegymászás (random--restart hill-climbing), ami véletlenül generált kiindulási pontokból indítja a kereséseket rögzített számú iterációs lépésekben. A keresés hamarabb is leállhat, ha néhány iterációs lépés során a megtalált legjobb megoldás nem mutat javulást. Ha kellő számú iterációt engedélyezünk, akkor a hegymászó kereső meg fogja találni az optimális megoldást. Azonban a keresés időtartamát és sikerességét nagyban befolyásolja az adott probléma állapotterének a felszíne, de ha a keresőt csak az ahhoz illő problémákra alkalmazzuk, akkor viszonylag kevés iterációs lépés után is megkaphatjuk az optimális megoldást. [3] [5]

2.4.2.1.2. Szimulált lehűtés

A szimulált lehűtés (simulated annealing) alapgondolata az, hogy a keresőnek bizonyos esetekben lehetősége legyen egy-két lefelé vezető lépést tennie, hogy ezáltal kimeneküljön a lokális maximumok fogságából. Mivel ez a módszer a hegymászó keresés egy változatának tekinthető, az alap algoritmusa nagyon hasonlít az elődjére. A különbség csak az, hogy míg a hegymászó kereső mindig a lehető legjobb állapotot terjesztette ki, a szimulált lehűtés mindig egy véletlenül kiválasztott irányba folytatja a keresést. Természetesen az állapotok nem ugyanakkora eséllyel kerülnek végrehajtásra, hiszen a kereső teljesen használhatatlan lenne. A felfelé vezető lépések kiválasztásakor azokat minden esetben végre is hajtja, de a lefelé vezető lépések végrehajtásának esélye ΔE-vel arányosan exponenciálisan csökken. Minden állapothoz meghatározható ez a ΔE mennyiség, amely megmutatja, hogy az állapot a szülőjéhez képest milyen mértékű romlást mutat, de a valószínűségek meghatározásához szükség van egy T paraméterre is. Az újonnan generált állapotokhoz tartozó ΔE mennyiségek meghatározása után minden állapot $e^{\Delta E/T}$ valószínűséggel kerülhet végrehajtásra. A T paramétert minden iterációs lépésben meghatározza a keresés előtt bemenetként kapott hűtési karakterisztika. Az iterációs lépések során a T paraméter a nullához tart, így a lefelé vezető lépések végrehajtásának esélye folyamatosan csökken.

Látható, hogy a kereső nem véletlenül kapta a nevét egy természetben megfigyelhető folyamatról, hiszen annak az analógiájára fejlesztették ki. Ha a kiértékelő függvényt az anyag összenergiájának, a T paramétert pedig a hőmérsékletnek tekintjük, akkor megkapjuk a lehűlés folyamatát leíró fizikai összefüggést. Bizonyított, hogy ha egy anyagot megfelelően lassan hűtenek le, akkor az anyag kisebb energiájú állapotba kerül dermedéskor, mint ha gyorsan menne végbe a hűlés folyamata. Amennyiben a T paraméter a jól megválasztott hűtési karakterisztika hatására kellő mértékben csökken, akkor a kereső megtalálja a feladat globális minimumát. Az ilyen keresési módszereket leggyakrabban termelési folyamatok és elrendezési problémák optimalizálásához használják. [3]

2.4.2.2. Legjobbat-először (best-first) keresők

Az előzőekben olyan problémákról esett szó, amelyeknél csak a megoldás megkeresése volt a cél. Most tekintsük azokat a problémákat, amelyeknél a keresés elsődleges célja a gyökércsomóponttól egy célállapotot tartalmazó csomópontig vezető út feltárása. Ehhez a csomópontban nem elég az állapotot és a kiértékelő függvény által szolgáltatott értéket tárolni, hanem a korábban tárgyalt adattagok is megjelennek, hogy a nem informált keresőknél tapasztaltakhoz hasonlóan a keresés végén a megoldást jelentő út visszavezethető legyen. Az általános keresési algoritmust használva a meglévő tudás a kiterjesztendő csomópont megválasztásakor jelenik meg. Amennyiben a frissen generált csomópontok közül mindig a legjobb kiértékelésű csomópontot terjesztjük ki legelőször, akkor az így kapott stratégiát legjobbat-először keresésnek nevezzük. A név csalóka lehet, mert nem biztos, hogy tényleg az a legjobb csomópont, amelyet a kiértékelő függvény annak ítél. A különböző típusú legjobbat-először stratégiák algoritmusa majdnem megegyezik, eltérést leginkább a kiértékelő függvény milyenségében tapasztalhatunk. A keresők általában a megoldás valamilyen költségbecslését használják, ilyen volt a vak keresők esetében az útköltség. Ez a mérték azonban nem tartalmaz információt az adott állapotból a legközelebbi célállapotba vezető út költségéről, így nem fókuszálja a keresést. Ahhoz hogy a keresést hatékonyabbá tehessük, ki kell tudnunk választani a kiterjesztésre váró csomópontok közül a célhoz legközelebb eső csomópontot vagy a célhoz vezető legkisebb költségű úton található csomópontot. [3] [5]

2.4.2.2.1. Mohó keresés

Az egyik legegyszerűbb legjobbat-először keresési stratégia a mohó keresés (greedy search). A mohó keresés mindig azt a csomópontot terjeszti ki először, amelyiket kiértékelő függvénye legközelebbinek ítéli meg a célhoz. Az ilyen jellegű költségbecslést kiszámító függvényeket heurisztikus függvényeknek nevezzük, és h betűvel jelöljük őket.

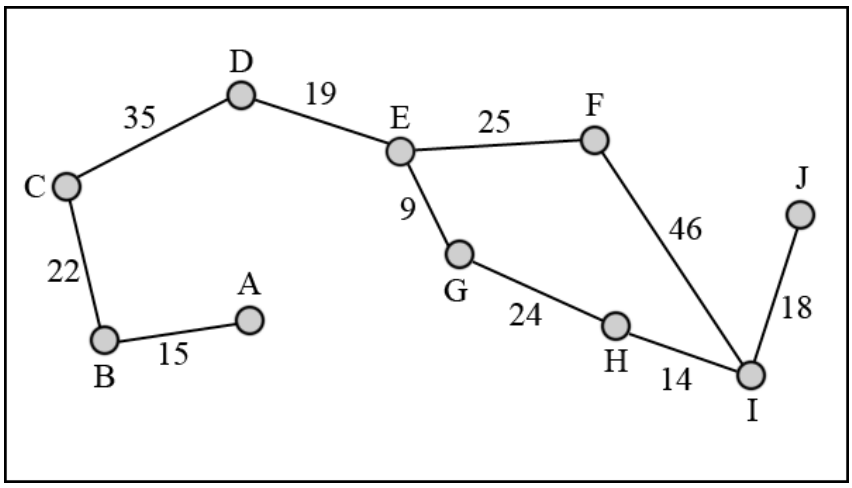
h(n) = az n csomópont állapotából egy célállapotba vezető legolcsóbb út becsült költsége.

Ez a h függvény szinte bármilyen lehet, de feltesszük hogy h(n) = 0, ha n egy célállapotot tartalmazó csomópont.

Ahhoz, hogy a mohó keresés természetét megismerjük, egy konkrét példán kell megvizsgálni annak működését. Ha veszünk egy útvonal-keresési problémát, jó heurisztikának számít a

h(n) = az n csomópont célcsomóponttól légvonalban mért távolsága.

A h függvény abban segíti a keresőt, hogy kiválasszon egy lehetőleg minél jobban a cél felé vezető úton lévő csomópontot. A kereső azonban így nem optimális és nem is teljes, amit a következő példák be is bizonyítanak.



10. ábra: egy elképzelt térképrészlet városokkal és az utak hosszával

Tekintsük a 10. ábrát és tegyük fel, hogy J-ből szeretnénk eljutni E-be. A kereső kiterjeszti J-t és az egyetlen lehetséges utat választja I-be, majd kiterjeszti I-t is. Látható, hogy a heurisztikus függvény a két lehetséges város közül F-et fogja előnyben részesíteni H-val szemben, mert az F szemmel láthatóan közelebb fekszik E-hez. Ezután F-et kiterjesztve megtalálja E-t. A keresés tehát sikeresen befejeződött, de ha kicsit utánaszámolunk, akkor láthatjuk, hogy az E-F-I-J utak összesen 89 km-t tesznek ki, míg az E-G-H-I-J utak csak 65 km-t. Ezek szerint a megoldás tényleg nem optimális. Most nézzünk egy másik példát, melynél B-ből szeretnénk eljutni az E pontba. A kereső kiterjeszti B-t és azt látja, hogy a legkedvezőbb az, ha A felé tart, hiszen az közelebb fekszik E-hez mint C. Ebben az esetben az ismétlődő csomópontok figyelése nélkül a kereső sohasem fejezi be a keresést, hiszen A és B közt fog ugrálni. Azonban ha figyeljük az állapotok ismétlődését, még akkor is előfordulhat, hogy egy végtelen úton elindulva a kereső eltéved és soha nem tér vissza, hogy felfedezze a helyes irányt.

Látható, hogy a kereső tényleg rászolgál a mohó névre, hiszen egyáltalán nem érdekli, hogy hosszú távon mi lenne a kifizetődőbb, mindig mérlegelés nélkül az éppen legjobbat választja, még ha ki is derülhet, hogy a lehető legrosszabbat választotta. A keresés időigénye és tárigénye így a legrosszabb esetet feltételezve egyaránt $O(b^m)$, de ha a hozzá illő problémára alkalmazzuk és a heurisztikus függvény is jó, akkor ez a szám jelentősen csökkenhet. [3] [5]

2.4.2.2.2. A* keresés

Az A* keresés nagyon hasonlít a mohó keresési stratégiára, de kiértékelő függvénye összetettebb. A mohó kereséssel ellentétben az A* kereső döntéskor felhasználja az általa addig megtett út költségét, s így próbál meg olyan megoldást találni, melynél a teljes út költsége minimális. Ennek megvalósításához kiértékelő függvénye két összetevőből áll. Az egyik a célig még hátralévő út becsült költsége, a másik a gyökércsomóponttól a vizsgált csomópontig már megtett út költsége. A mennyiségeket összeadva kapjuk az

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

kiértékelő függvényt. Az A* kereső esetén egy csomópont költsége:

f(n) = a legolcsóbb és az n csomóponton keresztül vezető megoldási út becsült költsége.

Az A* kereső több kereső pozitív tulajdonságát ötvözi, miközben elhagyja azok negatív tulajdonágait. Korábban láthattuk, hogy az egyenletes költségű keresés teljes és optimális is, hiszen minimalizálja a keresés során megtett út költségét. Ennek viszont következménye, hogy a keresés időtartama $O(b^m)$. A mohó kereső minimalizálja a hátralévő út becsült költségét, így a kereső költségét jelentősen lecsökkenti, cserébe viszont nem teljes és nem is optimális. Az f(n) kiértékelő függvény segítségével az A* kereső nemcsak teljes és optimális, de a keresés költségét is sikerül alacsonyan tartania. Ennek egyetlen feltétele, hogy a heurisztikának elfogadhatónak kell lennie, azaz sohasem szabad túlbecsülnie a megoldásig vezető út költségét. Ez hatással van az f kiértékelő függvényre is, hiszen ha a h elfogadható, akkor f sem fogja túlbecsülni az adott csomóponton áthaladó legjobb megoldás költségét. Azokat a keresőket, melyek f kiértékelő függvényt használnak és heurisztikájuk elfogadható, A* keresésnek nevezzük. [3] [5]

2.5. A program felépítése

A program Java nyelven íródott, NetBeans IDE 6.1 fejlesztői környezetben. A probléma megoldásához használt osztályok logikai hovatartozásuk alapján csomagokba vannak rendezve, így a program bemutatása során az a legegyszerűbb, ha ezeket a csomagokat vesszük sorra.

2.5.1. Az *allapotter* csomag

Az *allapotter* csomagban két absztrakt osztály és egy interfész található, melyek rögzítik az állapotterek általános elemeit. Ezeket az elemeket konkretizáljuk egy adott probléma állapottér-reprezentációjának implementálásakor. Például az *Allapot* absztrakt osztályt kiterjesztve meg kell határoznunk annak az absztrakt metódusnak a törzsét, mely ellenőrzi egy adott állapotról, hogy az célállapot-e? Az *Operator* absztrakt osztálynak egy adattagja sincs, mert a problémák sokszínűségére való tekintettel az operátorokról általánosságban nem sokat tudunk elmondani, viszont puszta jelenlétével figyelmeztet egy konkrét operátorosztály szükségességére. Az absztrakt osztályok a keresők problémafüggetlensége miatt is nélkülözhetetlenek. A *HeurisztikusAllapot* interfész jelzi, hogy ha egy állapot heurisztikus, akkor van egy állapot-kiértékelő függvénye is, melyet ugyancsak implementálnunk kell. [6]

2.5.2. A *kocka* csomag

Az *allapotter* csomag osztályait és interfészét felhasználva készített osztályok a *kocka* csomagban foglalnak helyet, és ezek az osztályok már szorosan a Rubik-kockához kapcsolódnak. A *KockaAllapot* osztály példányai az állapottér egy-egy állapotát írják le, de az osztály tartalmazza a Rubik-kocka állapotát nyilvántartó 54 elemű byte-tömbön kívül az osztály példányosításához használt konstruktorokat és az állapotokra meghívható összes metódust is, melyek a következők:

- Célállapot ellenőrző függvény, melynek visszatérési értéke igaz vagy hamis lehet. Három egymásba ágyazott for-ciklus segítségével végigjárja a kocka állapotát leíró háromdimenziós tömböt, és ha egy oldalon oda nem illő színt talál, akkor azonnal hamissal tér vissza. Ha sikerült végigjárnia a tömböt, akkor minden szín a helyén van és visszatérési értéke igaz lesz.

- Operátor-alkalmazási előfeltételt ellenőrző függvény, mely megvizsgálja, hogy az adott állapotra alkalmazható-e a paraméterként kapott operátor. Visszatérési értéke ennek is logikai érték, ami a Rubik-kocka esetében minden esetben igaz.
- Egy *alkalmaz* függvény, mely a paraméterként kapott operátort alkalmazza az adott állapotra, visszatérési értéke a keletkezett állapot. Ehhez másolatot készít a kocka állapotáról, majd a másolat tömbjén elvégzi az operátornak megfelelő összes értékmásolást és visszatér a másolattal.
- Az adott állapotot egy paraméterként kapott állapottal összehasonlító függvény. Visszatérési értéke logikai, mely igaz, ha a vizsgált állapotok tömbjének minden eleme megegyezik, egyébként hamis.
- Egy kiértékelő függvény, mellyel részletesebben is foglalkozunk.
- Az állapotot nyilvántartó adattag lekérdező metódusa.
- Kiíratással kapcsolatos metódusok.

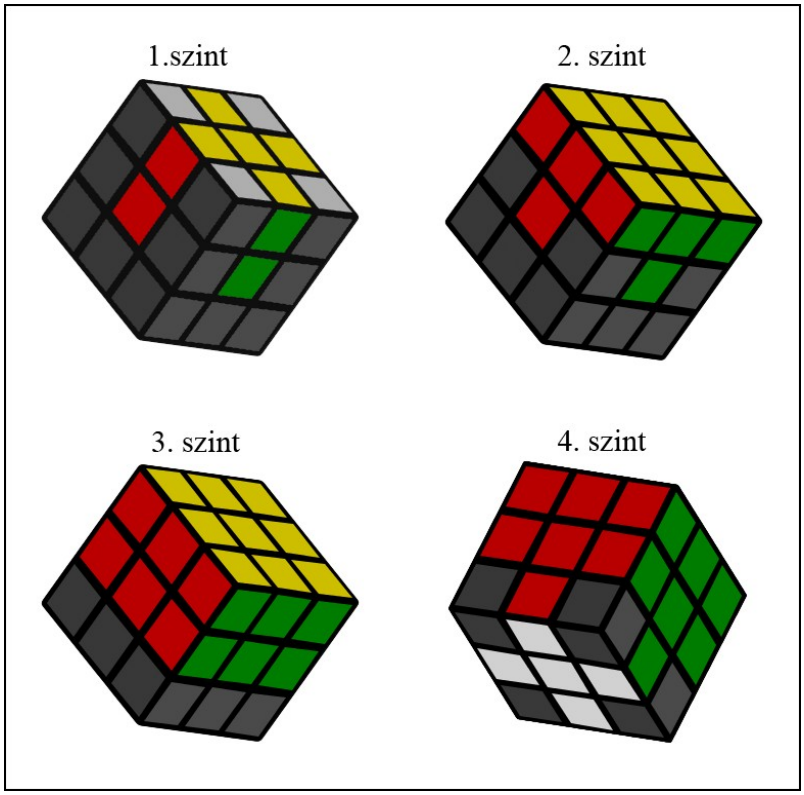
A *Teker* osztály példányai a Rubik-kocka operátorainak felelnek meg, és különbséget köztük az irányuk alapján tehetünk. Az osztály tartalmaz egy byte típusú *irány* adattagot, mely 1 és 12 között veszi fel értékeit a 12-féle lehetséges tekerésnek megfelelően, valamint az ezt lekérdező *getIrany* metódust, melynek segítségével a *KockaAllapot* osztály *alkalmaz* metódusa lekérdezheti a paraméterként kapott operátorok irányát. [4] [6]

2.5.2.1. A layer by layer módszer és a kiértékelő függvény

Ahogy azt még látni fogjuk, a program egy mohó keresőt használ a kocka megoldásához, és ennek megfelelően a kiértékelési függvény egyedül a heurisztikus függvényből áll, amit jelen esetben a *KockaAllapot* osztály *heurisztika* metódusa implementál. A metódus az adott állapotokat a korábban már említett soronként haladó, vagy más néven layer by layer módszer szerint pontozza le. Ez azt jelenti, hogy a heurisztika hatására a kereső a módszernek megfelelő úton keresi a megoldást, még ha manapság nem is ez a leghatékonyabb módszer. Legelső dolgunk, hogy kiválasszuk azt az oldalt, amelyikkel kezdeni szeretnénk. Mivel az esetek túlnyomó részében előre nem láthatjuk, hogy melyik oldallal kezdve lesz a végjáték egyszerűbb, ezért a program egy hatszor hosszabb kiértékelő függvény használata helyett mindig a sárga oldallal kezd. A módszerben könnyen ellenőrizhető rétegekről (layer), vagy

más szóval szintekről van szó, ezért a heurisztikus függvény is a szint ellenőrzésével kezdődik, hogy elkerülje az aktuálisnál csak alacsonyabb szinteken fontos ellenőrzéseket. Ezek a szintek a következők:

- 0. szint: a kocka még az 1. szint követelményeinek sem felel meg.
- 1. szint: a sárgát is tartalmazó élek a helyükön vannak és jó irányban állnak, azaz elkészült a sárga „kereszt”.
- 2. szint: a sárgát is tartalmazó sarkok a helyükön vannak és jó irányban állnak, azaz elkészült a legfelső sáv is.
- 3. szint: kész van a középső sáv is.
- 4. szint: a fehéret is tartalmazó élek a helyükön vannak és jó irányban állnak, azaz elkészült a fehér „kereszt”.
- 5. szint: a fehéret is tartalmazó sarkok a helyükön vannak és jó irányban állnak, azaz a kocka célállapotban van.



11. ábra: a layer by layer módszer szintjei

Előfordulhat olyan eset, amikor a heurisztika vagy akár a szint lerontása nélkül nem tudunk továbblépni. Segítségünkre lehetnek azonban az úgynevezett receptek. Ezek olyan lépéssorozatok, melyeket a megfelelő állapotra alkalmazva először lerontják ugyan a

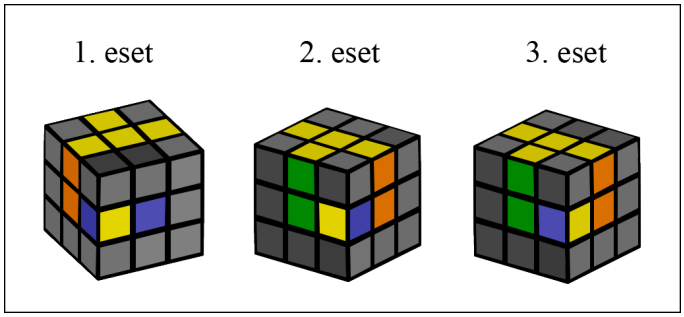
heurisztikát, de a lépéssorozat végére közelebb kerülünk a célhoz, mint a recept alkalmazása előtt voltunk. Az első szintet szerencsés esetben elérhetjük receptek nélkül is, mégis ez a legnehezebben megvalósítható része a heurisztikus függvénynek. Ennek oka, hogy a magasabb szintekkel ellentétben, ahol egy-két tekerést leszámítva már csak recepteket alkalmazunk, itt sokkal nehezebb az emberi tudást átadni a gépnek. A felsőbb szinteken már nincs min hezitálni, minden állapotra illik néhány recept, csak azt kell eldöntenünk, hogy melyiket alkalmazzuk előbb.

A továbbiakban egy állapot heurisztikája alatt az állapotra meghívott heurisztikus függvény visszatérési értékét, azaz az állapot „jóságát" értem. Az elgondolásom az volt, hogy minél alacsonyabb szinten vagyunk, az állapot heurisztikája annál magasabb értékről indul, majd a szintnek megfelelő vizsgálatok során minél messzebbinek tűnik az eggyel magasabb szint, ez az érték annál jobban növekszik. A növekedés mértékét tehát a szint teljesítéséhez szükséges élek vagy sarkok helyzete és/vagy iránya határozzák meg. Minden ilyen él vagy sarok többé-kevésbé növeli meg a heurisztikát, attól függően, hogy mennyire messze vagy közel esik a helyétől vagy egy olyan helytől, ahonnan egy recept segítségével helyére tehető. Egy szinten belül sohasem nő meg annyival a heurisztika értéke, hogy egy alacsonyabb szintű állapotnak a heurisztikája kisebb legyen. Ez a feltétel elengedhetetlen ahhoz, hogy a kereső a módszerhez mérten a lehető legrövidebb megoldást találja meg, valamint egyik következménye, hogy ha elértünk egy bizonyos szintet, akkor elég csak ahhoz a szinthez kapcsolódó vizsgálatokat elvégezni, hiszen a többi már úgyis vagy még úgysem teljesül. Ezeknek megfelelően a pontozás a következő módon zajlik:

- A pontozás alapjául szolgáló szint meghatározása a kiértékelés első lépése. Minél magasabb szinten vagyunk, annál kisebb a heurisztika kezdőértéke, ötös szint esetén a kiértékelő függvény visszatérési értéke 0.

- 0. szinten: Egy helyén lévő és jó irányban álló él alig növeli meg a heurisztikát, míg a többi él minél messzebb esik a helyétől, annál jobban növeli azt. Ha már legalább két él áll jó helyen és irányban, akkor megengedjük a receptek használatát is, és ezzel egy időben az élek kisebb növekedést okoznak, ha közel vannak ahhoz, hogy egy recept segítségével helyre rakjuk őket. Ezek a receptek csak 3-5 lépésből állnak, viszont mellékhatással is rendelkeznek. Összesen három ilyen recept alkalmazhatóságára kell figyelnünk oldalanként. Ennek oka, hogy a gép rögzített szemszögből értelmezi az operátorokat, mindig a sárga oldal van fent és a kék oldal szemben, s emiatt ugyanaz a

mozdulatsor más operátorokból épül fel a különböző oldalakra nézve. Példaként tekintsük meg a kék-sárga él három olyan helyzetét, melyből csak recept segítségével tehető helyre:

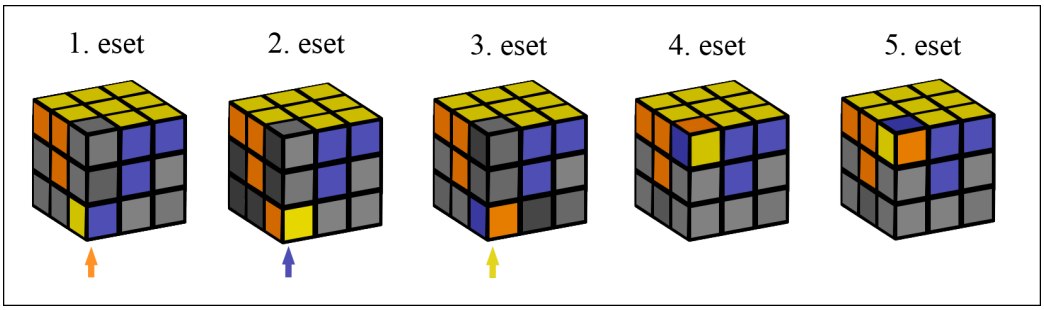


12. ábra: csak recept alkalmazásával helyre rakható élek

1. recept: UR, LB, UL. 2. recept: UR, LF, UL. 3. recept: UR, UR, RR, UL, UL.

A rövidítések az angol front, rear, left, right, upper, lower, forward és backward szavak kezdőbetűiből állnak össze.

- 1. szinten: Mivel ezen a szinten már csak recepttel lehet helyrerakni egy sarkot, a heurisztika a szintnek megfelelő kezdőértékén túl a sarkok „receptelhetőségtől” való távolságától függően nőhet. Ezen a szinten már 4·5 receptre kell figyelnünk. A példában nézzük meg a kék-sárga-narancs sarokra vonatkozó öt receptet:



13. ábra: sarkok „receptelhető” pozíciói

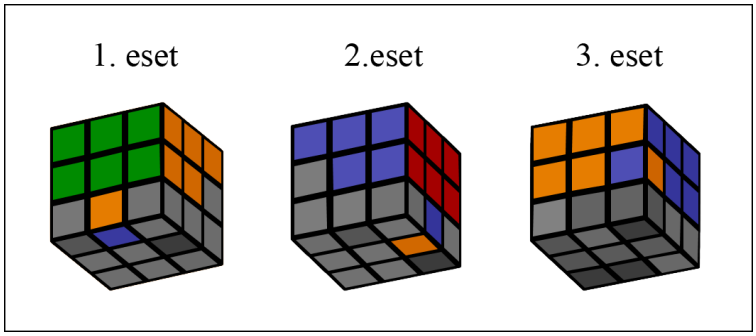
1. recept: LF, LL, LB. 2. recept: FL, LR, FR. 3. recept: LF, LR, LB.

4. recept: FL, LL, FR. 5. recept: LF, LL, LB.

A 13. ábrán látható szituációk közül csak az 1. és 2. esetben szolgál a recept a sarok helyrerakására. A 3., 4., illetve az 5. esetben a receptek csak az első két eset valamelyikére vezetnek minket vissza, valamint az 5. recept akkor is alkalmazható, ha

a kék-narancs-sárga helyén egy másik, sárgát tartalmazó sarok áll, mert előfordulhat, hogy két sarok egymás helyét foglalja el.

- 2. szinten: A heurisztika itt is az élek „receptelhető” helyeitől való távolságát pontozza. Ezen a szinten oldalanként három recept alkalmazhatóságát kell vizsgálni. Ezek a kék-narancs oldalra nézve a következők:



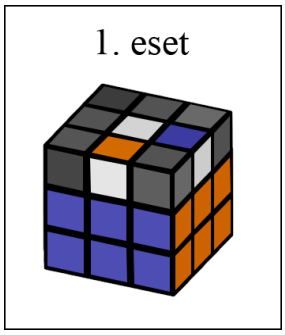
14. ábra: második sáv éleinek „receptelhető” pozíciói

1. recept: FL, LL, FR, LB, FR, LF, FL. 2. recept: LF, LR, LB, FR, LB, FL, LF.

3. recept: LF, LL, LB, LR.

A 14. ábrán látható állapotok közül csak az első kettőnél szolgál a recept az élek helyrerakására, a 3. recept az első szint 2. esetére vezet vissza minket. Ez korántsem baj, mert az oda illő recept végrehajtása után ismét 2. szintű állapotot kapunk, ráadásul az első eset egy tekeréssel elérhető lesz. A 3. recept akkor is alkalmazható, ha a kék-narancs él helyén egy másik, fehéret nem tartalmazó él foglal helyet, ekkor azonban nem feltétlenül a példán bemutatott esetekre jutunk vissza. Erre azért van szükség, mert előfordulhat, hogy két él egymás helyén áll.

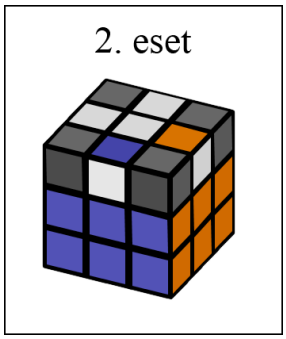
- 3. szinten: Ezen a szinten a fehér éleket először helyükre kell tenni, majd irányba kell forgatni őket. Hogy az összes él a helyén legyen, az embernek elég egy mozdulatsort megjegyezni, tehát a gépnek 4 recept alkalmazhatóságát kell figyelnie. A recept mindig két szomszédos élt cserél meg, így maximum kétszer kell végrehajtanunk és közben maximum egy-két tekerésre lehet szükség a „receptelhetőség” érdekében. Nézzük meg a mozdulatsort a kék-fehér és narancs-fehér él cseréjének esetében, de vegyük figyelembe, hogy a cserénél nem számít a két cserélt él iránya:



15. ábra: élcserére érdemes állapot

1. recept: LF, LL, LL, LB, LR, LF, LR, LB, LR.

Ha minden él a helyére került, akkor ideje jó irányba forgatni őket. Ehhez is elég egy mozdulatsort ismerni, mely egyszerre két szomszédos él irányát fordítja meg. A gépnek tehát itt is négy recept alkalmazhatóságát kell figyelnie. Ha egyik él sem áll jó irányban, akkor tetszőleges oldalon végezhető el az élcsere, a következő lépésben pedig olyan oldalra kell alkalmazni a receptet, melynek bal oldali szomszédjának éle jó irányban áll. Ha két szemközti él áll jó irányban, akkor hasonlóképpen járunk el. Amennyiben két szomszédos él áll jó irányban, akkor az előző esetek második lépése szerint kell cselekednünk. Tekintsük meg ez utóbbi esetet a kék-fehér és narancs-fehér élre nézve:

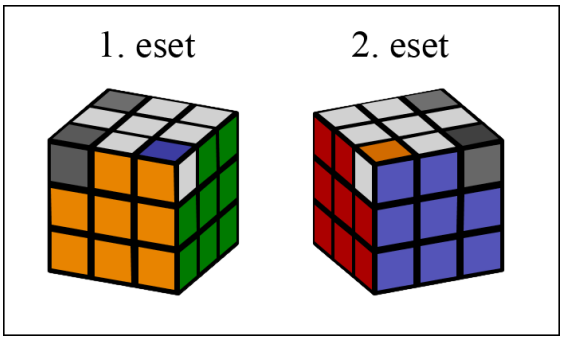


16. ábra: élforgatásra érdemes állapot

2. recept: LB, RB, FL, LF, RF, LR, LB, RB, FL, LF, RF, LR, LB, RB, FL, LF, RF, LR.

- 4. szinten: Már csak a maradék négy sarkot kell először helyére tenni, majd jó irányba forgatni. Az élek helyre rakásához két mozdulatsort kell megjegyeznünk, ezért a gépnek nyolc különböző recept alkalmazhatóságára kell odafigyelnie. Mivel egy sarokcserélő recept egyszerre három élt cserél meg körkörösen, ezért az előzőekben

látottakhoz hasonlóan itt is maximum két receptet kell alkalmaznunk, hogy a sarkok a helyükre kerüljenek. Amennyiben egy sarok sincs a helyén, tetszőleges oldalra alkalmazhatjuk a két recept egyikét, s ettől biztosan lesz egy olyan sarok, ami a helyére kerül. Ha egy sarok már a helyén van, akkor egy rossz helyen álló sarok a két recept egyikének segítségével mindkét lehetséges rossz helyről a jó helyre mozgatható, ezzel egy időben a másik két rossz helyen lévő sarok is a helyére kerül. Nézzük meg a második eshetőség két lehetséges variációját a kék-narancs-fehér sarokra nézve, de ne felejtsük el, hogy a sarok iránya lényegtelen:



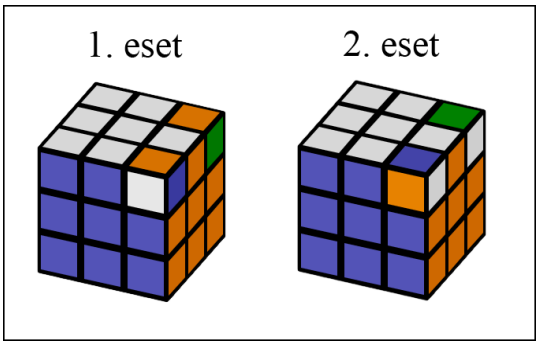
17. ábra: a kék-narancs-fehér sarok lehetséges helyzetei

1. recept: LB, LL, RB, LR, LF, LL, RF, LR.

2. recept: FR, LR, RR, LL, FL, LR, RL, LL.

A sarkok helyrerakása után az utolsó dolgunk, hogy azok irányát is megfelelően állítsuk be. Ehhez két mozdulatsort alkalmazhatunk, ami a gép számára nyolc lehetséges receptet jelent. Az egyik mozdulatsor a másik ellentettje, és így éppen az ellenkező irányba forgatja a sarkokat. Ebben az esetben is háromféle szituációval találhatjuk szemben magunkat. Az első, amikor egyetlen sarok sem áll a megfelelő irányban. Ekkor a második fajta mozdulatsor megfelelő receptváltozatát kell alkalmaznunk a kocka egy olyan oldalára, melynek jobb oldali szomszédjának az alsó sávjában két fehér lap is található. Ezzel eljutunk a harmadik fajta eshetőséghez. A második eshetőség, hogy egy sarok már jól áll. Ekkor ennek a jobb oldali szomszédját a megfelelő recepttel irányba állítva az átellenben lévő sarok is olyan irányt vesz fel, mellyel már a következő lépésben az ő saját jobb oldali szomszédjával egyszerre lehet a megfelelő irányba állítani. A harmadik eshetőség, hogy két sarok áll jó irányban, ekkor már csak a megfelelő receptet kell alkalmazni a maradék két sarok jó irányba

állítására. Nézzük meg a két mozdulatsort az utolsó esetre illő kék-narancs-fehér, illetve zöld-narancs-fehér sarkokra nézve:



18. ábra: a rossz sarkok lehetséges irányai

1. recept: RB, LL, RF, LL, RB, LR, LR, RF, LB, LR, LF, LR, LB, LR, LR, LF.

2. recept: LB, LL, LL, LF, LL, LB, LL, LF, RB, LL, LL, RF, LR, RB, LR, RF.

Az utolsó recept lefutása után elértük az 5. szintet, ami azt jelenti, hogy megoldást találtunk és a kereső befejezi működését. [3] [5] [6] [7]

2.5.3. A *kereso* csomag

Ez a csomag tartalmazza azt a *Kereso* és *Csucs* osztályt, melyek a programban használt keresők és az azok által generált csúcspontok ősosztályai. Ezekben az ősosztályokban olyan adattagok és metódusok találhatóak, melyek a legtöbb kereső és az általuk használt csúcspontok alapjául szolgálnak. A különböző típusú keresőket ezen osztályokat kiterjesztve hozhatjuk létre, így azok osztályaiban csak azokat az eszközöket kell deklarálni, melyek csak arra a konkrét keresőre vagy keresők egy csoportjára jellemzőek.

A *Kereso* egy absztrakt osztály, melynek a kereső-specifikus metódusait a leszármazott osztályokban kell implementálni. Csak azok a metódusok nem absztraktak, melyek minden egyes keresőben biztosan ugyanúgy kell, hogy működjenek. Ilyen például a keresés során talált terminális csúcsokat lekérdező metódus és a kereső jellemzőinek kiíratása. Ezek a jellemzők kapcsolóként működő adattagok, melyeket a kereső példányosításakor adhatunk meg a konstruktoroknak. Ezek mondják meg, hogy csak célállapot kerülhet-e be a terminális csúcsok halmazába, avagy az első célállapotot megtalálva a kereső folytassa-e a keresést jobb megoldások után kutatva.

A *Csucs* osztály tartalmazza az *allapot*, *szulo*, *operator*, *melyseg* adattagokat, melyeket a keresési fa nyilvántartására használunk. Az iteratívan javító kereső csúcspontja is lehet ezen osztály leszármazottja, legfeljebb az adattagok egy részét nem használjuk és értékük mindig *null*. Az osztályban ezen kívül helyet kaptak az adattagokhoz tartozó lekérdező metódusok, az összehasonlító és kiíratást szolgáló metódusok, illetve az osztály konstruktorai. [3] [5] [6]

2.5.4. A *backtrack* csomag

A *backtrack* csomag a *kereso* csomagon belül kapott helyet, ezzel is jelezve az általuk tartalmazott osztályok közötti logikai viszonyt. A *backtrack* csomag osztályai kiterjesztik a *kereso* csomag osztályait, létrehozva ezzel egy konkrét példányosítható és használható keresőosztályt.

Mivel ez a visszalépéses kereső nem használ keresőgráfot, ezért a csomópontokban tartja nyilván az állapotokra még nem alkalmazott operátorokat. Ehhez a *BacktrackCsucs* osztály bevezet egy új adattagot, a *nemAlkalmazottOperatorok* halmazt, amelyhez egy lekérdező metódus is tartozik, hogy a kereső lekérdezhesse egy adott csomópontra még nem alkalmazott operátorok halmazát.

A *BacktrackKereso* osztály az örökölt kapcsolók mellé két újat is felvesz, ezek a *korfigyeles* és az *uthosszkorlat* adattagok. Ezekkel a fogalmakkal már foglalkoztunk korábban. A kereső egy *backtrackCsucsok* nevű vermet használ az aktuális út nyilvántartására, valamint az osztály saját konstruktorokat is tartalmaz, hogy az új kapcsolók is beállíthatóak legyenek a kereső példányosításakor. A *keres* metódus a legfontosabb, hiszen a keresés ennek meghívásával történik. Vegyük szemügyre a metódus 19. ábrán látható kódját. Egy végtelen ciklus magjában zajlik a keresés, míg bizonyos feltételek mellett a metódus vissza nem tér egy *reurn* utasítás hatására. Az első *if* utasítás megvizsgálja, hogy üres-e a verem. Ha igen, az azt jelenti, hogy a gyökércsomópontra is alkalmaztuk már az összes alkalmazható operátort, és onnan is visszalépés történt. Ekkor a keresés befejeződik, ellenkező esetben következik a célvizsgálat. Amennyiben az aktuális csúcs nem célállapotot tartalmaz, vagy célállapotot tartalmaz, de az *osszesMegoldas* kapcsoló igaz, akkor a keresés folytatódik, de a második esetben előbb még az aktuális csomópont bekerül a terminális csúcsok halmazába. A következő *if* utasítás ellenőrzi a visszalépés szükségességét, majd ezen túljutva következik a mélységkorlát ellenőrzése. Végül a kereső alkalmazza az aktuális

állapotra az egyik még nem alkalmazott operátort, majd a *korfigyeles* kapcsolót és a verem tartalmát is figyelembe véve vagy elmenti a keletkezett csomópontot a verembe, vagy nem. Az aktuális csomópont *nemAlkalmazottOperatorok* halmazából az éppen alkalmazott operátor törlésre kerül, majd következik a ciklusmag legelső vizsgálata. [3] [5] [6]

```
public void keres() {
    while (true) {
        if (backtrackCsucsok.empty()) {
            return;
        }
        BacktrackCsucs aktcsucs = backtrackCsucsok.peek();
        Set<Operator> nemAlkalmazottOperatorok = aktcsucs.getNemAlkalmazottOperatorok();
        if (aktcsucs.getAllapot().isCelallapot()) {
            terminalisCsucsok.add(aktcsucs);
            if (!osszesMegoldas) {
                return;
            }
            backtrackCsucsok.pop();
            continue;
        }
        if (nemAlkalmazottOperatorok.isEmpty()) {
            backtrackCsucsok.pop();
            continue;
        }
        if (uthosszkorlat > 0 && aktcsucs.getMelyseg() == uthosszkorlat) {
            backtrackCsucsok.pop();
            continue;
        }
        if (!nemAlkalmazottOperatorok.isEmpty()) {
            Iterator iterator = nemAlkalmazottOperatorok.iterator();
            Operator operator = (Operator) iterator.next();
            BacktrackCsucs uj = new BacktrackCsucs(aktcsucs, operator);
            if (!(korfigyeles && backtrackCsucsok.contains(uj))) {
                backtrackCsucsok.push(uj);
            }
            aktcsucs.getNemAlkalmazottOperatorok().remove(operator);
        }
    }
}
```

19. ábra: a *BacktrackKereso* osztály *keres* metódusa.

2.5.5. A *keresografos* csomag

A csomag szintén a *kereso* csomagban foglal helyet. A csomagban található *KeresografosKereso* osztály egy keresőcsalád alapjait fekteti le. A *Kereso* osztályt egy *nyiltak* és egy *zartak* nevű listával egészíti ki, melyekkel a keresőfán bejárt csomópontokat tartják nyilván. A *BacktrackKereso* osztályhoz hasonlóan ez az osztály is bevezeti a mélységkorlátozást kapcsoló adattagot, melynek a nulla kezdőértékét szintén konstruktor segítségével bírálhatjuk felül. [6]

2.2.6. A *heurisztikus* csomag

A csomag a *keresografos* csomagban található, azaz az osztályai által megvalósított keresési mechanizmus a keresőgráfos családba tartozik, azon belül is egy még szűkebb családba. Ezek a heurisztikus keresők. A program egy legjobbat-először keresőt használ, ennél is pontosabban egy mohó keresőt, ahogy erről korábban már szó esett.

A *BestFirstCsucs* osztály egy *heurisztika* adattaggal egészíti ki a *Csucs* osztályt. Ebben az adattagban tárolódik el a kiértékelő függvény visszatérési értéke, amikor a konstruktorban meghívjuk a kiértékelő függvényt a csomópontban eltárolni készült állapotra.

A *BestFirstKereso* példányosításával létrehozott kereső tehát egy mohó kereső, amely mindig az éppen legkisebb heurisztikájú állapotot terjeszti ki. Valahogy azonban lehetővé kell tenni a receptek használatát is, mert ha a kereső így kezdene bele egy receptbe, ami kezdetben lerontja a heurisztikát, akkor a recept első tekerése után elkezdené helyrehozni a károkat egy alacsonyabb szinten. Ezt elkerülendő a keresőnek egy recept végrehajtása közben semmi másra sem szabad figyelnie. Ehhez a kereső egy operátorokat tartalmazó puffert alkalmaz, és az abban szereplő operátorokat egy ciklus segítségével minden esetben megszakítás nélkül üríti ki. A puffer csak üres állapotban írható, hogy a receptek ne váltsák ki futásuk közben egy másik recept végrehajtását. Amennyiben a puffer használatát mellőzzük, a kereső hétköznapi mohó keresőként viselkedik. Az osztály a szokásos kiíratő metódusokon kívül tartalmaz egy a puffer írására használható *addToPuffer* metódust és egy *getLegjobb* metódust is, ami a nyílt csomópontok listájából kikeresi a legkedvezőbb heurisztikával rendelkező csomópontotok egyikét. Továbbá a pufferhez egy lekérdező metódus is társul. A keresést végző *keres* metódus kódja a 20. ábrán tekinthető meg.

```
public void keres() {
    while(!nyiltak.isEmpty()) {
        aktualis = (BestFirstCsucs) nyiltak.get(getLegjobb(nyiltak));
        if(!puffer.isEmpty()) {
            zartak.add(nyiltak.remove(nyiltak.indexOf(aktualis)));
            while(!puffer.isEmpty())
                aktualis=new BestFirstCsucs(aktualis,puffer.remove(0));
            nyiltak.add(aktualis);
            continue;
        }
        if(aktualis.getAllapot().isCelallapot()) {
            terminalisCsucsok.add(aktualis);
            if(osszesMegoldas) {
                zartak.add(nyiltak.remove(nyiltak.indexOf(aktualis)));
                continue;
            }
            else
                break;
        }
        for(Operator op : Allapot.getOperatorok())
            if(aktualis.getAllapot().elofeltetel(op)) {
                BestFirstCsucs uj = new BestFirstCsucs(aktualis,op);
                 if(!nyiltak.contains(uj) && !zartak.contains(uj)) {
                    nyiltak.add(uj);
                 }
            }
        zartak.add(nyiltak.remove(nyiltak.indexOf(aktualis)));
    }
}
```

20. ábra: a *BestFirstKereso* osztály *keres* metódusa

A *while* utasítás feltételében jól látszik, hogy a keresés véget ér, ha elfogytak a kiterjesztésre várakozó csomópontok, igaz a Rubik-kocka problémájának esetében ez aligha fordulhatna elő. Viszont már tudjuk, hogy a kereső probléma-független kell, hogy legyen, ezért a vizsgálat elengedhetetlen. Az *aktualis* változóba mindig bekerül az éppen legjobb heurisztikával rendelkező csomópont, majd a puffer vizsgálata következik. Ha egy csomópont generálásakor a pufferbe operátorok kerültek, akkor az aktuális csomópont átkerül a nyíltak listájából a zártakéba. Ezután addig generáljuk az új csomópontokat a pufferben található operátorokkal, míg a puffer ki nem ürül, s amikor ez megtörtént a legvégül kapott csomópontot bekerül a nyíltak listájába. Ekkor a vezérlés visszakerül a ciklusmag legelejére. A következő körben a recept végén kapott csomópont állapota lesz a legkedvezőbb, így az lesz az aktuális csomópont. Ez azonban attól is függ, hogy mennyire jók a receptek. Egy jó recept sohasem növeli meg a megoldás hosszát, hanem csökkenti azt. Ha a puffer tartalmát vizsgáló *if* utasítás

feltétele hamis, azaz a puffer üres, akkor a célteszt következik. Ha az aktuális állapot nem célállapot, vagy ha célállapot, de az *osszesMegoldas* kapcsoló igaz, akkor a generálás során keletkezett csomópontok bekerülhetnek a nyíltak listájára, az aktuális csomópont pedig átkerül a zártakéba. Ellenkező esetben a keresés sikeresen véget ér. [3] [5] [6]

2.2.7. A *kep* csomag

Ezen csomag osztályai felelősek a felhasználói interfész megjelenítéséért. A *Polygon3D* osztály a *Polygon* osztályt terjeszti ki, mivel a *Polygon* osztály példányai csak kétdimenziós objektumok, a kocka megjelenítéséhez pedig háromdimenziósokra van szükség. Emellett a gyári *Polygon* osztály koordinátákat tároló adattagjai csak egész típusúak, márpedig egy ilyen jellegű feladat során a rengeteg koordináta-transzformáció miatt ez nem kielégítő. A kockát tízszer körbeforgatva már szemmel látható lenne a kerekítési hibák okozta torzulás. Mégis hasznosak az eredeti egész típusú koordináták is, mert használatukkal elérhető az összes gyári grafikus eszköz, mint a kirajzolás és a kiöntés is. Az új lebegőpontos és a teljesen új z koordináták miatt az osztálynak rendelkeznie kell saját koordináta-transzformációt elvégző metódusokkal is.

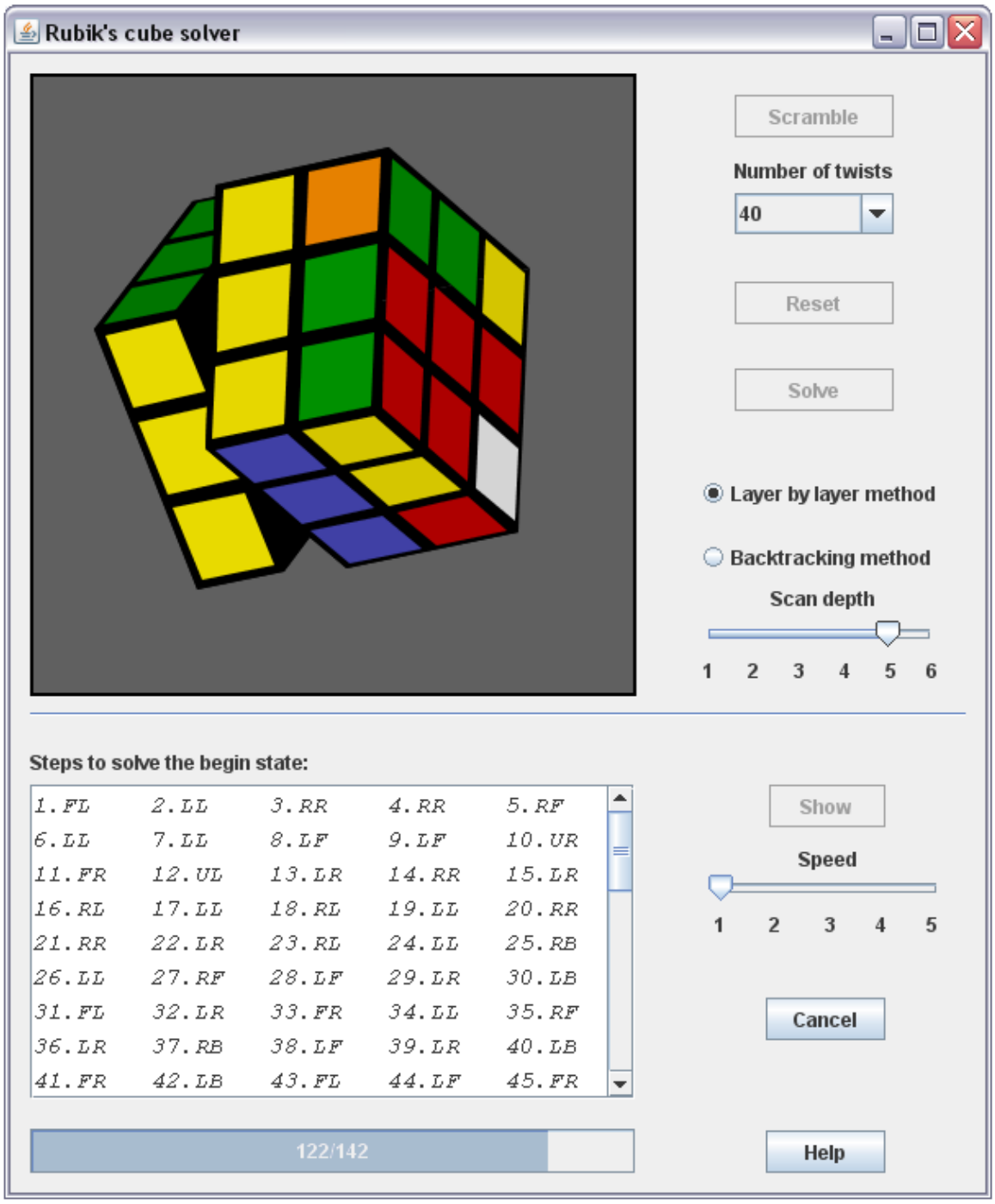
A háromdimenziós poligonokat a *Kep* osztályban használjuk fel. Ennek az osztálynak a példánya jeleníti meg a felhasználói interfész bal felső sarkában látható háromdimenziós képet. A képen látható kocka teljes mértékben a fent említett poligonok összessége. A kockát forgathatjuk és tekerhetjük közvetlenül az egér segítségével, azonban a felhasználói interfész elemei is manipulálják a képet.

Az osztály elején található a megjelenítéshez használt jó néhány adattag deklarációja. Ezt követi az ugyancsak hosszú inicializáló blokk, melyben a poligonokat tartalmazó adattagok kerülnek feltöltésre, így többek között létrejön a kockát megjelenítő poligonok sokasága is. A blokk végén a kezdeti elfogatásra és a képen megjelenített kocka állapotának inicializálására is sor kerül. A képen látható kockához tehát tartozik egy *KockaAllapot* típusú adattag is, melyből a kocka kirajzolásakor kiolvassuk a kocka lapocskáinak színét. A keresők és a kép kapcsolata egy operátor-puffer segítségével valósul meg, amibe a keresők által szolgáltatott megoldáshoz tartozó operátorsorozatot belemásoljuk. Az operátor-pufferhez tartozik egy metódus, mellyel operátorokat írhatunk bele, valamint egy *doPuffer* metódus is, amely egy külön szálat indítva végzi el a kocka megfelelő poligonjain a kellő koordináta-

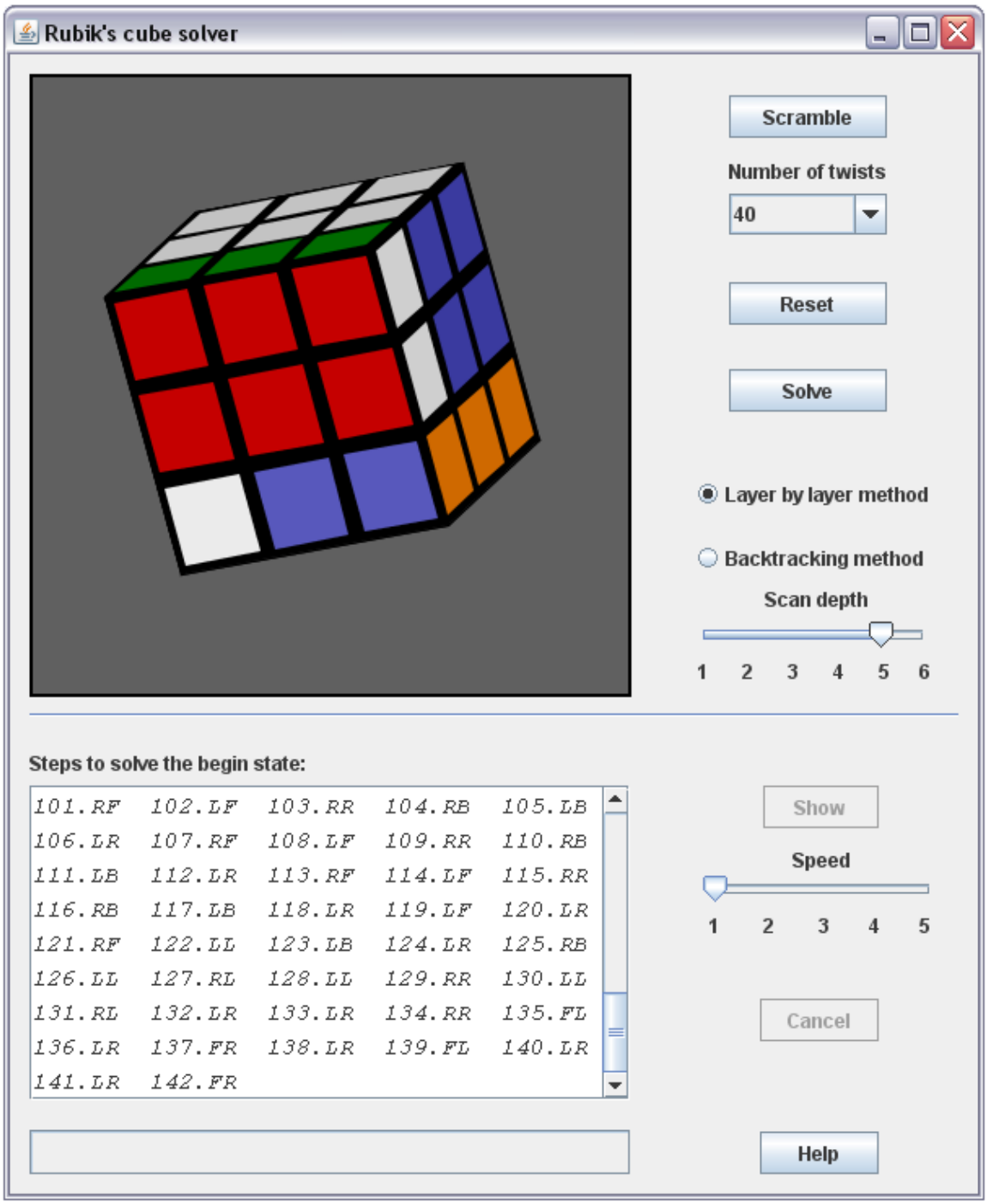
transzformációkat. Az egérrel történő forgatás és tekerés is egy ilyen szálon fut, hogy a keresők működése és a megoldás mutatása közben is használhassuk az egeret. Az inicializáló blokk után következik az osztály konstruktora, melyben az egérkezeléshez szükséges névtelen osztályokat adjuk hozzá az éppen példányosított képhez.

A *paint* metódus hívására rajzolódik ki maga a kép. A metódus lényegében a poligonok láthatóság-vizsgálataiból, a látható poligonok megfelelő sorrendben történő kirajzolásából és azok megfelelő színnel történő kiöntéséből áll. A színek megállapításakor figyelembe veszi a lap láthatóságának mértékét, és ez alapján sötétebb, illetve világosabb tónust használ a színezésre, hogy árnyékolási effektust érjen el. A metódus írásakor nem annak tömörségét tartottam a legfontosabbnak, hanem azt, hogy minél kevesebb láthatósági vizsgálatot kelljen elvégezni, mert ezek nagyon megterhelik a gépet. Bár a *lathatosag* metódus ránézésre egy-két műveletből áll ugyan, de gondolni kell arra is, hogy forgatáskor egy hosszabb egérmozdulat során a *paint* metódus akár több százszor is lefuthat egy másodperc alatt. Az osztályban található még egy *kever* és egy *reset* metódus is, amelyeket az interfészen látható „Scramble” és „Reset” gombok lenyomásakor hívunk meg.

A *Gui* osztály a *JFrame* osztályt terjeszti ki, és a program indításakor megjelenő felhasználói interfész ennek az osztálynak egy példánya. Egyik adattagja a bal felső sarkában megjelenített *Kep* típusú objektum, mely a kocka megjelenítéséért felelős. Az ablakban látható gombok, csúszkák, és egyéb vizuális eszközök is mind a *Gui* osztály adattagjai. Az egyes gombok lenyomásakor a gombhoz tartozó különböző típusú *Listener*-ek meghívják a *Kep* típusú adattag megfelelő metódusát, vagy a csúszkánkon beállított értékeknek megfelelően indítanak el szálakat. A „Help” gomb egy *JFrame* típusú tájékoztató ablakot jelenít meg. A „Solve” gomb lenyomásakor egy keresőszál indul, mely futtatja a kiválasztott típusú keresést, majd a kapott eredményt az egymást kioltó operátorsorozatoktól megtisztítva bemásolja a kép operátor-pufferébe. Emellett az eredményt is kiíratja a bal alsó sarokban látható *textField*-be. A „Scan depth” feliratú csúszkán a legnagyobb választható mélységkorlát a hatos, mert a visszalépéses kereső csak érdekességképpen szerepel a programban, és a b = 12-es elágazási tényezőjű Rubik-probléma esetében maximum 5-6-os mélységkorlát mellett szolgáltat értelmes időn belül megoldást. [6]



21. ábra: kép a programról



22. ábra: kép a programról

3. Összefoglalás

A szakdolgozat készítésekor elsődleges célom egy olyan program írása volt, mely képes megoldani a Rubik-kockát a korábban már tárgyalt rétegenként haladó módszer (layer by layer method) segítségével, méghozzá a módszer képességeihez mérten a lehető legkevesebb tekerés használatával. A módszert alkalmazó heurisztikus keresőn kívül a programban található egy visszalépéses kereső, melynek gyenge teljesítménye világít rá a heurisztikus függvény használatának szükségességére. A játék kategóriájú problémák közül nem véletlenül esett a kockára a választásom, hiszen annak grafikai megjelenítése is tartogatott kihívásokat, és ezen a téren érzésem szerint tovább jutottam, mint azt kezdetben gondoltam volna.

A szakdolgozat papír alapú része a megértést szolgálja. Ennek érdekében betekintést nyerhettünk a mesterséges intelligencia keresési módszerekkel kapcsolatos fejezeteibe, valamint megnéztük, hogyan épül fel a program, és mi célt szolgálnak annak különböző elemei. Emellett a könnyebb értelmezés érdekében szemléletes ábrákkal egészítettem ki a szöveges részt. Azonban annak megítélését, hogy céljaimat milyen mértékben sikerült elérnem, a felhasználóra, illetve az olvasóra bízom.

4. Irodalomjegyzék

[1] http://hu.wikipedia.org/wiki/Rubik-kocka,

CD-n tárolt változat: \Hivatkozások\Rubik-kocka - Wikipédia.htm

[2] http://en.wikipedia.org/wiki/Rubik's_cube,

CD-n tárolt változat: \Hivatkozások\Rubik's Cube - Wikipedia, the free encyclopedia.htm

[3] Stuart J. Russel, Peter Norvig Mesterséges Intelligencia Modern Megközelítésben. Panem Könyvkiadó Kft. Budapest, 2000

[4] http://www.inf.unideb.hu/~varteres/mi/part1/index.htm,

CD-n tárolt változat: \Hivatkozások\Mesterséges intelligencia 1.htm

[5] Alison Cawsey Mesterséges Intelligencia Alapismeretek. Panem Könyvkiadó Kft. Budapest, 2002

[6] http://java.sun.com/javase/6/docs/api/,

CD-n tárolt változat: \Hivatkozások\Java 6 API & Documentation\docs\api\index.html

[7] http://peter.stillhq.com/jasmine/rubikscubesolution.html,

CD-n tárolt változat: \Hivatkozások\LayerByLayerRubikSolution.pdf

5. Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Dr. Halász Gábor témavezetőmnek, aki szakmai tanácsaival, tapasztalataival segítette a szakdolgozat létrejöttét. Továbbá szeretném megköszönni Kósa Márknak és Ledeczky Gábornak, hogy átadták gyakorlati tudásukat a mesterséges intelligencia és a számítógépes grafika terén, mellyel megkönnyítették a munkámat.