Floppy-kocka szimulációja 16+8-bites ábrázolással

Dokumentáció Műszaki és Fizikai Problémák Számítógépes Szimulációja című tantárgy házi feladatához.

A programot megíró hallgató:

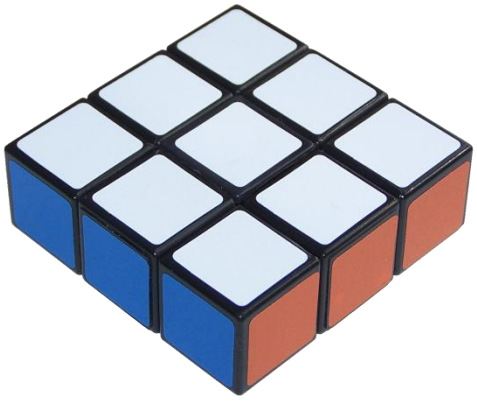
- neve: Koics Dániel
- neptun-kódja: D1V90V
- hallgatói státusza: elsőéves, aktív, alkalmazott fizikus mesterképzéses hallgató
- e-mail címe: koics.dani@gmail.com

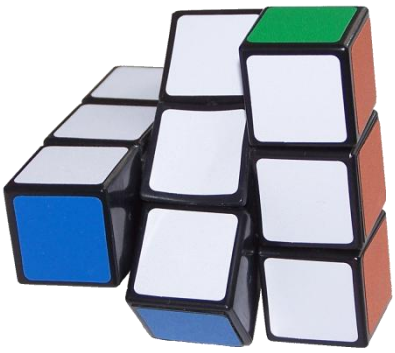
A feladat kiírása

A feladat egy olyan szoftver fejlesztése, mely a triviálisnál kisebb memóriaterületen szimulálja a 3x3x1-es méretű Rubik-kockát (más néven floppy-kockát). A szoftvernek tartalmaznia kell egy megoldót is, mely képes megállapítani, milyen műveletekkel vihető át két kockaállapot egymásba. A szoftvernek képesnek kell lennie a pillanatnyi kockaállapot, valamint a megoldói paraméterek és eredmények mentésére és újra betöltésére. Végül annak diszkutálása, hogyan lehetne a szoftvert továbbfejleszteni a hétköznapokban is megszokott 3x3x3-as Rubik-kockára.

A kitűzött problémáról hosszabban

A 3x3x1-es méretű Rubik-kocka modellezéséről van szó. Ez egy 9 elemből álló, négyzetes eszköz, mely oldalsó elemeinek az oldal középtengelye körüli 180°-os elforgatásával újabb és újabb állapotába vihető át:





Egy ilyen eszköz szimulációja legegyszerűbben úgy történhet, hogy az oldalak minden egyes négyzetfelületének különb memóriaelemet foglalunk, s ebben az elemben tároljuk az adott négyzet színét. Ekkor csak a színértékeket szükséges a memóriaterületek között cserélgetni. Azonban esetünkben egy másik módszer került alkalmazásra.

Megállapítható, hogy a színes négyzetek valójában 9 elem különböző oldalai. Az azonos elemeken szereplő négyzetek mindig együtt mozognak, ami a mozgásnak egyfajta kényszerfeltételt ad. Elméleti mechanikából megszoktuk, hogy ha különböző pontok mozgása között kényszerfeltételek vannak jelen, akkor a rendszer szabadsági foka lecsökken, és a pillanatnyi helyzetek leírására kevesebb paraméter (általános koordináta) is elegendő, mint amennyi a pontok összes helykoordinátájának száma. Arra a kérdésre kerestem a választ, hogy a floppy-kocka esetében mik ezek az általános koordináták.

Végül arra a következtetésre jutottam, hogy 4 oldalelem forgatásáról, valamint 4 sarokelem forgatva permutálásáról van szó, és a kocka állapotát érdemes 16 biten tárolni,

mégpedig a következőképpen: A 4 oldalelemet 1-1 bit jellemzi, mely megadja, hogy az adott sarok ellentétes oldalát mutatja-e a néző felé, mint amit a kocka kirakott állapotában mutat. A sarokelemeknek is van ugyanilyen forgatási bitjük, de ezen kívül rendelkeznek még 2-2 az eltoltságot jellemző bittel. Utóbbi bitek adják meg, hogy a sarok a kirakott állapotbeli pozíciójában (mindkét bit alacsony), valamelyik a kirakott állapotbelivel szomszédos (egy alacsony és egy magas érték), vagy esetleg a kirakott állapotbelivel szemközti (két magas érték) pozíciójában van-e.

A bitek állását az ún. kockamátrixszal érdemes szemléltetni:

$$\begin{array}{ccccc} & 0 & & 0 & \\ 0 & 0 & 0 & 0 & 0 \\ & 0 & & 0 & \\ 0 & 0 & 0 & 0 & 0 \\ & 0 & & 0 & \end{array}$$

Itt a középső, lyukas 3x3-as mátrix tartalmazza a forgatási biteket, míg a felső és alsó hiányos sor a függőleges, a bal és jobb oldali hiányos oszlop a sarokelemek vízszintes irányú eltoltságát kódolják.

A fenti, csak alacsony értékeket tartalmazó kockamátrix a kocka kirakott állapotát jelenti. Ez az ábrázolásmód a kocka belső állapotát adja meg, s nem törődik a kocka háromdimenziós térbeli helyzetével, hiszen a két dolog egymástól független, jóllehet egy kirakott kocka bármilyen módon elforgatva kirakott marad. (A térbeli helyzetet a programban egy külön tájolási mátrix rögzíti.) Bár a 16 bit összesen 2^{16}=65536 állapotot tesz lehetővé, látni fogjuk, hogy a floppy-kockának ennél lényegesen kevesebb állapota lehetséges. (Szabályosan manipulált kocka esetében 192, míg elemeire bontható és tetszés szerint összerakható kocka esetében 32x192=6144 állapot járható be.) Vagyis az így kapott ábrázolásmód még mindig meglehetősen túlhatározott. Ennek ellenére elmondhatjuk, hogy sikerült memóriát megtakarítani, hiszen 30db 6-állapotú színváltozó ($30x\log_2 6 \sim 30x3$=90 bit) helyett elég 16 db logikai változót használni.

Végeredményben elmondhatjuk, hogy sikerült egy olyan szoftvert írni *MATLAB* nyelven, mely a 3x3x1-es Rubik-kockát nemcsak szimulálja 16 biten, hanem képes próbálgatással meghatározni, hogyan lehet eljutni a kocka egy adott kiindulási állapotából egy előre meghatározott célállapotába vagy állapothalmazába. (A megoldó programrész határozatlan célállapota miatt be kellett vezetni egy-egy újabb bitet mind a 8 elemhez, így a kockaobjektum memóriaterülete 24 bitre nőtt.) A szoftver tartalmaz olyan tesztfüggvényeket, melyek képesek eldönteni, hogy a kitűzött probléma megoldható-e. E tesztfüggvények megírásához meg kellett keresni a 16-bites ábrázolásmódot terhelő kényszerfeltételeket, és fel kellett térképezni a bejárható állapotcsoportokat. Jelen dokumentáció célja részben ezen vizsgálódás eredményeinek bemutatása (erről a következő szakaszban van szó).

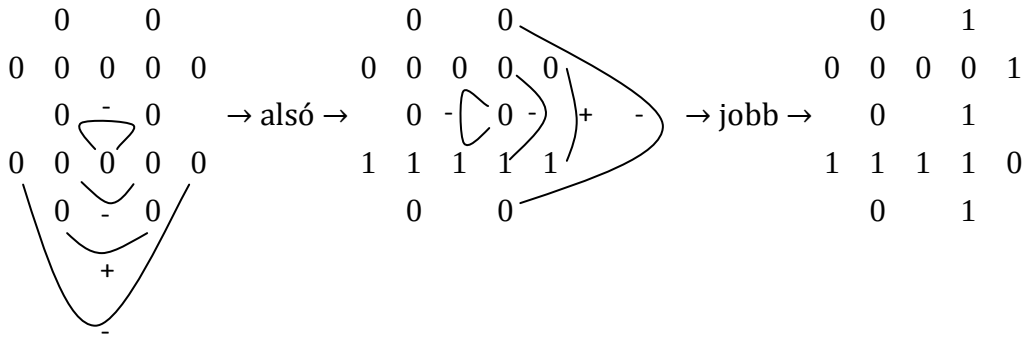
Az elméleti megfontolások

A kocka 4-féle művelettel manipulálható szabályosan: A bal, a lenti, a jobb és a fenti oldal 180°-os átfordításával. A lenti oldal forgatása a következőképp hat a kockamátrixra:

$$\begin{array}{ccccc} & x & & x & \\ x & x & x & x & x \\ & x & & x & \\ a & b & c & d & e \\ & f & & g & \end{array} \rightarrow \begin{array}{ccccc} & & x & & \\ x & & x & & x \\ & & x & & \\ neg(e) & neg(d) & neg(c) & neg(b) & neg(a) \\ & & g & & f \end{array}$$

Látható, hogy a két alsó hármas felcserélődik, miközben az alsó elemek forgatottságát és vízszintes eltoltságát jelző bitek negálódnak. Gyakorlatilag a mátrix alsó két sora tükröződik, s az alsó teljes sor emellett még negálódik is. Hasonlóképp a többi oldal csavarása a mátrix megfelelő oldalára hat.

Példa:



A szemléletesség kedvéért összekötöttem azokat az elemeket, melyek felcserélődnek, s az összekötő vonalakon - jelzi a negálásos, + pedig a negálás nélküli cserét.

Ami a kirakhatóság tesztelését illeti: Felvetődik a kérdés, hogy nem létezik-e egy (a kockamátrixból számítható) mennyiség, mely a szabályos műveletek során megmarad, de szabálytalan művelet esetén megváltozik. Akkor ugyanis csak azt kéne ellenőrizni, hogy a mennyiség tényleges értéke megegyezik-e a kirakott kocka megfelelő értékével. (Hasonló lenne ez, mint amikor egy mátrix invertálhatóságát a determináns alapján ellenőrizzük.) Végül sikerült olyan tesztfüggvényeket írni, melyek megmaradó logikai változókat adnak eredményül, s az összes változót együtt tekintve már megállapítható, hogy a kocka kirakható-e.

A tesztek megértéséhez két dolgot kell megfontolni. Az egyik az, hogy az egyes sarokelemek mozgásuk során mindig forgatottságot váltanak. Mivel a forgatottság kétféle lehet, és páros számú (4) oldalél van, elmondható, hogy a csúcsok forgatottsága a pozíciójuk egyértelmű függvénye: Alacsony a forgatási bit, ha az elem kirakott állapot szerinti vagy azzal szemközti pozícióban van (a két eltolási bit megegyezik). Magas a forgatási bit, ha a

sarokelem a kirakott állapot szerintivel szomszédos pozícióban van (a két eltolási bit eltérő). Vagyis ha megvizsgáljuk mind a négy sarokelemre, hogy teljesíti-e ezt a törvényt, akkor ki tudjuk szűrni a kirakhatatlan kockaállapotok egy részét. Az ezen az elven működő tesztet nevezem a későbbiekben sarok-tesztnek.

A másik megfontolás a következő: A kocka mozgatása közben a sarokelemek felcserélődnek, négy elem különböző permutációiról van szó. Mint matematikából ismeretes, a permutációknak van egy ún. paritásuk, mely két elem felcserélésekor (vagyis a kockajáték minden egyes lépésekor) mindig ellentétesre változik. Ugyanakkor, ha megnézzük a felfordított oldalelemek számának paritását, arról is azt mondhatjuk, hogy minden egyes lépésben ellentétesre változik. Ha tehát a kirakott kocka sarokpermutációjának paritását tekintjük párosnak, akkor elmondható, hogy egy kirakható kockában a sarokpermutáció és a magas forgatási bitű oldalelemek számának paritása meg kell, hogy egyezzen. Ha a kétféle paritás eltérő, a kocka biztosan nem rakható ki. Ezt nevezem paritás-tesztnek.

Felvetődik a kérdés, hogy a kétféle teszt elegendő-e ahhoz, hogy eldöntsük egy kocka kirakhatóságát, vagyis ha átmegy a kocka e teszteken, akkor elmondhatjuk-e, hogy kirakható. Más gondolatmenettel: A paritás-teszt és a négy sarokra vonatkozó sarok-teszt a kocka összes lehetséges (szabályosan nem feltétlenül elérhető) állapotait $2^{1+4}=32$ csoportra osztja (ennyi a lehetséges együttes kimenetelek száma). A kérdés az, hogy a kirakott állapotból kiindulva az azt tartalmazó csoport a szabályos lépések által bejárható-e. Nos, a válasz igen.

Ennek belátása: A sarkak permutációja olyan, mintha egy 2x2-es mátrixot 1-től 4-ig feltöltenénk egész számokkal, s az azonos sorba ill. az azonos oszlopba tartozó számokat cserélgetnénk. Annak ellenére, hogy nem cserélhető meg bármelyik két elem, mégis mind a 4!=24 permutáció előállítható. Jelölje ugyanis a felső sor két elemének cseréjét *f*, az alsó sorét *a*, a bal oszlopét *b*, a jobb oszlopét pedig *j*. Ekkor:

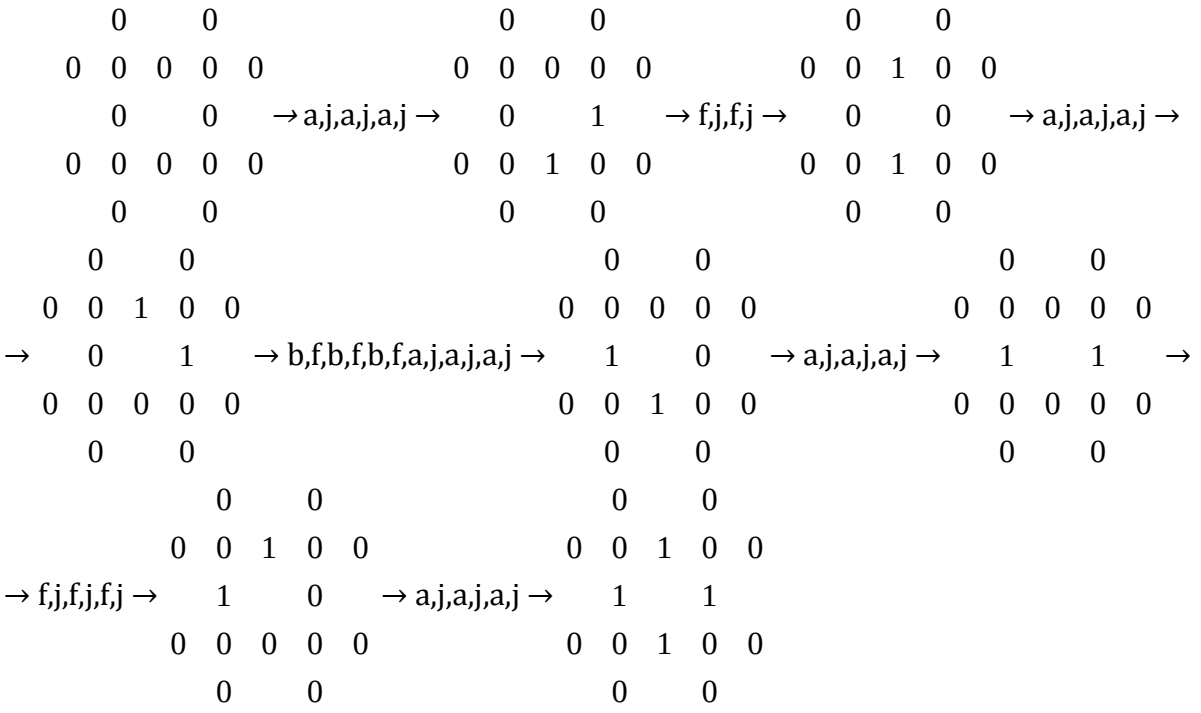
$$\begin{matrix}1&2\\3&4\end{matrix}\xrightarrow{a}\begin{matrix}1&2\\4&3\end{matrix}\xrightarrow{j,a}\begin{matrix}1&3\\2&4\end{matrix}\xrightarrow{a}\begin{matrix}1&3\\4&2\end{matrix}\xrightarrow{a,j}\begin{matrix}1&4\\2&3\end{matrix}\xrightarrow{a}\begin{matrix}1&4\\3&2\end{matrix}\xrightarrow{j,f}$$

$$\xrightarrow{}\begin{matrix}2&1\\3&4\end{matrix}\xrightarrow{a}\begin{matrix}2&1\\4&3\end{matrix}\xrightarrow{j,a}\begin{matrix}2&3\\1&4\end{matrix}\xrightarrow{a}\begin{matrix}2&3\\4&1\end{matrix}\xrightarrow{a,j}\begin{matrix}2&4\\1&3\end{matrix}\xrightarrow{a}\begin{matrix}2&4\\3&1\end{matrix}\xrightarrow{j,b}$$

$$\xrightarrow{}\begin{matrix}3&1\\2&4\end{matrix}\xrightarrow{a}\begin{matrix}3&1\\4&2\end{matrix}\xrightarrow{j,a}\begin{matrix}3&2\\1&4\end{matrix}\xrightarrow{a}\begin{matrix}3&2\\4&1\end{matrix}\xrightarrow{a,j}\begin{matrix}3&4\\1&2\end{matrix}\xrightarrow{a}\begin{matrix}3&4\\2&1\end{matrix}\xrightarrow{f,j}$$

$$\xrightarrow{}\begin{matrix}4&1\\2&3\end{matrix}\xrightarrow{a}\begin{matrix}4&1\\3&2\end{matrix}\xrightarrow{j,a}\begin{matrix}4&2\\1&3\end{matrix}\xrightarrow{a}\begin{matrix}4&2\\3&1\end{matrix}\xrightarrow{a,j}\begin{matrix}4&3\\1&2\end{matrix}\xrightarrow{a}\begin{matrix}4&3\\2&1\end{matrix}$$

A sarkak forgatottságával nem kell foglalkoznunk, mivel azt a sarkak pozíciója a paritás-tesztnél elmondottak miatt egyértelműen meghatározza. Ami az oldalelemek forgatottságát illeti: A 4 bitnek összesen 2^4 állapota létezik, de a paritás-tesztnél elmondottak megszorítást jelentenek rájuk nézve: Ha három elem forgatottságát ismerjük, a negyedik már csak egyféleképp képes kielégíteni a paritás-tesztet. Így végül is csak $2^3=8$ állapotuk lehetséges egy adott sarokpermutáció mellett. A továbbiakban már csak azt kell bizonyítani, hogy e 8 állapot elérhető-e egymásból kiindulva. Jelölje az alsó oldal átfordítását *a*, a felső oldalét *f*, a jobb oldalét *j*, a bal oldalét *b*. Ekkor:



Ez a módszer alkalmazható tetszőleges sarokpermutáció mellett. Látható, hogy az egy sarokpermutációhoz tartozó mind a 8 oldalállapot egymásból elérhető.

Ezzel nemcsak azt bizonyítottuk, hogy a sarok- és a paritás-teszt elégséges a kirakhatóság eldöntéséhez, de azt is megmutattuk, hogy a floppy-kockának $4!*2^3=192$ megengedett állapota van. Sőt ha veszünk két tetszőleges, nem feltétlenül megengedett állapotot, azt is meg tudjuk mondani, hogy szabályos lépésekkel a két állapot elérhető-e egymásból, hiszen csak azt kell ellenőrizni, hogy a tesztek ugyanazt az eredményt szolgáltatják-e mindkét állapotra. Azt kaptuk, hogy a kocka állapotai 32 csoportba oszthatóak, melyek közül mindegyik 192 elemet tartalmaz. A szabályos műveletekkel egy csoporton belül bárhova eljuthatunk, de két csoport között csak akkor van átjárás, ha a kockát szétszedjük. Kirakott állapotot csak egyetlen csoportban találunk.

A 16 bites ábrázolásmódnak ennél jóval több, $2^{16}=65536$ állapota lehetséges. Ez azt jelenti, hogy 2^{16}-16*192=62464 állapot olyan esetnek felel meg, amikor a kockát szétszedték, s összerakás előtt a sarokelemeket más szétszedett kocka sarkaival összekeverték, így a kocka nem tartalmaz 4 különböző sarokelemet, hanem bizonyos sarkak ismétlődnek.

Nincs jelentősége, de fizikus szemszögből nézve érdekes a következő megmaradó mennyiség: Mivel két sarok cseréjekor elmozdulásvektoruk ellentett vektor, a sarkak helyvektorának összege – egy zárt mechanikai rendszer tömegközéppontjának impulzusához hasonlóan – megmaradó mennyiség. Ez azt jelenti, hogy ha egy négy különböző sarokelemet tartalmazó kocka mátrixában az x és y irányú eltolásbiteket megfelelő előjellel összegezzük, 0-t kell kapnunk. Ezt beépítettem a sarkak különbözőségét tesztelő *IsLapos()* függvénybe a kockaobjektum osztálydefiníciójában (*LAPOS.m*).

A programkód felépítése

A programkódot felépítő elemek vázlatosan:

- *LAPOS.m*: A floppy-kocka objektumtípusát leíró osztálydefiníció
  Ez a sarok és oldalelemeken, mint belső objektumokon kívül tartalmaz:
  - konstruktort
  - oldalforgató függvényeket (*fenti(),lenti(),bal(),jobb*) és az azokat véletlenszerűen meghívó keverő függvényt (*kever()*)
    A keverő függvény argumentumként egy természetes számot vár, mely megadja, hogy hány lépésnyi keverés történjen, s visszatérési értékként adja az összekevert kockát.
    Mivel ez az osztály nem alosztálya a *handle* osztálynak, a tagfüggvények az eredeti objektum másolatával dolgoznak, így lefutáskor nem képesek átállítani a kérdéses változókat, csupán olyan objektummal térnek vissza, amelyben a kérdéses változók már át vannak állítva.
  - tesztfüggvényeket
    - *IsDefinit()*
      Azt ellenőrzi, hogy minden elem határozott-e (az összes *definit* változó magas-e).
    - *CanbeLapos()*
      Azt ellenőrzi, hogy a határozott elemek alapján lehet-e különböző a 4 sarokelem.
    - *IsLapos()*
      Annak ellenőrzése, hogy a kocka minden eleme definiált-e és mind a 4 sarokelem különböző-e. Meghívja az *IsDefinit()* függvényt, s ha az igazat ad, egy sajátos, tömegközéppont megmaradási törvény által ihletett módszerrel (lásd: Elméleti megfontolások) ellenőrzi a sarkak különbözőségét.
    - *CornerTest()*
      Ez a korábban említett sarok-teszt, vagyis annak ellenőrzése, hogy a sarokelemek forgatottsága és pozíciója között megbomlik-e az összhang. Ennek megfelelően egy 2x2-es logikai mátrixszal tér vissza, ahol az alacsony érték jelent összhangot, a magas pedig problémát. A logikai értékek mátrixban való elhelyezkedése a sarokelemek kirakott kockabeli pozíciójára utal. Csak akkor szabad parancssorból meghívni, ha meggyőződtünk róla, hogy az *IsLapos()* függvény magassal tér vissza, ellenkező esetben használata hibás következtetésekhez vezethet.
    - *UCornerTest()*
      Ez a sarok-teszt határozatlan kockák esetén is használható változata. Logikai helyett 8 bites integerekből álló mátrixszal tér vissza, és ha határozatlan sarokelemet talál, azt -1 értékkel jelzi. Az előző

függvényhez hasonlóképp ezt a függvényt csak akkor hívjuk meg, ha a *CanbeLapos()* függvény igazzal tér vissza.

- *ParityTest()*
  Paritás-teszt, vagyis annak ellenőrzése, hogy a sarokelemek permutációjának és az átfordított oldalelemek számának paritása eltérő-e. Ezt a függvényt is csak akkor szabad a parancssorból meghívni, ha az *IsLapos()* függvény magas értéket ad eredményül.
- *UParityTest()*
  Az előző teszt határozatlan kockára is alkalmazható változata. A visszatérési érték 8 bites egész, 1 az éltérő (hibás), 0 az azonos (helyes) paritást, -1 pedig határozatlan oldalalelem jelenlétét jelzi. Természetesen használat előtt meg kell róla győződni, hogy *CanbeLapos()* visszatérési értéke igaz-e.
- *CanbeSolvable()*
  Annak ellenőrzése, hogy a kocka határozott elemei alapján lehet-e kirakható. Meghívja a *CanbeLapos()* függvényt, és ha az magassal tér vissza, akkor az *UCornerTest()* és az *UParityTest()* függvényt is, s azok kimenete alapján dönt. A magas visszatérési érték jelzi azt, hogy a kocka lehet kirakható.
- *IsSolvable()*
  Annak ellenőrzése, hogy a kocka minden eleme határozott, és a kocka kirakható-e. Meghívja az *IsLapos()* függvényt, és ha az igazzal tér vissza, akkor a *CornerTest()* és a *ParityTest()* Függvényt is. (Minden elemében határozott, 4 különböző sarokelemet tartalmazó kocka pontosan akkor rakható ki, ha *CornerTest()* eredményeképp kapott mátrix minden eleme és a *ParityTest()* eredménye is alacsony. Ekkor az *IsSolvable()* függvény magasat ad vissza.)
- *CanbeSolved()*
  Annak ellenőrzése, hogy a határozott elemek a kirakott állapot szerinti pozíciójukban és forgatottsági bitekkel szerepelnek-e.
- *IsSolved()*
  Ez a függvény csak és kizárólag a minden elemében határozott, kirakott kockára ad magas értéket.

- kockamátrixot kiíró függvényt és 3D-s vizualizáló függvényt
  - *WriteM()*
    A kockamátrixot parancssorba vagy fájlba író, visszatérési érték nélküli függvény. Paraméterként egy fájl azonosító egész számot vár. Az 1 mindig a parancssort jelenti, ha nem adunk meg számot, ez az alapértelmezett. A kezelőobjektum mentő függvénye is meghívja ezt a függvényt.
  - *Visual3()*
    A kockát a *surf()* függvény segítségével ábrázoló függvény. Szabályos tájolás esetén a kocka fektetve jelenik meg, egységnyi tájolási mátrix és kirakott kocka esetén a teteje sárga, az alja (nem látható) fekete, a

szemlélő felé néző, jobb oldali 1x3-as oldallap kék, a bal oldali zöld, a zölddel szemközti (nem látható) lap narancssárga, a kékkel szemközti pedig vörös.
A függvény mind a 6 oldallaphoz külön meghívja a *surf()* függvényt, amihez a szüksége, x, y, és z koordinátákat tartalmazó mátrixokat egy-egy dupla for ciklussal állítja elő.

- csoportműveleteket megvalósító függvényeket
A kocka állapotai azonosíthatók a kirakott állapotot az adott állapotba vivő hatással, s így a hatások egymás utáni alkalmazására, mint a kockaállapotok egyfajta szorzására nézve csoportot kapunk. A csoport egy olyan egy műveletes algebrai struktúra, melyben az egyetlen (általában szorzásnak nevezett, mindig asszociatív, de nem feltétlenül kommutatív) műveletre nézve létezik neutrális elem, mellyel akármelyik elemet akár jobbról, akár balról megszorozva a kérdéses elem maga lesz a szorzás eredménye, továbbá minden elemnek van inverze, mellyel összeszorozva a neutrális elemet kapjuk eredményül. A pozitív számok szorzása esetében a neutrális elem az 1, az inverz pedig a reciprok. A Rubik-kocka esetében neutrális a kirakott állapot, inverz pedig az az állapot, amibe a kockát kirakó hatás a már eleve kirakott kockát viszi. E műveletek a mátrixok szorzására és invertálására hasonlítanak, ezért a mátrixoknál használt *, /, \ és ^ operátorokat újradefiniáltam a *LAPOS* osztály számára.
A definiált csoportműveletek (a szimulátor és a megoldó szkript nem használja ezeket):
  - *C=mtimes(A,B)*
  A * operátorral is meghívható: *C=A*B*
  A *B* állapotra hattatja a kirakott kockát *A*-ba vivő hatást, s az eredményül kapott *C* állapottal tér vissza.
  - *C=mrdivide(A,B)*
  A / operátorral is meghívható: *C=A/B*
  Megoldja *C*-re az *A=C*B* kocka-egyenletet.
  - *C=mldivide(A,B)*
  A \ operátorral is meghívható: *C=A\B*
  Megoldja *C*-re a *B=A*C* kocka-egyenletet.
  - *C=inv(A)*
  Az *A* állapot inverzét adja eredményül a *C* kockában.
  - *C=mpower(A,N)*
  Operátorral: C=A^N
  Az A kockaállapot N-szer ismételt szorzása.

  Persze ezek a műveletek csak akkor működnek helyesen, ha a kockák minden eleme határozott, és 4 különböző sarokelemet tartalmaznak (vagyis ha az *IsLapos()* függvény igaz értékkel tér vissza).

- *VIS_LAP.m*: Osztálydefiníció a kezelőobjektumhoz
A kezelőobjektum tartalmaz egy floppy-kocka objektumot (*Kocka*), valamint egy tájolási mátrixot (egy *Nezet* nevű, 3x3-as méretű ortogonális mátrix), mely a kocka

térbeli forgatottságát hivatott megadni. Ezen kívül találunk itt egy logikai változót, mely megadja, hogy a kocka állapota és a nézet az utolsó ábrázolás óta módosult-e (*Changed*).
Ez az objektum biztosítja a kapcsolatot a felhasználó külső koordináta-rendszere és a kocka belső mátrixa között, így oldalforgató függvényei ennek megfelelően hívják meg a kocka saját oldalforgató függvényeit.
A kezelőobjektum osztálya a *MATLAB* beépített *handle* osztályának alosztályaként lett definiálva, így tagfüggvényei meghíváskor referenciát (és nem másolatot) kapnak az objektumról, és az objektum belső változóinak módosulásához nincs szükség arra, hogy a tagfüggvények visszatérési értékként megadják a módosított objektumot.
A tagfüggvények:

- konstruktor
- oldalforgató, nézetforgató, keverő és alapállapotba állító függvények
  - Az oldalforgató függvények (*elso(),hatso(),bal(),jobb()*) a kocka középpontjából a megfelelő oldallap felé, az oldallapra merőlegesen mutató egységvektorra hattatják az tájolási mátrix inverzét (transzponáltját), s az így kapott egységvektor iránya alapján döntik el, hogy a kockaobjektum melyik oldalforgató függvényét hívják meg.
  - A keverő függvény (*kever()*) bemenő argumentumként a keverési lépések számát várja, és azt továbbadja a kockaobjektum saját keverőfüggvényének.
  - A nézetforgató függvények a tájolási mátrixot változtatják. Közülük *X()* az egyik, *Y()* a másik vízszintes tengely körül 180°-kal, *Zp()* és *Zm()* pedig a függőleges tengely körül 90°-kal egyik illetve másik irányba forgatja el a kockát.
  - A helyreállító (*reset()*) függvény alapállapotba helyezi mind a kockamátrixot, mind a tájolási mátrixot.
- a tájolást leíró mátrix ortogonalitását és a tájolás használhatóságát ellenőrző függvények
  - *IsOrtog()*: Ez a függvény a tájolási mátrix ortogonalitását ellenőrzi.
  - *IsLapos()*: Ez a függvény azt vizsgálja meg, hogy az oldalforgató függvények az adott tájolási mátrix mellett működőképesek-e. (Vagyis itt kipróbálásra kerül mind a 4 egységvektor.)
- a kocka mátrixkiíró és vizualizáló függvényét a megfelelő paraméterekkel meghívó függvény (*Visual()*)
  Ez meghívja mind a kocka mátrixkiíró (*WriteM()*), mind a vizualizáló (*Visual3()*) függvényét, utóbbit csak akkor, ha az utolsó ábrázolás óta történtek módosítások. Ennek jelzésére a kezelőobjektum *Changed* nevű logikai változója szolgál, melynek értékét az oldalforgató és a nézetet állító függvények magasra, a *Visual()* függvény pedig alacsonyra állítja.
- mentő és betöltő függvény (*Ment(),Tolt()*)
  Ezek a függvények argumentumként a fájlra hivatkozó sztringet várják, s visszatérési értékük logikai, mely akkor alacsony, ha sikerült a művelet. A

mentő függvény használja a kockaobjektum mátrixkiíró (*WriteM()*) függvényét is.

- *SOL_LAP.m*: Osztálydefiníció a megoldó objektumhoz
  Ez az objektumtípus két floppy-kocka objektumot is tartalmaz, egyik a kezdeti, másik a célállapotot testesíti meg. Ezen kívül itt is található egy tájolási mátrix, valamint további változók.
  A kezdő és végállapotok szabályos, oldalforgató manipulációján kívül lehetőség van szabálytalan manipulációra is. Gyakorlatilag lehetőség van a kocka bármely sarkába bármelyik sarokból származó elemet beilleszteni, és tetszés szerint megadható mind a sarokelemek, mind az oldalelemek forgatottsága. Olyan ez, mintha a kockát elemeiből építenénk fel. A célállapot esetében arra is lehetőség van, hogy egyes sarkakat ne adjunk meg, vagy ne adjuk meg egyes oldalelemek forgatottságát. Ilyenkor a megoldó algoritmusok ezekre az elemekre nem végeznek ellenőrzést, s így célállapotként tulajdonképpen több állapot unióját adhatjuk meg. Evégett mind a sarok-, mind az oldalelemek rendelkeznek egy definiáltsági bittel is, és az ábrázolás így 16 helyett valójában 24 bites. Annak biztosítása érdekében, hogy a felhasználó meggyőződhessen róla, hogy valóban azt csinálja-e, amit szeretne, mind a kezdeti, mind a végállapotot tartalmazó kockaobjektumból van egy átmeneti másolat.
  Az osztály rekurzív megoldó függvényeket használ. A túl hosszú ideig tartó és túl sok stack-szintet felölelő rekurziók elkerülése végett előre meg kell adni korlátozásokat a maximális lépésszámra és a maximális futási időre, továbbá a futási időre előzetes becslés kérhető, és a megoldó függvények lefutása során a futási idő is tárolásra kerül.
  Ez az osztály is alosztálya a *handle* osztálynak.
  A változók teljes listája:
  - *Alfa*: Kezdeti állapot kockaobjektuma.
  - *Omega*: Célállapot kockaobjektuma
  - *Nezet*: Tájolási mátrix
  - *StepMax*: Maximális lépésszám a megoldáshoz
  - *TEstimateFcn*: Inline függvény a futási idő előzetes becsléséhez
  - *TimeMax*: Időkorlát a megoldó függvények futásához
  - *AlfaTemp*: Átmeneti kezdőállapot kockaobjektum a szabálytalan manipulációkhoz
  - *ATmpTovalidate*: logikai változó annak jelzésére, hogy van érvényesíthető átmeneti kezdőállapot
  - *OmegaTemp*: Átmeneti célállapot kockaobjektum a szabálytalan manipulációhoz
  - *OTmpTovalidate*: Logikai változó annak jelzésére, hogy van érvényesíthető átmeneti célállapot
  - *AlfaChanged*: Logikai változó annak jelzésére, hogy az utolsó vizualizáció óta módosult a kezdőállapot
  - *OmegaChanged*: Logikai változó annak jelzésére, hogy az utolsó vizualizáció óta módosult a célállapot
  - *Solved*: Logikai változó annak jelzésére, hogy az utoljára beállított paraméterek mellett lefutottak a megoldófüggvények

- *Solution*: Cella tömb a megoldások listázására
- *NSol*: A talált megoldások száma
- *Time*: A megoldás során ténylegesen mért futási idő

A tagfüggvények:

- Konstruktor, mely a *Timer.mat* fájlból betölti az időbecsléshez szükséges inline függvényt.
- A tájolást állító függvények (*X(), Y(), Zp(), Zm()*), és a tájolási mátrix használhatóságát ellenőrző függvények (*IsOrtog(), IsLapos()*)
  Ugyanaz a szemlélet, mint a kezelő osztályában. Itt azonban két kocka van, de csak egyetlen tájolási mátrix, így a tájolás változtatásakor a két kockát egyszerre látjuk elfordulni.
- A kezdő és a végállapotot szabályosan manipuláló függvények (*AlfaElso(), AlfaHatso(), AlfaBal(), AlfaJobb(), OmegaElso(), OmegaHatso(), OmegaBal(), OmegaJobb()*)
  Itt is ugyanaz a szemlélet, mint a kezelőobjektum esetében, csak itt külön függvények szolgálnak a kezdőállapot és a végállapot kockájának manipulációjára.
- A kezdő és a végállapotot szabálytalanul manipuláló függvények
  Ilyen jellegű függvényeket az egyszerű kezelő nem tartalmaz.
  - *AddtoAlfa(), AddtoOmega()*:
    Ezek a függvények a kezdőkocka ideiglenes módosítására, és a módosítások *AlfaTemp,* illetve *OmegaTemp* nevű kockaobjektumba mentésére szolgálnak. Egy *where* nevű vektort, és egy *what* nevű sztringet várnak argumentumként.
    A *where* vektort a tájolási mátrix transzponátjával megszorozva kapják meg a vektort, mely elárulja, hogy a kockamátrix melyik elemét kell módosítani. Oldalelemre koordinátatengely (±X vagy ±Y) irányú egységvektorokkal, sarokelemre az XY síkban lévő, a koordinátatengelyekkel 45°-os szöget bezáró, $\sqrt{2}$ hosszúságú vektorral hivatkozunk.
    A *what* nevű sztring értéke oldalelemek esetében *'def'* vagy *'roll'* lehet. Előbbi az adott oldalelem határozottá tételét, utóbbi az elem megfordítását végzi el.
    Sarokelemek esetében a *what* sztring lehetséges értékei: *'rg', 'gb', 'bo', 'or'* és *'roll'*. Az első négy különböző sarokelemek beillesztésére szolgál, és a betűk az oldalirányba néző lapok színeire utalnak. A kockamátrix bal oldala a kék, jobb oldala a narancssára, teteje vörös, alja pedig zöld. (Így pl *'rg'* az egyik oldalán vörös, a másikon zöld (a mátrixban eredetileg bal felső), *'bo'* pedig az egyik oldalán kék, a másikon narancssárga (a mátrixban eredetileg jobb alsó) sarokelemet jelenti.) A *'roll'* a már meglévő sarokelem megfordítására szolgál.
    Az AddtoOmega() függvény esetében a *what* paraméternek van még egy lehetséges értéke, mely mind sarok-, mind oldalelemek manipulációjára használható. Ez pedig az *'udef'*, mellyel az adott elem

határozatlanná tehető.
Az *AddtoAlfa()* függvény magassá teszi az *ATmpTovalidate* nevű logikai változót, ezzel jelezve, hogy van véglegesítésre váró átmeneti kezdőállapot. Hasonlóképp az *AddtoOmega()* függvény pedig az *OTmpTovalidate* változó értékét teszi igazzá. Mindkét függvény azonnal meghívja a *Visual3()* függvényt az átmeneti kockaállapotra, s a megjelenő ábra címsorában jelzi, hogy csak átmeneti állapotról van szó.

- *ValidateAlfa(), ValidateOmega()*:
  Ezek a függvények az átmeneti kezdő- és célállapotok véglegesítésére, illetve elvetésére szolgálnak. Egy *decision* nevű logikai változót várnak argumentumként. Ha ez a változó igaz, akkor az átmeneti állapot elfogadásra kerül, ha hamis, akkor az átmeneti állapot elvetődik. Ezek a függvények is meghívják a 3D kockaábrázolást, hogy láthatóvá tegyék a végleges állapotot.
- *UdefOmega()*:
  A célállapotot teljesen határozatlanná tevő függvény.
- *SolvedAlfa()*:
  Kirakott kezdőállapotot beállító függvény.
- *SolvedOmega()*:
  Kirakott célállapotot beállító függvény.
- *SetOmegaToAlfa()*:
  A célállapotot a kezdőállapottal egyenlővé tevő függvény.

o a megoldhatóságot tesztelő függvény, mely összehasonlítja a kezdő és a végállapotra meghívott tesztfüggvények visszatérési értékét; valamint az időbecslő függvény

- *Solvability()*:
  Ez a függvény teszteli a probléma megoldhatóságát. Egy *result* és egy *message* nevű változóval tér vissza. Előbbi egy logikai érték, mely magas, ha a probléma megoldható. Utóbbi pedig egy sztring, mely szöveges üzenet a döntés mellé.
  A függvény először azt ellenőrzi, hogy a kezdeti állapot teljesen határozott-e (*IsDefinit()* hívása *Alfa*-ra). Majd azt vizsgálja, hogy a kezdeti kocka 4 különböző sarokelemet tartalmaz-e (*IsLapos()*), illetve a célkocka határozott sarokelemei különböznek-e (*CanbeLapos()*). Csak ezek után következik a sarok- és paritástesztek futtatása. A kezdeti és a célkockára kapott eredményeket a *TestCmp()* függvény segítségével hasonlítja össze.
- *TEstimate()*:
  A függvény visszatérési értéke a maximális lépésszámból becsült futási idő másodpercben kifejezve. Ha a becslés nem lehetséges (mert például a konstruktor lefutásakor az inline függvény nem volt elérhető), a visszatérési érték *NaN*.

o a megoldó függvények és a megoldásokat listázó függvény
A megoldás rekurzív módon történik. Mivel azonban a szkript által meghívott

megoldófüggvény a megoldást az objektum *Solution* változójában kell, hogy tárolja, továbbá gondoskodnia kell az időmérésről, a tényleges rekurziót nem ez a függvény végzi el. Végeredményben *Solution* néven egy sztringekől álló tömb lesz a megoldóobjektumban, s a sztringek elemei az a, s, d, és w betűk lesznek, melyek rendre a bal, az első, a jobb ill. a hátsó oldal megfordítását jelentik. A sztringek betűrendben követik egymást, a megoldófüggvények minden ágon az első találatig keresnek. Ha a kezdőállapotból már 0 lépésben elérhető a célállapot, az a legelső sztringben jelzésre kerül, de ettől függetlenül elindul a keresés.

- *Solve()*:
  Ezt hívja meg a megoldószkript. Ő maga alapesetben 4-szer hívja meg a rekurzív függvényt. Argumentumként egy pozitív egészet vár, mely a maximális lépésszámot adja meg. Ha nem kap ilyen argumentumot, alapértelmezettként a megoldóobjektum *StepMax* változóját használja. A visszatérési érték logikai, mely siker esetén alacsony, kudarc esetén magas.
  Első lépésben elveti a véglegesítésre váró átmeneti kezdő és célállapotokat, törli a *Solution* tömb tartalmát, és első elemébe a *'<Nincs találat.>'* sztringet rakja arra az esetre, ha a beállított paraméterek mellett nem található megoldás a problémára. Ezután a *tic* paranccsal elindítja az időmérést.
  Miután az időmérő elindult, megvizsgálja, hogy a célállapot határozott elemei rendre megegyeznek-e a kiindulási állapot elemeivel. Amennyiben teljes az egyezés, a *Solution* tömb első elemébe az '<A kiindulás állapotot már eleve tartalmazza a célállapotot.>' sztring kerül. Csak ezek után következnek a rekurzív függvény hívása. Az *Iterate()* függvény hívása megtörténik a kezdőállapotból mind bal, mind első, mind jobb, mind pedig hátsó oldalfordítással nyert új állapotra. Az *Iterate()* függvény pedig a megfelelő ági megoldásokat tartalmazó sztringtömbbel tér vissza, a *Solve()* függvény feladata már csak e tömbök egyesítése a *Solution* változóban.
  Végül a *toc* paranccsal lekérdezésre kerül az eltelt idő, és tárolódik a *Time* változóban, a *Solved* változó értéke pedig magasra állítódik. (Ezt az értéket a kezdő és végállapotot manipuláló függvények állítják alacsonyra.)
- *Iterate()*:
  Ez a rekurzív megoldófüggvény, mely egy meghívása után tipikusan háromszor hívja meg önmagát. A megoldóobjektum referenciáján túl három változót vár: a pillanatnyi kockaállapotot (*Alfa*), egy tiltott lépést kifejező karaktert (*StepDisa*) és a maximálisig hátralévő lépésszámot (*StepMax*). A visszatérési érték szringekből álló tömb, mely az adott oldalághoz tartozó megoldásokat sorolja fel.
  A függvény először a toc parancs segítségével ellenőrzi, hogy a eltelt idő nem haladja-e meg a beállított korlátot (*TimeMax*). Ha meghaladja,

akkor egyetlen sztringgel tér vissza, melynek tartalma *'#runtime'*. A *Solve()* függvény ezzel a sztringgel találkozva alacsonyan hagyja a *Solved* változó értékét, és maga pedig magassal tér vissza.
Természetesen az *Iterate()* függvény maga is vizsgálja, hogy rekurzió során kap-e ilyen sztringet, s ha kap, akkor azt a hívó függvénynek továbbítva azonnal visszatér, meggátolva a további rekurziót.
A StepDisa változóra azért van szükség, hogy a megoldásban ne szerepeljen egymás után kétszer ugyanaz a lépés, hiszen az ismételt lépések (180°-os forgatásról lévén szó) kioltják egymást (ezért van szükség 4 helyett csak 3 rekurzióra).
A *StepMax* változó pedig minden egyes rekurziós lépésben egyre kisebb, amíg el nem éri a rekurziót megszakító 0 értéket. A visszatérő tömbben lévő sztringek maximális hosszát adja meg ez a változó.

- *List()*:
  Ez a függvény a parancssorba írja a *Solution* változó tartalmát feltéve, hogy a *Solved* változó értéke magas.

- a kockamátrixokat kiíró és a két kocka vizualizációját meghívó függvény (*Visual()*)
  Ez lényegében a kezelőobjektum *Visual()* függvényével azonos szerepű. Kiírja a kockamátrixokat, a tájolást, meghívja magas *AlfaChanged* és/vagy *OmegaChanged* érték esetén meghívja a megfelelő kockára a *Visual3()* függvényt, és végül alacsonnyá teszi az *AlfaChanged* és az *OmegaChanged* értéket. E logikai értékeket természetesen a kockákat véglegesen manipuláló függvények teszik igazzá.
- a két kockát szabályosan változtató, de az új állapotot visszatérési értékként megadó segédfüggvények (*elso(), hatso(), bal(), jobb()*)
  Ezek a függvények egy kockaobjektumot várnak bemenő paraméterként, de közben a megoldóobjektumról is kapnak egy referenciát, így mindig tisztában vannak a tájolási mátrixszal, s annak figyelembe vételével képesek meghívni a kockaobjektum oldaltekerő függvényeit. Végül a módosított kockát visszatérési értékül adják. A megoldófüggvények ezeket a függvényeket hívják meg, s így biztosítható, hogy a megoldófüggvények minden a tájolási mátrix minden megengedett értéke mellett helyesen működjenek.

- *MatrixKocka_3x3x1.m*: Főszkript
  A programot ennek a szkriptnek a meghívásával kell elindítani. A szkript létrehoz egy kezelő objektumot, és tartja a kapcsolatot a felhasználóval. Szükség esetén ez hívja meg a megoldó szkriptet. Magja egy while-ciklus, mely lényegében egy menürendszert működtet. A ciklus újra és újra sztringet kér a parancssorból, és meghívja a kezelőobjektum megfelelő tagfüggvényeit. A ciklus feltételéül megadott változóra, mely egy egész szám, a ciklus belsejében egy switch-elágazás épül. A switch egyes ágai felelnek meg az egyes menüképernyőknek. Kilépéskor a változó nullává válik, s így a ciklus megszakad.
  A menürendszer pontos leírása a „Szoftver használata" című szakaszban látható.

- *Solver_3x3x1.m*: Megoldó szkript
  Hasonló szemléletű a főszkripthez, de ez a megoldó objektumot hozza létre, és annak a tagfüggvényeit hívja meg a megfelelő időben.
- *TestCmp.m*: Határozatlan kockához használt tesztfüggvények kimenetének összehasonlítására is alkalmas függvény definíciója
  Mint azt korábban említettem, a határozatlan kockához is használható tesztfüggvények logikai változók helyett 8 bites egész számokat használ, hogy *-1* értékkel jelezze, ha egy kérdés nem dönthető el az elemek definiálatlansága miatt. A *TestCmp()* függvény úgy hasonlítja össze a 8 bites egészeket illetve a belőlük álló azonos méretű mátrixokat, hogy a -1 értéket mind a 0, mind az 1 értékkel azonosnak tekinti, de a 0 és az 1 értéket nem tekinti egymással azonosnak. Ha a két mátrixot ilyen értelemben azonosnak véli, magas értékkel tér vissza. (A cél nem annak eldöntése, hogy két kockaállapot biztosan egy állapotcsoportba tartozik-e, hanem az olyan esetek kiszűrése, amikor a két kockaállapot biztosan nem tartozik egy állapotcsoportba.)
- *VISUAL.m*: Általános kezelőosztály definíciója
  A kezelőobjektum osztálya (*VIS_LAP*) tulajdonképp ennek az alosztálya, és ezen keresztül alosztálya a *handle* osztálynak is. A kockaobjektum és a tájolási mátrix, valamint annak ortogonalitását ellenőrző függvény itt van definiálva, és a *VIS_LAP* konstruktor függvényében csak az dől el, hogy a kockaobjektum floppy-kocka (*LAPOS* osztályú). Ha a későbbiekben ki szeretnénk bővíteni a szoftvert más kockatípusok szimulációjára, az új kezelőosztály ennek alosztályaként definiálva nem kell a tájolási mátrixot, valamint annak ortogonalitását ellenőrző függvényt újra definiálni.
- *SOLVER.m*: Általános megoldóosztály definíciója
  Az előzőhöz hasonló gondolatmenet: A megoldó objektum osztálya (*SOL_LAP*) pedig ennek az alosztálya, s az ezen az osztályon keresztül alosztálya a *handle* osztálynak. A *SOL_LAP* osztálydefiníciónál említett változók tulajdonképp itt vannak definiálva, és a *SOL_LAP* konstruktorában csak az dől el, hogy a kezdő és a célkocka *LAPOS* osztályú. Ha más kockatípusokhoz tartozó megoldóosztályt ennek alosztályaként definiálunk, nem szükséges a változók újbóli definiálása.
- *SAROK.m, OLDAL.m, OLDALAK.m*: Osztálydefiníciók a kocka elemeihez
  A korábbiaknak megfelelően az *OLDAL* osztályú objektumok egy forgatási (*forgatas*) és egy definiáltsági (*definit*) bittel, a *SAROK* osztályúak pedig ezen kívül a pozíció kódolásához egy két logikai változóból álló tömbbel (*eltolas*) is rendelkeznek. Midkét osztálynak van a forgatást átállító függvénye, az *OLDAL* esetében *forgat()*, a *SAROK* esetében *eltol()* a függvény neve, utóbbi az eltolási biteket is átállítja. (Mivel azonban ezek az osztályok nem alosztályai a *handle* osztálynak, itt szükség van visszatérési értékek használatára.) A négy oldalobjektum (*bal, jobb, fenti* és *lenti* néven) az *OLDALAK* osztályban lett összefoglalva. A kockaobjektumban egy *OLDALAK* osztályú objektumot (*oldalak*) és egy *SAROK* osztályú objektumokból álló 2x2-es mátrixot (*sarkak*) találunk. Természetesen a bitek kezdeti értékei úgy vannak definiálva, hogy a *LAPOS()* konstruktor meghívásakor egy határozott kirakott floppy-kockát kapjunk.

- *Timer.mat*: Egy MATLAB adatfájl, mely az időbecsléshez szükséges inline függvényt tartalmazza
  Mint már korábban említettem, a megoldó objektum konstruktora olvassa be a fájlt.
- *Header_3x3x1.txt*: Egy szöveges fájl, mely fejlécet tartalmaz a programhoz
  A főszkript beolvassa a fájlt, mielőtt belép a menüciklusba, s a fájl tartalma mindig megjelenik a parancssor tetején.
- *mtimesTestLAPOS.m*: Szkript a csoportműveletek teszteléséhez
  Ez a szkript elvégez néhány egyszerű műveletet, s az eredményeket összehasonlító szándékkal feliratozva vizualizálja.

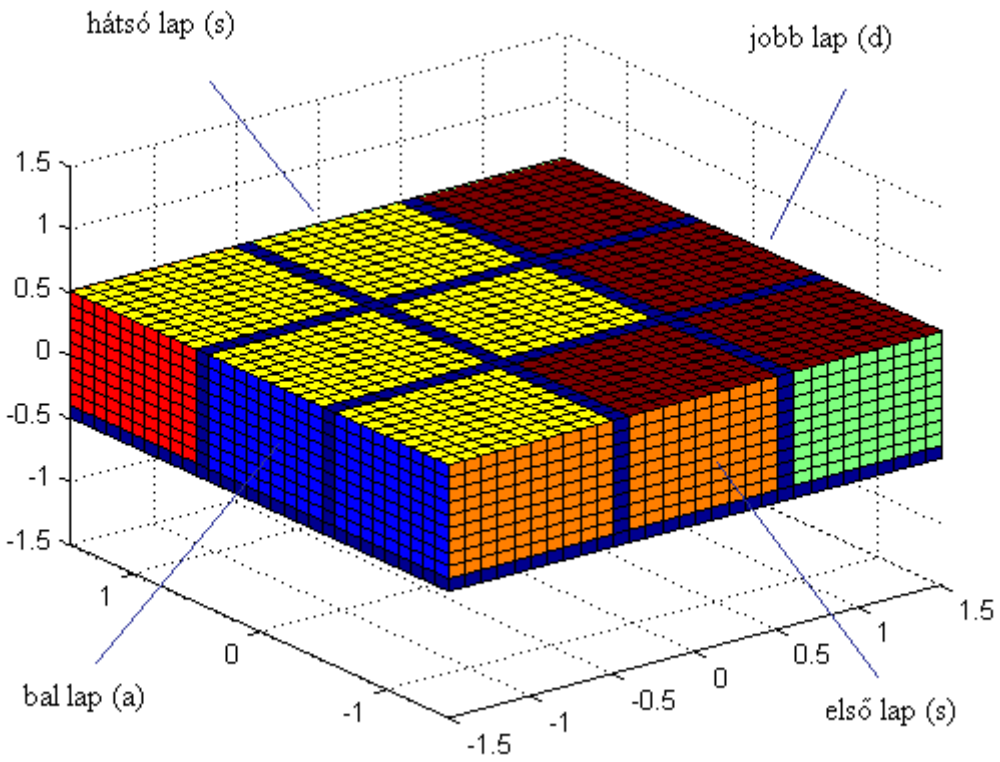
A szoftver használata

A szimulátor a *MatrixKocka_3x3x1.m* szkript aktiválásával indítható. Az indulás után a parancssoron az alábbiak láthatók:

```
Mátrixkocka
A Rubik-kocka szimulációja.
Méret: 3x3x1
Fejlesztette: Koics Dániel
A szoftvert ajánlom Balda Péter barátomnak.

A kockamátrix:
 1 0
01111
 0 1
10000
 1 0
A tájolás: [-y,x,z]

Lépések (folytonosan is):
   Hátsó: w
   Első: s
   Bal: a
   Jobb: d
Nézetváltások (folytonosan is):
   Megfordítás X körül: x
   Megfordítás Y körül: y
   Forgatás a síkban jobbra: c
   Forgatás a síkban balra: í
Menü: q

MK>>
```

Legfelül látható a mindig visszatérő fejléc. Alatta látjuk a kockamátrixot és a tájolási mátrixot. Utóbbinak rövidített az írásmódja: x jelenti a [1;0;0], y a [0;1;0], z pedig a [0;0;1] oszlopvektort. Ezek alatt információt találunk a felhasználó által küldhető karakterekről, sztringekről, és azok funkcióiról. Legalul találjuk a promptot (*MK*>>), mely után a felhasználó beírhatja parancsát.

Egy másik ablakban pedig megjelenik a kocka képe:



A fenti ábrán látható feliratok egyértelműsítik a parancsokat. Az oldalakat és a nézetet forgató parancsokból egyszerre több is küldhető, aminek hasznos dolog, ha például egy a megoldó programrész által adott megoldást akarunk tesztelni. A *q* parancsra a parancsokat jelző információk egy menüt jelenítenek meg, melynek pontjai:

- Folytatás: c
  Ez kilép a menüből, és a kockával tovább lehet játszani.
- Mentés: s
  Kér egy fájlnevet. Ez tetszőleges sztring lehet, mely automatikusan *.dan* végződést kap. Ekkor a kocka pillanatnyi állapota egy a megadott névvel ellátott fájlba mentődik a szimulátor fájljaival azonos mappában. A parancssor alján megjelenik a mentés sikerét jelző felirat.
- Betöltés: l
  Felsorolja a *.dan* végű fájlokat a mappában, és kér egy fájlnevet. A fájlnevet kiterjesztéssel együtt kell megadni. Végül megjelenik a parancssor aljában a betöltés sikeréről, és sikeres betöltés esetén a kockaállapot és a tájolási mátrix tesztjeiről szóló üzenet.
- Keverés: x
  Ez a menüpont kér egy természetes számot, mely megadja, hogy hány véletlen lépést akarunk végrehajtani a kockán.
- Visszaállítás: r
  Ez alaphelyzetbe állítja a kockát.

- Megoldó: d
  Meghívja a megoldó szkriptet.
- Kilépés: q
  Kilép a szimulátorból.

Ha a mentési fájlt kézzel módosítjuk, elő tudunk állítani egy olyan helyzetet, hogy a betöltés sikeres ugyan, de a kockaállapot vagy a tájolási mátrix nem megy át valamelyik teszten. Ekkor a fő képernyőn a tájolási mátrix és a billentyűparancsok kijelzése között egy hibaüzenetet látunk mindaddig, amíg ki nem javítjuk a problémát. Ugyanezen a helyen láthatunk gratuláló üzenetet, amikor sikerül kirakni a kockát.

A megoldó programrész is hasonlóképp néz ki, de ott két kockamátrix van, s a tájolás kijelzése alatt a beállított maximális lépésszám és a másodpercekben mért futási időkorlát is megjelenik. Alapértelmezettként 5-ös értéket látunk mindkettőnél. A prompt most *MK>>* helyett *MK/S>>*.

```
Mátrixkocka
A Rubik-kocka szimulációja.
Méret: 3x3x1
Fejlesztette: Koics Dániel
A szoftvert ajánlom Balda Péter barátomnak.
Megoldó programrész.

A kockamátrixok:
Alfa:  Omega:
 1 0    0 0
01000  00000
 1 0    0 0
11101  00000
 0 1    0 0
A tájolás: [x,y,z]
Beállított maximális lépésszám: 5
Beállított maximális futási idő: 5 másodperc

Kezdeti állapot módosítása:
   Hátsó: w
   Első: s
   Bal: a
   Jobb: d
   Elem szabálytalan hozzáadása: r
Célállapot módosítása:
   Hátsó: i
   Első: k
   Bal: j
   Jobb: l
   Elem szabálytalan hozzáadása/eltávolítása: u
Nézetváltások (2-esével is):
   Megfordítás X körül: x
   Megfordítás Y körül: y
   Forgatás a síkban jobbra: c
   Forgatás a síkban balra: í
Menü: q
Megoldó indítása: start

MK/S>>
```

Külön parancsok szolgálnak a kezdeti (*a,s,d,w*) és a célállapot (*j,k,l,i*) módosításához. Ezek a parancsok csak egyesével küldhetők (nem a folytonos játék a cél, ezért talán biztosabb, nem megengedni az összetett műveleteket), viszont a kényelem kedvéért a nézetváltó parancsok küldhetők kettesével.

A szabálytalan kockaépítés célját szolgáló *r* ill. *u* parancsra az *'Elem hozzáadása a kezdő állapothoz'* (*r*) illetve *'Elem hozzáadása/eltávolítása a célállapothoz/tól'* (*u*) menü jelenik meg, melyben ki lehet választani, hogy a kocka mely elemét kívánjuk módosítani:

- Oldalak:
  - Első: *e*
  - Hátsó: *h*
  - Bal: *b*
  - Jobb: *j*
- Csúcsok:
  - Bal első: *be*
  - Jobb első: *je*
  - Bal hátsó: *bh*
  - Jobb hátsó: *jh*
- Mégse: *c*

Az elem kiválasztása után sarokelem esetén az elem oldalsó lapjainak színét kérdezi meg a menü:

- Piros-Zöld: *pz*
- Zöld-Kék: *zk*
- Kék-Narancs: *kn*
- Narancs-Piros: *np*
- Eltávolítás: *hn*
  Ez a menüpont az elem határozatlanná tételére szolgál, és csak akkor jelenik meg, amikor a célállapotot kívánjuk módosítani.
- Mégse: *c*

Ha a célállapot egyik oldalelemét választjuk, akkor az alábbi 3 lehetőségünk van:

- Hozzáadás: ht
  Ez határozottra állítja az adott oldalelemet.
- Eltávolítás: hn
  Ez határozatlanná teszi az oldalelemet.
- Mégse: c

Ha a kezdeti állapot egy oldalalelemét akarjuk módosítani, akkor a fenti menüképernyő kimarad, s egyből a véglegesítő képernyő következik:

- Elem forgatása: r
  Ez az épp állítani kívánt elemet 180°-kal elforgatja, így a teteje alulra, az alja felülre kerül. Minden ilyen átfordítás után az átmeneti kockaállapot kirajzolódik. Nem jelenik meg ez a menüpont, ha épp határozatlanná teszünk egy elemet.
- Véglegesítés: ok
  Ez véglegesíti az új kockaállapotot. Ennek választásával automatikusan lefut a

megoldhatósági tesztfüggvény, s az eredmény a parancssor alján a következő billentyűnyomásig látható.

- Mégse: c
  Ennek választásával is látható egy üzenet a képernyő alján, mely a módosítás elvetését nyugtázza.

A szabálytalan módosítás után mindig lefut a megoldhatóságot tesztelő függvénylánc, és ha az nem megoldhatónak ítéli a problémát, a következő szabálytalan módosításig figyelmeztető üzenetet látunk, és a megoldófüggvény nem hívható meg.

A *q* parancsra előjövő menü:

- Folytatás: *c*
  Kilép a menüből és visszatér a megoldó főképernyőjéhez.
- Kirakott kezdeti állapot: *as*
  Kirakott állapotot állít be kezdeti állapotként.
- Kirakott célállapot: *os*
  Kirakott állapotot állít be célként.
- Határozatlan célállapot: *ou*
  Minden elemében határozatlan kockát állít be célként.
- A kezdetivel megegyező célállapot: *oa*
  Az épp aktuális kezdeti állapotot állítja be célként.
- Max. lépésszám állítása: *sm*
  Kér egy számot, melyet beállít maximális lépésszámként. A parancssor alján nyugtázza az új lépésszámot, és az új lépésszám alapján becsült futási időt. Egy figyelmeztetés is megjelenik, ha utóbbi meghaladja a beállított korlátot.
- Max. futási idő megadása: *tm*
  Kér egy számot, mely a maximális futási idő lesz másodpercben kifejezve. Ezután nyugtázza az új beállított időt. Egy figylemeztetés is megjelenik, ha e korlát nem éri el a beállított lépésszám alapján becsült futási időt. (Utóbbi zárójelben meg is jelenik.)
- Vissza a szimulátorhoz: *q*
  Visszatér az egyszerű szimulátorhoz. A megoldó kezdőállapota betöltődik a kockába.
- Vissza a MATLAB-hoz: *M*
  Kilép a szimulátorból.

A futási idő becslése egy exponenciális alakú függvény által történik, melyről a következő szakaszban lehet olvasni.

A parancssor alján a *'Megoldó indítása: start'* sor csak akkor jelenik meg, ha a probléma a megoldhatósági tesztfüggvény szerint megoldható, és a megoldófüggvény is csak ebben az esetben indítható. Ellenkező esetben a maximális lépésszám és az időkorlát kijelzése alatt egy figyelmeztető üzenet látható. Szintén figyelmeztető üzenet látható, ha a lépésszámból becsült

futási idő és a beállított időkorlát között nincs összhang, de a megoldófüggvény pusztán ilyen jellegű hiba esetén még elindítható.

A megoldófüggvény a *start* parancs beírásával indítható, ha a parancssor alján látható az erre vonatkozó információ. A megoldó lefutása után visszatér a főképernyő, és a megoldási lista az időkorlát kijelzése (és az esetleges időkorlátra vonatkozó figyelmeztető üzenet) alatt megjelenik. A lista mindaddig megjelenik a parancssorban, amíg nem módosítunk valamilyen paraméteren. A listán az *a* karakter jelenti a bal oldal, az *s* karakter az első oldal, a *d* karakter a jobb oldal, a *w* karakter pedig a hátoldal átfordítását.

Egy példa:

```
A kockamátrixok:
Alfa:  Omega:
 0 1    0 0
00010  00000
 0 1    0 0
00010  00000
 0 1    0 0
A tájolás: [x,y,z]
Beállított maximális lépésszám: 5
Beállított maximális futási idő: 5 másodperc

A találatok száma: 4
Futási idő: 0.76254 másodperc
A találati lista:
ada
d
swswd
wswsd

Kezdeti állapot módosítása:
   Hátsó: w
   Első: s
   Bal: a
   Jobb: d
   Elem szabálytalan hozzáadása: r
Célállapot módosítása:
   Hátsó: i
   Első: k
   Bal: j
   Jobb: l
   Elem szabálytalan hozzáadása/eltávolítása: u
```

Látható, hogy a paraméterek alatt mintegy a megoldási lista fejlécében megjelenik a találatok száma és a futás során ténylegesen eltelt idő. A kockamátrixokra nézve látható, hogy a célállapot a kirakott állapot (csupa 0), a kezdeti állapot pedig annyiban tér el a célállapottól, hogy a kocka jobb oldalát átfordítottuk. Ennek megfelelően a találati listán látunk egy egykarakteres *d* sort, ez jelenti azt, hogy a kocka egyetlen lépéssel, a jobb oldal átfordításával

kirakható. Az *ada* sor háromkarakteres, így háromlépéses megoldást jelöl. Annak megfelelője, hogy a jobb oldal átfordítása előtt és után is átfordítjuk a bal oldalt. A kétféle művelet persze felcserélhető, a műveletek öninverz tulajdonsága miatt ennek is pont ugyanaz a hatása, mintha csak a jobb oldalt fordítanánk át. A szolver talált még két darab, négylépéses megoldást (*swsd* és *wswsd*). Ebben az esetben a megoldó a kocka első és hátsó oldalát kívánja átfordítani kétszer, mielőtt a jobbhoz nyúlna.

Egy másik példa:

```
A kockamátrixok:
Alfa:  Omega:
 0 1
00010   uuu
 0 1    u u
00010   uuu
 0 1
A tájolás: [x,y,z]
Beállított maximális lépésszám: 5
Beállított maximális futási idő: 5 másodperc

A találatok száma: 5
Futási idő: 0.026696 másodperc
A találati lista:
<A kezdeti állapotot már eleve tartalmazza a célállapot.>
a
d
s
w
```

Ebben az esetben teljesen határozatlan célállapottal van dolgunk, vagyis a megoldó a kocka bármilyen állapotát megoldottnak tekinti. Ennek megfelelően a találati lista legfelső sora jelzi, hogy a megadott kezdőállapot már eleve megoldás. Ettől függetlenül elindul a legalább 1 lépésből álló megoldások keresése, s mivel a megoldó mind a 4 lehetséges irányba (*a,d,s,w*) lépve ismét megoldást lát, 1-nél több lépésből álló megoldást már nem is keres. (A megoldó a 0-lépéses megoldástól eltekintve minden ágon az első megoldásig keres.) Ezért lett a futási idő körülbelül 20-szor rövidebb, mint az előző esetben.

Ha a megoldó időkorlátba ütközik futás közben, akkor a parancssor alján az *'Időtúllépési hiba!'* üzenet jelenik meg, majd 1,5 másodperc után visszaáll a kezdeti képernyő. Ebben az esetben nem jelenik meg megoldási lista.

Egy rejtett funkció, hogy a megoldó főképernyője mellett az *exit* parancsot beírva a teljes szimulátor bezáródik.

A tesztelésről és a futásidő becsléséről

Egy ilyen bonyolult program esetében az egyes elemeket célszerű a fejlesztés során külön-külön tesztelni. Szerencsére az osztályok a konstruktort parancssorból meghívva tesztelhetők egyesével. Rendelkezem egy floppy-kockával, melynek segítségével a LAPOS osztály működése ellenőrizhető. Továbbá a megoldó által adott megoldások az egyszerű szimulátor parancssorába beilleszthetők, s így a megoldó működése is tesztelhető. Ilyen módon folytonos tesztelést végeztem, és csak akkor léptem a fejlesztés következő szintjére, amikor az adott szinten már nem volt észlelhető hiba.

A futási időt becslő függvény:

$$\mathrm{T}(N) = a \cdot e^{bN}$$

ahol

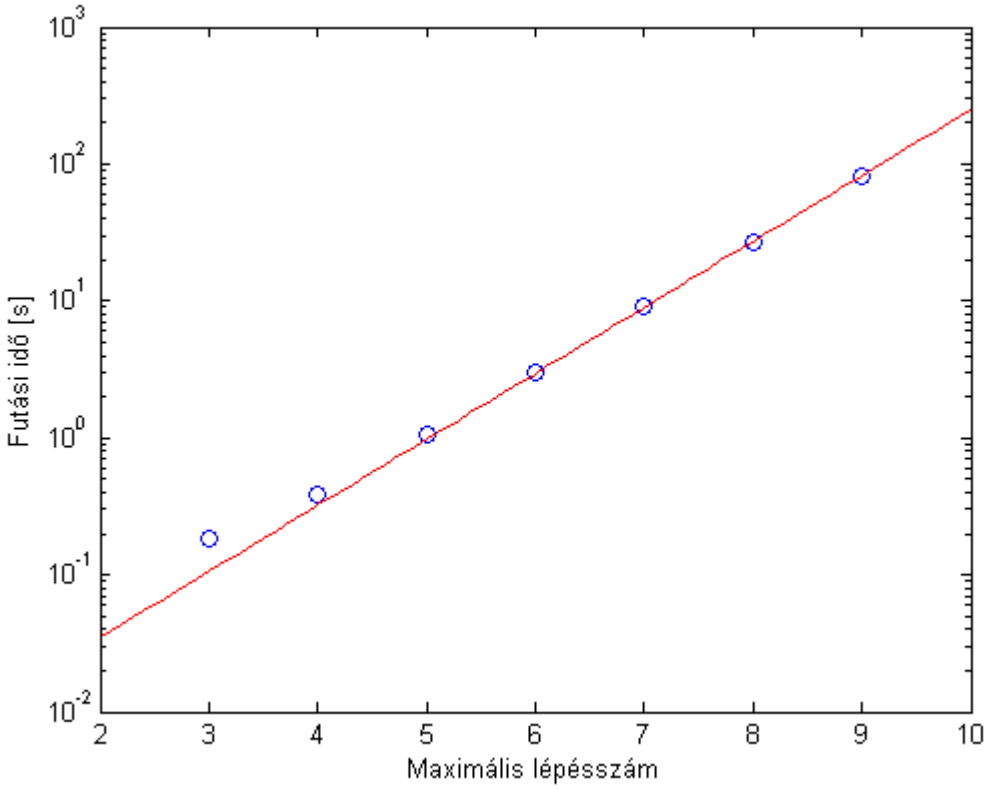
$$a = 0{,}003795s \ \ (0{,}003535..0{,}004055)$$
$$b = 1{,}11 \ \ (1{,}102..1{,}117)$$

és N a lépésszám, T pedig a futási idő másodpercben. Az exponenciális alak ötlete a rekurzióból természetesen adódik, hiszen minden egyes lépésben háromszor annyi függvény hívódik meg, mint az előzőben. Ami a paraméterek meghatározását illeti: Lehetővé tettem, hogy a megoldófüggvények akkor is hívhatók legyenek, ha a megoldhatósági teszten nem megy át a probléma, s megoldhatatlan problémákra indítottam el a megoldót, hogy a rekurziónak valóban csak a maximális lépésszám jelentsen korlátot. (Ehhez a *Solver_3x3x1.m* szkript 131. (*if Solvable*) és 139. (*end*) sorát kell kiiktatni egy % jel sor elejére történő beszúrásával.) Ekkor különböző lépésszámok mellett megmértem a futási időt, s a kapott adatokra a matlab illesztőjének 'exp1' nevű exponenciális görbéjét illesztettem. (A két paraméter után zárójelben a 95%-os konfidencia-intervallumot adtam meg.) Minden egyes adatpontot 4-szer mértem, s a mérések átlagát képeztem. A függvény a *Timer.mat* fájlban van mentve *FO* nevű *MATLAB*-változóként (illesztési objektumként). Módosítás esetén az új függvényt ennek a helyére kell beilleszteni. A teszt egy *Acer Aspire 5810T* típusú laptopon zajlott, mely 1,4GHz *Intel Core 2*-es Duo CPU-val és 4GB RAM-mal rendelkezik, s *R2009a* verziójú *MATLAB* futott rajta *Windows Vista* operációs rendszer alatt. Más konfiguráció mellett nem garantálom, hogy a becslő függvény helyesen működik (a megoldó ezért is indítható el akkor is, ha várhatóan túllépi az időkorlátot), ilyen esetben javasolt a mérést újra elvégezni.

A mért adatpontok:

Lépésszám	1. mérés	2. mérés	3. mérés	4. mérés	Átlag
3	*0,1737*	*0,19975*	*0,18725*	*0,1874*	*0,1870*
4	*0,42248*	*0,3975*	*0,38073*	*0,34464*	*0,3863*
5	*1,0515*	*1,0897*	*1,0172*	*1,0586*	*1,0543*
6	*3,0507*	*3,0199*	*3,0625*	*3,0202*	*3,0383*
7	*9,3337*	*9,0509*	*8,9057*	*8,9793*	*9,0674*
8	*27,1349*	*27,1923*	*26,9959*	*27,023*	*27,0865*
9	*82,6976*	*82,9803*	*81,9488*	*82,446*	*82,5182*

Az adatpontok és az illesztett görbe grafikonon, logaritmikus y tengellyel:



Megjegyzendő még, hogy az *UParityTest()* függvény határozatlan kimenetet ad, ha a kockában csak egy sarokelem is hiányzik, de az oldalelemek jól definiáltak. Pedig a sarokpermutáció paritása akkor is egyértelmű, ha 3 sarkat adott meg a felhasználó: a negyedik sarok helye kizárásos alapon meghatározható. Ennek az a következménye, hogy ha egy olyan célkockában, melynek elérhetősége csak a paritás-teszten bukik meg, kiiktatjuk az egyik sarokelemet, a szoftver a problémát tévesen megoldhatónak fogja ítélni.

Fejlesztési lehetőségek és konklúziók

Nem sikerült eljutni a megoldó által adott eredmények és a megoldói paraméterek elmentésére és betöltésére alkalmas függvények megírásához. Ennek pótlása az egyik továbblépési lehetőség. Másik lehetőség az *UParityTest()* függvény hibájának kijavítása, illetve a tesztfüggvények optimalizálása: Azok ugyanis kissé túlkomplikáltak, s nem elég felhasználóbiztosak.

A 3x3x3-as kockára való továbbfejlesztéshez szükséges az oldalforgató függvények egyesítése úgy, hogy az argumentumként kapott vektor döntse el, melyik oldalt merre kell elfordítani. A forgatottsági állapotokat nem lehet egyetlen bittel kifejezni, hanem valamilyen bonyolultabb objektumra (egy forgatási mátrixra) van szükség. A tesztelő algoritmusokat újra

kell gondolni. A 4x4x4-es eset még bonyolultabb, mert egy él mentén kettő középső elem van, s nincs az oldallapoknak középső, el nem mozduló elemük. Az 5x5x5x-ös viszont ismét tartalmaz ilyen jellegű középső elemeket, s visszavezethető a 3x3x3-as és a 4x4x4-es esetre.

Végeredményben elmondhatjuk, hogy sikerült írni egy sajátos szemléletű Rubik-kocka szimulátort és egy rekurzión alapuló megoldót egy egyszerűsített kockatípusra, s a szoftver képes bármilyen kockaállapotról eldönteni, hogy kirakható-e. Ami azt a kérdést illeti, hogy lehet-e a floppy-kocka esetében a színes négyzetek szimulációja helyett általános koordinátákkal élni, s ezzel memóriát megtakarítani: A válasz az, hogy bizonyos mértékben lehet, de cserébe a programkód elbonyolódik, hiszen a tömör adatokat használható formára hozni (a kocka kirajzolása a kockamátrix alapján) vagy épp a használható információt betömöríteni (mátrix-vektor szorzás az oldalforgató függvényekben) nem egyszerű. Bár a kockamátrixot viszonylag tömör formában valósítottam meg, a programkódok megírásához szükséges idő a gyakorlatban nem térült meg, feladatom inkább elméleti jelentőséggel bír. Egy gyakorlati programozónak tudnia kell optimalizálni az algoritmusok célravezetősége és a megírásukhoz szükséges idő között. Mindenesetre szórakoztató volt végiggondolni a floppy-kocka működését, és megírni ezt a programot.