SZAKDOLGOZAT

Szatmári László

Debrecen

2010

Debreceni Egyetem
Informatika Kar

Kereső algoritmusok hatékonyságának összehasonlítása a Hanoi problémán keresztül

Témavezető:
Dr. Várterész Magda
egyetemi docens

Készítette:
Szatmári László (Sotmari Laslo)
programtervező informatikus (BSc.)

Debrecen
2010

Tartalomjegyzék

Bevezetés

A hanoi tornyai matematikai játék, illetve feladvány egy legendából származó ötleten alapszik. A legenda szerint egykor az indiai Benaresben egy Shiva-templom jelölte a világ közepét. Fo Hi uralkodása idején a templomban élő szerzetesek feladatot kaptak Shiva istennőtől. A templom közepén volt egy márványtábla, melyből három gyémántrúd állt ki. Az első rúdon 64 különböző méretű arany korong volt. Legalul volt a legnagyobb, majd rajta méret szerint csökkenő sorrendben a többi. A szerzetesek feladata az volt, hogy áthelyezzék az összes korongot az első rúdról a másodikra, a következő szabályok betartása mellett. A korongok nagyon sérülékenyek, ezért egyszerre csak egy korong mozdítható, valamint egy korong nem helyezhető olyan rúdra, ahol van nála kisebb korong. A korongok megszenteltek, ezért csak ezt a három megszentelt rudat lehet használni, máshová nem rakhatják le a korongokat. Amint minden korong átkerült a második rúdra, a templom összeomlik, és a világ véget ér. Ha igaz ez a legenda, akkor jó lenne tudnunk, hogy ez mikor is következik be.

A játék leírása először 1883-ban jelent meg egy párizsi újságban, amit a Li-Sou-Stian egyetem egyik hallgatója, N. Claus de Siam publikált. Kiderült, hogy a szerző csak egy álnév, az igazi szerző nevének anagrammája. A cikk valódi írója a Lyce Saint-Louis egyetem professzora, Edouard Lucas D'Ameins volt.

A hanoi tornyai egy tipikus esete az úgynevezett „oszd meg és uralkodj" típusú problémáknak. A név onnan származik, hogy a megoldást legkönnyebben úgy találhatjuk meg, ha a problémát egyszerűbb, az eredetihez hasonló részproblémákra osztjuk, majd ha szükséges, a részproblémákat tovább osztjuk. Jelen esetben a problémának van egy kulcsmozzanata: a legnagyobb korong áthelyezése. Ha az első rúdról szeretnénk áthelyezni a korongokat a másodikra rúdra, akkor először a többi korongot kell átraknunk a harmadik rúdra, ezután át tudjuk rakni a legnagyobb korongot a második rúdra, majd a harmadik rúdra átrakott korongokat vissza kell rakni a második rúdra. A részproblémákat most a többi korong átrakása jelenti, először a harmadik rúdra, majd onnan vissza a másodikra. Ezeket a részproblémákat hasonló gondolkodásmód szerint oldhatjuk meg. Könnyen bizonyítható, hogy ezzel előállíthatjuk az optimális megoldást, ami n korong esetén $2^n - 1$ lépést jelent (a bizonyítást lásd később).

Ezek szerint 64 korong esetén minimum $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ lépésre van szükség. Ha a szerzetesek képesek lennénk egy korongot egyik rúdról a másikra egy másodperc alatt átrakni, a probléma megoldása még akkor is megközelítőleg 590 000 000 000

évet venne igénybe, úgyhogy egy ideig még nem kell félnünk a világ végétől akkor sem, ha a legenda igaz. Meg kell jegyeznünk, hogy az Univerzum jelen ismereteink szerint 13 700 000 000 éves.

Szakdolgozatom célja, hogy a hanoi problémát különböző módokon megoldjam, és ezeknek a megoldásoknak a hatékonyságát összehasonlítsam. Egy hagyományos mesterséges intelligenciai problémát általában legegyszerűbben állapottér-reprezentációval és az ismertebb kereső algoritmusokkal lehet megoldani. Természetesen ez közel sem a leghatékonyabb megoldás. Sokkal jobb eredményeket lehet elérni, ha elemezzük a problémát, és probléma specifikus algoritmust készítünk. Dolgozatom első felében két állapottér-reprezentációt fogok elemezni, majd a továbbiakban a problémát mélyebben megvizsgálva, ismertebb programozási technikákkal készítek többféle megoldást, és mindezeket összehasonlítom. Az algoritmusokat az egyszerűség kedvéért Java nyelven írtam, mivel célom nem a lehető legnagyobb hatékonyság elérése, hanem a különböző megoldások összehasonlítása.

I. Megoldás állapottér-reprezentációval

Készítsük el az állapottér-reprezentációt. A problémánk világa lényegében csak a rudakat és a korongokat tartalmazza. A rudak száma állandó, a korongok száma viszont a probléma méretével nő. Az állapotok tárolására két lehetőségünk van:

1. azt tároljuk, hogy az egyes rudakon milyen korongok vannak.
2. azt tároljuk, hogy melyik korong melyik rúdon van

I.1 Az első állapottér-reprezentáció

Az első esetben azt tároljuk, hogy az egyes rudakon milyen korongok vannak. A rudakon lévő korongokat halmazokkal tudjuk leírni. Legyen a három rudat leíró halmazok jelölése H_A, H_B és H_C. A korongokat méret szerinti sorrendben számozzuk 1-től n-ig. 1-es jelöli a legnagyobb korongot, értelemszerűen n jelöli a legkisebbet.

$$H_A = \{ \emptyset, \{1\}, \{2\}, \dots, \{n\}, \{1,2\}, \{1,3\}, \dots, \{1, n\}, \dots \{n-1, n\}, \dots, \{1, 2, 3, n-3, n\}, \dots, \{1, 2, \dots, n\} \}$$

$$H_B = \{ \emptyset, \{1\}, \{2\}, \dots, \{n\}, \{1,2\}, \{1,3\}, \dots, \{1, n\}, \dots \{n-1, n\}, \dots, \{1, 2, 3, n-3, n\}, \dots, \{1, 2, \dots, n\} \}$$

$$H_C = \{ \emptyset, \{1\}, \{2\}, \dots, \{n\}, \{1,2\}, \{1,3\}, \dots, \{1, n\}, \dots \{n-1, n\}, \dots, \{1, 2, 3, n-3, n\}, \dots, \{1, 2, \dots, n\} \}$$

A három halmaz Descartes-szorzata:

$$H_A \times H_B \times H_c = \{ (\emptyset, \emptyset, \emptyset), (\{1\}, \emptyset, \emptyset), \dots, (\{1, 2, \dots, n\}, \{1, 2, \dots, n\}, \{1, 2, \dots, n\}) \}$$

Erre a Descartes-szorzatra nem mondhatjuk, hogy ez az állapotok halmaza, mivel sok olyan elemhármast tartalmaz, amelyek nem lesznek állapotok, mivel az eredeti problémában nem fordulnak elő. Például négy vagy több korong esetén az $(\{1\}, \{2\}, \{1, 2, 4\})$ elemhármas nem jó, mert az 1-es és a 2-es korongok is egyszerre több rúdon vannak, valamint a 3-as korong nincs rajta semelyik rúdon. A $h = (h_A, h_B, h_C) \in H_A \times H_B \times H_C$ elemhármas a probléma állapota, ha teljesülnek rá a következő kényszerfeltételek:

- minden korongnak pontosan egy rúdon kell szerepelnie

- mindegyik korongnak szerepelnie kell valamelyik rúdon

Ezeket formalizálva a következőket kapjuk:

- $\forall i \forall j (h_i \cap h_j \neq \emptyset \supset i = j)$
- $h_A \cup h_B \cup h_C = \{1, 2, 3, \dots, n\}$

Összefoglalva:

$$\text{kényszerfeltétel(h)} = \forall i \forall j \left(h_i \cap h_j \neq \emptyset \supset i = j\right) \wedge h_A \cup h_B \cup h_C = \{1, 2, 3, \dots, n\}$$

Most már felírhatjuk az állapotok halmazát:

$$\mathcal{A} = \{ (h_A', h_B', h_C') \mid (h_A', h_B', h_C') \in H_A \times H_B \times H_C \wedge kényszerfeltétel(h) \}$$

Kezdőállapotnak az az állapot tekinthető, amikor minden korong egy rúdon van. Esetünkben legyen ez az első, vagyis az A rúd:

$$kezdő = (\{1, 2, \dots, \text{n}\}, \emptyset, \emptyset) \in \mathcal{A}$$

Azokat az állapotokat tekintjük célállapotnak, mikor minden korong ugyanazon a rúdon van, és ez a rúd különbözik a kezdőállapotban szereplő rúdtól. Két ilyen állapotunk van:

$$\mathcal{C} = \{ (\emptyset, \{1, 2, \dots, \text{n}\}, \emptyset), \ (\emptyset, \emptyset, \{1, 2, \dots, \text{n}\}) \} \subset \mathcal{A}$$

Ezután az operátorokat kell megadnunk. Ezeket kétféleképpen tehetjük meg. Első lehetőség, hogy az operátor paraméterben megkapja, hogy melyik korongot melyik rúdra kell áthelyezni. Ez kicsit problémás, mert így több rudat is meg kell vizsgálni, hogy megtaláljuk, melyiken van az adott korong. Egyszerűbb, ha paraméterben azt adjuk meg, hogy melyik rúdról melyik rúdra kell áthelyezni egy korongot. Ilyenkor értelemszerűen az adott rúdon lévő legkisebb, vagyis a legfelső korongot helyezzük át. Mivel három rúd van, és mindegyik rúdról szeretnénk mindegyikre (kivéve az adott rudat) korongot áthelyezni, ezért összesen hat operátorunk lesz:

$$\mathcal{O} = \{ Mozgat(honnan, hova) \} = \{ Mozgat(A, B), \ Mozgat(A, C), \ Mozgat(B, A), \ Mozgat(B, C), \ Mozgat(C, A), \ Mozgat(C, B) \}$$

Természetesen ezek az operátorok nem alkalmazhatóak bármikor, ugyanis nem létező korongot nem tudunk áthelyezni, valamint ott van az a szabály is, miszerint nagyobb korongot

nem rakhatunk kisebbre. Ezért egy $Mozgat(honnan, hova)$ operátor akkor alkalmazható egy $h = (h_A, h_B, h_C) \in \mathcal{A}$ állapotra, ha

- a $honnan$ rúdon van korong
- a $hova$ rúdon lévő minden korong nagyobb a $honnan$ rúdon lévő legkisebb korongnál

Formalizálás előtt vezessük be a $LK(h_i)$ jelölést, ami jelentse az i. rúdon lévő legkisebb korongot: $z = LK(h_i) \Leftrightarrow z \in h_i \wedge \forall x(x \in h_i \supset z \leq x)$

- $h_{honnan} \neq \emptyset$
- $\forall x(x \in h_{hova} \supset x > LK(h_{honnan}))$

A $Mozgat(honnan, hova)$ operátort egy $h = (h_A, h_B, h_C) \in \mathcal{A}$ állapotra alkalmazva a következőképp definiált $h' = (h_A', h_B', h_C') \in H_A \times H_B \times H_C$ elemhármast kapjuk:

$$h_i' = \begin{cases} h_i \setminus \{LK(h_i)\}, & ha\ i = honnan \\ h_i \cup \{LK(h_{honnan})\}, & ha\ i = hova \\ h_i, & egyébként \end{cases}$$

Mivel operátorainkat úgy sikerült definiálni, hogy állapotból bizonyíthatóan állapotot állítanak elő, és a kezdőállapotunk állapot, ezért a megoldáskeresés során előállított elemhármasokra a kényszerfeltételek ellenőrzése elhagyható.

Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az $< \mathcal{A}, kezdő, \mathcal{C}, \mathcal{O} >$ négyest, a probléma egy lehetséges állapottér-reprezentációját.

I.2 A második állapottér-reprezentáció

Ha a másik megoldást választjuk, vagyis azt tároljuk, hogy melyik korong, melyik rúdon van, akkor egy elem n-esre van szükségünk. Az elem n-es minden egyes eleme azt mondja meg, hogy az adott korong melyik rúdon van, így ezek az értékek csak $1, 2$ és 3 lehetnek.

$$H = \{(h_1, h_2, \ldots, h_n) \mid h \in \{1, 2, 3\}, i = 1, 2, \ldots, n\}$$

Ez a H nem tartalmaz olyan elem n-est, amely nem a probléma egy állapotát írná le, ezért nincs szükség kényszerfeltételekre. Ezzel az elem n-essel a probléma bármely állapotát leírhatjuk, ezért ez a probléma állapotainak halmaza:

$$\mathcal{A} = H$$

Kezdőállapotnak az az állapot tekinthető, amikor minden korong egy rúdon van. Esetünkben legyen ez az első rúd:

$$kezdő = (1, 1, \ldots, 1) \in \mathcal{A}$$

Azokat az állapotokat tekintjük célállapotnak, mikor minden korong ugyanazon a rúdon van, és ez a rúd különbözik a kezdőállapotban szereplő rúdtól. Két ilyen állapotunk van:

$$\mathcal{C} = \{(2, 2, \ldots, 2), (3, 3, \ldots, 3)\} \subset \mathcal{A}$$

Ezután az operátorokat kell megadnunk. Esetünkben egyetlen operátor is elegendő a probléma megoldásához, azonban ezt ismét kétféleképpen adhatjuk meg. Egyik esetben az operátornak azt adjuk meg, hogy melyik rúdról melyik rúdra kell egy korongot átrakni. Ebben az esetben meg kell keresni, hogy az adott rúdon melyik korong van legfelül. Egyszerűbb, ha rögtön azt adjuk meg, hogy melyik korongot kell átrakni. Vagyis az operátornak két paramétere lesz: mit és $hová$. A mit paraméter azt mondja meg, hogy melyik korongot szeretnénk átrakni, vagyis 1-től n-ig vehet fel értékeket. A $hová$ paraméter azt jelöli, hogy az adott korongot melyik rúdra kell tenni, vagyis értéke $1, 2$ vagy 3 lehet.

$$\mathcal{O} = \{Mozgat(mit, hová)\} = \{Mozgat(1, A),\ Mozgat(1, B),\ Mozgat(1, C),\ Mozgat(2, A), \ldots,\ Mozgat(n, B),\ Mozgat(n, C)\}$$

Az operátorunk csak a szabályok betartása mellett alkalmazható, valamint egy korongot nem rakhatjuk ugyanarra a rúdra, amelyiken van. Ezért egy $Mozgat(mit, hová)$ operátor akkor alkalmazható egy $h = (h_1, h_2, \ldots, h_n) \in \mathcal{A}$ állapotra, ha

- a $hová$ rúd vagy üres, vagy a $hová$ rúd minden korongja nagyobb a mit korongnál
- a mit korong rúdján nincs a mit korongnál kisebb korong

Az első feltétellel kizártuk azt a lehetőséget is, hogy a korongot ugyanarra a rúdra rakjuk vissza, amelyiken a korong volt. Formalizálva:

- $\forall i(h_i = hova \supset i > mit)$
- $\forall i(h_{mit} = h_i \supset mit \leq i)$

A $Mozgat(mit, hová)$ operátort egy $h = (h_1, h_2, \ldots, h_n) \in \mathcal{A}$ állapotra alkalmazva a következőképp definiált $h' = (h_1', h_2', \ldots, h_n') \in H$ elem n-est kapjuk:

$$h_i' = \begin{cases} hova, & ha\ i = mit \\ h_i, & egyébként \end{cases}$$

Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az $< \mathcal{A}, kezdő, \mathcal{C}, \mathcal{O} >$ négyest, a probléma egy lehetséges állapottér-reprezentációját.

I.3 Az állapottér-reprezentáció megvalósítása

A probléma állapottér-reprezentációval történő megoldásához Kósa Márk egyetemi tanársegéd által írt szélességi, mélységi és backtrack keresőket használtam. A keresőket A mesterséges intelligencia alapjai tárgy keretében használtuk. Ezek a keresők elérhetőek az említett tantárgy honlapján. [5]

A keresők használatához szükség van egy **Main** osztályra, ahonnan majd meghívjuk azokat, egy **Hanoi** osztályra, amely az állapotokat reprezentálja, valamint még annyi osztályra, ahány féle operátor használunk. Az operátorok osztályait az **Operator** osztályból kell származtatni, míg a **Hanoi** osztályt az **Allapot** osztályból.

Az **Operator** osztályban csak az operátor paramétereinek megfelelő adattagokra, valamint a hozzáférésükhöz szükséges get/set metódusokra van szükség, és természetesen egy konstruktorra, amely beállítja az adattagok értékét a paraméterben kapott értékekre. Ezek mellé ajánlott még megadni a **toString** metódust, ami az adott operátorról szöveges információt ad.

A **Hanoi** osztályt nem csak állapotok leírására használjuk, hanem ebben az osztályban valósítjuk meg az operátorok alkalmazási előfeltételének vizsgálatát, az operátorok alkalmazásának hatását, a célállapotok vizsgálatát, valamint a többi, a működéshez szintén szükséges metódust.

Az osztály elején egy statikus inicializáló blokkot használunk, amelyben feltöltjük az **operatorok** HashSet-et az összes operátor minden lehetséges paraméterezésével. Ezen kívül két konstruktort adunk meg: egy paraméter nélküli konstruktort, amelyik előállítja a kezdőállapotot, valamint egy másoló konstruktort, amelyik paraméterben a **Hanoi** osztály egy példányát kapja, és a példányban tárolt állapotra állítja be az aktuális állapotot. Ezenfelül a következő metódusokra van szükségünk:

- a **celAllapot** egy logikai értékkel visszatérő függvény, amely eldönti az aktuális állapotról, hogy az célállapot-e
- az **equals** metódus az aktuális állapot egy másik állapottal való egyezését vizsgálja, logikai függvény
- a **toString** metódus szolgál arra, hogy az adott állapotról szöveges információt adjunk. Visszatérési értéke **String** típusú

- az **elofeltetel** logikai típusú függvény, paraméterben egy operátort kap, majd eldönti, hogy a kapott operátor alkalmazható-e az adott állapotra
- az **alkalmaz** metódus paraméterként egy operátort kap, majd ezt alkalmazza az aktuális állapotra, és az eredményül kapott új állapottal tér vissza

I.4 Az első reprezentáció megvalósítása

Csak egyetlen operátorunk van, ezért a **Main** és a **Hanoi** osztályokon kívül csak egy **Mozgat** osztályra van szükségünk. Ennek az operátornak két paramétere van, ezért két adattagot használunk: **honnan** és **hova**. A konstruktor, valamint a get/set metódusok tartalma egyértelmű, nem szeretném részletezni. Az operátor egy korong áthelyezését jelenti, úgyhogy a **toString** metódusban ezt a lépést adjuk meg:

```
public String toString() {
    return (honnan + "->" + hova);
}
```

Vegyük most sorra a **Hanoi** osztály elemeit. Az osztály, mint azt már említettem, az **operatorok** kollekciót feltöltő statikus inicializáló blokkal kezdődik. A **Mozgat** operátor paraméterben rudakat kap, amiből csak három van, ezért ennek megfelelően végezzük el a feltöltést:

```
static {
    operatorok = new HashSet<Operator>();
    for ( int i = 1; i <= 3; ++i ) {
        for ( int j = 1; j <= 3; ++j ) {
            if ( i != j )
                operatorok.add( new Mozgat( i, j ) );
        }
    }
}
```

Szükség van egy **n** változóra, ami a probléma méretét jelöli. Az állapotok tárolásához elegendő egy kétdimenziós, logikai értékeket tartalmazó tömb. Legyen ez a **h** tömb, aminek három sorát használjuk, és annyi oszlopot, ahány korongunk van. A tömb egy eleme azt mondja meg, hogy az adott rúdon rajta van-e az adott korong.
Kezdőállapotban minden korong az első rúdon van, úgyhogy a tömb első sorát **igaz** értékekkel kell feltöltenünk, a többi elemet pedig **hamis** értékkel:

```
public Hanoi1() {
    h = new boolean[4][n+1];
    for ( int i = 1; i <= n; ++i ) {
        h[1][i] = true;
        h[2][i] = false;
        h[3][i] = false;
    }
}
```

A másik, paraméteres konstruktorban a **h** tömb értékeit a paraméterben kapott példány **h** tömbjének értékeire kell állítani:

```
public Hanoi1( Hanoi1 k ) {
    h = new boolean[4][n+1];
    for ( int i = 1; i <= 3; ++i ) {
        for ( int j = 1; j <= n; ++j ) {
            h[i][j] = k.h[i][j];
        }
    }
}
```

Célállapotnak azok az állapotok tekinthetőek, amikor minden korong ugyanazon a rúdon van, és ez nem a kiinduló rúd, vagyis minden korong a második rúdon, vagy minden korong a harmadik rúdon van. Először azt vizsgáljuk, hogy az első korong rajta van-e a két rúd valamelyikén, mert ha nem, akkor az biztosan nem célállapot. Ezután megvizsgáljuk az összes többi korongot, hogy ugyanazon a rúdon vannak-e, mint az első korong. Ha valahol hamis eredményt kapunk, akkor ez nem célállapot, egyébként pedig az.

```
public boolean celAllapot() {
    int rud;
    if ( h[2][1] == true )
        rud = 2;
    else if ( h[3][1] == true )
        rud = 3;
```

```
    else
        return false;
    for( int i = 2; i <= n; i++ )
        if ( h[rud][i] == false )
            return false;
    return true;
}
```

Az **equals** metódusban csak azt kell megvizsgálni, hogy a hasonlítani kívánt állapot **h** tömbjének minden eleme egyezik-e az aktuális állapot azonos tömbjének elemeivel. A hibák elkerülése végett érdemes a metódus elején ellenőrizni, hogy a kapott objektum valóban a **Hanoi** osztály példánya.

```
public boolean equals( Object obj ) {
    if ( obj == null || !( obj instanceof Hanoi1 ) ) {
        return false;
    }
    Hanoi1 k = ( Hanoi1 ) obj;
    for ( int i = 1; i <= 3; ++i ) {
        for ( int j = 1; j <= n; ++j ) {
            if ( h[i][j] != k.h[i][j] ) {
                return false;
            }
        }
    }
    return true;
}
```

A **toSring** metódus segítségével valamilyen szöveges formátumban kellene megjelenítenünk az aktuális állapotot. Ezt nagyos sokféle képen meg lehet tenni. Szerintem az egyik legegyszerűbb és mégis szemléletes mód az, ha három sorban kiíratjuk a tömb elemeit, és így jól látható hogy melyik korong, melyik rúdon van. Ha a korong rajta van az adott rúdon, akkor abban a sorban, a korong oszlopában egy ’**i**’ karakter jeleneik meg, egyébként pedig

egy ’**h**’ karakter. Persze ezek helyett használhatunk bármilyen egyéb karaktert, vagy karaktersorozatot.

```
public String toString() {
    StringBuffer sb = new StringBuffer();
    for( int i = 1; i <= 3; i++ ){
        for ( int j = 1; j <= n; j++ )
            if ( h[i][j] == true )
                sb.append( "i " );
            else
                sb.append( "h " );
        sb.append( "\n" );
    }
    return sb.toString();
}
```

Az **elofeltetel** metódusban azt kell vizsgálni, hogy az aktuális állapotra alkalmazni kívánt operátornak az állapottér-reprezentációban megadott alkalmazási előfeltételei teljesülnek-e erre az állapotra. Esetünkben, mint azt már az állapottér-reprezentációban is láttuk, két ilyen feltétel van:

- a *honnan* rúdon van korong
- a *hova* rúdon lévő minden korong nagyobb a *honnan* rúdon lévő legkisebb korongnál

Az első feltétel nagyon egyszerűen ellenőrizhető. Elegendő sorban vizsgálni a *honnan* rúd sorát a tömbben, és amint találunk egy igaz értéket, mondhatjuk, hogy teljesül a feltétel. Érdemes rögtön eltárolni, hogy ez hányadik korong, mivel ha sorban haladtunk, akkor ez a legkisebb korong, és a következő feltétel vizsgálatához szükségünk lesz erre az információra. A következő lépésben a *hova* rúdon lévő korongokat vizsgáljuk. Ha bármelyik kisebb, mint az előzőleg megtalált, *honnan* rúdon lévő legkisebb korong, akkor nem teljesül a feltétel. Egyébként az operátor alkalmazható az aktuális állapotra.

```
public boolean elofeltetel( Operator op ) {
    if ( op instanceof Mozgat ) {
        Mozgat m = ( Mozgat ) op;
```

```
            int honnan = m.getHonnan();
            int hova = m.getHova();
            boolean vanKorong = false;
            int legkisebb = 0;

            for ( int i = 1; i <= n && vanKorong == false; i++ )
                if ( h[honnan][i] != false ){
                    legkisebb = i;
                    vanKorong = true;
                }

            if ( vanKorong == false )
                return false;

            for ( int i = 1; i < legkisebb; i++ )
                if ( h[hova][i] != false )
                    return false;

            return true;
        }
        return false;
    }
```

Amennyiben egy operátor alkalmazható egy állapotra, azt valószínűleg alkalmazni is szeretnénk. Ehhez először létrehozunk egy új állapotot, majd ezen végezzük el a változtatásokat. A paraméterben kapott operátorból lekérdezzük annak paramétereit, majd ezek alapján a *honnan* rúdról átrakjuk a legkisebb korongot a *hova* rúdra, és végül a metódus visszatér az új állapottal.

```
public Allapot alkalmaz( Operator op ) {
    Hanoi1 uj = new Hanoi1( this );
    boolean megvan = false;
    if ( op instanceof Mozgat ) {
        Mozgat m = ( Mozgat ) op;
```

```
            int honnan = m.getHonnan();
            int hova = m.getHova();
            for ( int i = 1; i <= n && megvan == false ; i++ ){
                if ( uj.h[honnan][i] == true ){
                    uj.h[hova][i] = true;
                    uj.h[honnan][i] = false;
                    megvan = true;
                }
            }
        }
        return uj;
    }
```

I.5 A második reprezentáció megvalósítása

Ebben az esetben is előzőhöz hasonlóan **Main**, **Hanoi** és **Mozgat** osztályaink vannak. A **Mozgat** operátor most az előzőhöz hasonló, azzal a különbséggel, hogy a **honnan** paraméter helyett **mit** paraméter szerepel. Az adattagokat, metódusokat és kiírást is ennek megfelelően használjuk.

```
public String toString() {
    return (mit + "->" + hova);
}
```

Vegyük sorra a **Hanoi** osztály elemeit. Az osztály itt is az **operatorok** kollekciót feltöltő statikus inicializáló blokkal kezdődik. A **Mozgat** operátor paraméterben most egy korongot és egy rudat kap. A feltöltést ennek megfelelően végezzük el.

```
static {
    operatorok = new HashSet<Operator>();
    for ( int i = 1; i <= n; ++i ) {
        for ( int j = 1; j <= 3; ++j ) {
            operatorok.add( new Mozgat( i, j ) );
        }
    }
}
```

A probléma méretének tárolására itt is egy **n** változót használunk, az állapotok tárolásához azonban már elegendő egy egydimenziós tömb. Legyen ez a **h** tömb, melynek **n** eleme van, és minden elem azt jelöli, hogy az adott korong melyik rúdon van. Kezdetben minden korong az első rúdon helyezkedik el, ezért a tömböt egyesekkel kell feltölteni.

```
public Hanoi2() {
    h = new int[n+1];
    for ( int i = 1; i <= n; ++i ) {
        h[i] = 1;
    }
}
```

A paraméteres konstruktor a **h** tömb értékeit a paraméterben kapott példány **h** tömbjének értékeire állítja:

```
public Hanoi2( Hanoi2 k ) {
    h = new int[n+1];
    for ( int i = 1; i <= n; ++i ) {
        h[i] = k.h[i];
    }
}
```

Célállapotnak azok az állapotok tekinthetőek, amikor minden korong ugyanazon a rúdon van, és ez nem a kiinduló rúd, vagyis minden korong a második rúdon, vagy minden korong a harmadik rúdon van. Ez jelen esetben nagyon egyszerűen ellenőrizhető. Ez a feltétel teljesül a tömb minden elmének az értéke 2, vagy minden elem értéke 3.

```
public boolean celAllapot() {
    if ( h[1] == 2 ){
        for ( int i = 2; i <= n; i++ )
            if ( h[i] != 2 )
                return false;
    } else if ( h[1] == 3 ){
        for ( int i = 2; i <= n; i++ )
             if ( h[i] != 3 )
                return false;
    } else return false;
    return true;
}
```

Az **equals** metódus csak annyiban különbözik az előző verzió **equals** metódusától, hogy a **h** tömb most csak egydimenziós.

```
public boolean equals( Object obj ) {
    if ( obj == null || !( obj instanceof Hanoi2 ) ) {
        return false;
    }
```

```
    Hanoi2 k = ( Hanoi2 ) obj;
    for ( int i = 1; i <= n; ++i ) {
        if ( h[i] != k.h[i] ) {
            return false;
        }
    }
    return true;
}
```

A **toSring** metódus itt is nagyon egyszerű. Csak azt kell kiírni, hogy melyik korong melyik rúdon van, vagyis a **h** tömb elemeit.

```
public String toString() {
    StringBuffer sb = new StringBuffer();
    for( int i = 1; i <= n; i++ ){
        sb.append( h[i] + " " );
    }
    return sb.toString();
}
```

Az **elofeltetel** metódusban azt kell vizsgálni, hogy az aktuális állapotra alkalmazni kívánt operátornak az állapottér-reprezentációban megadott alkalmazási előfeltételei teljesülnek-e erre az állapotra. Esetünkben, mint azt már az állapottér-reprezentációban is láttuk, két ilyen feltétel van:

- a hová rúd vagy üres, vagy a hová rúd minden korongja nagyobb a mit korongnál
- a mit korong rúdján nincs a mit korongnál kisebb korong

Az első feltétel azt is magába foglalja, hogy a mit korongot nem rakhatjuk vissza ugyanarra a rúdra, amelyiken van. Az a leghatékonyabb, ha ezt külön ellenőrizzük. A másik két feltétel egy cikluson belül ellenőrizhető. Csak a mit-nél kisebb korongokat nézzük végig, és ezek nem lehetnek sem a hová rúdon, sem a mit korong rúdján.

```
public boolean elofeltetel( Operator op ) {
    if ( op instanceof Mozgat ) {
```

```
            Mozgat m = ( Mozgat ) op;
            int mit = m.getMit();
            int hova = m.getHova();

            if ( hova == h[mit] )
                return false;

            for ( int i = 1; i < mit; i++ )
                if ( h[i] == hova || h[i] == h[mit] )
                    return false;

            return true;
        }
        return false;
    }
```

Az **alkalmaz** metódusban először létrehozunk egy új állapotot, majd ezen végezzük el a változtatásokat. A paraméterben kapott operátorból lekérdezzük annak paramétereit, majd ezek alapján a mit korongot átrakjuk a hová rúdra. Ez az tömb egyetlen elemének átírásával megvalósítható. A metódus visszatérési értéke az új állapot.

```
public Allapot alkalmaz( Operator op ) {
    Hanoi2 uj = new Hanoi2( this );
    if ( op instanceof Mozgat ) {
        Mozgat m = ( Mozgat ) op;
        int mit = m.getMit();
        int hova = m.getHova();
        uj.h[mit] = hova;
    }
    return uj;
}
```

II. A minimum/maximum probléma

Annak függvényében, hogy a legkevesebb, vagy éppen a legtöbb lépésből álló megoldást keressük, minimum illetve maximum problémáról beszélünk.

A hanoi tornyai problémának négy paramétere van, ezért vezessük be a következő jelölést: $H(n, f, v, s)$, ahol

n: a korongok száma

f: a kiinduló/forrás rúd

v: a célrúd

s: segéd rúd

Vagyis a $H(n, f, v, s)$ jelölés esetén a feladatunk az, hogy h darab korongot mozgassunk f rúdról v rúdra, s rúd felhasználásával. Legyen a három rudunk a, b, c. Alapértelmezés szerint legyen a a forrás rúd, b a célrúd, c pedig a segéd rúd.

Mivel a legrövidebb és leghosszabb megoldást keressük, legyen a minimum verzió esetén a jelölés $H_{min}(n, a, b, c)$, maximum verzió esetén pedig $H_{max}(n, a, b, c)$. A maximális lépésszám eseténél azonban pontosítanunk kell. Ha például azt játszanánk, hogy a legkisebb korongot az a rúdról a b-re rakjuk, onnan pedig a c rúdra, majd ismét vissza az a-ra, akkor azt a végtelenségig ismételgethetnénk. Ezt ciklusnak nevezzük. Ciklusnak tekintünk minden olyan lépéssorozatot, amelyet végrehajtva visszakapjuk a lépéssorozat megkezdése előtti állapotot. A maximum lépések számának meghatározása során csak olyan megoldásokat veszünk figyelembe, amelyek nem tartalmaznak ciklust.

Vezessünk be két további jelölést. Jelölje a '→' szimbólum a legegyszerűbb lépést, vagyis egyetlen korong közvetlen áthelyezését egyik rúdról a másikra (pl. $a \rightarrow b$ jelentése: helyezd át az a rúdon lévő legfelső korongot a b rúdra). Továbbá jelentse a '$>>$' szimbólum azt a lépéssorozatot, amellyel több egymáson lévő korong áthelyezhető egyik rúdról a másikra.

Akár a minimális, akár a maximális számú lépéssorozatot keressük, a kulcsmozzanat legnagyobb korong áthelyezése. Minimális lépésszám esetén a legnagyobb korongot az a rúdról közvetlenül a b rúdra tesszük $(a \rightarrow b)$. Természetesen ezt csak akkor tehetjük meg, ha előtte a többi $n-1$ korongot már átraktuk az a rúdról a c rúdra $(a >> c)$. Miután a legnagyobb korongot áthelyeztük, rá kell raknunk a többi korongot is, vagyis a maradék

$n-1$ korongot átrakjuk a c rúdról a b rúdra $(c >> b)$. Természetesen ahhoz, hogy a probléma megoldása minimális lépésszámmal történjen, az $n-1$ darab korongot is a lehető legkevesebb lépésszámmal kell mozgatnunk, vagyis az éppen felvázolt módon. Ezen gondolkodásmód alapján megadhatunk egy rekurzív formulát a minimális lépésszámú megoldáshoz:

ha $n = 1$, akkor $H_{min}(n, a, b, c) = a \to b$

ha $n > 1$, akkor $H_{min}(n, a, b, c) = H_{min}(\mathrm{n}-1, \mathrm{a}, \mathrm{c}, \mathrm{b}),\ \mathrm{a} \to \mathrm{b},\ H_{min}(\mathrm{n}-1, \mathrm{c}, \mathrm{b}, \mathrm{a})$

Ebből felírhatunk egy általános képletet n korong esetére:

$$m_n = 2m_{n-1} + 1, m_1 = 1$$

m_{n-1}-re alkalmazzuk ugyanezt a képletet:

$$m_n = 2(2m_{n-2} + 1) + 1 = 2^2 m_{n-2} + 2 + 1$$

Ismét alkalmazva:

$$m_n = 2(2(2m_{n-3} + 1) + 1) + 1 = 2^3 m_{n-3} + 2^2 + 2 + 1$$

Ha ezt a helyettesítést m_1-ig ismételjük, a következő képletet kapjuk:

$$m_n = 2^{n-1} m_{n1} + 2^{n-2} + \ \ldots + 2^2 + 2 + 1 = 2^{n-2} + \ \ldots + \ 2^2 + 2 + 1$$

Vagyis $m_n = 2^n - 1$

Maximális lépésszám esetén a legnagyobb korongot nem közvetlenül az a rúdról a b rúdra helyezzük, hanem előbb a c rúdra, majd onnan b-re $(a \to c, c \to b)$. Természetesen közben a maradék $n-1$ korongot is át kell raknunk ahhoz, hogy a legnagyobb korongot mozgatni tudjuk. Mivel a legnagyobb korongot először a c rúdra fogjuk tenni, ezért a többi korongot a b rúdra rakjuk. Ezután rakjuk át a nagy korongot c-re, majd a többi korongot, visszarakjuk az a rúdra. Ezzel szabaddá válik a b rúd, ahová áttesszük a legnagyobb korongot, majd rá az összes többit. Be kell látnunk, hogy ha bármilyen egyéb lépést is végeznénk, az ciklust eredményezne. A minimális lépésszámú megoldáshoz hasonlóan itt is felírhatjuk a rekurzív formulát:

ha $n = 1$, akkor $H_{max}(n, a, b, c) = a \to c, c \to b$

ha $n > 1$, akkor $H_{max}(n, a, b, c) = H_{max}(n-1, a, b, c) = a \to c,$
$H_{max}(n-1, b, a, c), c \to b, H_{max}(n-1, a, b, c)$

Mint az előző esetben, itt is felírhatjuk az általános képletet:

$$M_n = 3M_{n-1} + 2,\ M_1 = 2$$

M_{n-1}-re alkalmazzuk ugyanezt a képletet:

$$M_n = 3(3M_{n-2} + 2) + 2 = 3^2 M_{n-2} + 2 \cdot 3 + 2$$

$$M_n = 3(3(3M_{n-3} + 2) + 2) + 2 = 3^3 M_{n-3} + 2 \cdot 3^2 + 2 \cdot 3 + 2$$

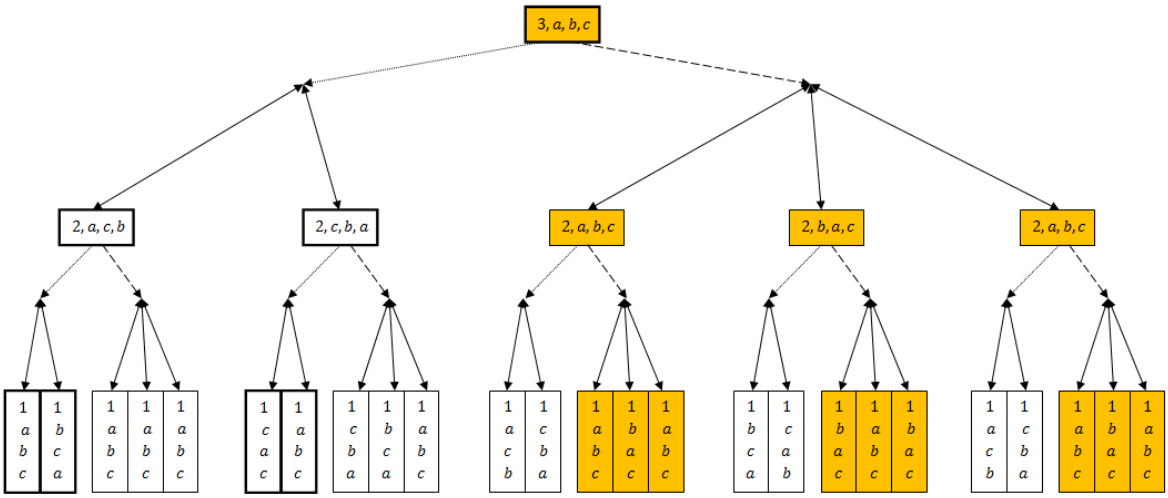
Ha ezt a helyettesítést M_1-ig ismételjük, a következő képletet kapjuk:

$$M_n = 3^{n-1} M_1 + 2 \cdot 3^{n-2} + \ \ldots + 2 \cdot 3^2 + 2 \cdot 3 + 2 = 2(3^{n-1} + 3^{n-2} + \ \ldots + 3^2 + \\ + 3 + 1)$$

$$M_n = 2\left(\frac{3^n - 1}{3 - 1}\right)$$

Vagyis $M_n = 3^n - 1$

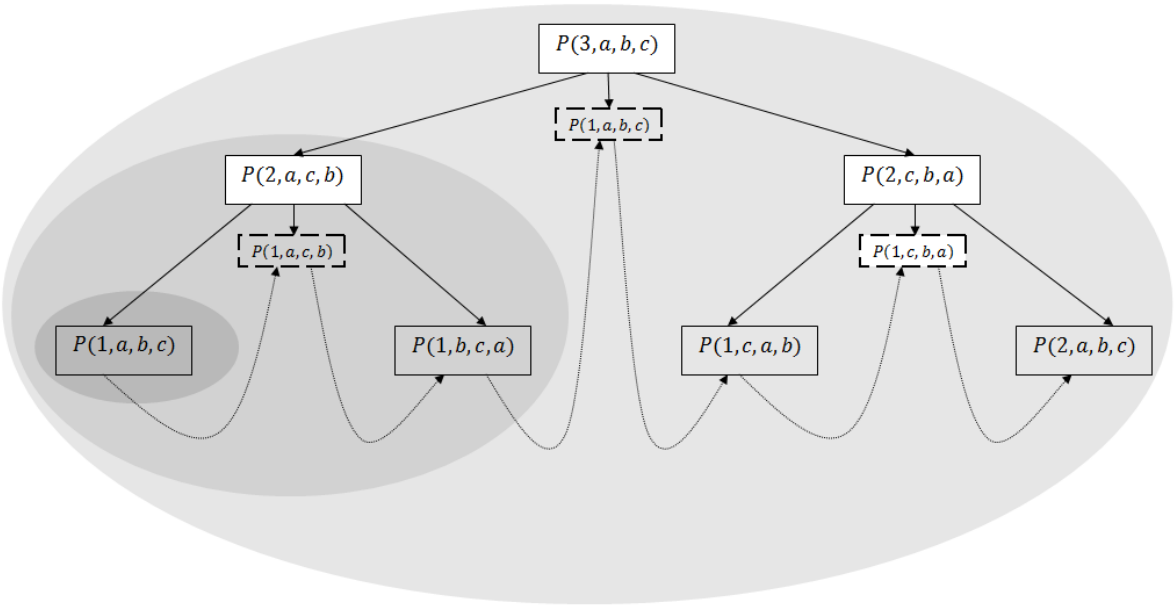
A minimális és a maximális lépésszámú megoldás annyiban tér el egymástól, hogy a legnagyobb korongot egy lépésben $(a \to b)$ vagy két lépésben $(a \to c, c \to b)$ rakjuk-e át. Más lehetőségünk nincs is. Ha mindig az egy lépéses áthelyezést választjuk, akkor minimális lépésszámot kapunk, ha mindig két lépést használunk, akkor pedig maximális lépésszámot kapunk. Az összes többi lehetőség a két módszer kombinálásával állítható elő.
Mindezeket egy gyökeres fával ábrázolhatjuk:



1. ábra

Az ábrán a gyökérelem jelöli az n méretű problémát, amit kétféleképpen oszthatunk részproblémákra. Lesz egy „minimum-ág” és egy „maximum-ág”. A minimum ágon két darab, míg a maximum ágon három darab, azaz összesen öt darab $n-1$ méretű részprobléma lesz. Ez azt jelenti, hogy egy teljes, n szinttel rendelkező fának minden egyes $k = 1, 2, \dots, n$ szintjén 5^{k-1} elem helyezkedik el. Vagyis a teljes fának $5^n - 1$ eleme van. Az ábrán a pontozott vonalak jelzik a „minimum-ágat”, a szaggatott vonalak pedig a „maximum-ágat”. Ha a teljes n szintű fában csak a „minimum-ág” elemei vannak, akkor a fának $2^n - 1$ eleme lesz. Ezeket az elemeket az ábrán a vastag keretes téglalapok jelölik. A „maximum-ág” elemeiből $3^n - 1$ darab van, ezeket a narancssárga téglalapok jelölik. Minden egyes részfa a probléma egy megoldását jelöli, annyi megoldása van, ahány részfa. Minimum és maximum megoldásból csak egy-egy létezik.

A rekurzív formulára úgy tekinthetünk, mint az a formula, amelyik leírja az optimális megoldás struktúráját. Ez a struktúra is egy-egy fával ábrázolható, méghozzá az 1. ábra „minimum-részfájával” és „maximum-részfájával”. A 2. ábrán látható a minimum verzió 3-méretű problémájához ($H_{min}\,(3, a, b, c)$) tartozó fa.



2. ábra

Az ábrán a részfák jelölik az optimális megoldásokat. Egy elem egy lépésnek felel meg. A pontozott vonalak által jelölt lépéssorozat, az optimális megoldást reprezentálja. A részfák

gyökérelemeit a fehér téglalapok jelölik. Ezek mindegyikének van szaggatott vonallal jelölt „saját-levele”. Ezek az elemek az adott problémához tartozó legnagyobb korong lépését írják le. Ezeket úgy vesszük, mintha a szülő elemükkel azonos szinten lennének (a szülő elemek „magukba foglalják” a „saját-leveliket”). Ha a korongokat a legnagyobbtól kezdve a legkisebbig megszámozzuk 1-től n-ig, és ugyanígy megszámozzuk a szinteket is, a gyökérelemtől a levél elemekig, akkor azt mondhatjuk, hogy fa k-adik szintjének lépései a k-adik korong lépéseit jelölik. A k-adik korongnak 2^{k-1} lépése van. A legalsó szinten lévő elemek jelentenek minden második lépést, ezek lesznek a legkisebb korong lépései.

III.1 A rekurzív implementáció

Az előzőekben felvázolt rekurzív formulákat implementálva a következő két algoritmust kapjuk:

```
public static void Rekurziv_min( int k, char forras, char cel,
char seged ){
    if ( k==1 ){
        System.out.println(forras + " -> " + cel);
    } else {
        Rekurziv_min(k-1, forras, seged, cel);
        System.out.println(forras + " -> " + cel);
        Rekurziv_min(k-1, seged, cel, forras);
    }
}

public static void Rekurziv_max( int k, char forras, char cel,
char seged ){
    if ( k==1 ){
        System.out.println( forras + " -> " + seged);
        System.out.println( seged + " -> " + cel);
    } else {
        Rekurziv_max(k-1, forras, cel, seged);
        System.out.println(forras + " -> " + seged);
        Rekurziv_max(k-1, cel, forras, seged);
        System.out.println(seged + " -> " + cel);
        Rekurziv_max(k-1, forras, cel, seged);
    }
}
```

III.2 A „mohó” algoritmus

Az előbb bemutatott rekurzív implementáció igazi „oszd meg és uralkodj” algoritmusnak tűnik, mivel a problémát a megoldás során két vagy három egyszerűbb, az eredetihez hasonló részproblémára osztjuk. Felmerül azonban a kérdés, hogy miért pont ilyen módon történik a részproblémákra bontás. Ezek a megoldások közvetlenül a legnagyobb korong kitüntetett szerepéből következnek, vagyis abból, hogy az minimális vagy maximális lépésszámmal rakjuk-e át. Ezt nevezhetjük „mohó döntés”-nek. Ebből a nézőpontból vizsgálva a rekurzív implementációt, az nem más, mint olyan „oszd meg és uralkodj” stratégia, amely „mohó” döntést használ a részproblémákra bontás folyamán.
Másfelől viszont minden részproblémának az „elvi” megoldása a legnagyobb korongra vonatkozó „mohó” döntéssel kezdődik, ami az adott problémát két vagy három, az eredetihez hasonló részproblémára osztja. A Hanoi tornyai probléma minimum lépésszámú esetében gyakorlatilag a rész-részproblémákat a „mohó” döntés végrehajtása előtt és után kell megoldani. Ebből kifolyólag a „mohó” döntések végrehajtásának a sorrendje az optimális megoldást reprezentáló fának (2. ábra) egyfajta in-order bejárását követi. Innen nézve viszont már azt mondhatjuk, hogy a rekurzív implementáció egy „mohó” stratégia, ami „oszd meg és uralkodj” szerű (in-order Depth-first search algoritmus) metódust használ a „mohó” döntések sorrendjének meghatározásához.
Az általános „mohó” algoritmus szerint először a „mohó döntést” kell végrehajtani, és majd csak aztán kell megoldani a részproblémákat. Ezt a sorrendet kapjuk a megoldási-fa pre-order Depth-first vagy Best-first szerinti bejárásával. Ahhoz, hogy megkapjuk a megfelelő lépéssorozatot, a megoldás során el kell tárolnunk minden egyes lépést. Sajnos ez az ötlet felvet néhány problémát. A megoldáshoz szükséges lépések száma exponenciálisan nő a probléma méretével. Hat lehetséges lépésünk van: $a \to b,\ a \to c,\ b \to c,\ c \to b,\ c \to a$ valamint $b \to a$. Ezekhez hozzárendelhetjük rendre a $0, 1, 2, 3, 4$ és 5 számjegyeket. Egy 64 korongot tartalmazó probléma esetén $2^{64} - 1$ számjegyre van szükség. Ekkora mennyiségű adat még a mai technológia mellett is csak nagyon nehezen lenne tárolható, és még nehezebben kezelhető. Kisebb n értékek esetén azonban működőképes a következő algoritmus.

A megoldási folyamat során először kiszámítjuk, hogy az adott n méretű probléma megoldásához hány lépésre van szükség, hogy megtudjuk, mekkora mennyiségű adatot kell

eltárolni. Szükségünk lesz egy ekkora méretű tömbre, hogy letároljunk minden egyes lépést. Ez legyen a **lepesekSzovegesen** tömb. Ahhoz, hogy meg tudjuk határozni a lépéseket, le kell tárolnunk a fa bejárását, amihez egy **q** segédtömböt használunk. A **q** tömb minden egyes eleme lényegében egy részproblémát ír le. Egy elemnek öt adattagja van: **f** a forrásrúd, **v** a célrúd, **s** a segédrúd, **b** és **j** pedig az adott lépéssorozat jobb és baloldali határait jelölik a **lepesek** tömbben. Továbbá használunk még **elso** és **utolso** nevű változókat, amikkel azt jelöljük, hogy a **q** tömbnek éppen melyik elemeit használjuk. Az **utolso** változó minidig a tömb utolsó eleme utáni helyre, azaz az első szabad helyre mutat.

A megoldás a következőképpen történik. Megvizsgáljuk a **q** tömbnek az **elso** nevű változó által mutatott elemét. Meghatározzuk, hogy a tömb ezen elemével leírt részproblémának a középső, vagyis a „mohó” lépése az eredeti problémának hányadik lépése lesz, majd meghatározzuk magát a lépést. Megvizsgáljuk, hogy hol vannak az adott részprobléma határai (**q[elso].l** és **q[elso].r**) és a „mohó” lépés az adott intervallum középső lépése lesz. A lépést a **q** tömb ugyanezen elemének **f** és **c** adattagjaiból határozzuk meg. Ezután előállítjuk az adott probléma részproblémáit, majd eltároljuk azokat a **q** tömb végén. Ezzel az adott részproblémát megoldottnak tekinthetjük, ezért léptetjük az **elso** változó értékét. Ilyen módon egy problémának mindig meghatározzuk a „mohó” lépését és a részproblémáit, majd ugyanezt tesszük a részproblémákkal. Ezt addig folytatjuk, amíg már csak egy lépésből álló problémák lesznek. Ezek megoldását már csak a „mohó” lépés meghatározása jelenti, ami csak 1-1 lépés. Amint ezzel megvagyunk, a **lepesekSzovegesen** tömbben előáll a kezdeti probléma megoldását jelölő lépéssorozat.

Ahhoz, hogy a lépéseket szöveges formában ki tudjuk írni, egy **helyettesito** nevű kétdimenziós String tömböt használunk. Minden egyes lépés meghatározásakor az dől el, hogy a **helyettesito** tömb melyik eleme kerüljön a **lepesekSzovegesen** tömb megfelelő helyére. A **helyettesito** tömb szerkezetét az 1. táblázat mutatja.

””	forras + „→” + cel	forras + „→” + seged
cel + „→” + forras	””	cel + „→” + seged
seged + „→” + forras	seged + „→” + cel	””

1. táblázat

```
public static void Greedy(int n, char a, char b, char c){

    int elso = 0;
    int utolso = 1;
    SorElem[] q = new SorElem[lepesekSzama];
    q[0] = new SorElem();
    q[0].f = a;
    q[0].v = b;
    q[0].s = c;
    q[0].b = 0;
    q[0].j = lepesekSzama - 1;
    int m;
    String[][] helyettesito = new String[3][3];
    for ( int i = 0; i < 3; i++ ){
        for ( int j = 0; j < 3; j++ ){
            if ( i != j )
                helyettesito[i][j] = (char)( 'a' + i ) +
                    " -> " + (char)('a'+j);
            else
                helyettesito[i][j] = "";
        }
    }
    while ( elso < utolso ){
        m = ( q[elso].b + q[elso].j ) / 2 ;
        lepesekSzovegesen[m] =
            helyettesito[q[elso].f -'a'][q[elso].c-'a'];

        if( q[elso].b < q[elso].j ){
            q[utolso] = new SorElem();
            q[utolso].f = q[elso].f;
            q[utolso].v = q[elso].s;
            q[utolso].s = q[elso].v;
            q[utolso].b = q[elso].b;
```

```
            q[utolso].j = m - 1;
            ++utolso;
            q[utolso] = new SorElem();
            q[utolso].f = q[elso].s;
            q[utolso].v = q[elso].v;
            q[utolso].s = q[elso].f;
            q[utolso].b = m + 1;
            q[utolso].j = q[elso].j;
            ++utolso;
        }
        ++elso;
    }
    for( int i = 0; i < lepesekSzama; i++ )
        System.out.println( lepesekSzovegesen[i] );
}
```

III.3 Dinamikus programozási megoldás

A rekurzív formulát úgy is értelmezhetjük, mint annak a leírását, ahogyan az optimális megoldás a részproblémák optimális megoldásából felépül (az optimalitás elve).

A dinamikus programozási megoldás a következőképpen működik: a megoldás a triviális részproblémák (esetünkben egyetlen korong áthelyezése) optimális megoldásával kezdődik, és folyamatosan építi fel az egyre összetettebb részproblémákat, és azok optimális megoldását, majd végül előáll az eredeti probléma az optimális megoldásával együtt. Erre mondjuk, hogy lentről felfelé haladunk. A mi esetünkben a lentről felfelé haladás azt jelenti, hogy megoldjuk a problémát $k = 1, 2, 3, \dots, n$ -re. Minden egyes k esetén hat különböző k-méretű részprobléma van: P(k, a, b, c), $P(k, a, c, b)$, $P(k, b, c, a)$, $P(k, b, a, c)$, $P(k, c, a, b)$ és $P(k, c, b, a)$. Egyszerű betűcserével a hat részprobléma közül akármelyik előállítható a többi részprobléma bármelyikéből. Például a $H_{min}(k, c, a, b)$ részprobléma optimális megoldásának lépéssorozata úgy állítható elő a $H_{min}(k, a, b, c)$ optimális megoldásának lépéssorozatából, hogy az a-t c-re cseréljük, a b-t a-ra, valamint a c-t b-re. Ebből kifolyólag minden
$k = 1, 2, 3, \dots, n$ esetén elég egyetlen k-méretű részproblémát megoldanunk.

A dinamikus programozás nehézsége abban rejlik, hogy általában nem egyszerű meghatározni azt, hogy melyik részproblémákat, és milyen sorrendben kell megoldani. A 2. ábrán az egymás fölé rétegeződő ellipszisek jelzik a lentről felfelé építkező stratégiát, amit a dinamikus programozás követ. Az ábrán azt is jól láthatjuk, hogy a $H_{min}(3, a, b, c)$ probléma megoldása két 2-méretű és öt 1-méretű részprobléma optimális megoldásából épül fel. $n = 3$ esetén a $H_{min}(3, a, b, c)$ probléma megoldási sorrendje dinamikus programozással a következő:

1. $H_{min}(1, a, b, c)$
2. $H_{min}(2, a, c, b)$ – a $H_{min}(1, a, b, c)$ és a $H_{min}(1, b, c, a)$ 1-méretű részproblémák megoldásából épül fel
3. $H_{min}(3, a, b, c)$ – a $H_{min}(2, a, c, b)$ és a $H_{min}(2, c, b, a)$ 2-méretű részproblémák megoldásából épül fel

Az ábrán az egymás fölé rétegeződő ellipszisek által tartalmazott növekvő rész-fa sorozat reprezentálja azt a növekvő részprobléma sorozatot, amelyet a dinamikus

programozási stratégia során meg kell oldani. Ebben a sorozatban az aktuális részprobléma a bal oldali részproblémáját követi, és megelőzi a szülő részproblémáját. A megoldandó részprobléma sorozat attól függ, hogy n páros vagy páratlan. A sorozat a következő, ha n páratlan:

$$H_{min}(1,a,b,c),\ H_{min}(2,a,c,b),\ H_{min}(3,a,b,c),\ H_{min}(4,a,c,b),\ \dots,\ H_{min}(n,a,b,c)$$

Ha n páros:

$$H_{min}(1,a,c,b),\ H_{min}(2,a,b,c),\ H_{min}(3,a,c,b),\ H_{min}(4\ a,b,c),\ \dots,\ H_{min}(n,a,c,b)$$

Elég nehéz egy olyan iteratív algoritmust készíteni, ami meghatározza ezt a „dinamikus programozási sorrendet", úgyhogy ajánlott lenne megpróbálni a rekurzív formula által meghatározott sorrendet használni. Sajnos a rekurzív formula közvetlen átírása általában egy nem túl hatékony „oszd meg és uralkodj" algoritmust eredményez.

Mivel a dinamikus programozási problémák megoldásánál gyakran újra előkerülnek ugyanazok a részproblémák, az „oszd meg és uralkodj" megközelítés általában azt eredményezi, hogy többször is megoldjuk ugyanazokat a részproblémákat. Az ilyen esetek elkerülésére szokták használni az úgynevezett „eredménytárolásos rekurzió" (memoization) módszert. A lényege az, hogy ha egy részproblémának már megtaláltuk az optimális megoldását, akkor azt eltároljuk, és a későbbiekben, ha újra találkozunk ugyanezzel a részproblémával, akkor egyszerűen megkeressük az eltárolt megoldást.

A hanoi probléma esetén egy részprobléma megoldását az jelenti, ha hatékonyan megoldjuk a rész-részproblémáit, vagyis minden részproblémát annyiszor kell megoldani, ahányszor az algoritmus találkozik velük. Ha a probléma nem túl nagy, akkor eltárolhatjuk a részproblémák megoldását. Ahogyan már előzőleg is szól volt róla, elég minden $k = 1, 2, 3, \dots, n$-hez egyetlen optimális megoldást eltárolni. Sajnos azonban a megoldások kezelése (eltárolás, visszakeresés, másik megoldás generálása, kiírás) legalább annyi időt vesz igénybe, mint újra előállítani ezeket a megoldásokat. Vagyis a mi esetünkben a tárolás nem csökkenti az algoritmus időigényét a közvetlen „oszd meg és uralkodj" formulához képest. Visszatérve az „eredménytárolásos rekurzió" módszerhez, megfigyelhetjük, hogy egy részprobléma sorozat megoldása során csak két féle problémával találkozunk: $H_{min}(k,a,b,c)$ és $H_{min}(k,a,c,b)$. Az első típusú ($H_{min}(k,a,b,c)$) részprobléma esetén a nagy korong áthelyezéséhez tartozó lépés $a \to b$ (0-val jelöljük). Később, miután megoldottuk ennek a

problémának a bal oldali részproblémáját ($H_{min}(k-1, a, c, b)$), annak megoldásából betűcserével generálhatjuk a jobb oldali részprobléma ($H_{min}(k, a, c, b)$) megoldását: az a-t c-re cseréljük, b-t a-ra, valamint c-t b-re. Használjuk a hat lehetséges lépéshez a következő kódokat:

0. $a \to b$
1. $a \to c$
2. $b \to c$
3. $c \to b$
4. $c \to a$
5. $b \to a$

Ilyen kódolás mellett, az előbb említett betűcserét a következő kódcserével valósíthatjuk meg: a 0-t 4-re cseréljük, 1-t 3-ra, 2-t 0-ra, 3-at 5-re, 4-t 2-re, valamint az 5-t 1-re.

A másik típusú ($H_{min}(k, a, c, b)$) probléma esetén a nagy korong áthelyezéséhez tartozó lépés $a \to c$ (1-el jelöljük). Az ehhez tartozó kódcserék: 0-t 2-re cseréljük, 1-t 5-re, 2-t 4-re, 3-t 1-re, 4-t 0-ra, valamint 5-t 3-ra.

A megoldás során ezeket a kódcseréket egy **lepesCserelo** nevű tömbben tároljuk. A tömbnek két sora van. Az első sorban az első típusú ($H_{min}(k, a, b, c)$), a másik sorában pedig a másik típusú ($H_{min}(k, a, c, b)$) részprobléma esetén használandó új kódokat tároljuk. A tömböt a metóduson kívül ajánlott deklarálni, mivel a metódust rekurzívan fogjuk hívni, és így nem kerül újra eltárolásra a memóriában.

```
static int[][] lepesCserelo = { {4, 3, 0, 5, 2, 1},
                                {2, 5, 4, 1, 0, 3} };
```

A **lepesek** tömbben a lépések kódjait tároljuk. A **minta** változóban tároljuk, hogy éppen melyik típusú problémáról van szó, és ez egyben a legnagyobb korong áthelyezéshez tartozó lépés kódját is jelenti. A függvény rekurzív hívásával folyamatosan előállítjuk mindig az adott részprobléma baloldali részproblémájának az optimális megoldását, majd ezután eltároljuk az adott problémához tartozó legnagyobb korong lépését. Ezután ezekből a megoldásokból kódcserével előállítjuk a jobboldali részproblémák megoldásait.

```
public static void Memoization( int k, int n ){
    int p, minta;
```

```
    minta = ( (n+k)&1 );
    if ( k == 1 )
        lepesek[0] = minta;
    else {
        Dynamic( k - 1, n );
        p = (int)( java.lang.Math.pow( 2, k - 1 ) - 1 );
        lepesek[p] = minta;
        for ( int i = 0; i < p ; i++ )
            lepesek[p+i+1] = lepesCserelo[minta][lepesek[i]];
    }
}
```

Összességében elmondhatjuk, hogy a Rekurziv_min (Rekurziv_max) és a Memoization algoritmusok megvalósítják az optimalitás elvét, és lentről felfelé építkezés miatt mondhatjuk, hogy ez dinamikus programozás.

Meg kell jegyeznünk, hogy ha a rekurzív formulákat fentről lefelé vizsgáljuk, akkor azt írják le, ahogy a „mohó” döntések a problémát hasonló részproblémákra osztja. Ugyannak a formulának a lentről felfelé haladó vizsgálata már azt mutatja, hogy hogyan épül fel részproblémák optimális megoldásából a probléma optimális megoldása. Ezért mondhatjuk, hogy az algoritmus egyszerre „mohó” és dinamikus programozási stratégia is.

III.4 Backtrack algoritmus

A backtrack (visszalépéses keresés) stratégia alapján az optimális megoldást, mint lépéssorozatot keressük. A probléma megoldási folyamat bármely állapotában három lehetséges lépés van. A három rudat a rajtuk lévő legkisebb korongok egymáshoz viszonyított mérete alapján elnevezhetjük „nagy”-nak, „közepes”-nek és „kicsi”-nek. Ezek alapján a három lehetséges lépés: „kicsi” → „közepes”, „kicsi” → „nagy”, „közepes” → „nagy”. A probléma állapotterét itt is gyökeres fával ábrázolhatjuk. A fa minden egyes csomópontja a probléma egy állapotának felel meg. Minden állapot egy halmaz hármassal írható le. Egy halmaz elemei egy adott rúdon lévő korongokat jelölik. Mivel a probléma minden állapotában három lehetséges lépés van, ezért minden csomópontnak (a levélelemek kivételével) három leszármazottja van. A gyökérelem reprezentálja a probléma kezdőállapotát, vagyis azt az állapotot, mikor minden korong az a rúdon van: $\{(1, 2, \dots, n); (); ()\}$. A célállapot (megoldás-levél) az, mikor minden korong a b rúdon van: $\{(); (1, 2, \dots, n); ()\}$. A gyökérelem és a célállapotot tartalmazó levélelem közötti legrövidebb (maximum verzió esetén a leghosszabb) út reprezentálja az optimális megoldást.

A backtrack algoritmus depth-first keresést alkalmaz (amit általában rekurzívan implementálunk), és az optimális megoldást az általános minimum-/maximum-kiválasztásos keresési algoritmus alapján választja ki. A hanoi probléma esetén a backtrack stratégia a maga primitív formájában csak a hurkokat kerüli el, nem túl hatékony. Például a minimum verzió optimális megoldását ábrázoló fának (2. ábra) $2^n - 1$ csomópontja van (feltéve, hogy a gyökér-elemek magukba foglalják a saját-levelüket), vagyis annyi, amilyen hosszú az optimális megoldás lépéssorozata. Az állapottér-fa esetében csak a gyökérelem és a megoldás-levél közötti útvonalon ennyi csomópont van. Mivel a backtrack algoritmusnak emellett még az aktuális lépéssorozatot is el kell tárolnia, ez az algoritmus csak kisebb n értékek esetén használható.

A backtrack algoritmushoz nem készítettem külön kódot, mivel ez lényegében ugyanaz a megoldás, mint az állapottér reprezentációval történő megoldás backtrack keresővel. Gyorsabb lenne ugyan, de ez a sebességnövekedés elhanyagolható, mert a backtrack algoritmus még így is nagyon lassú a többihez képest, nem is beszélve a hatalmas memóriaigényről.

III.5 Iteratív algoritmus

Az előzőleg bemutatott megoldások után felmerül a kérdés: egy iteratív algoritmus le tudná generálni az optimális megoldás lépéssorozatát? Beláthatjuk, hogy a Dynamic átírható egy iteratív dinamikus programozási algoritmussá.

Az előzőekben bemutatott „dinamikus programozási sorrend" alapján az algoritmus szülőről szülőre halad. Bármely szülő-probléma megoldása előállítható a bal oldali részproblémájának optimális megoldásából. Ha a lentről felfelé haladás során az aktuális részprobléma $H_{min}(k, x, y, z)$, akkor a jobb oldali testvérproblémája $H_{min}(k, y, z, x)$, a szülő problémája $H_{min}(k+1, x, z, y)$, és a szülő probléma legnagyobb korongjának áthelyezéséhez szükséges lépés $x \to z$. Mint azt már korábban említettük, a jobb oldali részprobléma optimális megoldása egyszerű betűcserével előállítható a bal oldali részproblémából. A $H_{min}(k+1, x, z, y)$ szülő probléma megoldása úgy áll elő, hogy miután megoldottuk a bal oldali részproblémát ($H_{min}(k, x, y, z)$), áthelyezzük a $(k+1)$-es korongot az x rúdról a z rúdra, majd legeneráljuk a jobb oldali testvérprobléma ($H_{min}(k, y, z, x)$) megoldását. A kiinduló probléma $H_{min}(1, a, b, c)$ vagy $H_{min}(1, a, c, b)$ attól függően, hogy n páratlan vagy páros. Az Iterativ1 eljárás ezt az iteratív dinamikus programozási stratégiát mutatja be.

```
public static void Iterativ1( int n ){
    int p, minta;
    lepesek[0] = minta = ( (n+1) % 2 );

    for ( int k = 2; k <= n; ++k ){
        p = (int)(java.lang.Math.pow(2, k-1) - 1);
        minta ^= 1;
        lepesek[p] = minta;
        for ( int i = 0; i < p; i++ ){
            lepesek[p+i+1] = lepesCserelo[minta][lepesek[i]];
        }
    }
}
```

A fő probléma az Iterativ1 eljárással továbbra is a számítógépek korlátozott memóriakapacitása. Hogyan tudnánk kiküszöbölni ezt az összetevőt? A megoldás a következő megfigyeléseken alapul (A probléma minimum verzióját vizsgáltuk, és az előzőleg felvázolt „kicsi", „közepes" és „nagy" rúdelnevezéseket használjuk):

- A probléma megoldási folyamat bármely közbenső állapotában három lehetséges lépés van: „kicsi" → „közepes", „kicsi" → "nagy", „közepes" → „nagy".
- A „kicsi" rúdon lévő legfelső korong mindig a legkisebb korong. Ahhoz, hogy elkerüljük a hurkokat, és hogy fenntartsuk a minimális lépésszámot, a legkisebb koronggal nem végezhetünk két egymás utáni lépést. Ha ez megvalósul, akkor a legkisebb korong és a „közepes" rúdon lévő legkisebb korong felváltva mozog.
- Ha a következő lépésben a „közepes" rúd legfelső korongját kell mozgatni, akkor egyértelmű, hogy a „közepes" → „nagy" lépést kell végrehajtani. Ezeket a lépéseket a saját-levelek jelölik. (2. ábra)
- A legkisebb korong az $(a, b, c, a, b, c, \dots)$ vagy az $(a, c, b, a, c, b, \dots)$ mintának megfelelően mozog attól függően, hogy az n páros vagy páratlan. Ebből kifolyólag a legkisebb korong lépései egyértelműen meghatározottak.

Az utolsó megjegyzés magyarázatra szorul. Ezek a lépésminták a lentről felfelé építkező folyamat elemzéséből kaphatóak meg, és helyességük matematikai indukcióval bizonyítható. Tegyük fel, hogy a megoldandó probléma $H_{min}(n, a, b, c)$, és n páratlan. Bebizonyítjuk, hogy ebben az esetben a legkisebb korong az $(a, b, c, a, b, c, \dots, c, a, b)$ lépésmintát követi, ami 2^{n-1} hosszúságú.

$n = 1$ esetén csak egyetlen korong van, ez a legkisebb korong. Ennek lépése $a \to b$. Tegyük fel, hogy adott $n > 1$ esetén, ahol n páratlan, a legkisebb korong 2^{n-1} hosszú lépéssorozata a következő: $a, b, c, a, b, c, \dots, c, a, b$. A $H_{min}(n, a, b, c)$ probléma jobb oldali testvérproblémája $H_{min}(n, b, c, a)$. Ennek megfelelően a legkisebb korong lépéssorozata: $b, c, a, b, c, a, \dots, a, b, c$. Ezeket a mintákat összefűzve megkapjuk a szülő problémának megfelelő $H_{min}(n + 1, a, c, b)$ probléma 2^n hosszú lépéssorozatát: $a, b, c, a, b, c, \dots, a, b, c$. Ezt az eljárást megismételve megkapjuk a $H_{min}(n + 2, a, b, c)$ nagyszülő probléma 2^{n+1} hosszú lépéssorozatát: $a, b, c, a, b, c, \dots, c, a, b$. Ugyanezt a gondolatmenetet alkalmazhatjuk páros n-re is.

$H_{min}\,(1, a, b, c)$: $(a \to b)$

$H_{min}\,(2, a, c, b)$: $\{(a \to b)\}, [a \to c], \{(b \to c)\}$

$H_{min}\,(3, a, b, c)$: $\{(a \to b), [a \to c], (b \to c)\}, [a \to b], \{(c \to a), [c \to b], (a \to b)\}$

a, b, c, a, b

3. ábra: Kerek zárójellel jelöltük a legkisebb korong lépéseit és szögletes zárójellel az aktuális probléma legnagyobb korongjának a lépését ($\mathbf{k} > 1$ esetén). A kapcsos zárójelpárok jelölik a testvér-részproblémákat.

Következtetésképpen mondhatjuk, hogy ezen iteratív algoritmus által alkalmazott stratégia nagyrészt dinamikus programozás, a következő okok miatt:

- Az algoritmus azt a lépéssorozatot generálja, amely a lentről felfelé építkező dinamikus programozási stratégiát implementálja.
- A lépéssorozat előállítása optimalizáláson alapul, beleértve az optimalitás elvét is.
- A legkisebb korong lépéssorozat-mintája megkapható a rekurzív formula lentről felfelé építkező elemzéséből.

Érdekesség, hogy a legkisebb korong által követett lépéssorozat megkapható a probléma „mohó” megközelítéséből is. A $H_{min}\,(n, a, b, c)$ probléma megoldása során a következő lépés-mintákat kapjuk:

- az összes (n darab) korongot át kell rakni a rúdról b rúdra: $(a >> b)$; (a és b állapot között lehetnek más állapotok is)
- az 1-es korongot átrakjuk a-ról b-re: $a \to b$; (a korongot közvetlenül rakjuk át egyik rúdról a másikra, ez a mohó lépés)
- a felső $n-1$ darab korong lépései: $a >> c >> b$; (a felső $(n-1)$ korong a, c és b állapotokon mennek keresztül
- a 2-es korong lépései: $a \to c \to b$, mohó lépések
- a felső $n-2$ darab korong lépései: $a >> b >> c >> a >> b$;
- a 3-as korong lépései: $a \to b \to c \to a \to b$, mohó lépések
- a felső $n-2$ darab korong lépései: $a >> c >> b >> a >> c >> b >> a >> c >> b$;

- a 4-es korong lépései: $a \to c \to b \to a \to c \to b \to a \to c \to b$
- …

Mivel minden $k > 1$ esetén a $H_{min}(k, x, y, z)$ részproblémát leegyszerűsítettük $H_{min}(k-1, x, z, y)$ és $H_{min}(k-1, z, y, x)$ részproblémákra, a lépéssorozatot úgy kapjuk az előzőből, hogy az egymás utáni állapotok közé befűzzük a „harmadik állapotot". Megfigyelhetjük, hogy csak kétféle minta van, egy páratlan számú korong esetére $(a, b, c, a, b, c, \dots)$ és egy páros számú korong esetére $(a, c, b, a, c, b, \dots)$.

Az **Iterativ2** algoritmusban először létrehozzuk a **p** tömböt, amelyben a probléma aktuális állapotát tároljuk. A tömb **i**-edik eleme az **i**-edik korong aktuális helyzetét jelzi. Az **i** változóban tároljuk, hogy éppen melyik korong mozog. Páratlan **k** lépések esetén a legkisebb korong mozog, páros lépések esetén pedig a legkisebb korong rúdjától különböző rúdon lévő legkisebb korong. Ha **i** páratlan, akkor az aktuális korong az $a, b, c, a, b, c, \dots$ „növekedő" lépésmintát követi, ha **i** páros, akkor a $c, b, a, c, b, a, \dots$ „csökkenő" mintát követi. Az algoritmus működése során először meghatározzuk, hogy melyik korong lép, ezt kiírjuk, majd eldöntjük, hogy melyik mintát követi a lépés, és kiírjuk az ennek megfelelő lépést.

```
public static void Iterativ2(int n, char a, char b, char c){

    char[] p = new char[n+1];
    int i;
    for ( i = 1; i <= n; i++ )
        p[i] = a;
    int k = 1;

    for ( k = 1; k <= lepesekSzama; k++ ){
        if ( (k&1) != 0 )
            i = n;
        else {
            i = n-1;
            while ( p[i] == p[n] ){
                --i;
            }
```

```
            }
            System.out.print( p[i] + " -> " );
            p[i] += 1 - ( (i&1) << 1 );
            if (p[i] == c + 1 )
                p[i] = a;
            else if ( p[i] == a - 1 )
                p[i] = c;
            System.out.println( p[i] );
        }
    }
```

IV. Elemzés

Az állapottér-reprezentációval készült megoldásokat a többitől külön vizsgálom, mivel ezek sokkal lassabbak, és nem is ugyanazt az eredmény nyújtják. A tesztkörnyezet:

- INTEL Core i5-750 2.66Ghz 4 magos processzor, engedélyezett Turbo Boost móddal
- GIGABYTE GA-P55-US3L alaplap
- 4GB Kingston HyperX DDR3 1600Mhz memória
- Windows 7 Professional Edition operációs rendszer
- NetBeans IDE 6.7.1 fejlesztői környezet

A két különböző állapottér-reprezentáció eredményei szinte ugyanazok voltak, főleg több korong esetén, ezért csak az eredményeik átlaga szerepel a táblázatban. A teszteket többször futtattam, a táblázatban látható eredmények az átlageredmények. A backtrack keresőt körfigyeléssel és az adott méretű probléma optimális megoldásának hosszúságával egyenlő úthossz korláttal használtam, hogy az optimális megoldást találja meg. Ez 6 koronggal már nem működött. Szélességi és mélységi kereső esetén az első megoldásig kerestem. Ezek eredménye nagyon hasonló volt, ezért a táblázatban egybefoglaltam. A futási idők milliszekundumban értendőek. Kevés korongnál az eredmények nagyon ingadozóak voltak, ezért a nagyobb korongszámú mérések többet mondanak.

Korongok száma	Lépések száma	Keresőgráfos keresők	Backtrack kereső
3	7	5	5
4	15	13	19
5	31	18	2580
6	63	55	-
7	127	320	-
10	1023	9560	-

2. táblázat

A Rekurziv, Greedy, Memoization és Iterativ2 algoritmusok eredményeit a 3. táblázat mutatja. A mért időeredményekben jelentős szerepet játszottak a kiíratások, ezért ezeket az algoritmusokat két különböző módon mértem. Először kiíratással, majd kiíratás nélkül.

Utóbbi esetben a lépések olyan szöveges adatként tárolódtak, amik egyébként kiíródnának. A mért eredményeket átlagolás során nagyobb súllyal vettem figyelembe a kiíratás nélküli eseteket. Emiatt a táblázatban nem időeredmények szerepelnek, de a számok jól tükrözik az algoritmusok egymáshoz viszonyított sebességét.

Korongok száma	Lépések száma	Rekurziv_min	Greedy	Memoization	Iterativ2
3	7	1	1	1	1
4	15	1	1	1	1
5	31	1	1	1	1
6	63	2	2	2	2
7	127	4	4	3	3
10	1023	25	32	5	5
15	32767	956	1253	110	139
20	1048575	30154	43257	5137	5240

3. táblázat

Láthatjuk, hogy az Iterativ2 és a Memoizaton algoritmusok sokkal hatékonyabbak a többinél. Mindemellett a többihez képest a rekurzív algoritmusnak sokkal nagyobb a memóriaigénye, a „mohó” algoritmusnak pedig a tárigénye. Az eltérő mérési módok és működés miatt nem látszik, de az állapottér-reprezentációval készült megoldások sokkal lassabbak a probléma specifikus megoldásoktól.

Irodalomjegyzék

1. Stuart Russell, Peter Norvig: Mesterséges intelligencia modern megközelítésben, Budapest 2005
2. Csákány Antal, Dr. Vajda Ferenc: Játékok számítógéppel, Budapest 1985
3. Zoltán Kátai, Lehel István Kovács: Towers of Hanoi – where programming techniques blend, Acta Universitatis Sapientaiae, Informatica Vol1, No1, 2009, 89-108 old.
4. Várterész Magda: Mesterséges intelligencia előadások http://www.inf.unideb.hu/~varteres/mi1folia/foliafo.pdf
5. A mesterséges intelligencia alapjai című tárgy honlapja: https://it.inf.unideb.hu/mestint/fooldal?action=show&redirect=FrontPage
6. Java SE 6 documentation http://java.sun.com/javase/6/docs/api/
7. http://mattort.fvt.hu/cikk.php?cikk=hanoi