
# Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába

Doktori (Ph.D.) értekezés

Kósa Márk Szabolcs

Témavezető: Dr. Várterész Magda

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika- és Számítástudományok Doktori Iskola

Debrecen, 2009


# Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába

Doktori (Ph.D.) értekezés

Kósa Márk Szabolcs

Témavezető: Dr. Várterész Magda

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika- és Számítástudományok Doktori Iskola

Debrecen, 2009

Ezen értekezést a Debreceni Egyetem Természettudományi Doktori Tanács Matematika és Számítástudományok Doktori Iskola *Informatika* programja keretében készítettem a Debreceni Egyetem természettudományi doktori (Ph.D.) fokozatának elnyerése céljából.

Debrecen, 2009. június 30.

Kósa Márk Szabolcs
doktorjelölt

Tanúsítom, hogy Kósa Márk Szabolcs doktorjelölt 2003–2006 között a fent megnevezett Doktori Iskola *Informatika* programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javasolom.

Debrecen, 2009. június 30.

Dr. Várterész Magda
témavezető

**Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába**

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében az informatika tudományágban

Írta: Kósa Márk Szabolcs okleveles programtervező matematikus

Készült a Debreceni Egyetem
Matematika és Számítástudományok Doktori Iskolája
(Informatika programja) keretében

Témavezető: Dr. Várterész Magda

A doktori szigorlati bizottság:

elnök: Dr. .....
tagok: Dr. .....
Dr. .....

A doktori szigorlat időpontja: 200... . ..... ... .

Az értekezés bírálói:

Dr. .....
Dr. .....
Dr. .....

A bírálóbizottság:

elnök: Dr. .....
tagok: Dr. .....
Dr. .....
Dr. .....
Dr. .....

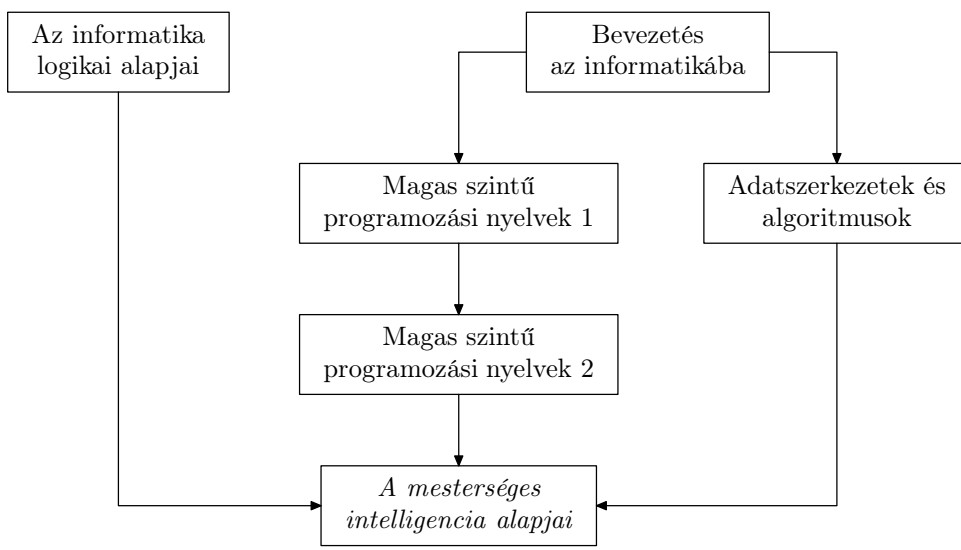
Az értekezés védésének időpontja: 200... . ..... ... .

# Tartalomjegyzék

1. FEJEZET

# Bevezetés

Az elmúlt évtizedben az objektumorientált szemléletmód széles körű elterjedését figyelhettük meg az informatikai világban. E technika elterjedésének szükségszerűen meg kellett jelennie az informatika oktatásában is. Dolgozatomban bemutatom azt, hogy milyen lehetőségeket rejt az objektumorientált programozási szemlélet térnyerése az egyetemi informatikai oktatásban és tehetséggondozásban. Vizsgálódásaim terepét a Debreceni Egyetemen a BSc szintű programtervező informatikus képzés egyik kötelező tárgyának, *A mesterséges intelligencia alapjai* című kurzusnak a gyakorlati foglalkozásai jelentették.




1.1. ábra. *A mesterséges intelligencia alapjai* című tárgy alapozó tárgyai

A Debreceni Egyetemen a BSc szintű programtervező informatikus képzésben *A mesterséges intelligencia alapjai* című kurzus négy olyan tantárgy ismereteire épül,

amelyekkel hallgatóink tanulmányaik korábbi féléveiben már találkozhattak. Ezek kapcsolatát szemlélteti az 1.1. ábra.

A tárgyat a képzési struktúra negyedik félévében vehetik fel először a hallgatók. Ekkor már – szerencsés esetben – túl vannak a két féléves programozás képzésen, ahol megismerkedhettek a C programozási nyelvvel és egy objektumorientált nyelv (Java vagy C#) alapjaival, valamint túl vannak az informatikai logika alapjait tárgyaló kurzuson is. Mindemellett hallgathattak előadást a különböző adatszerkezetekről és a hozzájuk kapcsolódó algoritmusokról is. Így aztán fontosnak tartom kiemelni, hogy a kurzust látogató hallgatók nem ezen a kurzuson kerülnek először kapcsolatba az objektumorientált világgal, már szerezhettek tapasztalatokat valamilyen objektumorientált programozási nyelvvel kapcsolatosan.

A tárgy előadásain ismerkedhetnek meg a hallgatók az MI klasszikus kutatási területeivel, alapvető módszereivel és eszközeivel, legfontosabb eredményeivel. Az alapvető módszerek közül a gráfokban megoldást kereső eljárásokat tanulják részletesen ebben a félévben.

A gráfban megoldást kereső eljárások tanításához először reprezentáljuk a problémát állapottéren vagy problémaredukcióval. Mindkét esetben megadjuk a megfelelő gráfreprezentációt. A megoldást kereső rendszerek felépítésének elemzése után rátérünk a konkrét eljárások tanítására. Az algoritmusainkat különböző szempontok alapján csoportosíthatjuk (nem módosítható – módosítható; nem informált – heurisztikus stratégiák). Elsősorban a módosítható kereső eljárásokkal (backtrack algoritmus; keresőgráffal történő keresések: szélességi, mélységi, optimális gráfkereső; heurisztikus gráfkeresők: best-first eljárás, A-algoritmusok) foglalkozunk. Vizsgáljuk ezen eljárások algoritmikus tulajdonságait is (teljesség, megfelelőség, bonyolultság).

A problémaredukciós feladatmegoldásnak megfelelő ÉS/VAGY gráfokban való megoldáskereséseket mint az eddigi algoritmusok általánosításait tárgyaljuk. Megadjuk a kétszemélyes, zérusösszegű teljes információjú játékok nyerő stratégiájának fogalmát, és módszert nyújtunk legjobbnak tűnő következő lépés kiválasztására (minimax, alfa-béta algoritmusok).

A mesterséges intelligencia tárgy gyakorlati kurzusának anyagát két nagy témakör: a gráfokban való keresések, valamint a kétszemélyes játékok algoritmusainak témaköre köré lehet csoportosítani ([7]). Mindkét témakör kiválóan alkalmas objektumorientált módon történő megközelítésre, ugyanakkor számtalan olyan optimalizálási lehetőséget is nyújt, amelyeket az objektumorientált megközelítés általánosításai elrejtenek a felületes szemlélő elől. Ez az utóbbi tény inspirált arra, hogy bizonyos problématípusok (problémaosztályok) esetén ne csak a magas szintű absztrakciót nyújtó objektumorientált lehetőségeket vizsgáljam meg, hanem más módszerekkel is megoldásokat adjak a kitűzött feladatokra.

Az egyes problémaosztályok optimalizálási lehetőségeinek feltérképezése azért is fontos, mert a tárgy keretében elsajátított algoritmusok az informatika más területein is visszaköszönnek a programozókra, hol látványosan (web- és mobilalkalmazások), hol kevésbé látványosan (adatbázis-alkalmazások). Napjainkban ugyan az erőforrások jelentősen megsokszorozódtak, de megfontolt felhasználásukat szintén ezeken a kurzusokon sajátíthatják el a hallgatók.

A hallgatók felé átadott ismeretek számonkérésének sokféle módja létezik. Egy le-

hetőség, amivel az elmúlt években többször is éltünk, a különféle versenyek szervezése. A mesterséges intelligencia kurzusokon elsajátított ismereteiket hallgatóink egyrészt a tárgy keretein belül meghirdetett háziversenyeken, másrészt pedig az Informatikai Kar[1] által szervezett programozói versenyeken tehetik próbára. Utóbbiak esetében ACM stílusú programozó versenyekről van szó, amelyeken az 5 óra programozási idő alatt megoldandó 7–10 feladat között sokszor szokott előfordulni olyan feladat, amelynek a megoldásához felhasználható a mesterséges intelligencia témaköreihez köthető algoritmus. Mindkét versenyfajtánál kulcsfontosságú a feladatok megoldásainak pontos ellenőrzése. Ezt 2002-től 2007-ig egy saját fejlesztésű szoftverrel, a Programming Contest Result Manager (PCRM) programmal végeztük. A programot eredetileg az Informatikai Karon szervezett ACM stílusú programozói versenyek lebonyolításához készítettük, de bebizonyosodott, hogy kiválóan alkalmazható néhány tantárgy (*Magas szintű programozási nyelvek*, *A mesterséges intelligencia alapjai*) házi feladatainak ellenőrzésére is.

Mindezen tényekből kiindulva dolgozatom megírásakor célkitűzéseim a következők voltak:

- a mesterséges intelligencia alapvető algoritmusainak bemutatása az informatikában ma elterjedt objektumorientált szemléletben, felhasználva és integrálva a tárgy előfeltétel-kurzusain (logika, programozás, adatszerkezetek és algoritmusok) tanult ismereteket;
- a Debreceni Egyetem Informatikai Karán, a mesterséges intelligencia tárgy bevezető kurzusának gyakorlatain a hallgatóknak bemutatott és tőlük számonkért állapottér-reprezentációs technikának a beillesztése az objektumorientált modellbe;
- az objektumorientált megközelítésből származó előnyök és hátrányok felmérése;
- a „hagyományos” technikák (elemi adatszerkezetek használata, dinamikus programozás) alkalmazhatóságának vizsgálata speciális problématípusok esetén;
- az ACM nemzetközi programozó versenyek néhány feladatának elemzése és felhasználása egyrészt a gyakorlati kurzusokon történő oktatás, másrészt a programozói versenyeken induló hallgatók felkészítésének eredményesebbé tétele érdekében.

A dolgozat egyes fejezeteiben látni fogjuk, hogy az egyes alapozó tárgyak ismeretanyaga milyen mértékben kapcsolódik a mesterséges intelligencia bevezető kurzusának anyagához. A 2. fejezetben bemutatjuk a problémák állapottér-reprezentálásának technikáját. Ugyanitt látni fogjuk, hogy a logika informatikai alapjai hogyan jelennek meg ebben a folyamatban. A 3. fejezetben ismertetjük a gráfokban megoldást kereső eljárásokat, a köztük lévő kapcsolatokat és ezek objektumorientált megközelítését. A 4. fejezet a kétszemélyes, véges, determinisztikus, zérusösszegű, teljes információjú stratégiai játékok lépésajánló algoritmusait mutatja be, szintén objektumorientált megközelítésben. Az 5. fejezetben részletesen olvashatunk az ACM stílusú programozói versenyekhez kifejlesztett PCRM szoftverről. Ebből a fejezetből külön is kiemelném a 5.7. alfejezetet, amely a PCRM-mel az oktatásban és a versenyek szervezésében szerzett tapasztalatokról szól.

[1] 2004 előtt a Természettudományi Kar (Matematikai és) Informatikai Intézete.

2. FEJEZET

# Állapottér-reprezentáció

## 2.1. Az állapottér-reprezentáció matematikai alapjai

Egy $p$ probléma állapottér-reprezentálásakor az $\langle \mathcal{A}, \mathit{kezdő}, \mathcal{C}, \mathcal{O} \rangle$ rendezett négyes elemeit kell definiálnunk, ahol

- $\mathcal{A}$ az állapottér, azon állapotok nem üres halmaza, amelyek a $p$ probléma világát alkotják,
- $\mathit{kezdő} \in \mathcal{A}$ egy kitüntetett állapot, amellyel a $p$ probléma kezdőállapotát írjuk le,
- $\mathcal{C} \subset \mathcal{A}$ a $p$ probléma explicit módon vagy célfeltételekkel definiált célállapotainak halmaza, és
- $\mathcal{O}$ az állapottéren értelmezett transzformációkat leíró operátorok nem üres halmaza.

Egyes problémák az operátorokhoz alkalmazási költségeket írhatnak elő. Az állapottér-reprezentáció elkészítése során ezeket az alkalmazási költségeket az operátorok definiálásával együtt adjuk meg. A problémák egységes kezelése érdekében célszerű akkor is értelmezni operátor-alkalmazási költségeket, ha az a kitűzött feladat leírásában explicit módon nem szerepel. Ezekben az esetekben az operátorok alkalmazási költségét egységnyinek fogjuk tekinteni.

A probléma világát alkotó állapotok halmaza (az állapottér) nem más, mint a problémát leíró, a feladat megoldása szempontjából fontosnak tartott tulajdonságokhoz tartozó halmazok Descartes-szorzatának a kényszerfeltételek által kijelölt részhalmaza. A Descartes-szorzatot alkotó halmazokra a továbbiakban a feladatra jellemző tulajdonságok alaphalmazaiként fogunk hivatkozni.

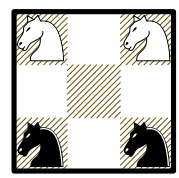
Látható, hogy ez a modell meglehetősen nagy szabadságot biztosít számunkra mind az állapotteret jellemző alaphalmazok megválasztásában, mind pedig az állapottéren értelmezett operátorok definiálásában. Ez azt jelenti, hogy egy feladathoz akár több állapottér-reprezentációt is készíthetünk. Természetesen egy modellező sem szeret feleslegesen dolgozni, az igazság azonban az, hogy a probléma részleteinek alapos elemzése nélkül, de legtöbbször még azzal együtt sem jelenthető ki egy állapottér-reprezentációról, hogy az az adott probléma legjobb reprezentációja. Fogadjuk el tehát

kiindulópontnak azt, hogy érdemes több, lehetőség szerint a probléma megközelítésében különböző reprezentációt készíteni, majd ezeket – akár mindegyiket is – sorra kipróbálni. A szakirodalomban gyakran tárgyalt iskolapéldák (a Sam Lloyd-féle nyolcas kirakójáték [4, 12], a Hanoi tornyai probléma [4], illetve a négy [4] vagy nyolc királynő [12] problémák) helyett néhány szintén ismert, de ritkábban említett probléma segítségével szeretném szemléltetni a lehetőségeinket.

## 2.2. *Huszárcsere*

### 2.2.1. A probléma

Adott egy 3×3-as sakktábla, amelynek az alsó sarkaiban sötét huszárok, a felső sarkaiban világos huszárok állnak úgy, ahogyan az az 2.1-es ábrán látható.

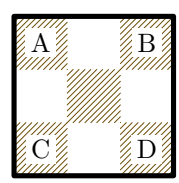


2.1. ábra. A huszárok kiinduló helyzete

A célunk az, hogy szabályos lólépésekkel felcseréljük a világos huszárokat a sötétekkel.

### 2.2.2. Egy mátrix négy különböző huszárral

Az egyik lehetséges megközelítés szerint a probléma jellemző tulajdonsága az, hogy a sakktábla 3×3 mezőjén egy adott pillanatban melyik sakkbábu áll. Mivel négy sakkbábunk van, jelölhetjük őket az ábécé első négy betűjével: A-val, B-vel, C-vel és D-vel. Attól a pillanattól fogva, ahogy ezt a négy betűt huszárjainkhoz hozzárendeljük, egyértelműen tudunk majd hivatkozni rájuk. Mondhatjuk például azt, hogy a kinduló helyzetben A-val a bal felső, B-vel a jobb felső, C-vel a bal alsó, D-vel pedig a jobb alsó sarokban álló huszárt fogjuk jelölni (2.2. ábra).




2.2. ábra. A huszárok egy lehetséges jelölése

Azt is jeleznünk kell valahogyan, hogy egy mezőn éppen nem áll rajta egyik huszárunk sem. Az ilyen mezőkhöz a továbbiakban a 0 szimbólumot fogjuk hozzárendelni.

A betűkkel ellentétben ezt a szimbólumot több mezőhöz is hozzárendelhetjük majd.

Ezek alapján meghatározhatjuk azokat az alaphalmazokat, amelyek a sakktábla egyes mezőihez hozzárendelhető értékeket tartalmazzák:

$$H_{i,j} = \{\, 0, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D} \,\}, \quad 1 \leq i \leq 3 \text{ és } 1 \leq j \leq 3.$$

A halmazok indexeléséből látható, hogy összesen kilenc ilyen alaphalmazunk van, a tábla minden mezőjéhez tartozik egy. Képezzük ezen halmazok Descartes-szorzatát:

$$H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3} =$$

$$\left\{ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{A} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{B} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{C} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{D} \end{pmatrix}, \right.$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{A} & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{A} & \mathrm{A} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{A} & \mathrm{B} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{A} & \mathrm{C} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{A} & \mathrm{D} \end{pmatrix},$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{B} & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{B} & \mathrm{A} \end{pmatrix}, \ldots, \begin{pmatrix} \mathrm{A} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{B} \end{pmatrix}, \begin{pmatrix} \mathrm{A} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{C} \end{pmatrix}, \begin{pmatrix} \mathrm{A} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{D} \end{pmatrix},$$

$$\left. \ldots, \begin{pmatrix} \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & 0 \end{pmatrix}, \begin{pmatrix} \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{A} \end{pmatrix}, \begin{pmatrix} \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{B} \end{pmatrix}, \begin{pmatrix} \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{C} \end{pmatrix}, \begin{pmatrix} \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \\ \mathrm{D} & \mathrm{D} & \mathrm{D} \end{pmatrix} \right\}$$

Ennek a Descartes-szorzatnak rendkívül sok 3×3-as mátrix alakban felírható elemkilences eleme van, szám szerint $5^9 = 1\,953\,125$ darab. Az összeset nem is lenne érdemes felsorolni, mert közülük úgyis csak azon

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \in H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3}$$

elemkilencesek lesznek problémánk állapotai, amelyekre teljesülnek a következő feltételek:

- az elemkilencesben pontosan egy A betű található:

$$\left( \sum_{i=1}^{3} \sum_{j=1}^{3} f(i, j, \mathrm{A}) \right) = 1, \tag{2.1}$$

ahol

$$f(i, j, \textit{betű}) = \begin{cases} 1, & \text{ha } h_{i,j} = \textit{betű}, \\ 0, & \text{egyébként;} \end{cases}$$

- az elemkilencesben pontosan egy B betű található:

$$\left(\sum_{i=1}^{3}\sum_{j=1}^{3} f(i,j,\mathrm{B})\right) = 1; \tag{2.2}$$

- az elemkilencesben pontosan egy C betű található:

$$\left(\sum_{i=1}^{3}\sum_{j=1}^{3} f(i,j,\mathrm{C})\right) = 1; \tag{2.3}$$

- az elemkilencesben pontosan egy D betű található:

$$\left(\sum_{i=1}^{3}\sum_{j=1}^{3} f(i,j,\mathrm{D})\right) = 1; \tag{2.4}$$

- a középső mezőre – figyelembe véve a kiinduló helyzetet – szabályos lépéssel nem juthat el egyik huszár sem:

$$h_{2,2} = 0. \tag{2.5}$$

Összességében azt mondhatjuk, hogy azok az elemkilencesek lesznek a probléma állapotai, melyekre teljesül az alábbi kényszerfeltétel, a megadott öt formula konjunkciója:

$$\textit{kényszerfeltétel}(h) \equiv (2.1) \wedge (2.2) \wedge (2.3) \wedge (2.4) \wedge (2.5)$$

Az előző pontokban mondatszerűen megfogalmazott megszorításokat mindenki magától értetődőnek találhatja, hiszen ezekben a mondatokban azt akartuk megfogalmazni, hogy a huszárok mindegyikének minden pillanatban ott kell lennie a táblán. Ami az egyes mondatok alá írt formulákat illeti, ezek már nem feltétlenül a naiv szemlélőnek szólnak, sokkal inkább azoknak a programozóknak, akiknek ezen állapottér-reprezentáció programozása lesz a feladatuk. Ezek egy elsőrendű logikai nyelv szabályos formulái, amelyeket nagyon könnyű lesz átültetni tetszőleges programozási nyelvre, amint azt a későbbiekben majd látni fogjuk.

A problémánk állapotterét tehát azok az elemkilencesek alkotják, amelyek eleget tesznek a kényszerfeltételeknek. Ez az állapottér definíció szerint részhalmaza az alaphalmazok Descartes-szorzatának:

$$\mathcal{A} = \left\{ h \;\middle|\; h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \wedge \textit{kényszerfeltétel}(h) \right\} \subset$$

$$\subset H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3}.$$

Ennek az állapottérnek azt az elemét fogjuk kezdőállapotnak tekinteni, amely a 2.2. ábrán látható helyzetet írja le:

$$\textit{kezdő} = \begin{pmatrix} \mathrm{A} & 0 & \mathrm{B} \\ 0 & 0 & 0 \\ \mathrm{C} & 0 & \mathrm{D} \end{pmatrix} \in \mathcal{A}$$

A célállapotok halmaza négy elemkilencest fog tartalmazni, hiszen a felső két és az alsó két sarokban álló huszárok négyféleképpen helyezkedhetnek el ezeken a mezőkön:

$$\mathcal{C} = \left\{ \begin{pmatrix} \mathrm{C} & 0 & \mathrm{D} \\ 0 & 0 & 0 \\ \mathrm{A} & 0 & \mathrm{B} \end{pmatrix}, \begin{pmatrix} \mathrm{C} & 0 & \mathrm{D} \\ 0 & 0 & 0 \\ \mathrm{B} & 0 & \mathrm{A} \end{pmatrix}, \begin{pmatrix} \mathrm{D} & 0 & \mathrm{C} \\ 0 & 0 & 0 \\ \mathrm{A} & 0 & \mathrm{B} \end{pmatrix}, \begin{pmatrix} \mathrm{D} & 0 & \mathrm{C} \\ 0 & 0 & 0 \\ \mathrm{B} & 0 & \mathrm{A} \end{pmatrix} \right\} \subset \mathcal{A}$$

Mindezek után álljunk meg egy pillanatra, és gondoljuk végig, hogy milyen változásokat fog eredményezni majd egy-egy operátor alkalmazása egy állapotra. Ha az operátorainkkal a sakk szabályos huszárlépését szeretnénk modellezni, akkor a lépés eredményeképpen az egyik (és csak az egyik) huszár helyzete megváltozik a táblán, amikor elugrik egy másik mezőre. Az operátorainkat úgy kell tehát megtervezni, hogy képesek legyenek ezt az ugrást megvalósítani:

- egyértelműen tudják azonosítani az ugró huszárt,
- egyértelműen tudják azonosítani azt a mezőt, ahová a huszár ugrani akar, és
- az ugrás végrehajtása után az előálló új helyzet is eleme legyen az állapottérnek.

Érezhető, hogy ilyen operátor(oka)t sokféleképpen lehetne definiálni. Mi most ezt a következőképpen tesszük: legyen

$$\mathrm{Ugrik}(h, \mathit{betű}, s, o) \colon \mathrm{dom}(\mathrm{Ugrik}) \to \mathcal{A},$$

ahol

$$\mathrm{dom}(\mathrm{Ugrik}) \subset \mathcal{A} \times \{\,\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\,\} \times \{\,1, 2, 3\,\} \times \{\,1, 2, 3\,\}.$$

Egy ilyen $\mathrm{Ugrik}(h, \mathit{betű}, s, o)$ operátor akkor alkalmazható, ha teljesülnek a következő alkalmazási előfeltételek:

- a célmező üres:
$$h_{s,o} = 0; \tag{2.6}$$
- a célmező pontosan egy lóugrásnyira van attól a mezőtől, ahonnan a *betű* betűjelű huszár elugrani készül:
$$\forall i \, \forall j \, (h_{i,j} = \mathit{betű} \supset (|i - s| = 1 \wedge |j - o| = 2) \vee (|i - s| = 2 \wedge |j - o| = 1)). \tag{2.7}$$

Értelemszerűen e két feltétel konjunkciója lesz az operátor alkalmazásának az előfeltétele:

$$\mathit{előfeltétel}(\mathrm{Ugrik}(h, \mathit{betű}, s, o)) \equiv (2.6) \wedge (2.7).$$

Az $\mathrm{Ugrik}(h, \mathit{betű}, s, o)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\mathrm{Ugrik}(h, \mathit{betű}, s, o) = \mathrm{Ugrik}\left( \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix}, \mathit{betű}, s, o \right) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} \end{pmatrix},$$

ahol minden $1 \leq i \leq 3$ és $1 \leq j \leq 3$ esetén

$$h'_{i,j} = \begin{cases} \textit{betű}, & \text{ha } i = s \wedge j = o, \\ 0, & \text{ha } h_{i,j} = \textit{betű}, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

Ebben a formalizmusban aposztróffal jelöltük az előálló új állapot komponenseit, hogy a definiáló egyenlőségben meg lehessen különböztetni őket az eredeti állapot komponenseitől. Érdemes megfigyelni még, hogy a definiáló egyenlőség jobb oldalán csak az eredeti állapot komponensei és az alkalmazott operátor paraméterei szerepelnek. Így egyrészt az új állapot egyértelműen meghatározható, másrészt látható, hogy az operátor alkalmazása az eredeti állapotot semmilyen körülmények között nem változtatja meg.

Mindezek után az operátoraink halmazát alkossák az ily módon definiált operátorok:

$$\mathcal{O} = \{\, \text{Ugrik}(h, \textit{betű}, s, o) \,\}$$

Ha utánaszámolunk, az operátoraink halmazát $4 \times 3 \times 3 = 36$ operátor alkotja (négyféle betűt és kilencféle koordinátapárt helyettesítve be paraméterként). Ebből a 36 operátorból az a négy, amely a tábla középső mezőjére navigálhatná a huszárokat, biztosan nem alkalmazható. A maradék 32-ből egy konkrét állapotra maximum 8 alkalmazható egyszerre, hiszen egy-egy huszár legfeljebb két helyre tud aktuális pozíciójáról tovább ugrani.

Az operátorunkkal szemben támasztott harmadik követelményt nem feltétlenül lett volna szükséges előírnunk. Az operátorunk előállíthatna bármilyen értékkilencest, hiszen már definiáltunk egy kényszerfeltételt, amelynek a segítségével ellenőrizhetnénk az előállított értékkilences állapot voltát. Így viszont – előrelátó módon – olyan operátorral rendelkezünk, amelynek az alkalmazása után már nem szükséges a kényszerfeltétel teljesülésének az ellenőrzése.

Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az

$$\langle \mathcal{A}, \textit{kezdő}, \mathcal{C}, \mathcal{O} \rangle$$

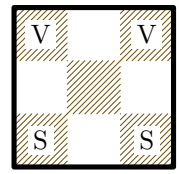
négyest, a probléma egy lehetséges állapottér-reprezentációját.

### 2.2.3. Egy mátrix kétféle huszárral

Az operátoraink számát tovább csökkenthetjük, ha végrehajtjuk a következő apró módosítást: az egyforma színű huszárokat jelöljük azonos betűvel, például a sötéteket S betűvel, a világosakat pedig V-vel.

Mivel két betűvel kevesebbet használunk most, mint korábban, az alaphalmazaink is módosulnak. A sakktábla egyes mezőihez most a $\{\, 0, \mathrm{S}, \mathrm{V} \,\}$ halmazból rendelhetünk értékeket:

$$H_{i,j} = \{\, 0, \mathrm{S}, \mathrm{V} \,\}, \quad 1 \leq i \leq 3 \text{ és } 1 \leq j \leq 3.$$




2.3. ábra. A világos huszárokat V-vel, a sötéteket S-sel jelöljük

Az alaphalmazok Descartes-szorzata így a következőképpen néz ki:

$$H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3} =$$
$$\left\{ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{S} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{V} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{S} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{V} \end{pmatrix}, \right.$$
$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{V} & \mathrm{S} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{V} & \mathrm{V} \end{pmatrix}, \ldots, \begin{pmatrix} \mathrm{S} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{S} \end{pmatrix}, \begin{pmatrix} \mathrm{S} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{V} \end{pmatrix},$$
$$\left. \ldots, \begin{pmatrix} \mathrm{S} & \mathrm{S} & \mathrm{S} \\ \mathrm{S} & \mathrm{S} & \mathrm{S} \\ \mathrm{S} & \mathrm{S} & \mathrm{S} \end{pmatrix}, \ldots, \begin{pmatrix} \mathrm{V} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{S} \end{pmatrix}, \begin{pmatrix} \mathrm{V} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \mathrm{S} & \mathrm{V} \end{pmatrix}, \ldots, \begin{pmatrix} \mathrm{V} & \mathrm{V} & \mathrm{V} \\ \mathrm{V} & \mathrm{V} & \mathrm{V} \\ \mathrm{V} & \mathrm{V} & \mathrm{V} \end{pmatrix} \right\}$$

Ennek a Descartes-szorzatnak – az előző reprezentáció Descartes-szorzatához viszonyítva – már jóval kevesebb, mindössze $3^9 = 19\,683$ darab 3×3-as mátrix alakban felírható elemkilences eleme van. Ezek közül is csak azon

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \in H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3}$$

elemkilencesek lesznek problémánk állapotai, amelyekre teljesülnek a következő feltételek:

- az elemkilencesben pontosan kettő darab S betű található:

$$\left( \sum_{i=1}^{3} \sum_{j=1}^{3} f(i, j, \mathrm{S}) \right) = 2; \tag{2.8}$$

- az elemkilencesben pontosan kettő darab V betű található:

$$\left( \sum_{i=1}^{3} \sum_{j=1}^{3} f(i, j, \mathrm{V}) \right) = 2; \tag{2.9}$$

- a középső mezőre – figyelembe véve a kiinduló helyzetet – szabályos lépéssel nem juthat el egyik huszár sem:

$$h_{2,2} = 0. \tag{2.10}$$

A formulákban használt $f(i, j, betű)$ függvény definíciója megegyezik az előző reprezentációban megadottéval (lásd a 2.1-es formulát). Ezúttal azok az elemkilencesek lesznek a probléma állapotai, melyekre teljesül a megadott három formula konjunkciója:

$$kényszerfeltétel(h) \equiv (2.8) \wedge (2.9) \wedge (2.10).$$

A problémánk állapotterét ismét csak azok az elemkilencesek alkotják, amelyek eleget tesznek a kényszerfeltételeknek. Ez az állapottér definíció szerint részhalmaza az alaphalmazok Descartes-szorzatának:

$$\mathcal{A} = \left\{ h \;\middle|\; h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix} \wedge kényszerfeltétel(h) \right\} \subset$$

$$\subset H_{1,1} \times H_{1,2} \times H_{1,3} \times H_{2,1} \times H_{2,2} \times H_{2,3} \times H_{3,1} \times H_{3,2} \times H_{3,3}.$$

Ebben a reprezentációban a kezdőállapot a 2.3. ábrának megfelelően a következő lesz:

$$kezdő = \begin{pmatrix} \mathrm{V} & 0 & \mathrm{V} \\ 0 & 0 & 0 \\ \mathrm{S} & 0 & \mathrm{S} \end{pmatrix} \in \mathcal{A}.$$

A célállapotok halmaza – az egyforma színű huszárok azonos betűjelei miatt – most egyelemű lesz:

$$\mathcal{C} = \left\{ \begin{pmatrix} \mathrm{S} & 0 & \mathrm{S} \\ 0 & 0 & 0 \\ \mathrm{V} & 0 & \mathrm{V} \end{pmatrix} \right\} \subset \mathcal{A}$$

Természetesen az, hogy az azonos színű huszárokat nem tudjuk egymástól megkülönböztetni, új operátorok bevezetését teszi szükségessé. Most úgy próbáljuk megtervezni az operátorainkat, hogy

- egyértelműen tudják azonosítani azt a mezőt, ahonnan az ugrás történik,
- egyértelműen tudják azonosítani azt a mezőt, ahová a huszár ugrani akar, és
- az ugrás végrehajtása után az előálló új helyzet is eleme legyen az állapottérnek.

Legyen ezúttal

$$\mathrm{Ugrik}(h, x, y, s, o) \colon \mathrm{dom}(\mathrm{Ugrik}) \to \mathcal{A},$$

ahol

$$\mathrm{dom}(\mathrm{Ugrik}) \subset \mathcal{A} \times \{\,1, 2, 3\,\} \times \{\,1, 2, 3\,\} \times \{\,1, 2, 3\,\} \times \{\,1, 2, 3\,\}.$$

Látható, hogy a korábbi *betű* paraméter helyére ezúttal egy $(x, y)$ koordinátapár került, ezzel fogjuk azonosítani azt a mezőt, ahonnan az ugrás történik.

Egy ilyen $\mathrm{Ugrik}(h, x, y, s, o)$ operátor akkor alkalmazható, ha teljesülnek a következő alkalmazási előfeltételek:

- az a mező, ahonnan az ugrás történik, nem üres:

$$h_{x,y} \neq 0; \tag{2.11}$$

- az a mező, ahová az ugrás történik, üres:

$$h_{s,o} = 0; \tag{2.12}$$

- a célmező pontosan egy lóugrásnyira van attól a kiindulási mezőtől:

$$(|x-s| = 1 \wedge |y-o| = 2) \vee (|x-s| = 2 \wedge |y-o| = 1). \tag{2.13}$$

E három feltétel konjunkciója lesz az operátor alkalmazásának az előfeltétele:

$$\textit{előfeltétel}(\mathrm{Ugrik}(h,x,y,s,o)) \equiv (2.11) \wedge (2.12) \wedge (2.13).$$

Az $\mathrm{Ugrik}(h,x,y,s,o)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\mathrm{Ugrik}(h,x,y,s,o) = \mathrm{Ugrik}\left(\begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{pmatrix}, x, y, s, o\right) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} \end{pmatrix},$$

ahol minden $1 \leq i \leq 3$ és $1 \leq j \leq 3$ esetén

$$h'_{i,j} = \begin{cases} h_{x,y}, & \text{ha } i = s \wedge j = o, \\ 0, & \text{ha } i = x \wedge j = y, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

Operátoraink halmazát alkossák tehát a fentebb definiált operátorok:

$$\mathcal{O} = \{\, \mathrm{Ugrik}(h,x,y,s,o) \,\}$$

Ennek az operátorhalmaznak az elemszámát úgy kaphatjuk meg, ha képezzük a lehetséges paraméterek összes kombinációját, melynek száma: $3 \times 3 \times 3 \times 3 = 81$. Ebből a 81 operátorból ténylegesen csak 16-ot fogunk tudni alkalmazni a huszárok ugrási szabálya miatt. Ráadásul, mivel a táblán csak négy huszár található, egy konkrét állapotra ebből a 16 operátorból is csak maximum 8 operátor alkalmazható egyszerre.

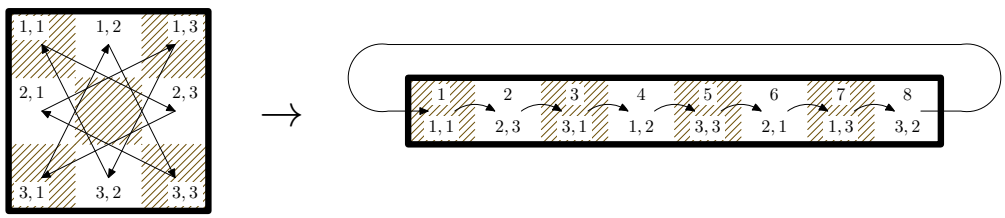
Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az

$$\langle \mathcal{A}, \textit{kezdő}, \mathcal{C}, \mathcal{O} \rangle$$

négyest, a probléma egy újabb állapottér-reprezentációját.

### 2.2.4. Egy vektor négy különböző huszárral

Többször is utaltunk már rá, hogy a sakktábla középső mezőjére a kiinduló állásban táblán álló huszárok szabályos sakklépésekkel nem juthatnak el. Korábbi reprezentációink másik fontos megfigyelése, hogy egy mezőről a rajta álló figura maximum két helyre tud továbblépni. Ezek a tények és [8] tanulmányozása adhatja az ötletet, hogy a tábla azon mezői között, amelyeket a figuráink egyáltalán elérhetnek, állítsunk fel egy sorrendet, sorszámozzuk be őket, és a bábuk mozgását ebben az új inerciarendszerben próbáljuk meg leírni.




2.4. ábra. A mezők a bejárás sorrendjében egy vektorra képezhetők le

A sorrend ezúttal, amint az a 2.4. ábrán is látható, az óramutató járásával megegyező irányú bejárási sorrend lesz. Érinti mind a nyolc mezőt, majd a nyolcadik mező érintése után visszatér az első mezőhöz. Ez tehát azt jelenti, hogy minden huszár, amely szabályos lólépésekkel végighalad ebben az irányban, előbb-utóbb visszatér(het) kiindulási pontjára. Ha előírnánk huszárjainknak, hogy mindannyian csak a bejárási sorrendnek megfelelően közlekedhetnek – mindig csak az óramutató járásával megegyező irányban –, akkor észrevehetjük, hogy ezt csak úgy tudják megtenni, ha a vektorban ciklikusan végighaladva megtartják egymáshoz viszonyított indulási sorrendjüket. Azt is látni kell, hogy mivel ily módon nem előzhetik meg egymást, annak sincs értelme, hogy saját célmezőjüknél tovább haladjanak.

A reprezentáció első lépése, miszerint újra négy különböző betűvel jelöljük a négy huszárt, látszólag visszalépés az előző reprezentációhoz képest. Jelölje tehát

$$H_i = \{\, 0, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D} \,\}, \quad 1 \le i \le 8$$

azokat a lehetőségeket, amelyek azt írják le, hogy milyen bábu tartózkodhat a sakktábla $i$-vel jelölt mezőjén. A 0 szimbólum ezúttal is azt jelöli, hogy az adott mezőn éppen nem tartózkodik huszár.

Képezzük ezen alaphalmazok Descartes-szorzatát:

$$\begin{aligned}
&H_1 \times H_2 \times H_3 \times H_4 \times H_5 \times H_6 \times H_7 \times H_8 = \\
&\quad = \{\, (0,0,0,0,0,0,0,0), (0,0,0,0,0,0,0,\mathrm{A}), (0,0,0,0,0,0,0,\mathrm{B}), \\
&\qquad (0,0,0,0,0,0,0,\mathrm{C}), (0,0,0,0,0,0,0,\mathrm{D}), (0,0,0,0,0,0,\mathrm{A},\mathrm{A}), \\
&\qquad (0,0,0,0,0,0,\mathrm{A},\mathrm{B}), (0,0,0,0,0,0,\mathrm{A},\mathrm{C}), (0,0,0,0,0,0,\mathrm{A},\mathrm{D}), \\
&\qquad (0,0,0,0,0,0,\mathrm{B},\mathrm{A}), (0,0,0,0,0,0,\mathrm{B},\mathrm{B}), (0,0,0,0,0,0,\mathrm{B},\mathrm{C}), \ldots, \\
&\qquad (\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A}), \ldots, (\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{A}), \\
&\qquad (\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{B}), (\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{C}), (\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D},\mathrm{D})\}
\end{aligned}$$

A $H_1 \times H_2 \times H_3 \times H_4 \times H_5 \times H_6 \times H_7 \times H_8$ Descartes-szorzatnak összesen $5^8 = 390\,625$ eleme van. Ezek közül csak azon

$$h = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8) \in H_1 \times H_2 \times H_3 \times H_4 \times H_5 \times H_6 \times H_7 \times H_8$$

elemnyolcasok lesznek a probléma állapotai, melyekre teljesülnek a következő kényszerfeltételek:

- az elemnyolcasban pontosan egy darab A betű szerepel:

$$\left|\{\, i \mid 1 \leq i \wedge i \leq 8 \wedge h_i = \mathrm{A} \,\}\right| = 1; \tag{2.14}$$

- az elemnyolcasban pontosan egy darab B betű szerepel:

$$\left|\{\, i \mid 1 \leq i \wedge i \leq 8 \wedge h_i = \mathrm{B} \,\}\right| = 1; \tag{2.15}$$

- az elemnyolcasban pontosan egy darab C betű szerepel:

$$\left|\{\, i \mid 1 \leq i \wedge i \leq 8 \wedge h_i = \mathrm{C} \,\}\right| = 1; \tag{2.16}$$

- az elemnyolcasban pontosan egy darab D betű szerepel:

$$\left|\{\, i \mid 1 \leq i \wedge i \leq 8 \wedge h_i = \mathrm{D} \,\}\right| = 1; \tag{2.17}$$

Láthatjuk, hogy a középső mezőre most nem kell kényszerfeltételt kimondanunk, hiszen a középső mező nem szerepel a vektorban. A huszárok sorrendjére vonatkozóan viszont kellene, hiszen azt mondtuk, hogy nem „előzhetik meg” egymást, miközben az óramutató járásával megegyező irányban ugrálnak, de mégsem tesszük. Ezzel ugyan sok olyan elemnyolcast megtartunk az állapotaink között, amelyeket lehet, hogy sehogyan sem tudnánk a későbbiek során definiálandó operátorainkkal elérni, de így legalább nem is korlátozzuk magunkat sem a kezdőállapot, sem az operátorok hatásainak a definiálásakor.

Kényszerfeltételünk tehát az előző négy formula konjunkciója lesz:

$$\mathit{kényszerfeltétel}(h) \equiv (2.14) \wedge (2.15) \wedge (2.16) \wedge (2.17).$$

A problémánk állapotterét ily módon azok az elemnyolcasok alkotják, amelyekre teljesül az előbb felírt kényszerfeltétel:

$$\mathcal{A} = \{h \mid h = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8) \wedge \mathit{kényszerfeltétel}(h)\} \subset \\ \subset H_1 \times H_2 \times H_3 \times H_4 \times H_5 \times H_6 \times H_7 \times H_8.$$

A kezdőállapotot a 2.2. és 2.4. ábrák alapján határozzuk meg:

$$\mathit{kezdő} = (\mathrm{A}, 0, \mathrm{C}, 0, \mathrm{D}, 0, \mathrm{B}, 0) \in \mathcal{A}.$$

A célállapotok halmaza egyetlen elemnyolcast fog tartalmazni:

$$\mathcal{C} = \{(\mathrm{D}, 0, \mathrm{B}, 0, \mathrm{A}, 0, \mathrm{C}, 0)\} \subset \mathcal{A}.$$

Operátorainktól ezúttal csak egyetlen dolgot várunk el, nevezetesen azt, hogy egyértelműen tudják azonosítani azt a mezőt, ahonnan az ugrás történik. Az óramutató járása ugyanis egyértelműen kijelöli az irányt (az egyébként fennálló két lehetőség közül), így azzal nem kell külön foglalkoznunk. Legyen tehát

$$\mathrm{Ugrik}(h,c)\colon\ \mathrm{dom}(\mathrm{Ugrik}) \to \mathcal{A},$$

ahol

$$\mathrm{dom}(\mathrm{Ugrik}) \subset \mathcal{A} \times \{\,1,2,3,4,5,6,7,8\,\}.$$

Az operátorok második paramétere szolgál annak a mezőnek az azonosítására, ahonnan az ugrás történik, a $c$ érték a mezőnek a vektorban elfoglalt pozícióját jelenti majd.

Egy ilyen $\mathrm{Ugrik}(h,c)$ operátor akkor alkalmazható, ha teljesülnek a következő alkalmazási előfeltételek:

- az a mező, ahonnan az ugrás történik, nem üres:

$$h_c \neq 0; \tag{2.18}$$

- az a mező, ahová az ugrás történik (ez a vektorban a $c$ indexű mezőt követő mező, ahogyan az a 2.4. ábrán is látható), üres:

$$h_{(c \bmod 8)+1} = 0; \tag{2.19}$$

- az 5-ös mezőről csak akkor ugorhatunk, ha ott nem az A betűjelű huszár áll:

$$c = 5 \supset h_c \neq \mathrm{A}; \tag{2.20}$$

- a 3-as mezőről csak akkor ugorhatunk, ha ott nem a B betűjelű huszár áll:

$$c = 3 \supset h_c \neq \mathrm{B}; \tag{2.21}$$

- a 7-es mezőről csak akkor ugorhatunk, ha ott nem a C betűjelű huszár áll:

$$c = 7 \supset h_c \neq \mathrm{C}; \tag{2.22}$$

- az 1-es mezőről csak akkor ugorhatunk, ha ott nem a D betűjelű huszár áll:

$$c = 1 \supset h_c \neq \mathrm{D}; \tag{2.23}$$

E hat feltétel konjunkciója lesz az operátor alkalmazásának az előfeltétele:

$$\textit{előfeltétel}(\mathrm{Ugrik}(h,c)) \equiv (2.18) \wedge (2.19) \wedge (2.20) \wedge (2.21) \wedge (2.22) \wedge (2.23).$$

Az $\mathrm{Ugrik}(h,c)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\mathrm{Ugrik}(h,c) = \mathrm{Ugrik}\big((h_1,h_2,h_3,h_4,h_5,h_6,h_7,h_8),c\big) = (h'_1,h'_2,h'_3,h'_4,h'_5,h'_6,h'_7,h'_8),$$

ahol minden $1 \leq i \leq 8$ esetén

$$h'_i = \begin{cases} h_c, & \text{ha } i = (c \bmod 8) + 1, \\ 0, & \text{ha } i = c, \\ h_i & \text{egyébként.} \end{cases}$$

Operátoraink halmazát alkossák tehát a fentebb definiált Ugrik$(h, c)$ operátorok:

$$\mathcal{O} = \{ \text{Ugrik}(h, c) \}$$

Ebben az $\mathcal{O}$ halmazban már csak 8 operátor található, annyi, ahány helyről egyáltalán tovább lehet lépni. A 8 operátorból egy adott állapotra maximum 4 alkalmazható, ennyi is csak akkor, ha mind a négy huszár tovább tud lépni az óramutató járásával megegyező irányban.

Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az

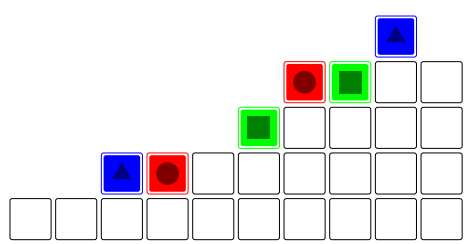
$$\langle \mathcal{A}, \textit{kezdő}, \mathcal{C}, \mathcal{O} \rangle$$

négyest, a probléma egy harmadik állapottér-reprezentációját.

## 2.3. *Szabadesés*

### 2.3.1. A probléma

Adott néhány színes doboz a 2.5. ábrán látható elrendezésben. A dobozokat jobbra és balra lehet tologatni egymáson és a tartóelemeken. Ha egy dobozt olyan helyre tolunk, ahol nincsen alatta sem tartóelem, sem másik doboz, akkor lehullik... egészen addig, míg a föld vagy egy másik színes doboz meg nem állítja az esését. Ha két vagy több azonos színű doboz kerül egymás mellé, akkor ezek az egymás mellett lévő egyforma színű dobozok eltűnnek.



2.5. ábra. Szabadesés előtt

Feladatunk az összes doboz eltüntetése. A feladat eredeti szövege, sőt maga a játszható feladvány is megtalálható a [6] URL-címen (ez ott az első feladvány).

### 2.3.2. Egy lehetséges állapottér-reprezentáció

A problémánk egy-egy állapotát egy-egy 5×10-es mátrixszal írhatjuk le. A mátrix $i$-edik sorában lévő $j$-edik elem azt fogja jelezni, hogy milyen elem található a játéktér $i$-edik sorának $j$-edik oszlopában. A lehetőségek halmazait jelöljük $H_{i,j}$-kkel:

$$H_{i,j} = \{\, -1, 0, 1, 2, 3 \,\}, \qquad 1 \le i \le 5 \text{ és } 1 \le j \le 10.$$

Az egyes számértékek jelentése a következő:

| | |
|---|---|
| $-1$, | ha a játéktér adott mezője tartóelem |
| $0$, | ha a játéktér adott mezője levegő |
| $1$, | ha a játéktér adott mezője piros dobozt tartalmaz |
| $2$, | ha a játéktér adott mezője kék dobozt tartalmaz |
| $3$, | ha a játéktér adott mezője zöld dobozt tartalmaz |

A fenti $H_{i,j}$ halmazok Descartes-szorzatát képezve a csillagászati elemszámú,

$$5^{50} = 88\,817\,841\,970\,012\,523\,233\,890\,533\,447\,265\,625$$

darab elemötvenest tartalmazó

$$H_{1,1} \times H_{1,2} \times H_{1,3} \times \cdots \times H_{5,9} \times H_{5,10}$$

halmazt kapjuk. Az már első pillantásra látszik, hogy ennek a halmaznak nem minden eleme lesz problémánk állapota, de hogy melyek lesznek azok, azt igencsak nehéz megfogalmazni. Kényszerfeltételt is csak a következőt tudjuk mondani kapásból: az a

$$\begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & h_{1,7} & h_{1,8} & h_{1,9} & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & h_{2,7} & h_{2,8} & h_{2,9} & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & h_{3,7} & h_{3,8} & h_{3,9} & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & h_{4,7} & h_{4,8} & h_{4,9} & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & h_{5,7} & h_{5,8} & h_{5,9} & h_{5,10} \end{pmatrix} \in$$
$$\in H_{1,1} \times H_{1,2} \times H_{1,3} \times \cdots \times H_{5,9} \times H_{5,10}$$

elemötvenes lesz a probléma állapota, amelyben a tartóelemek ott és csak ott állnak, ahol a kiinduló helyzetben is találhatók. A többi elemre, mivel sok helyre el lehet őket tologatni, egyszerűen nem merünk kényszerfeltételeket előírni. Állapotterünket, $\mathcal{A}$-t tehát a fenti kényszerfeltételnek eleget tevő elemötvenesek alkotják.

Kezdőállapotunk az az állapot, amelyik a 2.5. ábrán is látható:

$$\textit{kezdő} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 3 & -1 & -1 & -1 & -1 \\ 0 & 0 & 2 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix} \in \mathcal{A}.$$

A célállapotok halmazának azok az elemötvenesek lesznek az elemei, amelyek nem tartalmaznak egyetlen színes dobozt sem:

$$\mathcal{C} = \left\{ h \;\middle|\; h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & \dots & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & \dots & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & \dots & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & \dots & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & \dots & h_{5,10} \end{pmatrix} \wedge \forall i \forall j \, (h_{i,j} \leq 0) \right\} \subset \mathcal{A}.$$

Érdekességképpen jegyezzük meg, hogy bár ilyen elemötvenes most is csak egyetlenegy van, a célállapotok halmazának az elemeit most nem explicit módon, hanem célfeltétellel definiáltuk.

Ha a kényszerfeltételekkel korábban gondban voltunk, akkor az operátorok megválasztása sem bíztat minket sok jóval. Olyan operátort kellene ugyanis találnunk, amely

- képes jobbra vagy balra mozgatni egy dobozt,
- engedi lehullani a dobozt, ha nincsen alatta olyan elem, ami megtarthatná, továbbá
- észreveszi és eltünteti az egymás mellett lévő, azonos színű dobozokat.

Bizony, ez egyszerre sok tennivaló egy operátornak. De pontosan ez adhatja az ötletet: mi lenne, ha ezt a három (négy) funkciót szétválasztanánk, és mindegyiket egy külön operátorral valósítanánk meg?

A megoldás kereséséhez a következő operátorhalmazt definiáljuk:

$$\mathcal{O} = \{ \, \mathrm{Balra}(h, x, y), \mathrm{Jobbra}(h, x, y), \mathrm{Hullik}(h), \mathrm{Eltüntet}(h) \, \},$$

valamint a további definíciók egyszerűbb felírása érdekében legyen $S = \{ \, 1, 2, 3, 4, 5 \, \}$ és $O = \{ \, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \, \}$.

Az operátorok értelmezési tartománya és értékkészlete a következőképpen alakul:

$$\begin{aligned} \mathrm{Balra}(h, x, y)\colon &\ \mathrm{dom}(\mathrm{Balra}) \to \mathcal{A}, && \text{ahol } \mathrm{dom}(\mathrm{Balra}) \subset \mathcal{A} \times S \times O, \\ \mathrm{Jobbra}(h, x, y)\colon &\ \mathrm{dom}(\mathrm{Jobbra}) \to \mathcal{A}, && \text{ahol } \mathrm{dom}(\mathrm{Jobbra}) \subset \mathcal{A} \times S \times O, \\ \mathrm{Hullik}(h)\colon &\ \mathrm{dom}(\mathrm{Hullik}) \to \mathcal{A}, && \text{ahol } \mathrm{dom}(\mathrm{Hullik}) \subset \mathcal{A}, \\ \mathrm{Eltüntet}(h)\colon &\ \mathrm{dom}(\mathrm{Eltüntet}) \to \mathcal{A}, && \text{ahol } \mathrm{dom}(\mathrm{Eltüntet}) \subset \mathcal{A}, \end{aligned}$$

A $\mathrm{Balra}(h, x, y)$ operátor akkor alkalmazható egy

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & h_{1,7} & h_{1,8} & h_{1,9} & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & h_{2,7} & h_{2,8} & h_{2,9} & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & h_{3,7} & h_{3,8} & h_{3,9} & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & h_{4,7} & h_{4,8} & h_{4,9} & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & h_{5,7} & h_{5,8} & h_{5,9} & h_{5,10} \end{pmatrix} \in \mathcal{A}$$

állapotra, ha teljesülnek a következő alkalmazási előfeltételek:

- egyik doboz sem hullik éppen lefelé, azaz mindegyik doboz alatt vagy tartóelem, vagy egy másik doboz van:

$$\forall i \forall j \, (i < 5 \wedge h_{i,j} > 0 \supset h_{i+1,j} \neq 0), \tag{2.24}$$

- egyik doboz mellett sem áll vele azonos színű doboz:

$$\forall i \forall j \, (i < 5 \wedge j < 10 \wedge h_{i,j} > 0 \supset h_{i+1,j} \neq h_{i,j} \wedge h_{i,j+1} \neq h_{i,j}), \tag{2.25}$$

- az $x$-edik sor $y$-adik oszlopában egy színes doboz található:

$$h_{x,y} > 0, \tag{2.26}$$

- nem a játéktér bal szélén lévő dobozt akarjuk balra tolni:

$$y > 1, \tag{2.27}$$

- a mozgatandó dobozunktól balra nincs sem tartóelem, sem pedig másik doboz:

$$h_{x,y-1} = 0. \tag{2.28}$$

Ezeknek a feltételeknek a konjunkciója lesz az operátor alkalmazási előfeltétele:

$$\textit{előfeltétel}(\text{Balra}(h,x,y)) \equiv (2.24) \wedge (2.25) \wedge (2.26) \wedge (2.27) \wedge (2.28).$$

A Balra$(h,x,y)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\text{Balra}(h,x,y) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} & h'_{1,5} & h'_{1,6} & h'_{1,7} & h'_{1,8} & h'_{1,9} & h'_{1,10} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} & h'_{2,4} & h'_{2,5} & h'_{2,6} & h'_{2,7} & h'_{2,8} & h'_{2,9} & h'_{2,10} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} & h'_{3,4} & h'_{3,5} & h'_{3,6} & h'_{3,7} & h'_{3,8} & h'_{3,9} & h'_{3,10} \\ h'_{4,1} & h'_{4,2} & h'_{4,3} & h'_{4,4} & h'_{4,5} & h'_{4,6} & h'_{4,7} & h'_{4,8} & h'_{4,9} & h'_{4,10} \\ h'_{5,1} & h'_{5,2} & h'_{5,3} & h'_{5,4} & h'_{5,5} & h'_{5,6} & h'_{5,7} & h'_{5,8} & h'_{5,9} & h'_{5,10} \end{pmatrix},$$

ahol minden $1 \leq i \leq 5$ és $1 \leq j \leq 10$ esetén

$$h'_{i,j} = \begin{cases} 0, & \text{ha } i = x \wedge j = y, \\ h_{x,y}, & \text{ha } i = x \wedge j = y-1, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

A Jobbra$(h,x,y)$ operátor akkor alkalmazható egy

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & h_{1,7} & h_{1,8} & h_{1,9} & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & h_{2,7} & h_{2,8} & h_{2,9} & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & h_{3,7} & h_{3,8} & h_{3,9} & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & h_{4,7} & h_{4,8} & h_{4,9} & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & h_{5,7} & h_{5,8} & h_{5,9} & h_{5,10} \end{pmatrix} \in \mathcal{A}$$

állapotra, ha teljesülnek a következő alkalmazási előfeltételek:

- egyik doboz sem hullik éppen lefelé, azaz mindegyik doboz alatt vagy tartóelem, vagy egy másik doboz van:

$$\forall i \, \forall j \, (i < 5 \wedge h_{i,j} > 0 \supset h_{i+1,j} \neq 0), \tag{2.29}$$

- egyik doboz mellett sem áll vele azonos színű doboz:

$$\forall i \, \forall j \, (i < 5 \wedge j < 10 \wedge h_{i,j} > 0 \supset h_{i+1,j} \neq h_{i,j} \wedge h_{i,j+1} \neq h_{i,j}), \tag{2.30}$$

- az $x$-edik sor $y$-adik oszlopában egy színes doboz található:

$$h_{x,y} > 0, \tag{2.31}$$

- nem a játéktér jobb szélén lévő dobozt akarjuk jobbra tolni:

$$y < 10, \tag{2.32}$$

- a mozgatandó dobozunktól jobbra nincs sem tartóelem, sem pedig másik doboz:

$$h_{x,y+1} = 0. \tag{2.33}$$

Ezeknek a feltételeknek a konjunkciója lesz az operátor alkalmazási előfeltétele:

$$\textit{előfeltétel}(\text{Jobbra}(h, x, y)) \equiv (2.29) \wedge (2.30) \wedge (2.31) \wedge (2.32) \wedge (2.33).$$

A Jobbra$(h, x, y)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\text{Jobbra}(h, x, y) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} & h'_{1,5} & h'_{1,6} & h'_{1,7} & h'_{1,8} & h'_{1,9} & h'_{1,10} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} & h'_{2,4} & h'_{2,5} & h'_{2,6} & h'_{2,7} & h'_{2,8} & h'_{2,9} & h'_{2,10} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} & h'_{3,4} & h'_{3,5} & h'_{3,6} & h'_{3,7} & h'_{3,8} & h'_{3,9} & h'_{3,10} \\ h'_{4,1} & h'_{4,2} & h'_{4,3} & h'_{4,4} & h'_{4,5} & h'_{4,6} & h'_{4,7} & h'_{4,8} & h'_{4,9} & h'_{4,10} \\ h'_{5,1} & h'_{5,2} & h'_{5,3} & h'_{5,4} & h'_{5,5} & h'_{5,6} & h'_{5,7} & h'_{5,8} & h'_{5,9} & h'_{5,10} \end{pmatrix},$$

ahol minden $1 \leq i \leq 5$ és $1 \leq j \leq 10$ esetén

$$h'_{i,j} = \begin{cases} 0, & \text{ha } i = x \wedge j = y, \\ h_{x,y}, & \text{ha } i = x \wedge j = y + 1, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

A Hullik$(h)$ operátor akkor alkalmazható egy

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & h_{1,7} & h_{1,8} & h_{1,9} & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & h_{2,7} & h_{2,8} & h_{2,9} & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & h_{3,7} & h_{3,8} & h_{3,9} & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & h_{4,7} & h_{4,8} & h_{4,9} & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & h_{5,7} & h_{5,8} & h_{5,9} & h_{5,10} \end{pmatrix} \in \mathcal{A}$$

állapotra, ha teljesül a következő alkalmazási előfeltétel:

- van olyan doboz, amelyik alatt nincsen sem tartóelem, sem másik doboz:

$$\exists i\, \exists j\, (i < 5 \wedge h_{i,j} > 0 \wedge h_{i+1,j} = 0). \tag{2.34}$$

Itt nincs is szükség további feltételekre, ezért

$$\textit{előfeltétel}(\text{Hullik}(h)) \equiv (2.34).$$

A Hullik($h$) operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\text{Hullik}(h) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} & h'_{1,5} & h'_{1,6} & h'_{1,7} & h'_{1,8} & h'_{1,9} & h'_{1,10} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} & h'_{2,4} & h'_{2,5} & h'_{2,6} & h'_{2,7} & h'_{2,8} & h'_{2,9} & h'_{2,10} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} & h'_{3,4} & h'_{3,5} & h'_{3,6} & h'_{3,7} & h'_{3,8} & h'_{3,9} & h'_{3,10} \\ h'_{4,1} & h'_{4,2} & h'_{4,3} & h'_{4,4} & h'_{4,5} & h'_{4,6} & h'_{4,7} & h'_{4,8} & h'_{4,9} & h'_{4,10} \\ h'_{5,1} & h'_{5,2} & h'_{5,3} & h'_{5,4} & h'_{5,5} & h'_{5,6} & h'_{5,7} & h'_{5,8} & h'_{5,9} & h'_{5,10} \end{pmatrix},$$

ahol minden $1 \leq i \leq 5$ és $1 \leq j \leq 10$ esetén

$$h'_{i,j} = \begin{cases} 0, & \text{ha } i < 5 \wedge h_{i,j} > 0 \wedge h_{i+1,j} = 0, \\ h_{i-1,j}, & \text{ha } i > 1 \wedge h_{i-1,j} > 0 \wedge h_{i,j} = 0, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

Az Eltüntet($h$) operátor akkor alkalmazható egy

$$h = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & h_{1,5} & h_{1,6} & h_{1,7} & h_{1,8} & h_{1,9} & h_{1,10} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & h_{2,5} & h_{2,6} & h_{2,7} & h_{2,8} & h_{2,9} & h_{2,10} \\ h_{3,1} & h_{3,2} & h_{3,3} & h_{3,4} & h_{3,5} & h_{3,6} & h_{3,7} & h_{3,8} & h_{3,9} & h_{3,10} \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & h_{4,5} & h_{4,6} & h_{4,7} & h_{4,8} & h_{4,9} & h_{4,10} \\ h_{5,1} & h_{5,2} & h_{5,3} & h_{5,4} & h_{5,5} & h_{5,6} & h_{5,7} & h_{5,8} & h_{5,9} & h_{5,10} \end{pmatrix} \in \mathcal{A}$$

állapotra, ha teljesülnek a következő alkalmazási előfeltételek:

- egyik doboz sem hullik éppen lefelé, azaz mindegyik doboz alatt vagy tartóelem, vagy egy másik doboz van:

$$\forall i\, \forall j\, (i < 5 \wedge h_{i,j} > 0 \supset h_{i+1,j} \neq 0), \tag{2.35}$$

- van olyan színes doboz, amelynek közvetlen szomszédságában egy vele azonos színű doboz található:

$$\exists i\, \exists j\, (i < 5 \wedge j < 10 \wedge h_{i,j} > 0 \wedge (h_{i+1,j} = h_{i,j} \vee h_{i,j+1} = h_{i,j})). \tag{2.36}$$

E két feltétel konjunkciója lesz az operátor alkalmazási előfeltétele:

$$\textit{előfeltétel}(\text{Eltüntet}(h)) \equiv (2.35) \wedge (2.36).$$

Az Eltüntet($h$) operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$\text{Eltüntet}(h) = \begin{pmatrix} h'_{1,1} & h'_{1,2} & h'_{1,3} & h'_{1,4} & h'_{1,5} & h'_{1,6} & h'_{1,7} & h'_{1,8} & h'_{1,9} & h'_{1,10} \\ h'_{2,1} & h'_{2,2} & h'_{2,3} & h'_{2,4} & h'_{2,5} & h'_{2,6} & h'_{2,7} & h'_{2,8} & h'_{2,9} & h'_{2,10} \\ h'_{3,1} & h'_{3,2} & h'_{3,3} & h'_{3,4} & h'_{3,5} & h'_{3,6} & h'_{3,7} & h'_{3,8} & h'_{3,9} & h'_{3,10} \\ h'_{4,1} & h'_{4,2} & h'_{4,3} & h'_{4,4} & h'_{4,5} & h'_{4,6} & h'_{4,7} & h'_{4,8} & h'_{4,9} & h'_{4,10} \\ h'_{5,1} & h'_{5,2} & h'_{5,3} & h'_{5,4} & h'_{5,5} & h'_{5,6} & h'_{5,7} & h'_{5,8} & h'_{5,9} & h'_{5,10} \end{pmatrix},$$

ahol minden $1 \leq i \leq 5$ és $1 \leq j \leq 10$ esetén

$$h'_{i,j} = \begin{cases} 0, & \text{ha } i < 5 \wedge h_{i,j} > 0 \wedge h_{i+1,j} = h_{i,j}, \\ 0, & \text{ha } i > 1 \wedge h_{i,j} > 0 \wedge h_{i-1,j} = h_{i,j}, \\ 0, & \text{ha } j < 10 \wedge h_{i,j} > 0 \wedge h_{i,j+1} = h_{i,j}, \\ 0, & \text{ha } j > 1 \wedge h_{i,j} > 0 \wedge h_{i,j-1} = h_{i,j}, \\ h_{i,j} & \text{egyébként.} \end{cases}$$

Meg lehet mutatni, hogy az operátoraink előfeltételei és hatásdefiníciói biztosítják, hogy a műveletek végrehajtása során tartóelemek nem változtatják helyzetüket, nem tűnnek el, és nem is kerülnek az állapotokba újabbak, tehát az operátorok a kényszerfeltételeknek eleget tevő elemötveneseket állítanak elő.

Az állapottérnek, a probléma kezdőállapotának, a célállapotok halmazának, az operátorok alkalmazási előfeltételeinek és hatásának a definiálásával megadtuk az

$$\langle \mathcal{A}, \mathit{kezdő}, \mathcal{C}, \mathcal{O} \rangle$$

négyest, a probléma egy lehetséges állapottér-reprezentációját.

## 2.4. Az állapottér-reprezentáció osztálydiagramjai

Amikor azt a célt tűzzük ki magunk elé, hogy hatékony objektumorientált modellt készítsünk az állapottér-reprezentációhoz, meg kell találnunk azokat a jellegzetességeit, amelyeket kiemelhetünk belőle, és absztrakt módon kezelhetjük azokat a modellezés folyamán.

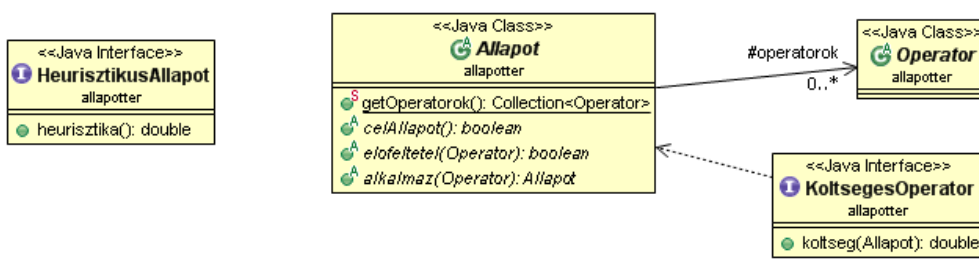
Amint az az előző alfejezetekből sejthetjük, ahány probléma, annyiféle állapottér-reprezentáció létezik. Sőt, a helyzet ennél sokkal rosszabb, hiszen egy-egy problémát alaposan végiggondolva, találhatunk több alkalmasnak tűnő reprezentációt is. A modellünkben tehát olyan jellemzőket kell kiemelnünk, amelyek a problémáktól függetlenek. Nézzük át, melyek is lehetnek ezek:

1. **Az állapottér és az állapotok.** Az állapotok típusára nincs külön megszorítás, azt az egyes állapottér-reprezentációkban definiáljuk. Minden állapotról el lehet viszont dönteni, hogy *alkalmazható-e rá* egy adott operátor, és meg lehet mondani, hogy milyen *új állapot* áll elő, ha alkalmazunk rá egy konkrét operátort. Az állapottérhez emiatt szorosan kötődik azoknak az operátoroknak a halmaza, amelyekkel az állapotok transzformációit elvégezhetjük.

2. **A kezdőállapot.** Minden problémához hozzátartozik a kezdőállapot definíciója, ez tehát egy közös pont. A kezdőállapot az állapottér egy eleme, típusára nincs külön megszorítás.

3. **A célállapotok halmaza.** A kezdőállapottal ellentétben a célállapotok halmazába tartozó állapotokat vagy konkrétan megadjuk az állapottér-reprezentációban (lásd a 2.2. alfejezet reprezentációit), vagy csak feltételt írunk elő rájuk (lásd a 2.3. alfejezet reprezentációját). Az viszont mindkét esetben igaz, hogy egy állapotról el kell tudnunk dönteni, hogy *célállapot-e* vagy sem.

4. **Az operátorok.** Egy állapottér-reprezentációnak önmagában is jellegzetes komponense az operátorainak a halmaza. Ha viszont több reprezentációt is megvizsgálunk, láthatjuk, hogy annál több közös tulajdonságot, miszerint egy halmazról van szó, nem sokat találunk. Az operátorok ugyanis lehetnek paraméteresek vagy paraméter nélküliek; ha paraméteresek, akkor a paramétereik típusa (tartománya) is egymástól teljesen eltérő lehet. Sőt, amint azt a 2.3. alfejezetben láthattuk, egy reprezentáción belül is több különböző (eltérő nevű és paraméterezésű) operátort is definiálhatunk egy feladathoz. Az operátorok mindegyike rendelkezhet alkalmazási költséggel is.




2.6. ábra. Az állapottér-reprezentáció osztályai és interfészei

Ezen megfigyelések alapján készítettük el a 2.6. ábrán látható osztályhierarchiát. Az ábrán két absztrakt osztály és egy interfész látható, mindhárman egy közös csomagban, az `allapotter` csomagban helyezkednek el.

Az `Operator` osztály az egyes problémákhoz tartozó operátorok, míg az `Allapot` osztály a problémákhoz tartozó állapotok absztrakt ősosztálya. Mivel egy állapot jellemzői mindig a konkrét problémától függnek, ezért ez utóbbi osztályban csak egyetlen adattagot definiáltunk, amely nem más, mint a probléma állapotaira alkalmazható összes operátor kollekciója. Természetesen – függetlenül a problémától – minden állapotnak meg kell tudnia mondani magáról, hogy célállapot-e (`celAllapot` metódus), hogy alkalmazható-e rá egy adott operátor (`elofeltetel` metódus), illetve hogy egy operátor alkalmazásával milyen új állapot jön létre belőle (`alkalmaz` metódus).

Lássuk ezek után először az `Operator` osztály Java kódját:

```
1  package allapotter;
```

```
2
3   public abstract class Operator
4   {
5   }
```

Majd következzék az `Allapot` osztályé:

```
1   package allapotter;
2
3   import java.util.Collection;
4
5   public abstract class Allapot
6   {
7     protected static Collection<Operator> operatorok;
8
9     public static Collection<Operator> getOperatorok()
10    {
11      return operatorok;
12    }
13
14    public abstract boolean celAllapot();
15    public abstract boolean elofeltetel( Operator op );
16    public abstract Allapot alkalmaz( Operator op );
17  }
```

A heurisztikus keresők (lásd a 3. fejezetet) olyan állapotokat használnak, amelyek a fentebb felsorolt jellemzőkön kívül még egy heurisztikaértéket is tartalmaznak. Ehhez nyújt segítséget a `HeurisztikusAllapot` interfész azáltal, hogy az ezt megvalósító állapotok rendelkezni fognak egy heurisztikus értéket szolgáltató módszerrel.

```
1   package allapotter;
2
3   public interface HeurisztikusAllapot
4   {
5     public double heurisztika();
6   }
```

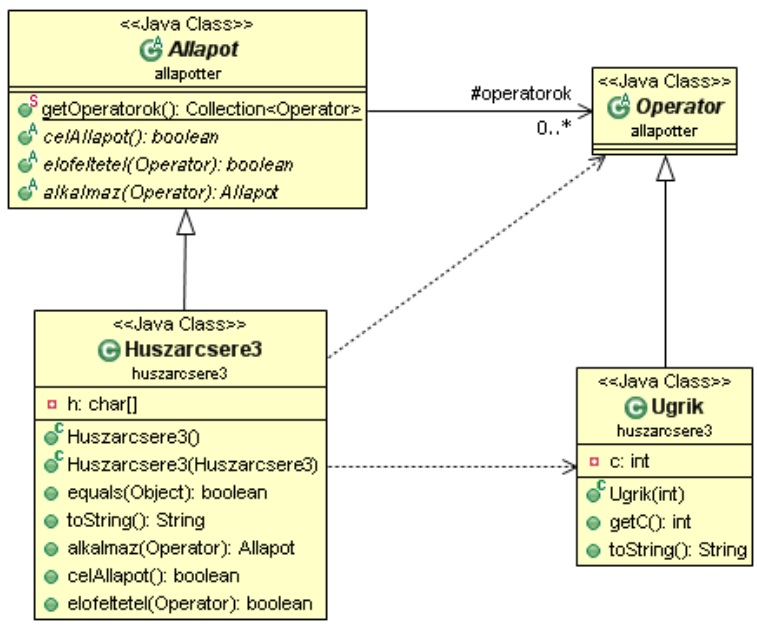
A `KoltsegesOperator` interfészt azok az operátorok fogják implementálni, amelyeket az operátorok költségét figyelembe vevő keresőalgoritmusok használnak. Az interfész kódja a következő:

```
1   package allapotter;
2
3   public interface KoltsegesOperator
4   {
5       public double koltseg( Allapot allapot );
6   }
```

Mindezek után válasszuk ki a 2.2.2. alfejezet harmadik reprezentációját, és nézzük, hogyan valósíthatjuk ezt meg Java programozási nyelven.[1]

## 2.5. A *Huszárcsere* harmadik reprezentációjának osztályai

Először is megjegyeznénk, hogy a problémáinkat megvalósító osztályokat a továbbiakban mindig külön csomagokban helyezzük el. A *Huszárcsere* harmadik reprezentációját megvalósító osztályok a `huszarcsere3` csomagba fognak kerülni.




2.7. ábra. A *Huszárcsere* harmadik reprezentációjának osztályhierarchiája

A programkódok elkészítését az operátorokat megvalósító osztályokkal érdemes kezdeni. Ezek az osztályok ugyanis éppúgy függetlenek egymástól, mint az `Allapot` osztálytól. Mivel a reprezentációban csak egy operátort definiáltunk, csak egy osztályt kell elkészítenünk:

```
1  package huszarcsere3;
2
3  import allapotter.Operator;
```

[1] Ezt az állapottér-reprezentációt egyrészt helytakarékossági okok miatt, másrészt pedig azért választottuk, mert egyszerűségével a legjobban átlátható kódot eredményezi a bemutatott három reprezentáció közül. A másik két reprezentációhoz is készült persze kód, a futási eredményeket ezek alapján hasonlítjuk majd össze a 62. oldalon.

```
 4
 5  public class Ugrik extends Operator
 6  {
 7    private int c;
 8
 9    public Ugrik( int c ) {
10      this.c = c;
11    }
12
13    public int getC() {
14      return c;
15    }
16
17    @Override
18    public String toString() {
19      return "Ugrik[" + c + "]";
20    }
21  }
```

Az osztály egyetlen adattagja az állapottér-reprezentációban definiált $c$ paraméter lesz, amely megtartja reprezentációbeli nevét. Ezen kívül csak egy, az adattagot beállító konstruktor, egy lekérdező metódus és az örökölt `toString` metódus felüldefiniálása szerepel ebben az osztályban.

Annak, hogy ez az osztály ilyen egyszerű lehet, persze ára van. Értelemszerűen az állapotteret megvalósító osztály lesz bonyolultabb, hiszen ott kell majd kezelnünk minden olyan kapcsolatot, amely ezzel, az Ugrik operátort megvalósító osztállyal hozható összefüggésbe. Lássuk tehát a `Huszarcsere3` osztály kódját!

```
 1  package huszarcsere3;
 2
 3  import java.util.HashSet;
 4
 5  import allapotter.Allapot;
 6  import allapotter.Operator;
 7
 8  public class Huszarcsere3 extends Allapot
 9  {
10    static {
11      operatorok = new HashSet<Operator>();
12      for ( int c = 1; c <= 8; ++c )
13        operatorok.add( new Ugrik( c ) );
14    }
15
16    private char[] h;
17
18    public Huszarcsere3() {
19      h = new char[9];
20      for ( int i = 1; i <= 8; ++i )
21        h[i] = '0';
```

```
22        h[1] = 'A';
23        h[7] = 'B';
24        h[3] = 'C';
25        h[5] = 'D';
26      }
27
28      public Huszarcsere3( Huszarcsere3 hcs ) {
29        h = new char[9];
30      }
31
32      @Override
33      public boolean celAllapot() {
34        return h[1] == 'D' && h[3] == 'B' && h[5] == 'A' && h[7] == 'C';
35      }
36
37      @Override
38      public boolean elofeltetel( Operator op ) {
39        if ( op instanceof Ugrik ) {
40          Ugrik u = ( Ugrik )op;
41          int c = u.getC();
42          if ( h[ c ] == '0' )
43            return false;
44          if ( h[ (c%8)+1 ] != '0' )
45            return false;
46          if ( c == 5 && h[c] == 'A' )
47            return false;
48          if ( c == 3 && h[c] == 'B' )
49            return false;
50          if ( c == 7 && h[c] == 'C' )
51            return false;
52          if ( c == 1 && h[c] == 'D' )
53            return false;
54          return true;
55        }
56        return false;
57      }
58
59      @Override
60      public Allapot alkalmaz( Operator op ) {
61        Huszarcsere3 uj = new Huszarcsere3( this );
62        if ( op instanceof Ugrik ) {
63          Ugrik u = ( Ugrik )op;
64          int c = u.getC();
65          for ( int i = 1; i <= 8; ++i )
66            if ( i == (c%8) + 1 )
67              uj.h[i] = h[c];
68            else if ( i == c )
69              uj.h[i] = '0';
70            else
```

```
71              uj.h[i] = h[i];
72        }
73        return uj;
74      }
75
76      @Override
77      public boolean equals(Object obj) {
78        if ( obj == null || !( obj instanceof Huszarcsere3 ) )
79          return false;
80        Huszarcsere3 hcs = ( Huszarcsere3 )obj;
81        for ( int i = 1; i <= 8; ++i )
82          if ( h[i] != hcs.h[i] )
83            return false;
84        return true;
85      }
86
87      @Override
88      public String toString() {
89        StringBuffer sb = new StringBuffer();
90        sb.append( h[1] );
91        for ( int i = 2; i <= 8; ++i )
92          sb.append( ' ' ).append( h[i] );
93        sb.append( System.getProperty( "line.separator" ) );
94        return sb.toString();
95      }
96    }
```

A kód számunkra érdekes részei a következők:

- **10–14. sor**:

  Ebben a pár sorban hozzuk létre az operátorok halmazát, és töltjük fel azzal a 8 darab operátorral, amelyet a paraméter összes lehetséges értékével előállíthatunk; látható, hogy mindezt egy statikus inicializáló blokkban tesszük, lévén az `operatorok` adattag maga is statikus.

- **16. sor**:

  Az állapotunk jellemzőit a karaktereket tartalmazó egydimenziós `h` tömbben tároljuk.

- **18–26. sor**:

  Az osztály egyik konstruktora, amely a kezdőállapot létrehozására szolgál. A `h` mátrix számára a minimálisan szükséges méretnél eggyel több tárhelyet foglalunk le, „túlfoglaljuk”, hogy egy az egyben követhessük a reprezentációban megadott indexelést. Ha ezt nem tennénk meg, akkor – a Java nyelv szabályai szerint – a mátrix mindkét dimenziójában a reprezentációban megadottaknál eggyel kisebb indexeket kellene használnunk.

- **32–35. sor**:

Azt, hogy célállapot-e az aktuális állapot, egy egyszerű logikai kifejezéssel el tudjuk dönteni. A `celallapot` metódus igaz értéket ad vissza, ha az aktuális állapot célállapot, egyébként hamisat.

- **37–57. sor**:

  Az `elofeltetel` metódusban hat kvantormentes formulát kell megvizsgálni (42–53. sorok). Ha bármelyikük is nem teljesül, a paraméterként kapott operátor nem alkalmazható.

- **59–74. sor**:

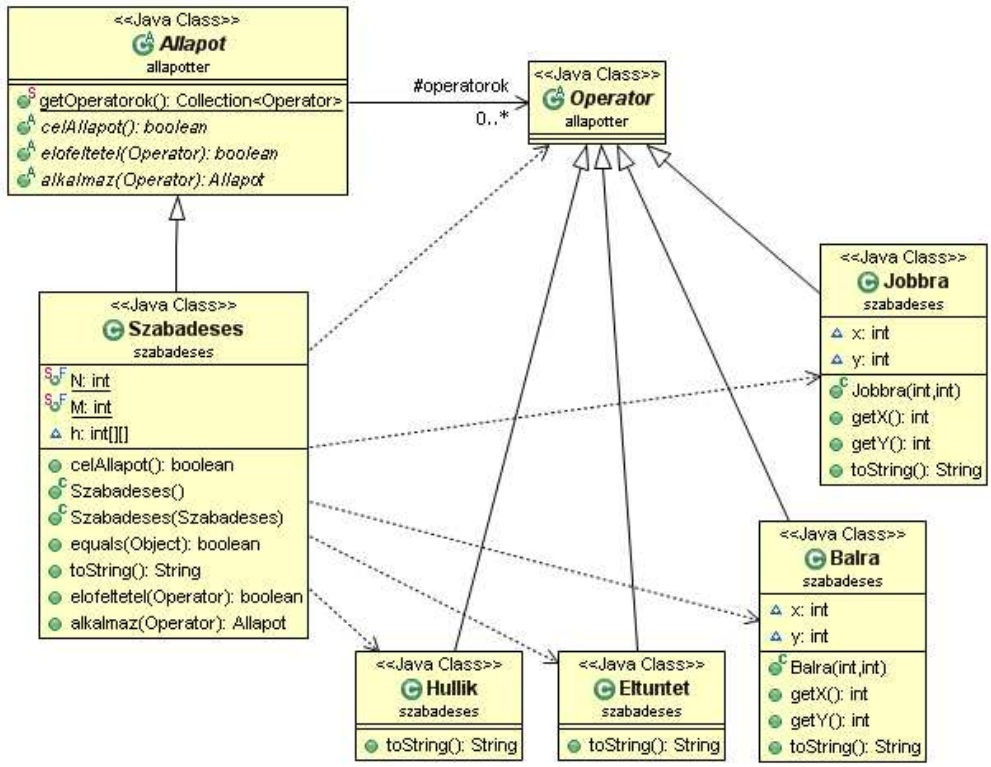
  Az operátorok hatását megvalósító `alkalmaz` metódus. Egyetlen operátorfajtát kell csak kezelnie, így ha beazonosította az Ugrik operátort, akkor a 65–71. sorokban végrehajtja az új mátrix feltöltését a régi mátrix és az operátor paramétere alapján.

## 2.6. A *Szabadesés* állapottér-reprezentációjának osztályai

A *Szabadesés* fantázianevű feladatnak az állapottér-reprezentációja egyetlen szempontból érdekes: az eddigi példáktól eltérően több, különböző paraméterezésű operátort definiál. A kérdés természetesen az, hogy hogyan lehet megkonstruálni az operátoroknak megfelelő osztályokat, valamint hogy hogyan lehet leprogramozni az előfeltételeiket vizsgáló és a hatásukat megvalósító metódusokat.

Amint az a 2.8. ábrán látható, az operátorokat megvalósító osztályok mindegyike az absztrakt `Operator` osztálynak lesz a közvetlen leszármazottja. Az osztályokat az operátorok alapján nevezzük el. Nézzük gyorsan végig először a Balra operátort megvalósító osztály kódját:

```
1  package szabadeses;
2
3  import allapotter.Operator;
4
5  public class Balra extends Operator {
6    int x, y;
7
8    public Balra( int x, int y ) {
9      this.x = x;
10     this.y = y;
11   }
12
13   public int getX() { return x; }
14   public int getY() { return y; }
15
16   @Override
17   public String toString() {
18     return "Balra[" + x + "," + y + "]";
19   }
```




2.8. ábra. A *Szabadesés* állapottér-reprezentációját megvalósító osztályhierarchia

```
20  }
```

Következzen a Jobbra operátort megvalósító osztály kódja:

```
1   package szabadeses;
2
3   import allapotter.Operator;
4
5   public class Jobbra extends Operator {
6     int x, y;
7
8     public Jobbra( int x, int y ) {
9       this.x = x;
10      this.y = y;
11    }
12
13    public int getX() { return x; }
14    public int getY() { return y; }
15
```

```
16    @Override
17    public String toString() {
18      return "Jobbra[" + x + "," + y + "]";
19    }
20  }
```

A Hullik és az Eltüntet operátorokat megvalósító osztályok még az előző kettőnél is egyszerűbbek, hiszen ők nem rendelkeznek adattagokkal:

```
1   package szabadeses;
2
3   import allapotter.Operator;
4
5   public class Hullik extends Operator {
6     @Override
7     public String toString() { return "Hullik[]"; }
8   }
```

```
1   package szabadeses;
2
3   import allapotter.Operator;
4
5   public class Eltuntet extends Operator {
6     @Override
7     public String toString() { return "Eltuntet[]"; }
8   }
```

Ahhoz, hogy lássuk, hogyan különböztetjük meg az egyes operátorokat egymástól, amikor az alkalmazási előfeltételeiket vizsgáljuk, vagy amikor a hatásdefiníciójukat programozzuk, bele kell néznünk a `Szabadeses` osztály kódjába is:

```
1   package szabadeses;
2
3   import java.util.HashSet;
4
5   import allapotter.Allapot;
6   import allapotter.Operator;
7
8   public class Szabadeses extends Allapot {
9     public static final int N = 5;
10    public static final int M = 10;
11
12    static {
13      operatorok = new HashSet<Operator>();
14      for ( int i = 1; i <= N; ++i )
15        for ( int j = 1; j <= M; ++j ) {
16          operatorok.add( new Balra( i, j ) );
17          operatorok.add( new Jobbra( i, j ) );
18        }
19      operatorok.add( new Hullik() );
```

```
20      operatorok.add( new Eltuntet() );
21    }
22
23    private int[][] h;
24
25    public Szabadeses() {
26      h = new int[][] { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
27                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0 },
28                        { 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, -1, -1 },
29                        { 0, 0, 0, 0, 0, 0, 0, 3, -1, -1, -1, -1 },
30                        { 0, 0, 0, 2, 1, -1, -1, -1, -1, -1, -1 },
31                        { 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 } };
32    }
33
34    public Szabadeses( Szabadeses sz ) {
35      h = new int[N + 1][M + 1];
36    }
37
38    @Override
39    public boolean celAllapot() {
40      for ( int i = 1; i <= N; ++i )
41        for ( int j = 1; j <= M; ++j )
42          if ( h[i][j] > 0 )
43            return false;
44      return true;
45    }
46
47    @Override
48    public boolean equals( Object obj ) {
49      if ( obj == null || !( obj instanceof Szabadeses ) )
50        return false;
51      Szabadeses sz = ( Szabadeses ) obj;
52      for ( int i = 1; i <= N; ++i )
53        for ( int j = 1; j <= M; ++j )
54          if ( h[i][j] != sz.h[i][j] )
55            return false;
56      return true;
57    }
58
59    @Override
60    public String toString() {
61      StringBuffer sb = new StringBuffer();
62      sb.append( System.getProperty( "line.separator" ) );
63      for ( int i = 1; i <= N; ++i ) {
64        switch ( h[i][1] ) {
65          case -1:  sb.append( "X" );  break;
66          case 0:   sb.append( " " );  break;
67          default:  sb.append( h[i][1] );  break;
68        }
```

```
69        for ( int j = 2; j <= M; ++j ) {
70          sb.append( " " );
71          switch ( h[i][j] ) {
72            case -1:  sb.append( "X" );  break;
73            case 0:   sb.append( " " );  break;
74            default:  sb.append( h[i][j] );  break;
75          }
76        }
77        sb.append( System.getProperty( "line.separator" ) );
78      }
79      return sb.toString();
80    }
81
82    @Override
83    public boolean elofeltetel( Operator op ) {
84      if ( op instanceof Balra ) {
85        Balra balra = ( Balra ) op;
86        int x = balra.getX();
87        int y = balra.getY();
88        if ( h[x][y] <= 0 )
89          return false;
90        if ( y <= 1 )
91          return false;
92        if ( h[x][y - 1] != 0 )
93          return false;
94        for ( int i = 1; i <= N; ++i )
95          for ( int j = 1; j <= M; ++j ) {
96            if ( i < N && h[i][j] > 0 && h[i + 1][j] == 0 )
97              return false;
98            if ( i < N && j < M && h[i][j] > 0 &&
99                 ( h[i + 1][j] == h[i][j] || h[i][j + 1] == h[i][j] ) )
100             return false;
101         }
102       return true;
103     } else if ( op instanceof Jobbra ) {
104       Jobbra jobbra = ( Jobbra ) op;
105       int x = jobbra.getX();
106       int y = jobbra.getY();
107       if ( h[x][y] <= 0 )
108         return false;
109       if ( y >= M )
110         return false;
111       if ( h[x][y + 1] != 0 )
112         return false;
113       for ( int i = 1; i <= N; ++i )
114         for ( int j = 1; j <= M; ++j ) {
115           if ( i < N && h[i][j] > 0 && h[i + 1][j] == 0 )
116             return false;
117           if ( i < N && j < M && h[i][j] > 0 &&
```

```
118                 ( h[i + 1][j] == h[i][j] || h[i][j + 1] == h[i][j] ) )
119               return false;
120           }
121         return true;
122       } else if ( op instanceof Hullik ) {
123         for ( int i = 1; i <= N; ++i )
124           for ( int j = 1; j <= M; ++j )
125             if ( i < N && h[i][j] > 0 && h[i + 1][j] == 0 )
126               return true;
127       } else if ( op instanceof Eltuntet ) {
128         for ( int i = 1; i <= N; ++i )
129           for ( int j = 1; j <= M; ++j )
130             if ( i < N && h[i][j] > 0 && h[i + 1][j] == 0 )
131               return false;
132         for ( int i = 1; i <= N; ++i )
133           for ( int j = 1; j <= M; ++j )
134             if ( i < N && j < M && h[i][j] > 0 &&
135                  ( h[i + 1][j] == h[i][j] || h[i][j + 1] == h[i][j] ) )
136               return true;
137       }
138       return false;
139     }
140
141     @Override
142     public Allapot alkalmaz( Operator op ) {
143       Szabadeses uj = new Szabadeses( this );
144       if ( op instanceof Balra ) {
145         Balra balra = ( Balra ) op;
146         int x = balra.getX();
147         int y = balra.getY();
148         for ( int i = 1; i <= N; ++i )
149           for ( int j = 1; j <= M; ++j )
150             if ( i == x && j == y )
151               uj.h[i][j] = 0;
152             else if ( i == x && j == y - 1 )
153               uj.h[i][j] = h[x][y];
154       } else if ( op instanceof Jobbra ) {
155         Jobbra jobbra = ( Jobbra ) op;
156         int x = jobbra.getX();
157         int y = jobbra.getY();
158         for ( int i = 1; i <= N; ++i )
159           for ( int j = 1; j <= M; ++j )
160             if ( i == x && j == y )
161               uj.h[i][j] = 0;
162             else if ( i == x && j == y + 1 )
163               uj.h[i][j] = h[x][y];
164       } else if ( op instanceof Hullik ) {
165         for ( int i = 1; i <= N; ++i )
166           for ( int j = 1; j <= M; ++j )
```

```
167             if ( i < N && h[i][j] > 0 && h[i + 1][j] == 0 )
168               uj.h[i][j] = 0;
169             else if ( i > 1 && h[i - 1][j] > 0 && h[i][j] == 0 )
170               uj.h[i][j] = h[i - 1][j];
171       } else if ( op instanceof Eltuntet ) {
172         for ( int i = 1; i <= N; ++i )
173           for ( int j = 1; j <= M; ++j )
174             if ( i < N && h[i][j] > 0 && h[i + 1][j] == h[i][j] )
175               uj.h[i][j] = 0;
176             else if ( i > 1 && h[i][j] > 0 && h[i - 1][j] == h[i][j] )
177               uj.h[i][j] = 0;
178             else if ( j < M && h[i][j] > 0 && h[i][j + 1] == h[i][j] )
179               uj.h[i][j] = 0;
180             else if ( j > 1 && h[i][j] > 0 && h[i][j - 1] == h[i][j] )
181               uj.h[i][j] = 0;
182       }
183       return uj;
184     }
185   }
```

A kód számunkra érdekes részei a következők:

- **12–21. sor**:

  Mivel most két darab két paraméteres és két darab paraméter nélküli operátorunk van, mindegyiket – a paramétereseket az összes megengedett paraméter-kombinációval – fel kell vennünk az operátorok halmazába. Ez történik ebben a statikus inicializáló blokkban.

- **23. sor**:

  Az állapotunk jellemzőit a karaktereket tartalmazó kétdimenziós `h` tömbben tároljuk.

- **25–32. sor**:

  Az osztály egyik konstruktora, amely a kezdőállapot létrehozására szolgál. A `h` mátrix számára a minimálisan szükséges méretnél mindkét dimenziójában eggyel több tárhelyet foglalunk le, hogy egy az egyben követhessük a reprezentációban megadott indexelést. Ha ezt nem tennénk meg, akkor – a Java nyelv szabályai szerint – a mátrix mindkét dimenziójában a reprezentációban megadottaknál eggyel kisebb indexeket kellene használnunk. A „túlfoglalt” tárhelyeket (a 0-ás indexű sor és oszlop elemeit) tetszőleges értékkel feltölthetjük, az egyszerűség kedvéért mi 0-val töltöttük fel őket.

- **38–45. sor**:

  A célfeltétel vizsgálatát két, egymásba ágyazott `for` ciklussal végezzük. Ez felel meg a

  $$\forall i \, \forall j \, (h_{i,j} \leq 0)$$

  formulának.

- **82–139. sor**:

  Az `elofeltetel` metódus egyetlen nagy `if`–`else if`–...–`else` szerkezetet tartalmaz, ebben vizsgáljuk, hogy a négy lehetséges operátor közül melyikkel kell foglalkoznunk. A Balra operátor előfeltételeit a 87–100., a Jobbra operátor előfeltételeit a 106–119., a Hullik operátor előfeltételeit a 122-125., míg az Eltüntet operátor előfeltételeit a 127–135. sorokban vizsgáljuk. Érdemes megfigyelni, hogy például a Balra operátor előfeltételeinek ellenőrzésénél előre kerültek a kvantormentes formulák ((2.26), (2.27) és (2.28)) kódjai, megelőzve a kvantoros formulákét ((2.25) és (2.26)), amelyeket ráadásul össze is vontunk közös `for`-ciklusokba. Ezt azért tettük, hogy az egy-egy elemre, értékre vonatkozó vizsgálatok kiértékelésével ne kelljen azokra a vizsgálatokra várni, amelyeknek a kétdimenziós tömb összes elemét végig kell vizsgálniuk.

- **141–185. sor**:

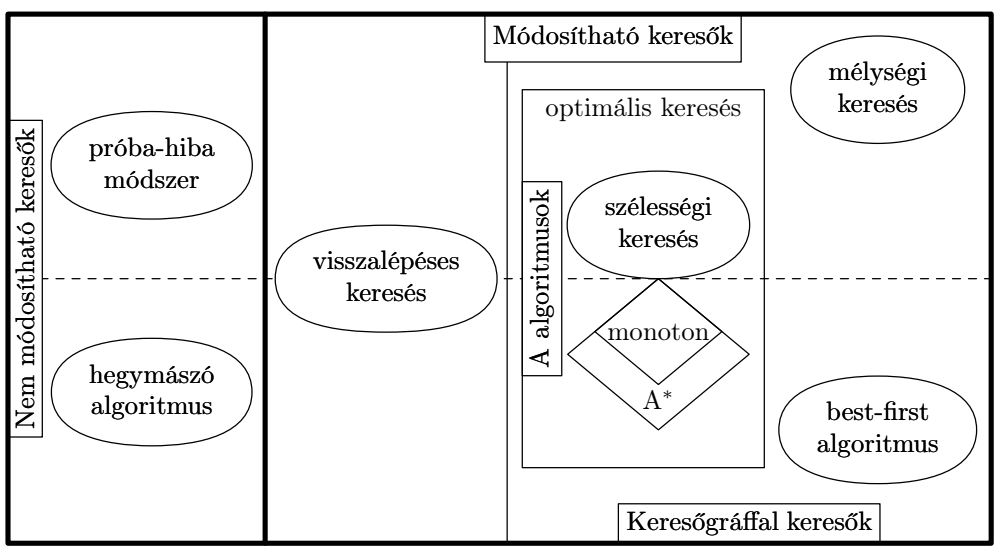
  Az alkalmazandó operátorokat az `alkalmaz` metódusban is egy `if`–`else if`–...–`else` szerkezettel válogatjuk szét. A 143–153. sorokban a Balra, a 153–163. sorokban a Jobbra, a 163–170. sorokban a Hullik, míg a 170–181. sorokban az Eltüntet operátor hatását programoztuk le.

3. FEJEZET

# Keresőalgoritmusok objektumorientált megközelítésben

## 3.1. Megoldást kereső rendszerek csoportosítása

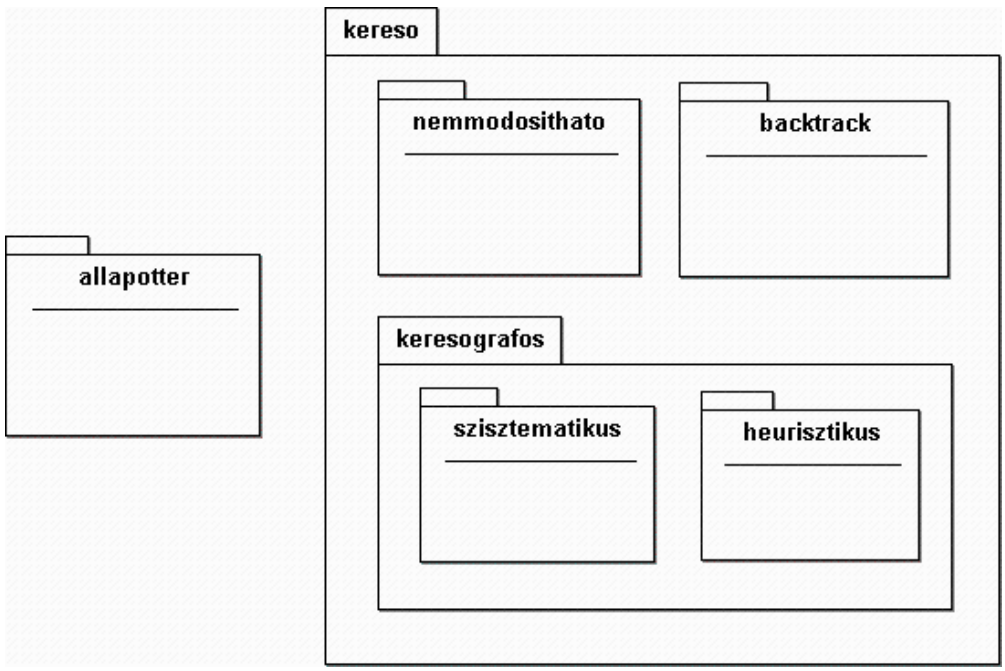
*A mesterséges intelligencia alapjai* kurzuson a hallgatók megismerkednek a megoldáskereső rendszerek komponenseivel (adatbázis, műveletek, vezérlő). A keresőrendszereket az adatbázisukban tárolt adatok, az adatbázist módosító műveleteik és vezérlési stratégiájuk alapján a 3.1. ábrán látható módon csoportosíthatjuk. Az ábrán egy vízszintes szaggatott vonal választja el a nem informált keresőket (fent) a heurisztikusaktól (lent).




3.1. ábra. A megoldást kereső rendszerek csoportosítása

A Venn-diagramra pillantva adja magát az ötlet, hogy a keresőalgoritmusoknak egy olyan hierarchiáját alakítsuk ki, amely hűen követi a diagram felépítését. Ezért az

előző fejezetben bemutatott `allapotter` csomag mellé egy másik csomaghierarchiát képzeltünk el, amelynek szerkezete a 3.2. ábrán látható.




3.2. ábra. A megoldást kereső rendszerek csomaghierarchiája

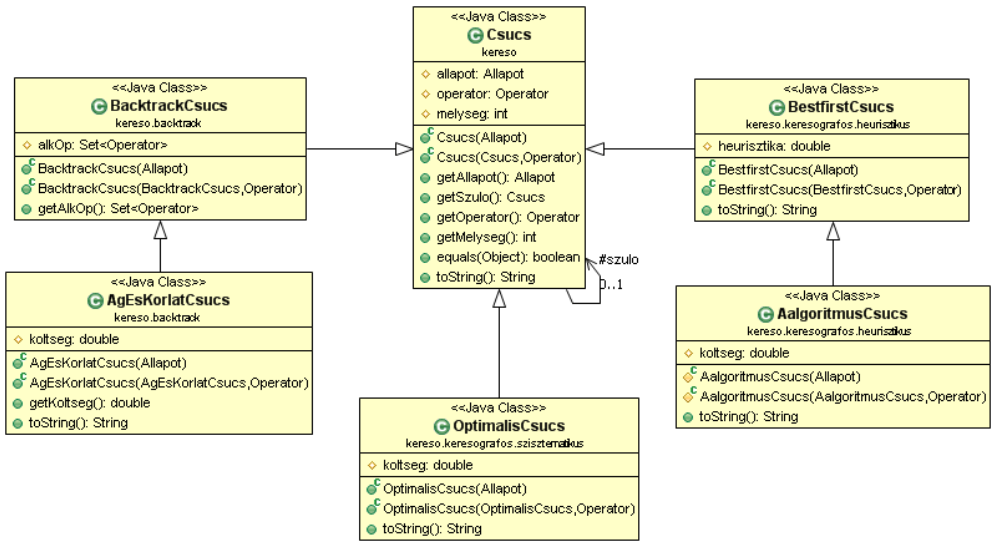
## 3.2. A javasolt objektumorientált megközelítés

A megoldáskereső algoritmusok csomagjában lévő osztályok két különálló hierarchiába rendeződnek. Az egyik (3.3. ábra) az állapottérgráfok csúcsainak, a másik (3.4. ábra) a különböző keresőalgoritmusoknak a modellezésére szolgál.

### 3.2.1. A reprezentációs gráf csúcsainak megvalósítása

A csúcsokra vonatkozó csaknem minden információ a különböző csúcstípusok közös ősosztályában, a `Csucs` osztályban található. Itt tároljuk a csúcs által szemléltetett állapotot, a csúcsnak a startcsúcstól mért távolságát (mélységét), a szülő csúcsát, illetve azt az operátort, amellyel a szülőjében tárolt állapotból az adott csúcsban tárolt állapotot előállítottuk. A `Csucs` osztály leszármazottai ezeken az attribútumokon túl a konkrét keresőalgoritmusok által igényelt további adattagokat tartalmaznak.

```
1  package kereso;
2
```




3.3. ábra. A reprezentációs gráf csúcsainak osztálydiagramja

```
3   import allapotter.*;
4
5   public class Csucs
6   {
7     protected Allapot allapot;
8     protected Csucs szulo;
9     protected Operator operator;
10    protected int melyseg;
11
12    public Csucs( Allapot kezdoAllapot ) {
13      allapot = kezdoAllapot;
14      szulo = null;
15      operator = null;
16      melyseg = 0;
17    }
18
19    public Csucs( Csucs szulo, Operator operator ) {
20      allapot = szulo.allapot.alkalmaz( operator );
21      this.szulo = szulo;
22      this.operator = operator;
23      melyseg = szulo.melyseg + 1;
24    }
25
26    public Allapot getAllapot()   { return allapot;  }
27    public Csucs getSzulo()       { return szulo;    }
28    public Operator getOperator() { return operator; }
```

```
29    public int getMelyseg()       { return melyseg;  }
30
31    @Override
32    public boolean equals( Object obj ) {
33      return obj instanceof Csucs &&
34             allapot.equals( ( ( Csucs )obj ).allapot );
35    }
36
37    @Override
38    public String toString() {
39      return ( operator == null ? "" : operator + " => " ) +
40               allapot + " (" + melyseg + ")";
41    }
42  }
```

A visszalépéses kereső fogja használni majd a `BacktrackCsucs` osztályt, amely egy `alkOp` adattaggal egészíti ki az örökölt tulajdonságait. Ebben a halmaz típusú adattagban tárolják majd az egyes `BacktrackCsucs` típusú objektumok a csúcsban tárolt állapotra alkalmazható, de még ki nem próbált operátorokat.

```
1   package kereso.backtrack;
2
3   import java.util.Set;
4   import java.util.HashSet;
5   import allapotter.*;
6   import kereso.Csucs;
7
8   public class BacktrackCsucs extends Csucs {
9     protected Set<Operator> alkOp;
10
11    public BacktrackCsucs( Allapot kezdoAllapot ) {
12      super( kezdoAllapot );
13      alkOp = new HashSet<Operator>();
14      for ( Operator op : Allapot.getOperatorok() )
15        if ( allapot.elofeltetel( op ) )
16          alkOp.add( op );
17    }
18
19    public BacktrackCsucs( BacktrackCsucs szulo, Operator operator ) {
20      super( szulo, operator );
21      alkOp = new HashSet<Operator>();
22      for ( Operator op : Allapot.getOperatorok() )
23        if ( allapot.elofeltetel( op ) )
24          alkOp.add( op );
25    }
26
27    public Set<Operator> getAlkOp() { return alkOp; }
28  }
```

Az „ág és korlát” algoritmus csúcsainak `AgEsKorlatCsucs` nevű osztálya a `BacktrackCsucs` osztályból öröklődik, és kiegészíti azt egy `koltseg` adattaggal, valamint a hozzá tartozó lekérdező metódussal. Ennél a csúcsfajtánál a költségértékek az operátorok alkalmazásaiból származnak, ezért a konstruktorait úgy terveztük meg, hogy képes legyen az ilyen csúcsokat is kezelni. Azt is mondhatnánk, hogy az „ág és korlát” kereső számára éppen az interfészt nem implementáló operátorok jelentik a kivételeket.

```
1  package kereso.backtrack;
2
3  import allapotter.*;
4
5  public class AgEsKorlatCsucs extends BacktrackCsucs
6  {
7    protected double koltseg;
8
9    public AgEsKorlatCsucs( Allapot allapot ) {
10     super( allapot );
11     koltseg = 0;
12   }
13
14   public AgEsKorlatCsucs( AgEsKorlatCsucs szulo, Operator operator ) {
15     super( szulo, operator );
16     koltseg = szulo.koltseg + ( operator instanceof KoltsegesOperator ?
17       ( ( KoltsegesOperator )operator ).koltseg( szulo.allapot ) : 1 );
18   }
19
20   public double getKoltseg() { return koltseg; }
21
22   @Override
23   public String toString() {
24     return super.toString() + ", koltseg=" + koltseg;
25   }
26 }
```

Az `OptimalisCsucs` osztályt az optimális keresőhöz terveztük. A szülő osztályától örökölt adattagok mellé egy `koltseg` adattagot definiál, amely a startcsúcsból az adott csúcsba vezető út költségét tárolja. Konstruktorában a `KoltsegesOperator` interfészt megvalósító operátorokat is képes kezelni, hasonlóan az `AgEsKorlatCsucs` osztályhoz.

```
1  package kereso.keresografos.szisztematikus;
2
3  import allapotter.*;
4  import kereso.Csucs;
5
6  public class OptimalisCsucs extends Csucs
7  {
8    protected double koltseg;
9
10   public OptimalisCsucs( Allapot kezdoAllapot ) {
```

```
11      super( kezdoAllapot );
12      koltseg = 0;
13    }
14
15    public OptimalisCsucs( OptimalisCsucs szulo, Operator operator ) {
16      super( szulo, operator );
17      koltseg = szulo.koltseg + ( operator instanceof KoltsegesOperator ?
18        ( ( KoltsegesOperator )operator ).koltseg( szulo.allapot ) : 1 );
19    }
20
21    @Override
22    public String toString() {
23      return super.toString() + ", koltseg=" + koltseg;
24    }
25  }
```

A `BestFirstCsucs` az első olyan osztály, amelyet egy heurisztikus keresőhöz terveztünk. Ennek megfelelően egy `heurisztika` adattagja van, amely a csúcsban tárolt állapotra vonatkozik, és az aktuális csúcsból egy terminális csúcsba történő eljutás becsült értékét tartalmazza. Természetesen leginkább akkor van értelme ilyen értékről beszélni, ha a csúcsban tárolt állapot implementálja a `HeurisztikusAllapot` interfészt. Ezt az ellenőrzést és az adattag beállítását a konstruktorok fogják elvégezni.

```
1   package kereso.keresografos.heurisztikus;
2
3   import allapotter.*;
4   import kereso.Csucs;
5
6   public class BestfirstCsucs extends Csucs
7   {
8     protected double heurisztika;
9
10    public BestfirstCsucs( Allapot kezdoAllapot ) {
11      super( kezdoAllapot );
12      heurisztika = allapot instanceof HeurisztikusAllapot ?
13        ( ( HeurisztikusAllapot )allapot ).heurisztika() : 0;
14    }
15
16    public BestfirstCsucs( BestfirstCsucs szulo, Operator operator ) {
17      super( szulo, operator );
18      heurisztika = allapot instanceof HeurisztikusAllapot ?
19        ( ( HeurisztikusAllapot )allapot ).heurisztika() : 0;
20    }
21
22    @Override
23    public String toString() {
24      return super.toString() + ", heurisztika=" + heurisztika;
25    }
26  }
```

Az `AalgoritmusCsucs` osztály esetén az osztályok között egyszeres öröklődést feltételezve kétféleképpen járhatunk el:

- vagy a `BestFirstCsucs` osztályból örököltetjük, majd kiegészítjük egy `koltseg` adattaggal,
- vagy az `OptimalisCsucs` osztályból örököltetjük, és aztán egy `heurisztika` adattaggal egészítjük ki.

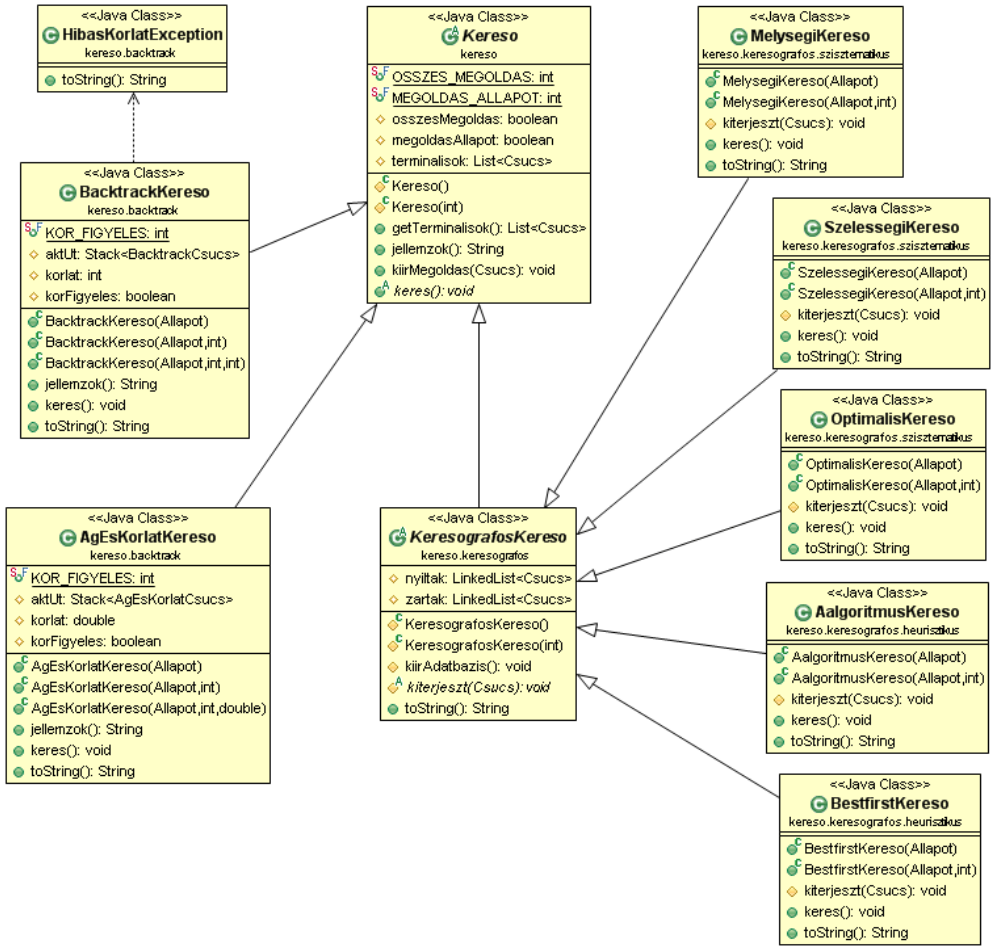
Mi az első megoldást választottuk, így a kód a következőképpen néz ki:

```
1  package kereso.keresografos.heurisztikus;
2
3  import allapotter.*;
4
5  public class AalgoritmusCsucs extends BestfirstCsucs
6  {
7    protected double koltseg;
8
9    protected AalgoritmusCsucs( Allapot kezdoAllapot ) {
10     super( kezdoAllapot );
11     koltseg = 0;
12   }
13
14   protected AalgoritmusCsucs(AalgoritmusCsucs szulo,Operator operator) {
15     super( szulo, operator );
16     koltseg = szulo.koltseg + ( operator instanceof KoltsegesOperator ?
17       ( ( KoltsegesOperator )operator ).koltseg( szulo.allapot ) : 1 );
18   }
19
20   @Override
21   public String toString() {
22     return super.toString() + ", koltseg=" + koltseg;
23   }
24 }
```

### 3.2.2. A megoldást kereső rendszerek megvalósítása

A keresőalgoritmusok ősosztálya az absztrakt `Kereso` osztály, amely éppen a legfontosabb metódusát nem implementálja, nevezetesen azt, hogy milyen stratégiával kell működnie a megoldáskeresés vezérlőjének. Ennek a metódusnak a kódját a leszármazott osztályok tartalmazzák majd. Ugyanakkor lehetőséget biztosít arra, hogy tároljuk a terminális csúcsokat, kiírjuk a megoldásokat, illetve hogy beállítsuk az egyes keresők algoritmustól független jellemzőit. A bemutatásra kerülő kódokból – helytakarékossági okok miatt – kihagyjuk a keresők jellemzőit `String` típusú értékként visszaadó `jellemzok` és `toString` metódusokat.

```
1  package kereso;
2
```




3.4. ábra. A keresőalgoritmusok osztálydiagramja

```
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public abstract class Kereso
7  {
8    public static final int OSSZES_MEGOLDAS = 1;
9    public static final int MEGOLDAS_ALLAPOT = 2;
10
11   protected boolean osszesMegoldas;
12   protected boolean megoldasAllapot;
13   protected List<Csucs> terminalisok;
14
```

```
15    protected Kereso() {
16      osszesMegoldas = false;
17      megoldasAllapot = false;
18      terminalisok = new ArrayList<Csucs>();
19    }
20
21    protected Kereso( int jellemzok ) {
22      this();
23      osszesMegoldas = ( jellemzok & OSSZES_MEGOLDAS ) != 0;
24      megoldasAllapot = ( jellemzok & MEGOLDAS_ALLAPOT ) != 0;
25    }
26
27    public List<Csucs> getTerminalisok() { return terminalisok; }
28
29    public void kiirMegoldas( Csucs cs ) {
30      if ( megoldasAllapot )
31        System.out.println( cs.getAllapot() );
32      else if ( cs != null ) {
33        kiirMegoldas( cs.getSzulo() );
34        System.out.println( cs );
35      }
36    }
37
38    public abstract void keres();
39  }
```

A `BacktrackKereso` osztály új jellemzői között találjuk az aktuális utat, valamint a körfigyelés és az úthosszkorlát opcionális lehetőségét. Igényünket az utóbbi két lehetőség kihasználására a konstruktorok paramétereként kell megadni a keresőnek. A `keres` metódust úgy írtuk meg, hogy a keresési paraméterek minden lehetséges kombinációjára fel legyen készítve.

```
1   package kereso.backtrack;
2
3   import java.util.Set;
4   import java.util.Stack;
5   import allapotter.*;
6   import kereso.Kereso;
7
8   public class BacktrackKereso extends Kereso
9   {
10    public static final int KOR_FIGYELES = 4;
11
12    protected Stack<BacktrackCsucs> aktUt;
13    protected int korlat;
14    protected boolean korFigyeles;
15
16    {
17      aktUt = new Stack<BacktrackCsucs>();
```

```
18      korlat = 0;
19      korFigyeles = false;
20    }
21
22    public BacktrackKereso( Allapot kezdoAllapot ) {
23      aktUt.push( new BacktrackCsucs( kezdoAllapot ) );
24    }
25
26    public BacktrackKereso( Allapot kezdoAllapot, int jellemzok ) {
27      super( jellemzok );
28      aktUt.push( new BacktrackCsucs( kezdoAllapot ) );
29      korFigyeles = ( jellemzok & KOR_FIGYELES ) != 0;
30    }
31
32    public BacktrackKereso( Allapot kezdoAllapot, int jellemzok, int k )
33      throws HibasKorlatException {
34      this( kezdoAllapot, jellemzok );
35      if ( k < 1 )
36        throw new HibasKorlatException();
37      this.korlat = k;
38    }
39
40    @Override
41    public void keres() {
42      while ( !aktUt.empty() ) {
43        BacktrackCsucs aktualis = aktUt.peek();
44        if ( aktualis.getAllapot().celAllapot() ) {
45          if ( !( megoldasAllapot && terminalisok.contains( aktualis ) ) )
46            terminalisok.add( aktualis );
47          if ( osszesMegoldas ) {
48            aktUt.pop();
49            continue;
50          }
51          else
52            break;
53        }
54        if ( korlat > 0 && aktualis.getMelyseg() == korlat ) {
55          aktUt.pop();
56          continue;
57        }
58        Set<Operator> alkOp = aktualis.getAlkOp();
59        if ( alkOp.isEmpty() ) {
60          aktUt.pop();
61          continue;
62        }
63        Operator op = alkOp.iterator().next();
64        BacktrackCsucs uj = new BacktrackCsucs( aktualis, op );
65        if ( korFigyeles && aktUt.contains( uj ) )
66          ;
```

```
67          else
68            aktUt.push( uj );
69          alkOp.remove( op );
70        }
71      }
72    }
```

Az algoritmus paramétereként megadható jellemzők listáját és szemantikáját az alábbiakban foglaljuk össze.

- `OSSZES_MEGOLDAS`: Azt írja elő a kereső számára, hogy ne álljon le az első megoldás megtalálásakor, hanem ha talál egy megoldást, akkor tárolja a hozzá tartozó terminális csúcsot a terminális csúcsok halmazában, hajtson végre egy visszalépést, és folytassa a keresést egészen addig, míg az adatbázisa ki nem ürül (az aktuális út hossza 0-ra nem csökken). Ilyen módon a keresés csak a startcsúcsból történő visszalépés után áll le, és minden olyan útvonalat megjegyez, amely a startcsúcsból egy tetszőleges terminális csúcsba vezet.

- `MEGOLDAS_ALLAPOT`: Ha be van állítva, akkor a megoldás(ok) kiíratásakor csak a megoldás útvonalának terminális csúcsában tárolt állapotot írja ki a teljes útvonal állapotai és a megoldás operátorai helyett. Olyan problémáknál érdemes használni, ahol

  – vagy nem lényeges az, hogy hogyan (milyen operátorsorozattal) állítottuk elő a célállapotot,

  – vagy az állapotok komponensei olyan értékeket tartalmaznak, amelyekből következtetni lehet az operátorok végrahajtási sorozatára.

  Ha az összes megoldását keressük egy problémának, és ez a jellemző be van állítva, akkor a terminális csúcsok közé csak az egymástól különböző állapotokat tartalmazó csúcsok kerülnek be. Ellenkező esetben, mivel egy csúcs szülő komponense eltérhet az ugyanolyan állapotot tartalmazó másik csúcsétól, mindkét csúcs tárolásra kerül.

- `KOR_FIGYELES`: Olyan problémáknál érdemes használni, amelyeknek a reprezentációs gráfjai köröket tartalmaznak. Ha be van állítva, akkor az algoritmusok minden új állapotnak az aktuális út végéhez történő hozzáfűzése előtt végrehajtanak egy ellenőrzést, hogy az új állapot szerepel-e az aktuális útvonal valamelyik csúcsában.

- Úthosszkorlátos visszalépéses keresés esetén a fenti három jellemző mellett megadhatunk egy plusz paramétert, amely az úthosszkorlát (a startcsúcstól való távolság) értékét tartalmazza. Ennek a paraméternek a megadása jelzi tulajdonképpen a keresőnek, hogy úthosszkorlátos keresést szeretnénk végezni. Az úthosszkorlát elérésekor a kereső visszalépést fog végrehajtani.

A visszalépéses kereső paraméterei közé tartozó úthosszkorlátot a kereső példányosításakor adhatjuk meg. Ha érvénytelen értéket (nullát vagy negatív számot) adunk meg, kiváltódik a `HibasKorlatException` kivétel, melynek kódja a következő:

```
1  package kereso.backtrack;
2
3  public class HibasKorlatException extends Exception
4  {
5    @Override
6    public String toString() {
7      return "A korlat nem pozitiv szam!";
8    }
9  }
```

Az `AgEsKorlatKereso` osztály annyiban tér el a `BacktrackKereso` osztálytól, hogy az úthosszkorlát helyett költségkorláttal dolgozik, amely itt nem opció, hanem az algoritmus szerves részét képezi.

```
1  package kereso.backtrack;
2
3  import java.util.Set;
4  import java.util.Stack;
5  import allapotter.*;
6  import kereso.Kereso;
7
8  public class AgEsKorlatKereso extends Kereso
9  {
10   public static final int KOR_FIGYELES = 4;
11
12   protected Stack<AgEsKorlatCsucs> aktUt;
13   protected double korlat;
14   protected boolean korFigyeles;
15
16   {
17     aktUt = new Stack<AgEsKorlatCsucs>();
18     korlat = 0;
19     korFigyeles = false;
20   }
21
22   public AgEsKorlatKereso( Allapot kezdoAllapot ) {
23     aktUt.push( new AgEsKorlatCsucs( kezdoAllapot ) );
24   }
25
26   public AgEsKorlatKereso( Allapot kezdoAllapot, int jellemzok ) {
27     super( jellemzok );
28     aktUt.push( new AgEsKorlatCsucs( kezdoAllapot ) );
29     korFigyeles = ( jellemzok & KOR_FIGYELES ) != 0;
30   }
31
32   public AgEsKorlatKereso( Allapot kezdo, int jellemzok, double k ) {
33     this( kezdo, jellemzok );
34     this.korlat = k;
35   }
```

```
36
37      @Override
38      public void keres()
39      {
40        while ( !aktUt.empty() ) {
41          AgEsKorlatCsucs aktualis = aktUt.peek();
42          if ( aktualis.getAllapot().celAllapot() &&
43               ( korlat <= 0 || aktualis.getKoltseg() <= korlat ) ) {
44            if ( aktualis.getKoltseg() < korlat )
45              terminalisok.clear();
46            if ( terminalisok.isEmpty() || osszesMegoldas &&
47                 !( megoldasAllapot && terminalisok.contains( aktualis ) ) )
48              terminalisok.add( aktualis );
49            korlat = aktualis.getKoltseg();
50            aktUt.pop();
51            continue;
52          }
53          if ( korlat > 0 && aktualis.getKoltseg() >= korlat ) {
54            aktUt.pop();
55            continue;
56          }
57          Set<Operator> alkOp = aktualis.getAlkOp();
58          if ( alkOp.isEmpty() ) {
59            aktUt.pop();
60            continue;
61          }
62          Operator op = alkOp.iterator().next();
63          AgEsKorlatCsucs uj = new AgEsKorlatCsucs( aktualis, op );
64          if ( korFigyeles && aktUt.contains( uj ) )
65            ;
66          else
67            aktUt.push( uj );
68          alkOp.remove( op );
69        }
70      }
71    }
```

Az „ág és korlát” algoritmus paramétereként megadható jellemzők listáját és szemantikáját az alábbiakban foglaljuk össze.

- `OSSZES_MEGOLDAS`: Mivel az „ág és korlát” algoritmussal egy probléma optimális megoldását találhatjuk meg, ezért itt ez a jellemző azt írja elő a kereső számára, hogy a keresés során talált összes optimális megoldást jegyezze meg. Nyilván, amíg nincs egyetlen megoldás sem a tarsolyában, addig nincs gond, az első megoldás terminális csúcsát biztosan meg fogja jegyezni. A következő megoldás, amit megtalál, az vagy ugyanolyan jó, mint az utoljára megjegyzett, vagy jobb annál. Utóbbi esetben az eddig feljegyzett terminális csúcso(ka)t törli, majd az újonnan megtalált megoldás terminális csúcsát mindenképpen feljegyzi a terminális csúcsok halmazába.

- `MEGOLDAS_ALLAPOT`: Hasonlóan értelmezhető, mint a visszalépéses keresőnél.
- `KOR_FIGYELES`: Hasonlóan értelmezhető, mint a visszalépéses keresőnél.
- A keresőalgoritmus konstruktoraiban opcionálisan használható negyedik paraméter a költségkorlát. Ha nem adjuk meg, alapértelmezett értéke 0, amely azt jelzi a keresőnek, hogy az első megoldás megtalálásáig nem kell a megoldás költségét figyelnie, azt elég az első megoldás megtalálásakor a megoldás költségére beállítania. Ha a konstruktorhívásban szerepel a kölségparaméter, akkor viszont csak olyan megoldásokat tudunk a keresővel meghatározni, amelynek a költsége kisebb a paraméterként megadottnál.

A keresőgráffal keresők közös absztrakt szülőosztálya a `KeresografosKereso` osztály, amely az adatbázisában tárolt csúcsok nyilvántartásához két listát használ: az egyiket a nyílt csúcsok, a másikat a zárt csúcsok számára. Attól lesz absztrakt osztály, mert a vezérlési stratégiától függő `kiterjeszt` módszerét nem implementáltuk.

```
1  package kereso.keresografos;
2
3  import java.util.LinkedList;
4  import kereso.*;
5
6  public abstract class KeresografosKereso extends Kereso
7  {
8    protected LinkedList<Csucs> nyiltak, zartak;
9
10   {
11     nyiltak = new LinkedList<Csucs>();
12     zartak = new LinkedList<Csucs>();
13   }
14
15   protected KeresografosKereso() {
16   }
17
18   protected KeresografosKereso( int jellemzok ) {
19     super( jellemzok );
20   }
21
22   protected void kiirAdatbazis() {
23     System.out.println( "Nyilt csucsok:" );
24     for ( Csucs cs : nyiltak )
25       System.out.println( cs );
26     System.out.println( "Zart csucsok:" );
27     for ( Csucs cs : zartak )
28       System.out.println( cs );
29     System.out.println();
30   }
31
```

```
32    protected abstract void kiterjeszt( Csucs csucs );
33  }
```

Ezen osztály leszármazottai (a konkrét gráfkereső algoritmusok) egyrészt a keresési stratégiában, másrészt pedig az adatbázisukban tárolt csúcsok osztályában térnek el egymástól.

A `MelysegiKereso` osztály `Csucs` típusú objektumokat használ a reprezentációs gráf csúcsainak tárolására. (Ezért nem absztrakt a `Csucs` osztály a csúcsok hierarchiájában.) Nem használja viszont ki a csúcsokban tárolt mélységi számot, mert a keresés során a nyílt csúcsok listáját veremként kezeli, amelyben így mindig a legnagyobb mélységi számmal rendelkező csúcsok lesznek elöl, függetlenül attól, hogy megvizsgáljuk-e a mélységi számukat vagy sem. Fontos még megjegyezni, hogy a bemutatott kódban nem hozzuk előre a kiterjesztéskor újonnan generált csúcsok tesztelését, azt csak kiterjesztésre kiválasztáskor tesszük meg.

```
1   package kereso.keresografos.szisztematikus;
2
3   import allapotter.*;
4   import kereso.Csucs;
5   import kereso.keresografos.KeresografosKereso;
6
7   public class MelysegiKereso extends KeresografosKereso
8   {
9     public MelysegiKereso( Allapot kezdoAllapot ) {
10      nyiltak.add( new Csucs( kezdoAllapot ) );
11    }
12
13    public MelysegiKereso( Allapot kezdoAllapot, int jellemzok ) {
14      super( jellemzok );
15      nyiltak.add( new Csucs( kezdoAllapot ) );
16    }
17
18    @Override
19    protected void kiterjeszt( Csucs csucs ) {
20      for ( Operator op : Allapot.getOperatorok() )
21        if ( csucs.getAllapot().elofeltetel( op ) ) {
22          Csucs uj = new Csucs( csucs, op );
23          if ( !( nyiltak.contains( uj ) || zartak.contains( uj ) ) )
24            nyiltak.addFirst( uj );
25        }
26    }
27
28    @Override
29    public void keres() {
30      while ( true ) {
31        if ( nyiltak.isEmpty() )
32          break;
33        Csucs aktualis = nyiltak.getFirst();
34        if ( aktualis.getAllapot().celAllapot() ) {
```

```
35          terminalisok.add( aktualis );
36          if ( osszesMegoldas ) {
37            zartak.add( nyiltak.removeFirst() );
38            continue;
39          }
40          else
41            break;
42        }
43        zartak.add( nyiltak.removeFirst() );
44        kiterjeszt( aktualis );
45      }
46    }
47  }
```

A `SzelessegiKereso` osztály nagyon hasonlít az előbb bemutatott `MelysegiKereso` osztályra, annyi különbséggel, hogy itt a nyílt csúcsok listája sorként működik. Ennél a keresőnél is kiterjesztésre kiválasztáskor tesztelünk, ami az esetek nagy részében az algoritmus lassú működését eredményezi.

```
1   package kereso.keresografos.szisztematikus;
2
3   import allapotter.*;
4   import kereso.Csucs;
5   import kereso.keresografos.KeresografosKereso;
6
7   public class MelysegiKereso extends KeresografosKereso
8   {
9     public MelysegiKereso( Allapot kezdoAllapot ) {
10      nyiltak.add( new Csucs( kezdoAllapot ) );
11    }
12
13    public MelysegiKereso( Allapot kezdoAllapot, int jellemzok ) {
14      super( jellemzok );
15      nyiltak.add( new Csucs( kezdoAllapot ) );
16    }
17
18    @Override
19    protected void kiterjeszt( Csucs csucs ) {
20      for ( Operator op : Allapot.getOperatorok() )
21        if ( csucs.getAllapot().elofeltetel( op ) ) {
22          Csucs uj = new Csucs( csucs, op );
23          if ( !( nyiltak.contains( uj ) || zartak.contains( uj ) ) )
24            nyiltak.addLast( uj );
25        }
26    }
27
28    @Override
29    public void keres() {
30      while ( true ) {
```

```
31        if ( nyiltak.isEmpty() )
32          break;
33        Csucs aktualis = nyiltak.getFirst();
34        if ( aktualis.getAllapot().celAllapot() ) {
35          terminalisok.add( aktualis );
36          if ( osszesMegoldas ) {
37            zartak.add( nyiltak.removeFirst() );
38            continue;
39          }
40          else
41            break;
42        }
43        zartak.add( nyiltak.removeFirst() );
44        kiterjeszt( aktualis );
45      }
46    }
47  }
```

Az `OptimalisKereso` osztály annyiban más az előző két keresőhöz képest, hogy ő már az őhozzá tervezett `OptimalisCsucs` típusú csúcsokat tárolja az adatbázisában, és minden kiterjesztés után növekvő sorrendbe teszi a nyílt csúcsait a startcsúcsból hozzájuk vezető út költsége alapján.

```
1   package kereso.keresografos.szisztematikus;
2
3   import java.util.Collections;
4   import java.util.Comparator;
5   import allapotter.*;
6   import kereso.Csucs;
7   import kereso.keresografos.KeresografosKereso;
8
9   public class OptimalisKereso extends KeresografosKereso
10  {
11    public OptimalisKereso( Allapot kezdoAllapot ) {
12      nyiltak.add( new OptimalisCsucs( kezdoAllapot ) );
13    }
14
15    public OptimalisKereso( Allapot kezdoAllapot, int jellemzok ) {
16      super( jellemzok );
17      nyiltak.add( new OptimalisCsucs( kezdoAllapot ) );
18    }
19
20    @Override
21    protected void kiterjeszt( Csucs csucs ) {
22      for ( Operator op : Allapot.getOperatorok() )
23        if ( csucs.getAllapot().elofeltetel( op ) ) {
24          OptimalisCsucs uj =
25                      new OptimalisCsucs( ( OptimalisCsucs )csucs, op );
26          int index;
```

```
27          if ( ( index = nyiltak.indexOf( uj ) ) != -1 ) {
28            OptimalisCsucs regi = ( OptimalisCsucs )nyiltak.get( index );
29            if ( uj.koltseg < regi.koltseg ) {
30              nyiltak.remove( regi );
31              nyiltak.add( uj );
32            }
33          }
34          else if ( !zartak.contains( uj ) )
35            nyiltak.add( uj );
36        }
37    }
38
39    @Override
40    public void keres()
41    {
42      while ( true ) {
43        if ( nyiltak.isEmpty() )
44          break;
45        OptimalisCsucs aktualis = ( OptimalisCsucs )nyiltak.getFirst();
46        if ( !terminalisok.isEmpty() && aktualis.koltseg >
47          ( ( OptimalisCsucs )terminalisok.iterator().next() ).koltseg )
48          break;
49        if ( aktualis.getAllapot().celAllapot() ) {
50          terminalisok.add( aktualis );
51          if ( osszesMegoldas ) {
52            zartak.add( nyiltak.removeFirst() );
53            continue;
54          }
55          else
56            break;
57        }
58        zartak.add( nyiltak.removeFirst() );
59        kiterjeszt( aktualis );
60        Collections.sort( nyiltak, new Comparator<Csucs>() {
61          @Override
62          public int compare( Csucs cs1, Csucs cs2 ) {
63            OptimalisCsucs ocs1 = ( OptimalisCsucs )cs1,
64                           ocs2 = ( OptimalisCsucs )cs2;
65            return new Double( ocs1.koltseg ).compareTo(
66                   new Double( ocs2.koltseg ) );
67          }
68        } );
69      }
70    }
71  }
```

A `BestFirstKereso` osztály az `OptimalisKereso` osztályhoz hasonlít leginkább, azzal a különbséggel, hogy az adatbázisában található `BestFirstCsucs` típusú nyílt csúcsokat minden kiterjesztést követően a bennük tárolt heurisztikus érték alapján

rendezi sorba.

```
1  package kereso.keresografos.heurisztikus;
2
3  import java.util.Collections;
4  import java.util.Comparator;
5  import allapotter.*;
6  import kereso.Csucs;
7  import kereso.keresografos.KeresografosKereso;
8
9  public class BestfirstKereso extends KeresografosKereso
10 {
11   public BestfirstKereso( Allapot kezdoAllapot ) {
12     nyiltak.add( new BestfirstCsucs( kezdoAllapot ) );
13   }
14
15   public BestfirstKereso( Allapot kezdoAllapot, int jellemzok ) {
16     super( jellemzok );
17     nyiltak.add( new BestfirstCsucs( kezdoAllapot ) );
18   }
19
20   @Override
21   protected void kiterjeszt( Csucs csucs ) {
22     for ( Operator op : Allapot.getOperatorok() )
23       if ( csucs.getAllapot().elofeltetel( op ) ) {
24         BestfirstCsucs uj =
25                     new BestfirstCsucs( ( BestfirstCsucs )csucs, op );
26         if ( !( nyiltak.contains( uj ) || zartak.contains( uj ) ) )
27           nyiltak.add( uj );
28       }
29   }
30
31   @Override
32   public void keres() {
33     while ( true ) {
34       if ( nyiltak.isEmpty() )
35         break;
36       BestfirstCsucs aktualis = ( BestfirstCsucs )nyiltak.getFirst();
37       if ( aktualis.getAllapot().celAllapot() ) {
38         terminalisok.add( aktualis );
39         if ( osszesMegoldas ) {
40           zartak.add( nyiltak.removeFirst() );
41           continue;
42         }
43         else
44           break;
45       }
46       zartak.add( nyiltak.removeFirst() );
47       kiterjeszt( aktualis );
```

```
48        Collections.sort( nyiltak, new Comparator<Csucs>() {
49          @Override
50          public int compare( Csucs cs1, Csucs cs2 ) {
51            BestfirstCsucs bcs1 = ( BestfirstCsucs )cs1,
52                           bcs2 = ( BestfirstCsucs )cs2;
53            return new Double( bcs1.heurisztika ).compareTo(
54                   new Double( bcs2.heurisztika ) );
55          }
56        } );
57      }
58    }
59  }
```

Befejezésképpen pedig következzék az `AalgoritmusKereso` osztály, amely ötvözi az `OptimalisKereso` és a `BestFirstKereso` osztályok keresési stratégiáját, mégpedig oly módon, hogy az adatbázisában tárolt `AalgoritmusCsucs` típusú nyílt csúcsokat a bennük tárolt költség- és heurisztikus értékek alapján rendezi növekvő sorrendbe minden kiterjesztés után.

```
1   package kereso.keresografos.heurisztikus;
2
3   import java.util.Collections;
4   import java.util.Comparator;
5   import allapotter.*;
6   import kereso.Csucs;
7   import kereso.keresografos.KeresografosKereso;
8
9   public class AalgoritmusKereso extends KeresografosKereso
10  {
11    public AalgoritmusKereso( Allapot kezdoAllapot ) {
12      nyiltak.add( new AalgoritmusCsucs( kezdoAllapot ) );
13    }
14
15    public AalgoritmusKereso( Allapot kezdoAllapot, int jellemzok ) {
16      super( jellemzok );
17      nyiltak.add( new AalgoritmusCsucs( kezdoAllapot ) );
18    }
19
20    @Override
21    protected void kiterjeszt( Csucs csucs ) {
22      for ( Operator op : Allapot.getOperatorok() )
23        if ( csucs.getAllapot().elofeltetel( op ) ) {
24          AalgoritmusCsucs uj =
25                       new AalgoritmusCsucs( (AalgoritmusCsucs)csucs, op );
26          int index;
27          if ( ( index = nyiltak.indexOf( uj ) ) != -1 ) {
28            AalgoritmusCsucs regi = (AalgoritmusCsucs)nyiltak.get( index );
29            if ( uj.koltseg < regi.koltseg ) {
30              nyiltak.remove( regi );
```

```
31              nyiltak.add( uj );
32            }
33          }
34          else if ( ( index = zartak.indexOf( uj ) ) != -1 ) {
35            AalgoritmusCsucs regi = (AalgoritmusCsucs)zartak.get( index );
36            if ( uj.koltseg < regi.koltseg ) {
37              zartak.remove( regi );
38              nyiltak.add( uj );
39            }
40          }
41          else
42            nyiltak.add( uj );
43        }
44      }
45
46      @Override
47      public void keres() {
48        while ( !nyiltak.isEmpty() ) {
49          kiirAdatbazis();
50          AalgoritmusCsucs aktualis = (AalgoritmusCsucs)nyiltak.getFirst();
51          if ( aktualis.getAllapot().celAllapot() ) {
52            terminalisok.add( aktualis );
53            if ( osszesMegoldas ) {
54              zartak.add( nyiltak.removeFirst() );
55              continue;
56            }
57            else
58              break;
59          }
60          zartak.add( nyiltak.removeFirst() );
61          kiterjeszt( aktualis );
62          Collections.sort( nyiltak, new Comparator<Csucs>() {
63            @Override
64            public int compare( Csucs cs1, Csucs cs2 ) {
65              AalgoritmusCsucs acs1 = ( AalgoritmusCsucs )cs1,
66                               acs2 = ( AalgoritmusCsucs )cs2;
67              return new Double( acs1.heurisztika + acs1.koltseg ).compareTo(
68                     new Double( acs2.heurisztika + acs2.koltseg ) );
69            }
70          } );
71        }
72      }
73    }
```

Az összes keresőgráffal kereső algoritmusnál megadhatjuk a keresés feladatspecifikus jellemzőit, amelyek szemantikája a következő:

- `OSSZES_MEGOLDAS`: Szemben a visszalépéses keresővel és az „ág és korlát” algoritmussal, ezeknél a keresőknél csak azt tudjuk kérni ennek a jellemzőnek a beállításával,

hogy a keresők minden terminális csúcsot keressenek meg. Mivel a keresési stratégiák olyanok, hogy egy-egy állapotot csak egy `Csucs` típusú objektumban tárolnak az adatbázisaikban, velük együtt csak egyetlen alkalmazott operátor tárolására képesek (mégpedig az utoljára megjegyzett operátoréra). Emiatt állhat elő az a helyzet, hogy bár egy problémát adott esetben többféleképpen is meg lehet oldani, ezek a keresők a valóságosnál sokkal kevesebb (szélsőséges esetben akár csak egyetlen) megoldást jeleznek működésük befejezésekor.

- `MEGOLDAS_ALLAPOT`: Hasonlóan értelmezhető, mint a visszalépéses keresőél és az „ág és korlát” algoritmusnál.

A körfigyelés mint jellemző, a keresőgráffal keresőknél értelmetlen opció, hiszen ezek a keresők – külön kéréstől függetlenül – minden esetben ellenőrzik, hogy egy kiterjesztéskor generált állapot szerepel-e már az adatbázisban vagy sem.

Ezzel bemutattuk az általunk javasolt osztályhierarciákat. A következő fejezetben megtekinthetjük, hogy hogyan lehet a korábban ismertetett állapottér-reprezentációkhoz készített osztályokkal végrehajtani egy tetszőleges keresést.

### 3.2.3. Példák és tapasztalatok

Válasszuk ki először korábbi példáink közül a *Szabadesés* című problémát, és vegyük a hozzá készített osztályokat! Bármelyik keresőalgoritmussal is szeretnénk megoldani a feladatot, célszerű a feladat osztályait tartalmazó csomagban létrehozni azt az osztályt, amelyben példányosítani fogjuk majd a kívánt keresést. Ez a mi esetünkben így nézhet ki:

```
1  package szabadeses;
2
3  import allapotter.Allapot;
4  import kereso.Csucs;
5  import kereso.Kereso;
6  import kereso.keresografos.szisztematikus.SzelessegiKereso;
7
8  public class Main {
9    public static void main( String[] args ) {
10     Allapot kezdo = new Szabadeses();
11     Kereso k =
12       new SzelessegiKereso( kezdo );
13     k.keres();
14     for ( Csucs cs : k.getTerminalisok() ) {
15       System.out.println( "Egy megoldas" );
16       k.kiirMegoldas( cs );
17     }
18     System.out.println( "Megoldasok szama: " +
19                         k.getTerminalisok().size() );
20   }
21 }
```

Minden keresőalgoritmusnak van olyan konstruktora, amelyben be tudjuk állítani a keresés jellemzőit. Ha nem adunk meg a `SzelessegiKereso` objektum példányosításakor jellemzőket, akkor a keresőt olyan jellemzőkkel hozzuk létre, melyek szerint a keresést az első megoldás megtalálásáig folytatjuk, és a startcsúcstól a terminális csúcsig vezető teljes operátor- és állapotsorozatot meg kívánjuk jeleníteni a kimeneten. A keresést elindítva, az nagyon gyorsan véget ér, és kiírja a képernyőre a probléma egyik legrövidebb (legkevesebb lépésben végrehajtható) megoldását.

```
Egy megoldas

                2
            1 3 X X
          3 X X X X
    2 1 X X X X X X
X X X X X X X X X X
 (0)
Balra[2,7] =>
                2
          1   3 X X
          3 X X X X
    2 1 X X X X X X
X X X X X X X X X X
 (1)
Balra[2,6] =>
                2
        1     3 X X
          3 X X X X
    2 1 X X X X X X
X X X X X X X X X X
 (2)
Hullik[] =>
                2
              3 X X
        1 3 X X X X
    2 1 X X X X X X
X X X X X X X X X X
 (3)
Balra[3,5] =>
                2
              3 X X
      1   3 X X X X
    2 1 X X X X X X
X X X X X X X X X X
 (4)
Eltuntet[] =>
                2
              3 X X
          3 X X X X
    2   X X X X X X
X X X X X X X X X X
 (5)
Jobbra[4,3] =>
                2
              3 X X
          3 X X X X
      2 X X X X X X
X X X X X X X X X X
 (6)
Balra[1,9] =>
              2
              3 X X
          3 X X X X
      2 X X X X X X
X X X X X X X X X X
 (7)
Balra[1,8] =>
            2
              3 X X
          3 X X X X
      2 X X X X X X
X X X X X X X X X X
 (8)
Hullik[] =>

            2 3 X X
          3 X X X X
      2 X X X X X X
X X X X X X X X X X
 (9)
Balra[2,7] =>

          2   3 X X
          3 X X X X
      2 X X X X X X
X X X X X X X X X X
 (10)
Balra[2,6] =>

```

```
          2     3 X X
            3 X X X X
        2 X X X X X X
X X X X X X X X X X
 (11)
Hullik[] =>

                3 X X
          2 3 X X X X
        2 X X X X X X
X X X X X X X X X X
 (12)
Balra[3,5] =>

                3 X X
        2   3 X X X X
        2 X X X X X X
X X X X X X X X X X
 (13)
Eltuntet[] =>

                3 X X
            3 X X X X
          X X X X X X
```

```
X X X X X X X X X X
 (14)
Balra[2,8] =>

              3   X X
            3 X X X X
          X X X X X X
X X X X X X X X X X
 (15)
Balra[2,7] =>

            3     X X
            3 X X X X
          X X X X X X
X X X X X X X X X X
 (16)
Eltuntet[] =>

                  X X
              X X X X
          X X X X X X
X X X X X X X X X X
 (17)
Megoldasok szama: 1
```

Megpróbálhatnánk összegyűjteni a probléma összes megoldását is, azonban ez nagyon hamar nehézségekbe fog ütközni. A keresőgráfos keresőkkel ez garantáltan nem fog menni, hiszen náluk az `OSSZES_MEGOLDAS` jellemző csak a különböző állapotokat tartalmazó terminális csúcsokban végződő megoldásokat gyűjti össze (lásd az 57. oldalt). Marad tehát a visszalépéses kereső vagy az „ág és korlát” algoritmus. Mielőtt bármelyiküket is választanánk, gondoljuk végig, mit is jelent ennél a feladatnál az „összes megoldás” kifejezés. Mivel a feladat reprezentációs gráfja sok kört tartalmaz – a legkisebb, kétlépéses köröket például azon állapotok között, amelyek áttranszformálhatók egymásba egy balra-jobbra vagy egy jobbra-balra lépéssorozattal –, ezért mindenképpen csak olyan megoldások jöhetnek szóba, amelyek nem tartalmaznak köröket; a köröket tartalmazó megoldások száma végtelen. Emiatt a visszalépéses kereső alkalmazásával az összes körmentes megoldás megtalálására, míg az „ág és korlát” algoritmussal az összes körmentes optimális (legkevesebb lépésszámú) megoldás megtalálására van esélyünk. A következő kérdés, hogy az ezen tulajdonságokkal rendelkező megoldásokat vajon mennyi idő alatt képesek ezek az algoritmusok megtalálni? Nyilvánvaló, hogy az összes körmentes megoldás megtalálása legalább annyi időt vesz igénybe, mint az összes körmentes optimális megoldásé. Érdemes lenne tehát ez utóbbit megvalósítani, ami amúgy is egy ésszerű gondolat, hiszen ki akarja azzal tölteni az idejét, hogy ide-oda tologatja a dobozokat, ha anélkül is megoldható a feladat. Az „ág és korlát” algoritmust tehát a következőképpen paraméterezzük:

```
1  package szabadeses;
2
```

```
3  import allapotter.Allapot;
4  import kereso.Csucs;
5  import kereso.Kereso;
6  import kereso.backtrack.AgEsKorlatKereso;
7
8  public class Main {
9    public static void main( String[] args ) {
10     Allapot kezdo = new Szabadeses();
11     Kereso k = new AgEsKorlatKereso( kezdo,
12       AgEsKorlatKereso.OSSZES_MEGOLDAS | AgEsKorlatKereso.KOR_FIGYELES );
13     k.keres();
14     for ( Csucs cs : k.getTerminalisok() ) {
15       System.out.println( "Egy megoldas" );
16       k.kiirMegoldas( cs );
17     }
18     System.out.println( "Megoldasok szama: " +
19                         k.getTerminalisok().size() );
20   }
21 }
```

A program elindításakor azt tapasztaljuk, hogy a kereső viszonylag hamar megtalál egy nagy költségértékű (nagy mélységi számú) megoldást, és onnantól kezdve iszonyatosan lassan tudja csak csökkenteni a korlát értékét. Ez nem csoda, hiszen ilyen nagy mélységben minden operátor kipróbálása nagy méretű részfák bejárását teszi szükségessé, sokszor a siker minden reménye nélkül. Ezen a tényen még az sem változtat sokat, hogy az első megoldás megtalálása – mivel a kereső algoritmusának a kódjában a Java `HashSet` típusát alkalmaztuk – nagyban függ az operátorok számára a JVM által generált hashkódtól. Két környezetben is teszteltük a fentebb kódolt programot: az egyik helyen egy 47-es mélységű, a másikon egy 35-ös költségű megoldás megtalálásával indított, de az optimális megoldásokat 8 órányi kereséssel sem találta meg.

Tekintsük most a *Huszárcsere* problémához készített állapottér-reprezentációinkat! Ezen reprezentációk mindegyikét leprogramozva összehasonlíthatjuk, hogy az egyes keresőalgoritmusok mennyi idő alatt képesek megtalálni a probléma egyik vagy éppen összes megoldását. A 3.1. táblázatban láthatjuk a korábban ismertett három állapottér-reprezentációval végrehajtott keresések eredményeit. A táblázat oszlopai a következő adatokat tartalmazzák:

1. A reprezentáció sorszáma. A kidolgozott állapottér-reprezentációk a 5., 9. és 13. oldalakon kezdődnek.

2. Az állapotok száma a kezdőállapotból az operátorok alkalmazásával elérhető állapotok darabszámát mutatja. Gyakorlatilag ennyi csúcsa van a feldolgozandó reprezentációs gráfnak.

3. Az alkalmazható operátorok száma az egyes reprezentációkban.

4. Visszalépéses kereséssel az elsőként megtalált megoldás hossza. Az „n/a” felirat azt jelzi, hogy az adatot nem adta meg véges, belátható időn belül az algoritmus.

5. Visszalépéses kereséssel az első megoldás megtalálásának az ideje. Az „n/a” felirat azt jelzi, hogy az adatot nem adta meg véges, belátható időn belül az algoritmus.

6. Szélességi kereséssel az elsőként megtalált megoldás hossza.

7. Szélességi kereséssel az első megoldás megtalálásának az ideje.

| | Állapotok száma | Operátorok száma | Visszalépéses keresés | | Szélességi keresés | |
|---|---|---|---|---|---|---|
| | | | Megoldás hossza | Keresési idő | Megoldás hossza | Keresési idő |
| 1. | 280 | 32 | 108 | 21 msec | 16 | 51 msec |
| 2. | 280 | 16 | n/a | n/a | 16 | 45 msec |
| 3. | 117 | 8 | 16 | 1 msec | 16 | 14 msec |

3.1. táblázat. Statisztika a *Huszárcsere* probléma reprezentációihoz

A 3.1. táblázatban azt láthatjuk, hogy a [8] könyvben is bemutatott ötletet megvalósító állapottér-reprezentációhoz a másik két reprezentációnál több mint 50%-kal kisebb méretű reprezentációs gráf tartozik. Ha megvizsgáljuk ezt a 117 csúcsot tartalmazó gráfot, akkor ráadásképpen azt is láthatjuk, hogy nem tartalmaz köröket (a huszárok csak az óramutató járásával egyező irányban közlekedhetnek, tehát az ideoda ugrálást magának a reprezentációnak a szabályrendszere zárja ki).[1] E két tényező együttesen eredményezi azt, hogy mindkét vizsgált keresővel ez a harmadik reprezentáció működik a leggyorsabban. Ebből azt a következtetést vonhatjuk le, hogy a probléma reprezentálása során hasznos lehet, ha a lehető legkisebb gráfot eredményező reprezentációt próbálunk készíteni, esetleg olyan, a probléma eredeti kiírásában nem szereplő megszorítások bevezetésével, amelyek nem vezetnek megoldások elvesztéséhez a feladat megoldása során. Szintén gyorsíthatja a keresést, ha a reprezentációs gráfunk nem tartalmaz köröket, illetve ha kevés olyan csúcsot tartalmaz, amely a startcsúcsból több útvonalon keresztül is elérhető.

## 3.3. Oktatási tapasztalatok

Tanulságképpen megfogalmazhatjuk, hogy egy probléma megoldásainak sikeres meghatározásához több lépésen keresztül vezet az út, ráadásul úgy, hogy egyik-másik lépést, ha nyilvánvalóan zsákutcába jutottunk, többször is meg kell ismételni:

1. a problémához okosan, több megközelítés szerint is készíteni kell állapottér-reprezentációkat (nem tudhatjuk előre, hogy az egyszerű és kevés komponens használata, vagy a bonyolult operátor-alkalmazási előfeltételek hozzák-e majd meg a kívánt eredményt);

[1] Így persze az eredeti feladat összes megoldásának a felét nem is fogja explicit módon meghatározni ezzel a reprezentációval egyik keresőalgoritmus sem, de a kapott megoldásokból tükrözéssel vissanyerhetők a hiányzó megoldások.

2. a kiválasztott állapottér-reprezentációt jól, pontosan kell kódolni (a debuggolás nem kis időt vehet el a tesztelésből, ha nem működik megbízhatóan a programunk);

3. gondosan fel kell mérni a kiválasztott állapottér-reprezentációhoz tartozó reprezentációs gráf tulajdonságait (eszerint fogjuk megadni a keresők jellemzőit, sőt, olykor-olykor ezek miatt a tulajdonságok miatt egyes keresőket akár egy kicsit módosítani is érdemes, új leszármazott osztály generálásával a gyorsabb futás érdekében);

4. a reprezentációs gráf mérete és szerkezete alapján egy ahhoz illeszkedő keresőalgoritmust kell választanunk, és megfelelően felparaméterezve lefuttatnunk.

A felsorolt pontok sikeres végrehajtásához elengedhetetlenül szükséges több korábbi tantárgyi ismeret szintézise és integrálása. A kellő rutint csak ezek együttes alkalmazásával képesek hallgatóink megszerezni. Az elmúlt három évben már egységesen a dolgozatban felvázolt koncepció tudatos alkalmazásával szerveztük meg *A mesterséges intelligencia alapjai* című tárgy gyakorlati foglalkozásait. Természetesen az itt felsoroltaknál jóval több példán keresztül mutattuk be az egyes technikák (állapottér-reprezentáció, majd objektumorientált programtervezés) módszereit és lépéseit. A gyakorlatok ezzel sokkal látványosabbá és lendületesebbé váltak, és hallgatóink is érdekesebbnek találták a foglalkozásokat.[2]

A 2007/2008-as tanév eredményei a koncepció helyességét igazolták, a gyakorlati aláírások megszerzése a hallgatók 74%-ának (135 hallgatónak a 181-ből) sikerült, ráadásul a gyakorlati aláírást megszerzők a vizsgán sem szerepeltek rosszul: 95% felett volt a tárgyat teljesítők aránya.[3]

A 2008/2009-es tanévben egyszerűsödtek a tárgy felvételének a követelményei, nem volt szükséges mind a négy alapozó tárgy (lásd a 2. oldalon) teljesítésének a megléte hozzá. Ez az eredményeken is meglátszódott: gyakorlati aláírást mindössze 110 hallgató szerzett a 186-ból, ez az összes hallgató 59%-a. Sajnos ebben az esetben a követelmények enyhítése nem feltétlenül szolgálta a hallgatók érdekeit, mert így azokat az ismereteket, amelyek hiányoztak az előtanulmányaikból, is most, a kurzus teljesítésével párhuzamosan kellett megpróbálniuk elsajátítani.

Meglepő módon volt még egy képzési terület, ahol a bevezetésre javasolt koncepció látványosan jó eredményeket hozott: ez a terület a levelező tagozatos informatikus hallgatók képzése volt. Mivel a számonkérések típusa megegyezett a nappali tagozatosokéval, ezért a továbbjutási arányt is nyugodtan össze lehet vetni az övékével. A gyakorlati aláírásért zajló számonkéréseken 36 hallgató jelent meg, közülük 19 szerzett aláírást, 7 hallgatónak az állapottér-reprezentáció elkészítése, 10-nek pedig a reprezentáció kódolása okozott gondot. A továbbjutási arány így itt 52%, ami sokkal jobb arány a 2007/2008-as tanévinél, különösen annak fényében, hogy a korábbi félévekben a gyakorlati aláírás megszerzése technikailag is sokkal jobban elhúzódott, mivel teljesen mások voltak a gyakorlati aláírás megszerzésének a feltételei.

Konklúzióként azt vonhatjuk le az eddig elmondottakból, hogy a jó eredmények eléréséhez, a később is hasznosítható tudás megszerzéséhez és begyakorlásához ennél a

[2] Mindenképpen megjegyzendő, hogy a tárgy foglalkozásain minden tanévben kínáltunk és kértünk olyan önállóan megoldandó feladatokat, amelyekkel a hallgatók tesztelhették felkészültségüket és kipróbálhatták tudásukat már a félév végi számonkérések előtt is.

[3] Az adatok a Debreceni Egyetem Neptun tanulmányi rendszeréből származnak.

tárgynál is – mint persze sok másiknál is – nagyon fontosak az alapos előtanulmányok: a programozói rutin és a megbízható elméleti (logikai stb.) tudás.

4. FEJEZET

# Kétszemélyes stratégiai játékok lépésajánló algoritmusai

## 4.1. A vizsgált játékok tulajdonságai

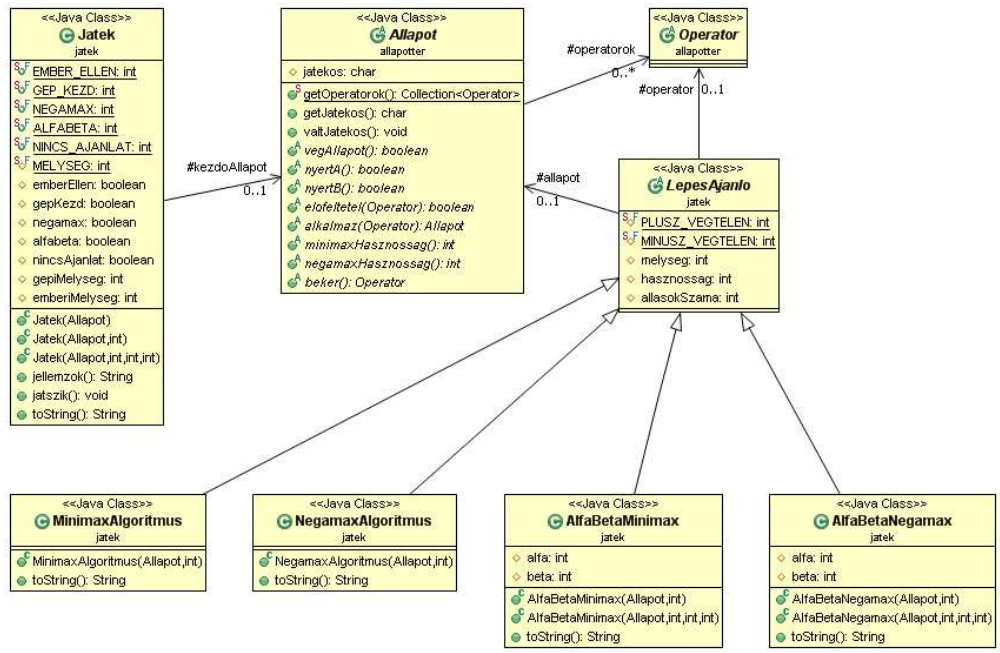
Az olyan játékokat, ahol a játékosoknak a játék kimenetelére nincs befolyásuk, *szerencsejátékoknak* nevezzük (rulett, kockajátékok). Ezeknek a játékoknak a tanulmányozása alapozta meg a 17. században a valószínűségszámítás tudományát. Ezzel szemben a *stratégiai játékokban* (sakk, bridzs) a játék kimenetelét a játékosok maguk is aktívan befolyásolják. A mesterséges intelligencia – többek között – az ilyen játékokkal is foglalkozik. A stratégiai játékokat a következő szempontok szerint osztályozhatjuk:

- a résztvevők száma szerint beszélhetünk *két-*, *három-*, ..., *n-személyes* játékokról;
- a játék hosszát tekintve vannak *véges* játékok, amelyekben minden játékos véges sok lépéslehetőség közül választhat, és minden játék véges sok lépés után véget ér, illetve vannak *végtelen* játékok, amelyek nem végesek;
- a játékosok össznyereségét és -veszteségét tekintve megkülönböztetünk *zérusösszegű* és *nem zérusösszegű* játékokat, előbbiekben a játékosok nyereségeinek és veszteségeinek az összege nulla;
- amennyiben a játék lefolyásában a véletlennek is van szerepe, akkor *sztochasztikus*, egyébként *determinisztikus* játékról beszélünk;
- ha a játék folyamán a játékosok a játékkal kapcsolatos minden információnak a birtokában vannak, akkor a játék *teljes információjú*, máskülönben nem az.

*A mesterséges intelligencia alapjai* tárgy keretein belül csak teljes információjú, kétszemélyes, véges, determinisztikus, zérusösszegű stratégiai játékokkal foglalkozunk. A gyakorlatokon először állapottér-reprezentálunk néhány ilyen játékot, amelyeket aztán Java programozási nyelven implementálunk, majd ki is próbálunk. Az implementálást segítendő kifejlesztettünk egy olyan Java csomagot, amely ügyesen felparaméterezve lehetőséget biztosít tetszőleges játék tetszőleges lépésajánló algoritmussal történő lebonyolításához.

## 4.2. Játékok objektumorientált szemléletben

A következőkben bemutatjuk a gyakorlatainkon használt osztályhierarchiát. Az általunk javasolt Java csomagok az állapottér-reprezentációval és a lépésajánló algoritmusokkal kapcsolatos osztályokat tartalmazza (4.1. ábra).




4.1. ábra. A lépésajánló algoritmusok osztálydiagramja

Az `allapotter` csomagban lévő `Operator` osztály az egyes játékokhoz tartozó operátorok, míg az `Allapot` osztály a játékokhoz tartozó állapotok absztrakt ősosztálya.

```
1  package allapotter;
2
3  public abstract class Operator
4  {
5  }
```

Mivel egy állapot jellemzői mindig a konkrét játéktól függnek, ezért ez utóbbi osztályban mindössze két adattagot definiáltunk. Az egyik a játék állapotaira alkalmazható összes operátor halmaza, a másik pedig az aktuális állapotban lépni következő játékos. A konkrét játéktól függetlenül minden állapot el tudja dönteni, hogy végállapot-e, és ha igen, akkor milyen eredménnyel ért véget a játék (melyik játékos nyert, ha nem döntetlennel végződött a játszma). Szintén minden állapotnál vizsgálható, hogy alkalmazható-e rá egy adott operátor, illetve megadható, hogy egy operátor alkalmazásával milyen új állapot jön létre belőle. A `beker` absztrakt metódus implementálásával valósíthatjuk meg az emberi játékos lépéseinek a beolvasását.

```
1  package allapotter;
2
3  import java.util.Collection;
4
5  public abstract class Allapot
6  {
7    protected static Collection<Operator> operatorok;
8    protected char jatekos;
9
10   public static Collection<Operator> getOperatorok() {
11     return operatorok;
12   }
13
14   public char getJatekos() {
15     return jatekos;
16   }
17
18   public void valtJatekos() {
19     jatekos = jatekos == 'A' ? 'B' : 'A';
20   }
21
22   public abstract boolean vegAllapot();
23   public abstract boolean nyertA();
24   public abstract boolean nyertB();
25   public abstract boolean elofeltetel( Operator op );
26   public abstract Allapot alkalmaz( Operator op );
27   public abstract int minimaxHasznossag();
28   public abstract int negamaxHasznossag();
29   public abstract Operator beker();
30 }
```

A játékokhoz kötődő algoritmusokat megvalósító osztályokat a `jatek` csomagban helyeztük el. A lépésajánló algoritmusok ősosztálya az absztrakt `LepesAjanlo` osztály, amelynek az attribútumai tárolják azt az *állapotot*, amelyhez a legjobb lépést szeretnénk meghatározni, a lépéskeresés *mélységét*, a keresés végrehajtását követően javasolt *operátort*, az ezzel az operátorral az adott mélységig előretekintve elérhető legjobb állapot *hasznosságát*, illetve a keresés során kiértékelt állapotok *darabszámát*. Ezeket a jellemzőket minden leszármazott osztályban kiszámítják a konkrét lépésajánló algoritmusok, melyeket a kiértékelt állapotok számának a segítségével tudunk majd összehasonlítani egymással.

```
1  package jatek;
2
3  import allapotter.*;
4
5  public class Jatek
6  {
7    public static final int EMBER_ELLEN = 1;
8    public static final int GEP_KEZD = 2;
```

```
 9     public static final int NEGAMAX = 4;
10     public static final int ALFABETA = 8;
11     public static final int NINCS_AJANLAT = 16;
12     protected static final int MELYSEG = 5;
13
14     protected boolean emberEllen;
15     protected boolean gepKezd;
16     protected boolean negamax;
17     protected boolean alfabeta;
18     protected boolean nincsAjanlat;
19     protected int gepiMelyseg;
20     protected int emberiMelyseg;
21
22     protected Allapot kezdoAllapot;
23
24     public Jatek( Allapot kezdoAllapot ) {
25       this( kezdoAllapot, 0, MELYSEG, MELYSEG );
26     }
27
28     public Jatek( Allapot kezdoAllapot, int jellemzok ) {
29       this( kezdoAllapot, jellemzok, MELYSEG, MELYSEG );
30     }
31
32     public Jatek( Allapot kezdoAllapot, int jellemzok,
33                   int gepiMelyseg, int emberiMelyseg ) {
34       this.kezdoAllapot = kezdoAllapot;
35       emberEllen = ( jellemzok & EMBER_ELLEN ) != 0;
36       gepKezd = ( jellemzok & GEP_KEZD ) != 0;
37       negamax = ( jellemzok & NEGAMAX ) != 0;
38       alfabeta = ( jellemzok & ALFABETA ) != 0;
39       nincsAjanlat = ( jellemzok & NINCS_AJANLAT ) != 0;
40       this.gepiMelyseg = gepiMelyseg;
41       this.emberiMelyseg = emberiMelyseg;
42     }
43
44     public void jatszik() {
45       Allapot allapot = kezdoAllapot;
46       LepesAjanlo lepesAjanlo;
47       while ( !allapot.vegAllapot() ) {
48         System.out.println( "Aktualis allapot: " + allapot );
49         if ( allapot.vegAllapot() )
50             break;
51         Operator op;
52         if ( !emberEllen && ( allapot.getJatekos() == 'A' && gepKezd ||
53                               allapot.getJatekos() == 'B' && !gepKezd ) ) {
54           if ( !negamax && !alfabeta )
55             lepesAjanlo = new MinimaxAlgoritmus( allapot, gepiMelyseg );
56           else if ( negamax && !alfabeta )
57             lepesAjanlo = new NegamaxAlgoritmus( allapot, gepiMelyseg );
```

```
58          else if ( !negamax && alfabeta )
59            lepesAjanlo = new AlfaBetaMinimax( allapot, gepiMelyseg );
60          else
61            lepesAjanlo = new AlfaBetaNegamax( allapot, gepiMelyseg );
62          op = lepesAjanlo.operator;
63          System.out.println( "A gep lepese: " + op +
64                              System.getProperty( "line.separator" ) );
65        }
66        else
67        {
68          if ( !nincsAjanlat ) {
69            if ( !negamax && !alfabeta )
70              lepesAjanlo =
71                        new MinimaxAlgoritmus( allapot, emberiMelyseg );
72            else if ( negamax && !alfabeta )
73              lepesAjanlo =
74                        new NegamaxAlgoritmus( allapot, emberiMelyseg );
75            else if ( !negamax && alfabeta )
76              lepesAjanlo =
77                        new AlfaBetaMinimax( allapot, emberiMelyseg );
78            else
79              lepesAjanlo =
80                        new AlfaBetaNegamax( allapot, emberiMelyseg );
81            op = lepesAjanlo.operator;
82            System.out.println( "Ajanlott lepes: " + op );
83          }
84          System.out.println( System.getProperty( "line.separator" )+ "'" +
85                              allapot.getJatekos() + "' jatekos lepese:" +
86                              System.getProperty( "line.separator" ) );
87          op = allapot.beker();
88        }
89        allapot = allapot.alkalmaz( op );
90      }
91      System.out.print( "A jateknak vege. " );
92      if ( allapot.nyertA() || allapot.nyertB() )
93        System.out.println( "A jatekot '" +
94                            ( allapot.nyertA() ? 'A' : 'B' ) + "' nyerte." );
95      else
96        System.out.println( "Az eredmeny dontetlen." );
97    }
98  }
```

A játék vezérlését a `Jatek` osztályban definiált `jatszik` metódus valósítja meg. Ez a metódus egy ciklust futtat mindaddig, míg az aktuális állapot példány végállapotba nem kerül. Mindeközben lépésenként megjeleníti az aktuális állást, és eldönti, hogy ki következik lépni. Ha interaktívan játszunk, és az emberi játékos lép, akkor beolvassa a játékos lépését, egyébként példányosít egy lépésajánlót, és annak segítségével meghatározza, hogy a gép mit lépjen. A lépés ismeretében mindkét esetben alkalmazza a lépésnek megfelelő operátort, és frissíti az aktuális állapotot.

A konkrét lépésajánló algoritmusok a legjobb lépés keresését rekurzívan, újabb és újabb lépésajánló algoritmusok példányosításával végzik el. Emiatt a keresések megvalósításához nem volt szükségünk más metódusok használatára a konstruktorokon kívül.

A következőkben végigböngészhetjük a négy lépésajánló algoritmus kódját, kezdve a minimax algoritmussal. (Az osztályok `toString` metódusait az egyszerűség és a jobb átláthatóság kedvéért elhagytuk a kódokból.)

```
1  package jatek;
2
3  import allapotter.*;
4
5  public class MinimaxAlgoritmus extends LepesAjanlo
6  {
7    public MinimaxAlgoritmus( Allapot allapot, int melyseg ) {
8      this.allapot = allapot;
9      this.melyseg = melyseg;
10     allasokSzama = 1;
11     if ( allapot.vegAllapot() || melyseg == 0 )
12       hasznossag = allapot.minimaxHasznossag();
13     else if ( allapot.getJatekos() == 'A' ) {
14       hasznossag = MINUSZ_VEGTELEN;
15       for ( Operator op : Allapot.getOperatorok() )
16         if ( allapot.elofeltetel( op ) ) {
17           Allapot uj = allapot.alkalmaz( op );
18           MinimaxAlgoritmus minimax =
19                              new MinimaxAlgoritmus( uj, melyseg - 1 );
20           if ( minimax.hasznossag > hasznossag ) {
21             hasznossag = minimax.hasznossag;
22             operator = op;
23           }
24           allasokSzama += minimax.allasokSzama;
25         }
26     }
27     else {
28       hasznossag = PLUSZ_VEGTELEN;
29       for ( Operator op : Allapot.getOperatorok() )
30         if ( allapot.elofeltetel( op ) ) {
31           Allapot uj = allapot.alkalmaz( op );
32           MinimaxAlgoritmus minimax =
33                              new MinimaxAlgoritmus( uj, melyseg - 1 );
34           if ( minimax.hasznossag < hasznossag ) {
35             hasznossag = minimax.hasznossag;
36             operator = op;
37           }
38           allasokSzama += minimax.allasokSzama;
39         }
40     }
41   }
42 }
```

Negamax algoritmus használatakor nem kell vizsgálnunk a lépő játékos személyét, igaz, más a kiértékelő függvényünk is:

```
1  package jatek;
2
3  import allapotter.*;
4
5  public class NegamaxAlgoritmus extends LepesAjanlo
6  {
7    public NegamaxAlgoritmus( Allapot allapot, int melyseg ) {
8      this.allapot = allapot;
9      this.melyseg = melyseg;
10     allasokSzama = 1;
11     if ( allapot.vegAllapot() || melyseg == 0 )
12       hasznossag = allapot.negamaxHasznossag();
13     else {
14       hasznossag = MINUSZ_VEGTELEN;
15       for ( Operator op : Allapot.getOperatorok() )
16         if ( allapot.elofeltetel( op ) ) {
17           Allapot uj = allapot.alkalmaz( op );
18           NegamaxAlgoritmus negamax =
19                              new NegamaxAlgoritmus( uj, melyseg - 1 );
20           if ( -negamax.hasznossag > hasznossag ) {
21             hasznossag = -negamax.hasznossag;
22             operator = op;
23           }
24           allasokSzama += negamax.allasokSzama;
25         }
26     }
27   }
28 }
```

Az alfabéta vágást végző algoritmusok osztályait kiegészítettük két olyan adattaggal, amelyek a játékfa egyes csomópontjaihoz tartozó aktuális *alfa* és *béta* értékeket tárolják. Lássuk, hogy néz ki a minimax algoritmus az alfa-béta vágással:

```
1  package jatek;
2
3  import allapotter.*;
4
5  public class AlfaBetaMinimax extends LepesAjanlo
6  {
7    protected int alfa;
8    protected int beta;
9
10   public AlfaBetaMinimax( Allapot allapot, int melyseg ) {
11     this( allapot, melyseg, MINUSZ_VEGTELEN, PLUSZ_VEGTELEN );
12   }
13
```

```
14    public AlfaBetaMinimax( Allapot allapot, int melyseg,
15                            int alfa, int beta ) {
16      this.allapot = allapot;
17      this.melyseg = melyseg;
18      this.alfa = alfa;
19      this.beta = beta;
20      allasokSzama = 1;
21      if ( allapot.vegAllapot() || melyseg == 0 )
22        hasznossag = allapot.minimaxHasznossag();
23      else if ( allapot.getJatekos() == 'A' ) {
24        for ( Operator op : Allapot.getOperatorok() ) {
25          if ( alfa >= beta )
26            break;
27          if ( allapot.elofeltetel( op ) ) {
28            Allapot uj = allapot.alkalmaz( op );
29            AlfaBetaMinimax abMinimax =
30                        new AlfaBetaMinimax( uj, melyseg - 1, alfa, beta );
31            if ( abMinimax.hasznossag > alfa ) {
32              this.alfa = alfa = abMinimax.hasznossag;
33              operator = op;
34            }
35            allasokSzama += abMinimax.allasokSzama;
36          }
37        }
38        hasznossag = alfa < beta ? alfa : beta;
39      }
40      else {
41        for ( Operator op : Allapot.getOperatorok() ) {
42          if ( alfa >= beta )
43            break;
44          if ( allapot.elofeltetel( op ) ) {
45            Allapot uj = allapot.alkalmaz( op );
46            AlfaBetaMinimax abMinimax =
47                        new AlfaBetaMinimax( uj, melyseg - 1, alfa, beta );
48            if ( abMinimax.hasznossag < beta ) {
49              this.beta = beta = abMinimax.hasznossag;
50              operator = op;
51            }
52            allasokSzama += abMinimax.allasokSzama;
53          }
54        }
55        hasznossag = alfa < beta ? beta : alfa;
56      }
57    }
58  }
```

Végül pedig következzék a nexamax algoritmusnak – a témakör népszerűsítő irodalmában ritkábban tárgyalt, de azért mégsem példa nélküli (lásd pl. [11]) – alfa-béta vágással kiegészített változata:

```
1  package jatek;
2
3  import allapotter.*;
4
5  public class AlfaBetaNegamax extends LepesAjanlo
6  {
7    protected int alfa;
8    protected int beta;
9
10   public AlfaBetaNegamax( Allapot allapot, int melyseg )
11   {
12     this( allapot, melyseg, MINUSZ_VEGTELEN, PLUSZ_VEGTELEN );
13   }
14
15   public AlfaBetaNegamax( Allapot allapot, int melyseg,
16                           int alfa, int beta ) {
17     this.allapot = allapot;
18     this.melyseg = melyseg;
19     this.alfa = alfa;
20     this.beta = beta;
21     allasokSzama = 1;
22     if ( allapot.vegAllapot() || melyseg == 0 )
23       hasznossag = allapot.negamaxHasznossag();
24     else {
25       for ( Operator op : Allapot.getOperatorok() ) {
26         if ( alfa >= beta )
27           break;
28         if ( allapot.elofeltetel( op ) ) {
29           Allapot uj = allapot.alkalmaz( op );
30           AlfaBetaNegamax abNegamax =
31                     new AlfaBetaNegamax( uj, melyseg - 1, -beta, -alfa );
32           if ( -abNegamax.hasznossag > alfa ) {
33             this.alfa = alfa = -abNegamax.hasznossag;
34             operator = op;
35           }
36           allasokSzama += abNegamax.allasokSzama;
37         }
38       }
39       hasznossag = alfa;
40     }
41   }
42 }
```

A játékok és a lépésajánló algoritmusok testreszabását a játék példányosításakor végezhetjük el. A játék jellemzőjeként állíthatjuk be azt, hogy ember ember ellen, vagy ember gép ellen játsszon-e. Utóbbi esetben azt is megadhatjuk, hogy ki kezdje a játékot. Eldönthetjük, hogy a lépésajánlást minimax vagy negamax algoritmussal végezze-e a program, illetve hogy használjanak-e alfabéta vágást az algoritmusok. Az emberi játékos számára is kérhetünk lépésajánlatot, amelyet a gépi lépésajánlathoz hasonlóan

egy lépésajánló példányosításával határoz meg a program. Ezt az ajánlatot aztán az emberi játékos vagy elfogadja, vagy elutasítja. Ráadásul külön szabályozhatjuk azt, hogy milyen mélységben tekintsen előre a program a gép és az ember lépésajánlatának a kiszámításakor.

A lépésajánlatok jóságát (erősségét) két tényező befolyásolja. Az egyik az előretekintés mélysége, a másik pedig az egyes állapotokra alkalmazott kiértékelő függvény megbízhatósága. Látni kell, hogy a játékfa elágazási tényezőjétől függően az előretekintés mélységének a növelésével a lépésajánlat ideje exponenciálisan nőhet. Érdemes tehát minél jobb kiértékelő függvényt írni a konkrét játék osztályában. Amennyiben olyan kiértékelő függvényt tudunk írni, amely a játék minden állapotáról biztosan el tudja dönteni, hogy az mennyire jó az egyes játékosok számára, akkor az előretekintés mélységét nem szükséges 1-nél nagyobb értékre állítani ([2]). Az éremnek persze két oldala van: egy-egy állapot kiértékelése is hosszú időt vehet igénybe, minimalizálni tehát valójában a kiértékelt állapotok számának és az egyes állapotok kiértékelésére fordított időnek a szorzatát kell.

A `jatek` csomag algoritmusainak használatát *A mesterséges intelligencia* tárgy gyakorlatain néhány egyszerű játék reprezentációján és Java nyelvű implementációján keresztül mutatjuk be hallgatóinknak (Nim, Bachet-féle játékok). Most egy ezekre épülő, a 2002-es ACM programozói verseny Varsóban megrendezett közép-európai selejtezőjén kiadott feladat kapcsán vizsgáljuk meg, hogy mennyire alkalmasak ezek az algoritmusok ilyen típusú feladatok megoldására. Látni fogjuk, hogy egy egyszerű kiértékelő függvénnyel csak viszonylag kevés lépésre előretekintve működőképesek, de a segítségükkel kinyerhető adatok már megalapozhatják a feladat későbbi sikeres megoldását és lehetővé teszik a más algoritmusokkal kapott eredmények ellenőrzését. A feladatnak ugyanis, ahogyan azt egy versenyfeladattól elvárhatjuk, nem ez lesz a legjobb módja, és ezért jól lehet majd látni az MI által favorizált módszerek és az egyéb algoritmikus feladatmegoldó technikák közötti különbségeket.

## 4.3. Példa – *Falánk Steve*

Steve és Digit vettek egy fánkokkal teli dobozt. Hogy felosszák maguk között a fánkokat, egy általuk kitalált speciális játékot játszanak. A játékosok felváltva vesznek ki legalább egy fánkot a dobozból, de nem többet, mint egy adott egész szám. A játékosok a fánkjaikat maguk előtt gyűjtik. Az a játékos, amelyik kiüríti a dobozt, megeszi az összegyűjtött fánkjait, míg a másik a sajátjait visszarakja a dobozba, és a játékot a „vesztes” játékos (aki visszarakta a fánkjait) folytatja. A játék addig tart, amíg az összes fánkot meg nem eszik. A játékosok célja, hogy minél több fánkot egyenek meg. Hány fánkra számíthat Steve, aki a játékot kezdi, feltéve, hogy mindkét játékos a legjobb stratégiája szerint játszik?

### Feladat

Írj programot, amely

- beolvassa a standard inputról a játék paramétereit,

- kiszámolja, hogy hány fánkra számíthat Steve,
- kiírja az eredményt a standard outputra.

## Input

A bemenet első és egyetlen sora pontosan két egész számot tartalmaz ($n$ és $m$), amelyeket egyetlen szóköz választ el egymástól, $1 \leq m \leq n \leq 100$. Ezek a játék paraméterei, ahol $n$ a játék kezdetekor a dobozban lévő fánkok száma, $m$ pedig az egy játékos által egy lépésben kivehető fánkok számának a felső korlátja.

## Output

A kimenetnek pontosan egy egész számot kell tartalmaznia, amely azon fánkok száma, amennyire Steve számíthat.

## Példa input

```
5 2
```

## Példa output

```
3
```

### 4.3.1. A játék állapottér-reprezentációja

De hogyan is használhatjuk fel ennek a feladatnak a megoldásában a lépésajánló algoritmusokat? Az első ötletünk az lehet, hogy segítségükkel egészen a játékfa levélelemeiig előretekintve megnézzük, hogy az egyes végállásokban mennyi fánkot tud Steve összegyűjteni. Ehhez persze a kiértékelő függvényünket úgy kell megírni, hogy az csak és kizárólag ezt az egyetlen tényt vegye figyelembe, nevezetesen azt, hogy kinek van több fánkja a játék befejeződésekor. Mivel a teljes játékfát kiértékeljük, az eljárás által javasolt lépésnél semmiképpen sem lesz „jobb” lépés, már ami Steve számára a játék végkimenetelét illeti.

Ahhoz, hogy a lépésajánló algoritmusok a játékfát teljes magasságában kiértékelhessék, két lehetőség közül választhatunk:

1. vagy átírjuk egy kicsit a kódokat, és kivesszük a rekurziókat megállító `if` utasításokból a lépéskeresés mélységét vizsgáló

   ```
   melyseg == 0
   ```

   logikai kifejezéseket;

2. vagy a játék paramétereitől függően kellően nagy értékre (pl. $n^2$-re) állítjuk az előretekintés mélységét, mivel nem szabad elfelejtkeznünk arról, hogy a fánkok újraosztását mindig újra kell kezdenünk ugyanazon játszmán belül, ha valakinek éppen sikerült néhány fánkot megennie.

Mindkét esetben módosítani kell a programban (a `jatszik` metódusban) azt is, hogy a játszmát a program automatikusan játssza le, hiszen a jelenlegi változatban az egyik játékos mindenképpen az ember.

Nem remélhetjük viszont, hogy nagy input adatok mellett is gyors lesz ez a fajta játékfa-generálás, hiszen nagy $n$ érték esetén a teljes játékfa magassága, nagy $m$ érték esetén pedig ráadásul a szélessége is nagyon nagyra nő. Kiindulásnak viszont mindenképpen megteszi, ezért lássuk egy lehetséges reprezentációját!

A játék egyes állapotainak fontos jellemzője, hogy hány darab fánk található éppen a dobozban, Steve és Digit előtt, hányat evett már meg Steve és Digit, valamint hogy ki a soron következő játékos. Definiáljuk tehát ennek megfelelően ilyen sorrendben alaphalmazainkat:

$$\begin{aligned} H_{doboz} &= \{\, 0,1,2,3,\ldots,n \,\}, \\ H_{D\_előtt} &= \{\, 0,1,2,3,\ldots,n \,\}, \\ H_{S\_előtt} &= \{\, 0,1,2,3,\ldots,n \,\}, \\ H_{Digit} &= \{\, 0,1,2,3,\ldots,n \,\}, \\ H_{Steve} &= \{\, 0,1,2,3,\ldots,n \,\}, \\ J &= \{\, D,S \,\}. \end{aligned}$$

A fenti halmazok Descartes-szorzatából azok a

$$\begin{aligned} h = (h_{doboz}, h_{D\_előtt}, h_{S\_előtt}, h_{Digit}, h_{Steve}, j) \in \\ \in H_{doboz} \times H_{D\_előtt} \times H_{S\_előtt} \times H_{Digit} \times H_{Steve} \end{aligned}$$

elemhatosok lesznek a játék állapotai, melyekre teljesül a következő kényszerfeltétel: a dobozban lévő, a két játékos előtt lévő, valamint a két játékos által megevett fánkok száma pontosan megegyezik a játék paramétereként megadott $n$ számmal, azaz

$$h_{doboz} + h_{D\_előtt} + h_{S\_előtt} + h_{Digit} + h_{Steve} = n. \tag{4.1}$$

Állapotterünk tehát így definiálható:

$$\begin{aligned} \mathcal{A} = \{\, h \mid h = (h_{doboz}, h_{D\_előtt}, h_{S\_előtt}, h_{Digit}, h_{Steve}, j) \;\wedge\; \textit{kényszerfeltétel}(h) \,\} \supset \\ \subset H_{doboz} \times H_{D\_előtt} \times H_{S\_előtt} \times H_{Digit} \times H_{Steve}, \end{aligned}$$

ahol

$$\textit{kényszerfeltétel}(h) \equiv (4.1).$$

Kezdőállapotunk az lesz, amikor még egyik játékos sem nyúlt a fánkokhoz, és Steve következik lépni:

$$\textit{kezdő} = (n,0,0,0,0,\mathrm{S}) \in \mathcal{A}.$$

A játék végállásainak a halmazát azok az állapotok alkotják, amelyekben már minden fánkot megettek a játékosok:

$$\mathcal{V} = \{\, h \mid h = (0,0,0,h_{Digit}, h_{Steve}, j) \,\} \subset \mathcal{A}.$$

Egyetlen operátorunk az

$$\text{Elvesz}(h, c)\colon\ \text{dom}(\text{Elvesz}) \to \mathcal{A}$$

operátor lesz, ahol

$$\text{dom}(\text{Elvesz}) \subset \mathcal{A} \times \{\, 1, 2, 3, \ldots, n \,\}.$$

Az operátor második paramétere az a szám lesz, amely azt jelzi, hogy az éppen soron következő játékos hány darab fánkot szeretne elvenni a dobozból.

Egy $\text{Elvesz}(h, c)$ operátor akkor alkalmazható, ha $c$ értéke nem nagyobb $m$-nél, és van még a dobozban legalább $c$ darab fánk, azaz

$$\textit{előfeltétel}(\text{Elvesz}(h, c)) \equiv c > 0 \ \wedge\ c \leq m \ \wedge\ h_{doboz} \geq c.$$

Az $\text{Elvesz}(h, c)$ operátor alkalmazásának a hatását a következőképpen definiálhatjuk:

$$h'_{\text{doboz}} = \begin{cases} h_{\text{D\_előtt}}, & \text{ha } j = \text{S} \ \wedge\ h_{\text{doboz}} - c = 0, \\ h_{\text{S\_előtt}}, & \text{ha } j = \text{D} \ \wedge\ h_{\text{doboz}} - c = 0, \\ h_{\text{doboz}} - c, & \text{egyébként;} \end{cases}$$

$$h'_{\text{S\_előtt}} = \begin{cases} h_{\text{S\_előtt}} + c, & \text{ha } j = \text{S} \ \wedge\ h_{\text{doboz}} - c > 0, \\ 0, & \text{ha } h_{\text{doboz}} - c = 0, \\ h_{\text{S\_előtt}} & \text{egyébként;} \end{cases}$$

$$h'_{\text{D\_előtt}} = \begin{cases} h_{\text{D\_előtt}} + c, & \text{ha } j = \text{D} \ \wedge\ h_{\text{doboz}} - c > 0, \\ 0, & \text{ha } h_{\text{doboz}} - c = 0, \\ h_{\text{D\_előtt}} & \text{egyébként;} \end{cases}$$

$$h'_{\text{Steve}} = \begin{cases} h_{\text{Steve}} + h_{\text{S\_előtt}} + c, & \text{ha } j = \text{S} \ \wedge\ h_{\text{doboz}} - c = 0, \\ h_{\text{Steve}} & \text{egyébként;} \end{cases}$$

$$h'_{\text{Digit}} = \begin{cases} h_{\text{Digit}} + h_{\text{D\_előtt}} + c, & \text{ha } j = \text{D} \ \wedge\ h_{\text{doboz}} - c = 0, \\ h_{\text{Digit}} & \text{egyébként;} \end{cases}$$

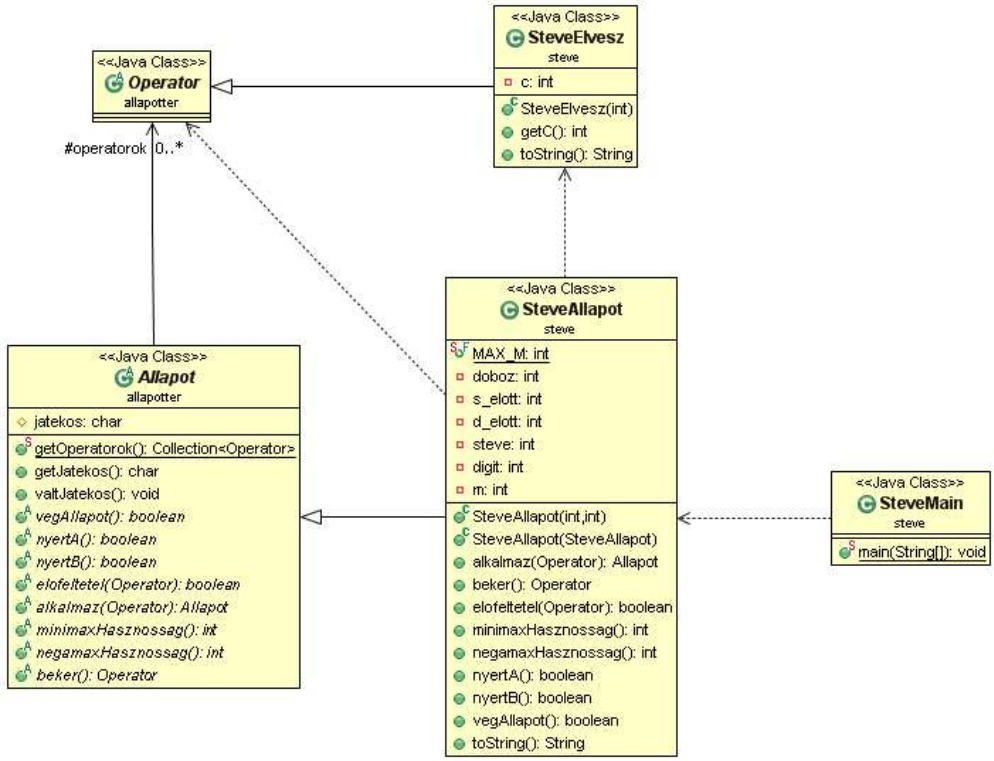
$$j' = \begin{cases} \text{S}, & \text{ha } j = \text{D}, \\ \text{D} & \text{egyébként.} \end{cases}$$

Operátoraink halmazát alkossák tehát a fentebb definiált $\text{Elvesz}(h, c)$ operátorok:

$$\mathcal{O} = \{\, \text{Elvesz}(h, c) \mid 1 \leq c \leq m \,\},$$

## 4.3.2. Programkódok a játékhoz

Nincs más dolgunk, mint származtatni egy-egy osztályt az `Operator` és az `Allapot` osztályokból, amelyekben aztán a játék állapottér-reprezentációjában leírtakat fogjuk kódolni.




4.2. ábra. A *Falánk Steve* játék osztálydiagramja.

Elsőként az operátort megvalósító osztály kódját nézzük meg. Ez az osztály nagyon egyszerű, nem tartalmaz semmi mást, csak az Elvesz operátor $c$ paraméterét egy egész típusú adattagként, a hozzá tartozó lekérdező metódussal.

```
1  package steve;
2
3  import allapotter.Operator;
4
5  public class SteveElvesz extends Operator {
6
7    private int c;
8
9    public SteveElvesz(int c) {
10     this.c = c;
11   }
12
13   public int getC() {
14     return c;
15   }
```

```
16
17    @Override
18    public String toString() {
19      return "SteveElvesz[" + c + "]";
20    }
21  }
```

A játék állapotterének megvalósításához a `SteveAllapot` osztályt a következőképpen írjuk meg:

```
1   package steve;
2
3   import java.util.HashSet;
4   import java.util.Scanner;
5
6   import allapotter.Allapot;
7   import allapotter.Operator;
8
9   public class SteveAllapot extends Allapot {
10
11    public static final int MAX_M = 100;
12
13    static {
14      operatorok = new HashSet<Operator>();
15      for ( int c = 1; c <= MAX_M; ++c )
16        operatorok.add( new SteveElvesz( c ) );
17    }
18
19    private int doboz, s_elott, d_elott, steve, digit;
20    private int m;
21
22    public SteveAllapot( int n, int m ) {
23      this.doboz = n;
24      this.s_elott = this.d_elott = this.steve = this.digit = 0;
25      this.m = m;
26      this.jatekos = 'A';
27    }
28
29    public SteveAllapot( SteveAllapot s ) {
30      this.m = s.m;
31    }
32
33    @Override
34    public boolean elofeltetel(Operator op) {
35      if ( op instanceof SteveElvesz ) {
36        SteveElvesz se = ( SteveElvesz )op;
37        int c = se.getC();
38        return c > 0 && c <= m && this.doboz >= c;
39      }
```

```
40      return false;
41    }
42
43    @Override
44    public Allapot alkalmaz( Operator op ) {
45      SteveAllapot uj = new SteveAllapot( this );
46      if ( op instanceof SteveElvesz ) {
47        SteveElvesz se = ( SteveElvesz )op;
48        int c = se.getC();
49        if ( this.jatekos == 'A' && this.doboz - c == 0 )
50          uj.doboz = this.d_elott;
51        else if ( this.jatekos == 'B' && this.doboz - c == 0 )
52          uj.doboz = this.s_elott;
53        else
54          uj.doboz = this.doboz - c;
55        if ( this.jatekos == 'A' && this.doboz - c > 0 )
56          uj.s_elott = this.s_elott + c;
57        else if ( this.doboz - c == 0 )
58          uj.s_elott = 0;
59        else
60          uj.s_elott = this.s_elott;
61        if ( this.jatekos == 'B' && this.doboz - c > 0 )
62          uj.d_elott = this.d_elott + c;
63        else if ( this.doboz - c == 0 )
64          uj.d_elott = 0;
65        else
66          uj.d_elott = this.d_elott;
67        if ( this.jatekos == 'A' && this.doboz - c == 0 )
68          uj.steve = this.steve + this.s_elott + c;
69        else
70          uj.steve = this.steve;
71        if ( this.jatekos == 'B' && this.doboz - c == 0 )
72          uj.digit = this.digit + this.d_elott + c;
73        else
74          uj.digit = this.digit;
75        if ( this.jatekos == 'A' )
76          uj.jatekos = 'B';
77        else
78          uj.jatekos = 'A';
79      }
80      return uj;
81    }
82
83    @Override
84    public Operator beker() {
85      Operator op;
86      Scanner sc = new Scanner( System.in );
87      while ( true ) {
88        System.out.print( "Kerem a lepeset: " );
```

```
89          int c = sc.nextInt();
90          op = new SteveElvesz( c );
91          if ( elofeltetel( op ) )
92            break;
93          else
94            System.out.println( "Ez az operator nem alkalmazhato!" );
95        }
96        System.out.println();
97        return op;
98      }
99
100     @Override
101     public int minimaxHasznossag() {
102       return this.steve - this.digit;
103     }
104
105     @Override
106     public int negamaxHasznossag() {
107       if ( this.jatekos == 'A' )
108         return minimaxHasznossag();
109       else
110         return -minimaxHasznossag();
111     }
112
113     @Override
114     public boolean vegAllapot() {
115       return this.doboz == 0 && this.s_elott == 0 && this.d_elott == 0;
116     }
117
118     @Override
119     public boolean nyertA() {
120       return vegAllapot() && this.steve > this.digit;
121     }
122
123     @Override
124     public boolean nyertB() {
125       return vegAllapot() && this.steve < this.digit;
126     }
127
128     @Override
129     public String toString() {
130       return "SteveAllapot[doboz=" + doboz + ", d_elott=" + d_elott +
131              ", s_elott=" + s_elott + ", digit=" + digit + ", steve=" +
132              steve + ", m=" + m + ", jatekos='" + jatekos + "']";
133     }
134   }
```

A kód számunkra érdekesebb, az állapottér-reprezentáció szempontjából fontos részei a következők:

- **13–17. sor**:

  A statikus osztályinicializáló blokkban gyűjtjük össze a lehetséges operátorokat. Az osztályt annyira általánosan írtuk meg, hogy képes legyen a maximálisan lehetséges $m = 100$ értékkel is végrehajtani a fánkok elvételét.

- **19. sor**:

  Az osztálynak az itt definiált öt adattagja fogja tárolni a játék egy állapotának az értékeit. A hatodik komponenst, az aktuálisan lépő játékost tartalmazó `jatekos` adattagot az osztály a szülő osztályától örökli.

- **22–27. sor**:

  Konstruktor, amely a játék kezdőállapotában leírt értékekkel inicializálja az adattagokat.

- **33–41. sor**:

  Az `elofeltetel` metódusban vizsgáljuk meg a paraméterként megadott operátor alkalmazhatóságának az előfeltételét. A 38. sorban szereplő `return` utasításban szereplő kifejezés egy az egyben az operátor alkalmazási előfeltételénél megadott formula Java nyelvi megfelelője.

- **43–81. sor**:

  Az alkalmazás hatását az állapottér-reprezentációban a hat komponens új értékeinek esetszétválasztásos definícióival adtuk, amelyeket az `alkalmaz` metódusban `if`–`else if`–`else` szerkezetekkel valósítottunk meg.

- **100–103. sor**:

  A játék egy-egy állapotának hasznosságát Steve szemszögéből nagyon egyszerűen kiszámíthatjuk: Steve megevett fánkjaiból kivonjuk Digit megevett fánkjait.

- **105–111. sor**:

  A `negamaxHasznossag`-gal nem foglalkozunk sokat: azt mondjuk, hogy megegyezik a `minimaxHasznossag` metódus által visszaadott értékkel, ha Steve következik lépni, és pont az ellentettje annak, ha nem ő jön.

- **113–116. sor**:

  A játék a reprezentáció szerint akkor ér véget, ha a játékosok már megették az összes fánkot. Ezt ellenőrizzük le a `vegallapot` metódusban.

- **118–121. sor**:

  Ha a játék véget ért, és Steve evett meg több fánkot, akkor ő nyert.

- **123–126. sor**:

  Ha a játék véget ért, és Digit evett meg több fánkot, akkor ő nyert.

Az elkészült osztályt példányosítani kell, majd a kezdőállapot átadható egy `Jatek` típusú objektum konstruktorának. Ezzel máris ki tudjuk próbálni a kiválasztott lépésajánló algoritmust:

```
1  package steve;
2
3  import jatek.Jatek;
4
5  public class SteveMain
6  {
7    public static void main( String[] args ) {
8      Jatek jatek = new Jatek( new SteveAllapot( 10, 3 ), 0, 100, 100 );
9      System.out.println( jatek );
10     jatek.jatszik();
11   }
12 }
```

A fenti példában egy olyan játékot játszhatunk, ahol $n = 10$ fánk van kezdetben a dobozban, amiből a játékosok egyszerre legfeljebb $m = 3$ darab fánkot vehetnek el.

### 4.3.3. A lépésajánló algoritmusok korlátai

Kezdjünk el játszani különböző paraméterezésű játékokat a kész programunkkal, és a kapott adatokat összegezzük egy táblázatban. A 4.1. táblázat sorait $n$, oszlopait $m$ lehetséges értékeivel címkéztük, az egyes celláiban szereplő értékek pedig azt mondják meg, hogy hány fánkot eszik meg Steve a játék végéig, ha végig a legjobb döntéseket hozza.

Az igazság az, hogy a 4.1. táblázat utolsó soraiban szereplő értékeinek a kiszámításához nagyon türelmesnek kell lennie az embernek. Ezek a számítások a játékfa korábban már említett nagy mérete miatt sokáig tartanak ezekkel az algoritmusokkal. Kicsit gyorsíthatunk a számításokon, ha alkalmazzuk az alfa-béta vágást is a lépésajánlat meghatározásához, de ez még mindig csak 2–3 szint növekedést hoz ugyanannyi időre vetítve. Minden jel arra mutat tehát, hogy valami más ötlet kell ennek a feladatnak a megoldásához – ahogyan azt előre sejtettük is.

Ha oszloponként, rögzített $m$ érték mellett tanulmányozni kezdjük a táblázatunkat, néhány triviális észrevételt tehetünk:

- ha a fánkok kezdeti száma $m$ és $2m$ közé esik (beleértve ezeket a határokat is), akkor Steve $m$ darab fánkot fog tudni megenni;
- ha a fánkok kezdeti száma $2m+1$, akkor Steve $m+1$ darab fánkra számíthat legjobb stratégiáját alkalmazva;
- ha a fánkok kezdeti száma $2m + 2$, akkor Steve – az első esethez hasonlóan – $m$ darab fánkot tud összegyűjteni.

Ezekkel a megfigyelésekkel a táblázatban az egyes oszlopok „fölső” sorait viszonylag gyorsan meg lehet határozni anélkül, hogy a játékot le kellene játszani. Az oszlopok

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | – | – | – | – | – | – | – | – | – | – | – | – | – | – | – |
| 2 | 1 | 2 | – | – | – | – | – | – | – | – | – | – | – | – | – | – |
| 3 | 2 | 2 | 3 | – | – | – | – | – | – | – | – | – | – | – | – | – |
| 4 | 1 | 2 | 3 | 4 | – | – | – | – | – | – | – | – | – | – | – | – |
| 5 | 4 | 3 | 3 | 4 | 5 | – | – | – | – | – | – | – | – | – | – | – |
| 6 | 2 | 2 | 3 | 4 | 5 | 6 | – | – | – | – | – | – | – | – | – | – |
| 7 | 5 | 5 | 4 | 4 | 5 | 6 | 7 | – | – | – | – | – | – | – | – | – |
| 8 | 1 | 6 | 3 | 4 | 5 | 6 | 7 | 8 | – | – | – | – | – | – | – | – |
| 9 | 8 | 3 | 6 | 5 | 5 | 6 | 7 | 8 | 9 | – | – | – | – | – | – | – |
| 10 | 4 | 7 | 7 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | – | – | – | – | – | – |
| 11 | 7 | 8 | 8 | 7 | 6 | 6 | 7 | 8 | 9 | 10 | 11 | – | – | – | – | – |
| 12 | 2 | 6 | 6 | 8 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | – | – | – | – |
| 13 | 11 | 7 | 7 | 9 | 8 | 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | – | – | – |
| 14 | 5 | 8 | 8 | 10 | 9 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | – | – |
| 15 | 10 | 7 | 9 | 8 | 10 | 9 | 8 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 16 | 1 | 9 | 8 | 8 | 11 | 10 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 16 | 10 | 9 | 9 | 12 | 11 | 10 | 9 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 18 | 8 | 8 | 10 | 10 | 10 | 12 | 11 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

4.1. táblázat. Falánk Steve legjobb eredményei $n$ és $m$ lehetséges értékei esetén

további értékeinek a meghatározásához észre kell vennünk, hogy ez a játék a (többek között [4]-ből) jól ismert Nim játéknak egy változata. Jelöljük egy rögzített $m$ érték esetén az oszlop értékeit kiszámító függvényt $f$-fel.

Képzeljük magunkat Steve helyébe úgy, hogy mi kezdjük a játékot, és próbáljuk kiszámítani az $f(i)$ értékeket $i > 2m + 2$ esetén! Tudjuk, hogy első lépésként legalább 1, legfeljebb $m$ darab fánkot vehetünk el, jó lenne tehát látni, hogy mi lesz ezekben az állapotokban a helyzet, ha megtesszük az adott lépést. A Nim alapjátékából tudhatjuk, hogy ha $i \bmod (m + 1) = 0$, akkor a kezdő játékos a játékot elveszti. Ha pedig $i \bmod (m + 1) \neq 0$, akkor két részjátszma összegét kell megvizsgálnunk: egyrészt azt, amelyik a lépés után megmaradt fánkokkal kezdődik, másrészt azt, amelyik az elvett fánkokkal kezdődik. Igen ám, de az első lépésünktől függően más és más kezdő fánkszámú részjátszmákkal játszhatunk tovább. Ezek közül a játszmák közül kell a számunkra legjobbal foglalkozni. Vizsgáljuk meg tehát azokat az $f(j)$ játszmákat, ahol

$$q = \left\lceil \frac{i}{m+1} \right\rceil \quad \text{és} \quad q \cdot m - m < j \leq q.$$

E játszmáknak az eredményei már ott szerepelnek az $m$-mel címkézett oszlopban, mielőtt még nekikezdenénk kiszámítani a számunkra érdekes, $f(i)$ értéket. Válasszuk ki

tehát közülük azt, amelyikben a legtöbb fánkot nyerhetjük:

$$\mathit{maxfánk} = \max\{\, f(j) \;\big|\; q \cdot m - m < j \le q \,\},$$

és határozzuk meg az ehhez tartozó legnagyobb indexet:

$$\mathit{index} = \{\, \mathit{idx} \;\big|\; q \cdot m - m < \mathit{idx} \le q \;\wedge\; f(\mathit{idx}) = \mathit{maxfánk} \,\}.$$

Mivel $i \bmod (m+1) = 0$ esetén az első részjátszmát mindenképpen mi (Steve) veszítjük el, ezért az össznyereségünk csupán $f(\mathit{index})$ lesz. Ha $i \bmod (m+1) \neq 0$, akkor viszont az $f(\mathit{index})$ és az $i - f(\mathit{index})$ értékek közül a nagyobbikkal számolhatunk.

Összefoglalva az eddigieket, az $f(i)$ függvény definíciója rögzített $m$ érték esetén a következő:

$$f(i) = \begin{cases} i, & \text{ha } 0 \le i < m, \\ m, & \text{ha } m \le i \le 2m \text{ vagy } i = 2m+2, \\ m+1 & \text{ha } i = 2m+1, \\ \max\{\, i - f(\mathit{index}), f(\mathit{index}) \,\}, & \text{ha } i > 2m+2 \text{ és } i \bmod (m+1) \neq 0, \\ f(\mathit{index}) & \text{egyébként.} \end{cases}$$

Az $f(i)$ függvény értékeit tehát dinamikusan, a korábban már kiszámított értékekből számíthatjuk. A program kódja a következőképpen néz ki:

```
1   package steve;
2
3   import java.util.Scanner;
4
5   public class SteveACM {
6     public static void main( String[] args ) {
7       Scanner sc = new Scanner( System.in );
8       int n = sc.nextInt();
9       int m = sc.nextInt();
10      if ( n <= 2 * m || n == 2 * (m+1) )
11        System.out.println( m );
12      else if ( n == 2 * m + 1 )
13        System.out.println( m + 1 );
14      else {
15        int f[] = new int[ 101 ], idx;
16        for ( int i = 0; i < m; ++i ) {
17          f[ i ] = i;
18        }
19        for ( int i = m; i <= 2 * m; ++i ) {
20          f[ i ] = m;
21        }
22        f[ 2 * m + 1 ] = m + 1;
23        f[ 2 * m + 2 ] = m;
24        for ( int i = 2 * m + 3; i <= n; ++i ) {
25          int j, seged;
```

```
26            idx = i / ( m + 1 ) * m;
27            for ( seged = idx - m, j = idx; j > seged; --j ) {
28              if ( f[ j ] > f[ idx ] ) {
29                idx = j;
30              }
31            }
32            if ( i % ( m + 1 ) == 0 )
33              f[ i ] = f[ idx ];
34            else
35              f[ i ] = Math.max( i - f[ idx ], f[ idx ] );
36          }
37          System.out.println( f[ n ] );
38        }
39      }
40    }
```

A futási idő tetszőleges megengedett bemeneti számpár esetén ezzel a számítási módszerrel a másodperc töredékére csökken. Két módon tovább gyorsítható a kimeneti adatok előállítása:

1. ha olyan programozási nyelven kódolunk, és azon olyan kódot írunk, amelyből gyorsabban futó program generálható;

2. a viszonylag kevés lehetséges bemeneti számpár mindegyikéhez előre kiszámítjuk és tároljuk a kimeneti értéket, majd az adatok beolvasásakor a tárolt értéket írjuk a kimenetre.

Az első esettel azért nem foglalkozunk, mert nem szolgáltat új algoritmust, a másodikkal pedig azért nem, mert a kód nagyobb része mindössze egy $\frac{100 \cdot 101}{2} = 5\,050$ elemet tartalmazó tömböt tartalmazna, amelynek a leprogramozása, és amelyből az adat kiolvasása nem okozhat különösebb gondot.

Hogy mire elég a dinamikus megközelítés ennél a feladatnál? Ezt láthatjuk a 4.3. ábrán, amely a [1] honlapról elérhető 2669-es feladat 2005. május 1-i kimutatása alapján készült.

A sikeres dinamikus megoldás ismeretében módosíthatjuk a `SteveAllapot` osztály `minimaxHasznossag` metódusát úgy, hogy az ne csak az aktuális állapotban megevett fánkokat vegye figyelembe az állapot hasznosságának a meghatározásakor, hanem azt is, hogy hány fánkra számíthat még Steve azokból, amelyeket még nem ettek meg. Ezzel a módosítással az előretekintés mélységét is csökkenthetnénk a lépésajánló algoritmusok konstruálásakor, hiszen gyakorlatilag egy perfekt kiértékelő függvény lenne a birtokunkban a játék állapotainak a kiértékeléséhez.

## 4.4. További feladatok a kétszemélyes játékok témaköréből

Az ACM programozó versenyek korábbi feladatait számos helyen megtalálhatjuk az Interneten ([16, 1]). A feladatgyűjteményekben több ezer feladat található, ezek között

**2669**
**Voracious Steve**

**Ranklist** (Best *Accepted* or *Presentation Error* from each author)
Last update on **Sun May 1 08:57:36 UTC 2005**

| | CPU Time | Memory | Author | Source | Date (UTC) | ID (host) |
|---|---|---|---|---|---|---|
| 1 | 0.002 | Minimum | Márk Kósa | C | 2005-05-01 08:59:24 | 48858 (H1) |
| 2 | 0.756 | 528 | Koo Kyung-ryeol | C++ | 2005-02-24 04:25:53 | 44496 (H1) |
| 3 | 0.793 | 540 | junhee hong | C++ | 2005-02-21 06:09:30 | 44109 (H1) |
| 4 | 1.363 | 404 | Adrian Kügel | C | 2005-01-15 22:02:09 | 41450 (H1) |
| 5 | 1.945 | 20864 | PRITCHARD David | C++ | 2005-03-18 20:24:26 | 46618 (H1) |
| 6 | 2.092 | 4928 | Guo Yunsong | C++ | 2005-02-16 09:27:37 | 43470 (H1) |
| 7 | 2.439 | 4540 | Aaron Chan | C++ | 2005-02-14 22:15:25 | 43406 (H1) |
| 8 | 5.189 | 4596 | PPP | C++ | 2005-03-03 02:41:33 | 45313 (H1) |

4.3. ábra. Statisztika a http://acmicpc-live-archive.uva.es/nuevoportal/ oldalról

pedig soknak valamilyen kétszemélyes játék az alapproblémája. Ezeket a feladatokat alapvetően kétféle módon hasznosítjuk az oktatásban:

1. Az alapprobléma általában egy-egy jól állapottér-reprezentálható kétszemélyes játék. Ezek közül az elmúlt években sok játéknak el is készítettük a reprezentációját *A mesterséges intelligencia* tárgy gyakorlatain, másokat pedig gyakorlásra, házi feladatnak jelöltünk ki. Ezeknek a játékoknak a reprezentáció alapján történő implementálása szintén érdekes feladat szokott lenni hallgatóink számára, különösen ha az elkészült játékprogramokat egymás ellen kell kipróbálni. A játékok implementálása szokott az a pont lenni, ahol az általunk ismertetett algoritmusok akkor működnek igazán hatékonyan, ha a játékokhoz sikerül „erős” játékállás-kiértékelő függvényeket találni. Ez hallgatóinknak is érdekes és nagy kihívás, és mindenképpen jó alkalom arra, hogy jobban elmélyedjenek a probléma hatékony megoldásának a részleteiben.

2. A másik lehetőség a feladatok természetes hasznosítása: eredeti szövegezésükkel feladhatók a helyi szervezésű programozó versenyeinken. Ebben az esetben e feladatok megoldásakor általában nem a bemutatott lépésajánló algoritmusoknak az alkalmazását várjuk a versenyzőktől, mert az esetek döntő többségében mindig található egy olyan feladatmegoldási technika, amely gyorsabban szolgáltatja az elvárt eredményt, mint ezek az algoritmusok, amelyek egyébként kiindulópontként és az eredmények ellenőrzésére itt is remekül használhatók.

A 2008/2009-es tanévben a korábban áttanulmányozott és a helyi programozó versenyeinken feladott ACM versenyfeladatokból a http://it.inf.unideb.hu/honlap/acm címen készítettünk egy gyűjteményt. Ezek között külön kategóriát alakítottunk ki a kétszemélyes játékokra épülő feladatoknak ([9]).

5. FEJEZET

# PCRM: kiértékelő szoftver programozói versenyekhez

## 5.1. Bevezető

A Debreceni Egyetem programozó matematikus, programtervező matematikus és programtervező informatikus szakos hallgatói 1995-től vesznek részt az ACM nemzetközi programozói verseny regionális fordulójában[1]. Bár a versenyen hallgatóként akár én magam is indulhattam volna, a sors úgy hozta, hogy csak a 2000-es évek elejétől, ám már a szervezők, felkészítők és a zsűri oldalán állva tapasztalhattam meg e verseny izgalmait.

A verseny regionális fordulójára egyetemünk minden évben két-két háromfős csapatot nevez(ett) be, a csapatok kiválasztására minden év szeptemberében-októberében kerül(t) sor egy helyi és egy országos forduló keretében. Míg a helyi forduló feladatainak összeállítása és a verseny lebonyolítása mindig saját hatáskörben zajlott, addig az országos fordulóban a feladatokat a Budapesti Műszaki Egyetem és az Eötvös Loránd Tudományegyetem lelkes kollégái állították össze, és az azokra beküldött megoldásokat is ők értékelték.

A helyi egyetemi fordulók lebonyolítását a kezdeti időkben igencsak megnehezítette, hogy a versenyzők által beküldött megoldások ellenőrzését kézzel kellett elvégeznünk, ami a kényelmetlenségeken túl a zsűri hatékony munkáját is akadályozta, és rengeteg hibalehetőséget rejtett magában. E problémák kiküszöbölésére határoztuk el, hogy elkészítünk egy olyan szoftvert, amely nagy mennyiségű megoldást képes feldolgozni mind versenykörülmények között, mind pedig offline módon. Ebben a fejezetben ezt a szoftvert, a Programming Contest Result Manager (PCRM) programot mutatom be, amelyet Pánovics János egyetemi tanársegéd kollégámmal és Gunda Lénárddal, 2005-ben végzett programtervező matematikus hallgatónkkal közösen fejlesztettünk ki [10]. A program készítésekor fő feladatom a beküldött feladatok ellenőrzését végző rutinok fejlesztése és tesztelése volt, különös tekintettel a külső ellenőrző programok rendszerhez történő illesztésére. Emellett még az eredményeket jelző HTML felület is az én ötleteimnek megfelelően készült el. A PCRM program C++ programozási nyelven íródott, fejlesztése 2004-ben fejeződött be.

[1] 1998–2000 között három éven keresztül a Debreceni Egyetem nem nevezett a versenyre.

### 5.1.1. A programozói versenyekről

Mielőtt a részletekbe belevágnánk, fontos tisztázni, hogy a PCRM készítésekor milyen konrét kívánalmakat fogalmaztunk meg a majdani kiértékelő programunkkal kapcsolatosan. A legfontosabb célunk az volt, hogy a saját szervezésű programozó versenyeinken megkönnyítsük a zsűri munkáját a feladatok kiértékelésekor. Emellett voltak olyan ötleteink is, hogy a programot úgy írjuk meg, hogy azt „békeidőben", a versenyeken kívül is fel lehessen használni. Erre két olyan lehetőséget láttunk, ahol a későbbiekben ki is próbáltuk a programot: a *Programozás 1–2* és a *Mesterséges intelligencia 1* tárgyakat[2], ahol a hallgatói létszámok és a beadandó házi feladatok nagy száma miatt szükség is lehetett egy ilyen fajta ellenőrző segédprogram alkalmazására.

Az elsődleges célunk megvalósításához először tisztáztuk azt, hogy mit is tekintünk valójában programozó versenynek? A mi elképzelésünk szerint egy *programozó verseny* folyamán a *versenyzőknek* (csapatoknak) *feladatokat* kell megoldaniuk. A versenyzők az egyes feladatok megoldását valamilyen programozási nyelven megírt *forráskód* formájában készítik el. A lefordított forráskódból előállított *program* a *bemeneti adatokból* egy *kimeneti adathalmazt* hoz létre. A kimeneti adathalmaz vagy a standard kimeneti csatornán, vagy egy (ritkán több) állományban áll elő.

Nézzük, az előzőekben leírtak mellett mi a *zsűri* feladata egy programozó versenyen? Amikor a zsűri megkapja egy feladat megoldását egy versenyzőtől, lefordítja azt a megfelelő fordítóprogrammal, és futtatja különböző tesztesetekre. Minden feladathoz tartozik legalább egy olyan *állomány*, amely a *teszteseteket* tartalmazza. Ezeket az állományokat kell a beküldött programoknak feldolgozni, és a helyes kimenetet előállítani.[3] A kimenet akkor *helyes*, ha vagy megegyezik egy előre generált, tárolt állománnyal, vagy egy *külső ellenőrző program* szerint helyes. A zsűri a kimenet helyességének az ismeretében *értékeli* a beküldött forráskódot, a legtöbb esetben valamilyen *pontszámot* adva a megoldásra. A zsűri feladata az is, hogy legkésőbb a versenyidő lejártakor *eredményt hirdessen*, azaz megállapítsa a versenyzők végső sorrendjét.

Hogyan zajlik a feladatok értékelése? Nos, ez az a pont, ahol az egyes programozó versenyek a kiírásaikban rögzített szabályoknak megfelelően egymástól gyökeresen eltérő elveket vallhatnak. Az alábbiakban az általam legfontosabbnak tartott értékelési alapelveket ismertetem:

- Az egyetlen lényeges értékelési szempont a helyesen megoldott feladatok száma. A feladat csak akkor tekinthető megoldottnak, ha a beküldött program *minden* tesztesetre helyes eredményt ad.

- Részpontszámok számítása. Ebben az esetben egy feladathoz több teszteset tartozik, egyszerűbbek és bonyolultabbak egyaránt. A tesztesetek mindegyike ér valahány pontot, nem feltétlenül ugyanannyit. A feladatra beadott megoldást minden tesztesetre lefuttatjuk, és összeadjuk azon tesztesetek pontszámait, amelyekre a megoldás helyes kimenetet állított elő. A megoldásra adott pontszám az elfogadott tesztesetek pontszámainak összege lesz.

[2] A BSc képzésekben *Magas szintű programozási nyelvek 1–2* és *A mesterséges intelligencia alapjai* néven.

[3] Ritkán, de előfordul, hogy egy feladathoz nincsenek bemeneti adatok. Ezt az esetet tekinthetjük úgy, hogy a beküldött programnak egy *üres* állományt kell feldolgoznia.

- Elfogadott tesztesetek számolása. Az előző számítási módhoz hasonló, azzal a különbséggel, hogy ebben az esetben minden teszteset azonos pontot ér, az egyszerűség kedvéért mondjuk azt, hogy 1-et. Látható, hogy ez a fajta pontszámítás csak a súlyozásában tér el az előzőtől.

Az ACM programozó versenyeken, ahol a feladatot csak akkor tekintik megoldottnak, ha minden tesztesetre helyes megoldást adott a versenyző által beküldött program, megengedik, hogy egy sikertelen kísérlet után a versenyző újra próbálkozhasson a feladat megoldásával a verseny időhatárain belül. Ha ezt nem tennék lehetővé, akkor – nehéz problémák és furfangos tesztesetek kitűzése esetén – előfordulhatna, hogy a versenyen kevés elfogadható megoldás születne, és így nehéz lenne rangsorolni a versenyzőket, arról nem is beszélve, hogy a verseny végeztével a versenyzők és a feladatok összeállítói (a zsűri) is sikerélmény nélkül maradhatnának.[4] Ugyanakkor a szituáció végső soron hasonlít a valósághoz: ha egy program nem képes egyből megoldani a kitűzött feladatot (azaz nem felel meg a megrendelő igényeinek), akkor van lehetőség a további finomításra.

### 5.1.2. A zsűri üzenetei

Az ACM versenyeken a megoldás beküldését követően a versenyző egy visszajelzést kap a megoldás helyességéről. A zsűri üzenete alapján aztán a versenyző eldöntheti, hogy mi legyen a következő lépése: újra próbálkozik a feladat megoldásával, vagy másik feladat megoldásába kezd bele.

A gyakorlatban mindez azt jelenti, hogy a versenyző a forrásállomány elküldése után megtekintheti egy weboldalon az eredményt, amely tartalmazza azt, hogy mikor küldte be a megoldást, és hogy milyen választ kapott rá a zsűritől.

Az alábbiakban látható a PCRM által küldött üzenetek listája:

- **Accepted (elfogadva)** – Azt jelenti, hogy a program átment minden teszteseten, és el lett fogadva.
- **Wrong answer (rossz válasz)** – A program lefutott, de az általa előállított kimenet nem egyezik meg a zsűriével, vélhetően hibás.
- **Compile error (fordítási hiba)** – Ezt az üzenetet akkor kapja a versenyző, ha a fordítóprogram nem tudta lefordítani a forrásállományt.
- **Runtime error (futási hiba)** – Azt jelzi, hogy a program szabálytalanul ért véget, amit általában valamilyen memóriahivatkozási hiba vagy kivétel kiváltódása okozhat. A programnak azt, hogy helyesen ért véget, 0 visszatérési értékkel kell jeleznie, különben szintén ezt a hibaüzenetet kapja a versenyző.
- **Time limit exceeded (időtúllépés)** – Minden problémához be lehet állítani egy időkorlátot (ezredmásodpercekben mérve). Ennyi idő alatt a megoldás programjá-

[4] A 2002-es ACM programozó verseny Varsóban megrendezett közép-európai regionális selejtezőjének résztvevőin még ez sem segített: a csapatok fele (61-ből 31 csapat) egyetlen feladatot sem tudott megoldani.

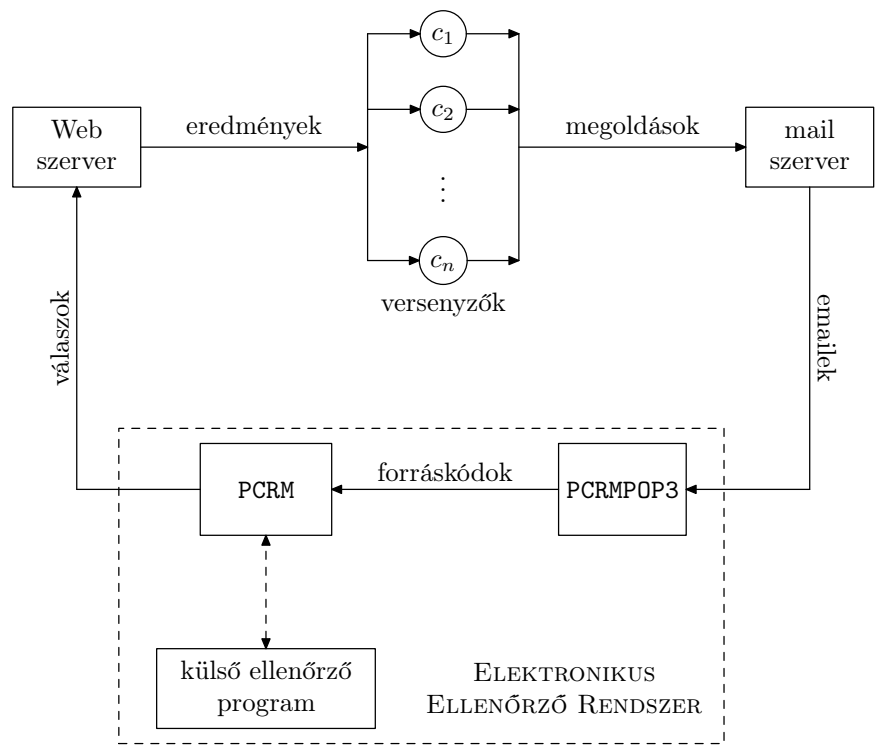
nak le kell futnia. Ha az időkorlátot a program túllépi, a PCRM megállítja azt, és ezt a hibaüzenetet küldi a versenyzőnek.

- **Memory limit exceeded (memóriakorlát-túllépés)** – Minden problémához be lehet állítani egy memóriakorlátot (kilobájtokban számolva). A PCRM ellenőrzi, hogy a megoldás programja futás közben elérte-e ezt a korlátot, és ha igen, megállítja a programot, miközben ezt a hibaüzenetet küldi a versenyzőnek. Megjegyzendő, hogy Windows operációs rendszer alatt az ellenőrzés csak a program futásának a befejeződése után történik meg, ha nem volt fordítási és futási hiba, illetve nem történt időtúllépés sem.

## 5.2. Hogyan működik?

Az elektronikus kiértékelő rendszert két program alkotja: a PCRMPOP3 és a PCRM.

A PCRMPOP3 egy egyszerű shell program, amely a POP3 protokoll segítségével üzeneteket tölt le egy mail szerverről, tárolja őket, majd egy MIME message parser segítségével kibányássza belőlük a forráskódokat. Ezután, ha a rendszert úgy paramétereztük fel, meghívja a PCRM programot, hogy végezze el a kapott megoldás ellenőrzését.




5.1. ábra. A rendszer funkcionális felépítése

A PCRM a konfigurációs állományában meghatározott beállításokat használja fel ahhoz, hogy meghatározza működési módját. Lefordítja a kapott megoldásokat, majd lefuttatja őket a tesztesetekre. Mind a fordítási, mind a futtatási parancssorokat egyenként lehet beállítani az egyes programozási nyelvekhez. Ily módon szabályozni tudjuk, hogy milyen nyelvű programokat kezeljen a rendszer, mint ahogyan arra is van lehetőség, hogy más programokat használjunk a lefordított programok futtatásához (ahogyan például a Java nyelvű programok esetében történik).

Ha a program a bemeneti adatokat a standard bemenetről olvassa, a PCRM átirányítja a standard bemenetet a program számára úgy, hogy az megkaphassa a szükséges tesztesetet. A standard kimenet hasonló módon irányítható át, ha a kimenetet oda kell írni.

Miután a program sikeresen befejeződött a kijelölt időkorláton belül úgy, hogy nem lépte túl a megadott memóriakorlátot, a PCRM összehasonlítja a kimenetet egy előre megadott kimeneti állománnyal, és ha egyezik a kettő, elfogadja a tesztesetet. A PCRM külső programot is meg tud hívni, amely ellenőrzi a kimenetet, és visszatérési értéke segítségével jelzi a PCRM-nek, hogy helyes-e a megoldás vagy sem.

### 5.2.1. A megoldásra vonatkozó követelmények

Először is, a megoldásokat olyan programozási nyelven kell elkészíteni, amelyet a PCRM támogat. A támogatott programozási nyelvekről a 5.3.4. fejezetben olvashatunk részletesebben.

Másodszor, a visszatérési érték. Minden megoldásnak, amit a versenyzők beküldenek, 0 értéket kell visszaadniuk, amikor befejezik működésüket. Ha ez nem teljesül, az ellenőrző program azt hiheti, hogy valamilyen futási idejű hiba lépett fel, és ezt fogja eredményként megjeleníteni még akkor is, ha egyébként a kimenet helyes.

C és C++ programozási nyelvek esetén ezt nagyon egyszerű megvalósítani, egyszerűen csak begépelünk egy `return 0;` utasítást a `main()` függvény végére. Más esetben tanulmányozzuk át az adott programozási nyelv dokumentációit arra vonatkozóan, hogy hogyan kezelik és állítják elő a visszatérési értéküket az ilyen nyelveken írt programok.

## 5.3. A `pcrm.ini` állomány

A `pcrm.ini` állomány a PCRM legfontosabb része. Ebben a fájlban található az összes olyan információ, ami egy verseny levezényléséhez szükséges. A következőkben végignézzük a `pcrm.ini` fájlban szereplő beállítási csoportokat.

### 5.3.1. A `[global]` csoport

A `[global]` csoport tartalmazza a programra és a futtatási környezetre vonatkozó általános beállításokat.

`numcount` A versenyben részt vevő versenyzők száma. Az értéknek nagyobbnak kell lennie 0-nál. Ha egyéni versenyzők helyett csapatok versenyeznek, akkor értelemszerűen a résztvevő csapatok számát jelzi.

`numprob` A versenyben kitűzött feladatok száma. Az értéknek nagyobbnak kell lennie 0-nál, de nem haladhatja meg a 64-et.

`mode` A pontszámítás módja. Alapvetően ez határozza meg, hogy a PCRM hogyan fogja a pontokat számolni az eredmények kiértékelése után. A programban három pontszámítási mód közül lehet választani:

`1` Jó/rossz

Ez a legegyszerűbb számítási mód, amely aszerint értékel, hogy melyik feladatot sikerült megoldani és melyiket nem.

`2` Pontszám alapú

Ebben a módban pontozzuk az összes feladat összes tesztesetét, összeadjuk a pontokat, és az a győztes, aki több pontot ér el.

`3` ACM stílusú

Az ACM stílus az ACM versenyek hagyományos pontszámítási módja. Itt az a versenyző a jobb, aki több feladatot oldott meg. Ha két versenyző ugyanannyi feladatot oldott meg jól, akkor kettejük közül az a jobb, aki kevesebb büntetőpontot gyűjtött. A részletes pontozási szabályokat lásd a 5.1.1. fejezetben.

`compout` Amikor a PCRM lefordítja a beküldött forráskódot, eldönthetjük, hogy szeretnénk-e látni a fordító üzeneteit vagy sem. Ez a beállítás szabályozza, hogy mi történjen az outputtal.

`0` nincs fordítási output (alapértelmezés)

`1` a fordítási üzenetek megjelenítése

`runout` Megjeleníti a futás kimenetét az egyes programok futtatásakor. Csak teszteléshez és nyomkövetéshez használatos.

`0` nincs output (alapértelmezés)

`1` a kimenet megjelenítése

`result` Az eredményszámítás módja.

`0` amikor egy új megoldás érkezik, teszteli a megoldást az összes tesztesetre, és minden tesztesetről készít egy bejegyzést a history állományba

`1` amikor egy új megoldás érkezik, teszteli a megoldást az összes tesztesetre, de csak egy bejegyzést ír a history állományba

`2` teszteli az összes tesztesetet az első hibáig, és készít egy bejegyzést a history állományba

ACM stílusú versenyekhez a 2-es mód az ajánlott.

`retest` Megoldás újratesztelésének beállítása. Ez a beállítás mondja meg a PCRM-nek, hogy mit csináljon, ha egy második vagy harmadik megoldás is érkezik ugyanarra a feladatra.

`0` amikor egy tesztesetre már tesztelt és elfogadott egy megoldást, nem teszteli újra

`1` mindig újratesztel

`2` újratesztel minden tesztesetet, hacsak nem volt már minden teszteset elfogadva

`pop3head` A pop3 fejléc ellenőrzése. A PCRMPOP3 egy rövid fejlécet ad hozzá minden állományhoz, amely információkat tartalmaz az állományról (versenyző, feladat, időbélyeg). Amikor engedélyezzük ezt a beállítást, a PCRM fel fogja használni ezeket az információkat, amellyel természetesen még pontosabbá tehetjük a zsűrizést. A fájlok integritását is ellenőrizhetjük ily módon.

`0` ne ellenőrizze

`1` ellenőrizze

`acmstart` A verseny kezdetének időpontja HHMM formátumban.

`acmstop` A verseny végének időpontja HHMM formátumban.

`acmpenal` Büntetőpontok száma rossz megoldás esetén. Az alapértelmezés 20 pont (ACM standard).

### 5.3.2. A `[html]` csoport

A PCRM elő tud állítani HTML formátumú kimenetet is. A kimenet formázásához a `html` csoport beállításait használhatjuk, melyek a következők:

`frame` Bizonyos esetekben szükséges lehet, a zsűri üzeneteket hozzon a versenyzők tudomására. Ezt meg tudja tenni az eredmények és a beküldési lista közé egy `<iframe>`-et beszúrva. Meg lehet adni annak az állománynak a nevét is, amelyből a frame tartalma származik, továbbá a frame egyéb jellemzőit is (lásd alább).

`0` a frame generálás kikapcsolása

`1` a frame generálás bekapcsolása, a frame megjelenítése az eredmények és a beküldési lista között

`framesrc` A frame szövegét tartalmazó állomány neve. Ez lesz az `<iframe> src` property-jének az értéke.

`framewidth` A frame szélessége pixelekben.

`frameheight` A frame magassága pixelekben.

`showlang` Ha ennek a beállításnak 1 az értéke, akkor a beküldött megoldás programozási nyelve megjelenik az eredménytáblában.

`destfile` Annak az állománynak a neve, amelybe a HTML formátumú kimenet generálódik. Alapértelmezés szerint az eredmény egy `pcrm_res.html` nevű állományba íródik ugyanabban a könyvtárban, ahol a PCRM is fut. Megadhatjuk a fájlnév elérési útját is (pl. `../public_html/index.html`).

`autorefresh` Ha ez az érték be van állítva, akkor a HTML állomány is tartalmazni fogja a beállítást, így a böngésző automatikusan frissíteni fogja a HTML oldalt bizonyos időközönként. Az értéket másodpercekben számolva kell megadni.

`gfxproblem` Ha ez az érték 1-es, akkor a probléma neve helyett egy betűvel ellátott kis karika jelenik meg a HTML oldalon. Csak akkor játszik szerepet, ha `resgfx` módban fut a program.

`historynum` Megszámozza a beküldési lista bejegyzéseit, 1-től kezdve.

- `0` számozás kikapcsolása (alapértelmezés)
- `1` számozás bekapcsolása

`noresultafter` Megadhatunk egy időt HHMM formátumban (hasonlóan a `global` csoport `acmstart`/`acmstop` beállításaihoz), amely után már nem generálódik a globális eredménylista (ezt az eredmény állományt a `destfile` beállításnál adhatjuk meg).

`judgeresults` Engedélyezi (1) vagy letiltja (0) az alapértelmezettől eltérő típusú kimenet generálását. Az alapértelmezett értéke 0. Ha az értéke 1, akkor több eredmény állományt generál (lásd a `judgefiles` beállítást). Ezek az állományok a részletes és teljes eredménylistán túl (amit alapértelmezett esetben generál a program, `_judge` kiegészítéssel a nevében) az egyes versenyzők személyenkénti eredményeit is tartalmazzák. A beállítás a `noresultafter` opcióval együtt használva azt eredményezi, hogy egy bizonyos idő után már csak a zsűri láthatja a teljes eredménylistát, míg a versenyzők csak a saját eredményeiket tekinthetik meg.

`judgefiles` Megadja a `judgeresult` által használt állományok elérési útját és neveit (kiterjesztésükkel együtt). Az állományok generálásakor `_judge` és `_#` kiegészítéseket ad az állományok neveihez (ahol `#` a versenyző sorszáma). Például ha a `../public_html/judgeonly/result.html` nevet adjuk meg, akkor az állományok nevei a következők lesznek ebben a könyvtárban: `result_judge.html`, `result_1.html`, `result_2.html` stb. Alapértelmezett értéke: `judge_res.html`.

`addtime` Ha 1 az értéke, az elfogadott programok mellett jelzi, hogy mennyi ideig futottak. Alapértelmezett értéke 0.

`addtime2` Ha 1 az értéke, az elfogadott programok mellett részletes információkat jelenít meg a futási időről. Használatához az `addtime` kapcsolónak engedélyezettnek kell lennie. A beállítás nem működik Linux rendszereken.

`addmemory` Ha 1 az értéke, az elfogadott programok mellett jelzi, hogy mennyi memóriát használtak futásuk során. Alapértelmezett értéke 0.

### 5.3.3. A [pop3] csoport

A `pop3` csoport tartalmazza azokat a beállításokat, melyek a PCRMPOP3 program működését szabályozzák. Ezek leginkább a levelek feldolgozására vontakozó beállítások, valamint olyan opciók, amelyek azt írják le, hogy hogyan induljon el maga a PCRM program.

`server` A POP3 szerver domain neve vagy IP címe.

`user` Felhasználói név a POP3 szerverhez.

`pass` A felhasználói névhez tartozó jelszó.

`checkwait` Egy egész szám, amely azt mutatja (másodpercekben), hogy `listen` és `autoall` módban milyen időközönként ellenőrizze a program a POP3 szerverre beérkezett leveleket.

`subject` A levelek tárgyának ellenőrzését szabályozza.

- `0` nincs megszorítás a tárgyra vonatkozóan
- `1` erős formájú tárgysort ír elő
- `11` jelszóval kiegészített tárgysort ír elő
- `12` kizárólag jelszót tartalmazó tárgysort ír elő

`savemail` Az üzenetek mentését szabályozza.

- `0` nincs mentés
- `1` minden üzenet elment a `pop3/` könyvtárba

`genresult` Azt jelzi, hogy generáljon-e eredményeket egy POP3 műveletet követően. Ha 1 vagy 2 az értéke, akkor `auto1` és `autoall` módban a PCRM program futása után eredményeket is generál.

- `0` nem generál eredményeket
- `1` a `reshtm` parancsot hajtja végre (egyszerű HTML állományt állít elő)
- `2` a `reshtm` parancsot hajtja végre (grafikus elemeket is tartalmazó HTML állományt készít), ez az alapértelmezett érték

`genalways` Ha 0-tól különböző az értéke, akkor `autoall` módban minden futás után újragenerálja az eredményeket, nemcsak akkor, ha frissülnek a feldolgozandó forrásállományok, és lefut a PCRM program. Ez a beállítás használható a hátralévő versenyidő egyenletes időközönként történő megjelenítésére.

- `0` csak akkor generálja az eredményeket, amikor szükséges
- `1` `autoall` módban a PCRM minden futása után újragenerálja az eredményeket

`pcrmpath` Jelzi, hogy hol található a `pcrm.exe` (vagy Linuxokon a `pcrm`) futtatható állomány. Alapértelmezés az aktuális könyvtár, de beállíthatunk mást is, `../debug/windows/pcrm.exe` formában.

`autoend` A verseny automatikus befejezését szabályozza. Ellenőrzi a `[global]` csoport `acmstop` idejét, és amint a verseny véget ér, megszakítja az `autoall` mód ellenőrző ciklusát. A `genresult` paraméternek megfelelően a verseny befejezésekor legenerálja a végeredményt tartalmazó állományokat. Megjegyzendő, hogy `listen` módban is működik.

### 5.3.4. A `[compilerwin]` és `[compilerlnx]` csoportok

A `[compilerwin]` és `[compilerlnx]` csoportok tartalmazzák a fordítási sorokat, amelyek megmondják a programnak, hogy hogyan fordítsák le a forrásállományokat. A `[compilerwin]` csoportot Windows, a `[compilerlnx]` csoportot Linux operációs rendszerek alatt használja a program.

A csoportban szereplő bejegyzések

```
lang=compileline
```

alakúak, ahol a `lang` a `c`, `cpp`, `asm`, `java`, `pas`, `bas` vagy `cs` karaktersorozatok valamelyike lehet. Ezek a karaktersorozatok rendre a C, C++, Assembly, Java, Pascal, Basic és C# nyelvekhez tartozó fordítási sorokat jelölhetik.

A `compileline` írja le a fordítót és a fordítási paramétereket. A benne szereplő százalékjel (`%`) karakterek helyére a forrásállomány neve helyettesítődik be teljes elérési úttal, de kiterjesztés nélkül.

Így például, ha van egy

```
cpp=g++ -Wall -o % %.cpp
```

sorunk, és a `problem3/p4c19.cpp` állományt szeretnénk lefordítani, akkor a `%` helyére behelyettesítődik a `problem3/p4c19.cpp` állomány neve, és az aktuális parancssor a következőképpen fog kinézni:

```
g++ -Wall -o problem3/p4c19 problem3/p4c19.cpp
```

Mindez hasonlóan működik Windows operációs rendszerek alatt is.

### 5.3.5. A [runwin] és [runlnx] csoportok

A `[runwin]` és `[runlnx]` csoportok tartalmazzák a futtatási sorokat, amelyek megmondják a programnak, hogy hogyan futtassák a lefordított megoldásokat. A `[runwin]` csoportot Windows, a `[runlnx]` csoportot Linux operációs rendszerek alatt használja a program.

A bejegyzés felépítése hasonló az előző alfejezetben leírt fordítási sorhoz, csak ezúttal a futtatási parancsot kell definiálnunk. A legtöbb esetben ez mindössze egy „`%`” vagy egy „`./%`” karaktersorozat, némely nyelv esetén azonban ettől eltérő is lehet, például: „`java %`”.

### 5.3.6. A [problem?] csoportok

Minden feladatnak saját `[problem?]` csoportja van (a `?` karakter helyére a feladat sorszámát helyettesítve). Minden feladathoz tartoznia kell legalább egy tesztesetnek.

`numtest` A tesztesetek száma.

`maxtime` Időkorlát az egyes tesztesetekre (ezredmásodpercben).

`maxmemory` Memóriakorlát a programok számára (kilobájtban). Alapértelmezett értéke 0, amely letiltja a memóriakorlát ellenőrzését. Linux és Windows NT4/2000/XP operációs rendszerek alatt használható.

`test?type` A `?` sorszámú teszteset típusa.

- `1` egy bemeneti állományhoz egy kimeneti állomány tartozik, a bemeneti adatokat a standard inputról olvassa, a kimenetet a standard outputra írja a tesztelendő program
- `2` egy bemeneti állományhoz egy kimeneti állomány tartozik, a bemeneti adatokat fájlból olvassa, a kimenetet fájlba írja a tesztelendő program; a bemeneti állomány nevét első argumentumként kapja meg futtatáskor

`11` egy bemeneti állományhoz több kimeneti állomány tartozik, a bemeneti adatokat a standard inputról olvassa, a kimenetet a standard outputra írja a tesztelendő program

`12` egy bemeneti állományhoz több kimeneti állomány tartozik, a bemeneti adatokat fájlból olvassa, a kimenetet fájlba írja a tesztelendő program; a bemeneti állomány nevét első argumentumként kapja meg futtatáskor

`test?parm1` 2-es és 12-es típusú tesztelés esetén a bemeneti állomány neve a `?` sorszámú tesztesetnél.

`test?parm2` 2-es és 12-es típusú tesztelés esetén a kimeneti állomány neve a `?` sorszámú tesztesetnél.

`test?chk` 11-es és 12-es típusú tesztelés esetén a tesztelőprogram neve a `?` sorszámú tesztesetnél.

A `parm1` és `parm2` azok a nevek, amelyet a tesztelendő program kér/vár inputként, illetve amelyet írnia kell outputként. Nincs közük a feladat `test/` könyvtárában lévő állományokhoz, amelyeket mindig azonos módon kell elnevezni (a részleteket lásd a 5.4.3. alfejezetben).

A `chk` paraméter az ellenőrzőprogram nevét adja meg. Ennek a programnak egy futtatható állománynak kell lennie a PCRM programrendszer gyökérkönyvtárában. Arról, hogy hogyan kell kinéznie egy megoldásellenőrző programnak, lásd a 5.4.2. alfejezetet.

### 5.3.7. A `[contestant]` csoport

Itt tároljuk az információkat a versenyzőkről. Kétféle bejegyzés állhat ebben a csoportban: az egyik a versenyzőnek vagy csapatnak a nevét jelzi, a másik egy jelszót, amelyet a PCRMPOP3 program használ, amikor az email üzeneteket kezeli.

```
?=name
p?=passphrase
```

A `?` karakter helyére a versenyzők sorszáma kerül. A sorszámozásnak 1-től kell indulnia és folyamatosnak kell lennie. A jelszó nem tartalmazhat szóköz karaktert.

## 5.4. A környezet konfigurálása

Mielőtt a programok használatát ismertetnénk, néhány szóban összefoglaljuk, hogy hogyan kell konfigurálni a program környezetét. Az általános dolgok beállítására szolgál a `pcrm.ini` állomány, melyről már a 5.3. fejezetben szóltunk. A következőkben a könyvtárstruktúra felépítését, valamint a feladatok ellenőrzéséhez szükséges állományok elnevezési konvencióit mutatjuk be.

### 5.4.1. A könyvtárstruktúra

Minden egyes feladathoz létre kell hozni egy `problem?` nevű könyvtárat, ahol a `?` karakter helyére egy számot, a feladat sorszámát kell behelyettesíteni. A feladatok

sorszámozása 1-től indul. A `problem?` könyvtárba fognak a megoldások forráskódjai bemásolódni, itt fordítja le őket automatikusan a PCRM, és ebben a könyvtárban fogja őket futtatni is.

Minden könyvtárban kell lennie egy `test` alkönyvtárnak, amely a `test?.in` és `test?.out` állományokat tartalmazza. Itt a `?` karakter helyére a tesztesetek 1-től indexelt sorszáma kerül. Ha saját megoldásellenőrző programot használunk, akkor a kimeneti tesztállományokat nem használja föl a rendszer.

Amennyiben a POP3 klienst is használni szeretnénk, szükségünk lesz még egy globális könyvtárra, amit `contestant`-nak kell elnevezni, és amely 1-től sorszámozott alkönyvtárakat fog tartalmazni, minden versenyző számára egyet-egyet. Amikor a PCRMPOP3 letölti a megoldásokat, minden megoldást a megfelelő versenyző alkönyvtárába fog elhelyezni.

Ha a beérkező leveleket (amelyek egyébként nem törlődnek a szerverről sem) tárolni szeretnénk, szükségünk lesz még egy `pop3` könyvtárra is. Minden email eredeti formájában itt kerül majd tárolásra időbélyeggel és sorszámmal ellátva.

Így például egy 6 feladatot tartalmazó, 12 résztvevős verseny könyvtárstruktúrája a következő lehet:

```
pcrm/
 + contestant/
 |  + 1/
 |  + 2/
 |  + 3/
 |  ...
 |  + 12/
 + pop3/
 + problem1/
 |  + test/
 + problem2/
 |  + test/
 | ...
 + problem6/
 |  + test/
```

Ha az összes paramétert beállítottuk a `pcrm.ini` állományban, lefuttathatjuk a PCRM programot a `dircrt` opcióval. Ez létre fogja hozni a kívánt könyvtárstruktúrát, s nekünk ezután már csak annyi dolgunk marad, hogy elhelyezzük a megfelelő állományokat a tesztkönyvtárakba.

### 5.4.2. A feladatok konfigurálása

Minden feladatnak saját könyvtára van. E könyvtárak mindegyike tartalmaz egy `test` alkönyvtárat, amelybe a feladat bemeneti és kimeneti tesztállományai kerülnek. Miután a `pcrm.ini` állományban megadtuk, hogy hány tesztállomány tartozik a feladathoz, ezeket a

```
test?.in
test?.out
```

állományokat el kell helyezni a `test` alkönyvtárban. Ezeket az állományokat (az `.in` fájlokat) kell a versenyzők megoldásként beadott programjainak feldolgozniuk, eredményeiknek pedig meg kell egyezniük a bemeneti adatokhoz tartozó output állományokkal (az `.out` fájlokkal).

Mielőtt a PCRM lefuttatna egy versenyző által beadott programot, másolatot készít a tesztállományokról, így semmiképpen sem történhet adatvesztés a tesztelés során.

Ha saját ellenőrző programot használunk, a kimeneti állományokat nem használja a program.

#### Saját eredményellenőrző program használata

Előfordulhatnak olyan feladatok, melyeknek több helyes megoldásuk is van. Ilyen esetben gondoskodnunk kell saját eredményellenőrző programról, mert a beépített ellenőrzés nem tesz lehetővé többszörös kimenetellenőrzést.

Az ellenőrző programot, amelynek futtathatónak kell lennie, majd a PCRM program fogja elindítani. A PCRM gyökérkönyvtárában kell elhelyezni, és amikor a PCRM elindítja, három argumentumot kell átadni neki:

```
executable <testcasenumber> <testcaseinput> <programoutput>
```

Az első argumentum a teszteset száma, egy sorszám, amely az ellenőrzéshez szükséges.

A második argumentum annak a bemeneti állomány relatív elérési útvonala, amelylyel a felhasználói programot meghívták. Ez a bemeneti állomány az adott tesztesethez.

A harmadik argumentum a relatív elérési útvonala annak az állománynak, amelyet a felhasználói program állít elő kimenetként, és amelyről az ellenőrző programnak el kell döntenie, hogy helyes kimenete-e a megadott tesztesetnek.

Az ellenőrző programnak egy 0 és 127 közötti számmal kell visszatérnie, jelezve a pontszámot. A 0 jelzi a rossz választ. Ha pontszám alapú értékelést használunk, a visszatérési érték a tesztesetre adott pontszám legyen. Ha ACM stílusú az értékelés, vagy egyszerűen csak azt akarjuk jelezni, hogy jó vagy rossz-e a megoldás, 1 legyen a visszatérési érték siker esetén, 0 pedig egyébként.

### 5.4.3. A forrásállományok elnevezése

A feladatok könyvtárainak kell tartalmazniuk azokat a forrásállományokat is, amelyeket a versenyzők hoztak létre. Mindegyik forrásállomány nevének

```
p?c?.ext
```

alakúnak kell lennie, ahol a `p` karakter után a feladat sorszáma áll, a `c` karaktert a versenyző sorszáma követi, míg az `ext` a forráskód kiterjesztése. A sorszámozás mind a feladatok, mind a versenyzők esetében 1-től kezdődik. A kiterjesztés a `c`, `cpp`, `asm`, `java`, `pas`, `bas` vagy `cs` karaktersorozatok valamelyike lehet. Egy szabályos név lehet például a következő:

```
p6c17.cs
```

Ez a 17-es sorszámú versenyzőnek (vagy csapatnak) a 6-os sorszámú feladatra beadott megoldása, amely C# programozási nyelven íródott.

Ha a PCRMPOP3 programot használjuk, akkor az automatikusan a helyes névvel másolja be a forrásállományt a megfelelő könyvtárba, nekünk nem kell az elnevezésekkel bajlódni. Egyébként viszont – bizonyos levelezőrendszerek használata esetén – fontos, hogy már maguk a versenyzők is helyesen nevezzék el a forrásokat.

## 5.5. A programok használata

Ebben a fejezetben a PCRM és a PCRMPOP3 programok használatával ismertetjük meg az olvasót.

### 5.5.1. A PCRM használata

Minden egyes verseny indítása előtt módosítanunk kell a `pcrm.ini` állományt, mert ez az állomány tartalmazza azokat az információkat, amelyek a versenyzők számára, a feladatok számára és más fontos, a versennyel kapcsolatos dolgokra vonatkoznak (lásd a 5.3. fejezetet).

A PCRM futtatásához a parancssorba a következőket kell begépelni:

```
pcrm <options> <command> [<command arguments>]
```

Legegyszerűbben mindenféle parancssori argumentum felsorolása nélkül, egyszerűen csak – az operációs rendszertől függően – a `pcrm.exe` vagy a `./pcrm` parancsok begépelésével indíthatjuk el a programot. Ekkor egy rövid áttekintést fogunk látni a futtatási opciókról és az elérhető parancsokról.

#### Parancssori opciók

A program működése az egyes opciók használatával a következőképpen alakul:

- `-q` Csendes üzemmód, amelyben nem ír a képernyőre.
- `-w` Nem írja ki a logó szövegét.
- `-l` Naplót (`pcrm.log`) készít.
- `-t` Időbélyeggel lát el minden bejegyzést a képernyőn vagy a naplóban.

Például a

```
pcrm -q -l
```

parancs futtatásával a program a képernyőre nem fog írni, csak naplót fog készíteni.

### Programparancsok

Az alábbiakban a PCRM által elfogadott parancsokat mutatjuk be. Amelyik parancs paraméterezhető, ott a paramétereit is jelezzük:

`ver` A program verzióját írja ki.

`ppl` Kilistázza a versenyzőket.

`dircrt` A `pcrm.ini` állomány alapján létrehozza a könyvtárszerkezetet.

`chk <problem> <contestant>` Ellenőrzi az adott versenyzőnek az adott feladatra beadott megoldásait.

`chkcon <contestant>` Ellenőrzi az adott versenyző összes beküldött megoldását.

`chkprb <problem>` Ellenőrzi az adott feladatra beadott összes megoldást.

`chkall` Az összes versenyző összes beadott megoldását ellenőrzi.

`clr` Törli az eredményeket.

`clrtst` Törli a tesztelések eredményeit.

`res` Kiírja az eredményeket.

`rescon <contestant>` Kiírja az adott versenyző eredményeit.

`reshtm` HTML kimenetet állít elő.

`resgfx` HTML kimenetet állít elő, melyhez a `htm/` könyvtárban lévő grafikus elemeket használja fel. (További részletek a HTML kimenetről az 5.3. fejezetben.)

### Eredménygenerálás

A PCRM az eredményeket különféle formákban tudja előállítani. A legegyszerűbb esetben egy szöveges kimenetet generál, amelyet a standard kimeneten (a képernyőn) jelenít meg. Van lehetőség különböző minőségű HTML kódok előállítására is: kaphatunk egyszerű szöveges vagy grafikus elemeket tartalmazó HTML oldalakat is.

A 5.3.2. fejezetben tekinthetjük meg azoknak az opcióknak a leírásait, amelyekkel lehetőségünk van a program által generált kimenet testreszabására. Számos olyan opciót találhatunk köztük, amelyeket a beállítások finomhangolására használhatunk fel.

A program azt is lehetővé teszi, hogy csak a zsűri számára generáljunk eredményeket. Ez persze azt is jelenti, hogy ezeket az eredményeket csak a zsűri láthatja, mivel ekkor az összeredmények generálása egy bizonyos előre beállított idő után már nem folyik tovább. Az ACM versenyeken szokás például az, hogy a verseny utolsó órájában az összeredményeket már nem frissítik a versenyzők és a külső szemlélődők számára, így is izgalmasabbá téve a verseny utolsó szakaszát. Valójában azért teszik ezt a szervezők, mert nem szeretnék, ha az eredményhirdetés előtt kiderülne a győztes kiléte.

A fenti lehetőségek miatt a zsűri és a versenyzők számára generált eredményeket külön-külön állományokban állítja elő a program, mindegyikben az adott szereplő számára szükséges adatokkal.

### 5.5.2. A PCRMPOP3 használata

A PCRMPOP3 programmal teljesen automatizálhatjuk a kiértékelési folyamatot. A program képes a beérkező emailek ellenőrzésére, a feladatok tárolására, amint azok beérkeznek, a PCRM program elindítására, amellyel a beérkezett feladatokat azonnal ellenőrizni lehet, valamint automatikusan újra tudja generálni a kimeneti html állományt az eredményekről. Tréfásan fogalmazva ez annyit jelent, hogy csak el kell indítani, és nem kell törődnünk semmivel, csak hátradőlnünk, és pihennünk, amíg a verseny véget nem ér.

A kapott leveleket bármilyen emailcímről be lehet küldeni. A levélnek **pontosan egy** csatolmányt kell tartalmaznia, amely a forráskódot tartalmazza. Ha a levél nem ilyen, a PCRMPOP3 egyszerűen nem foglalkozik vele. A forráskód nem állhat a levél törzsében sem, mindenképpen a **levélhez kell csatolni**. Fontos megszorítás még, hogy a forráskódot pontosan úgy kell elnevezni, ahogyan azt a PCRM beállításai megkövetelik. A forráskódok elnevezéséről lásd a 5.4.3. fejezetet.

Ha erős formájú tárgysort használunk az emailekben (lásd a 5.3.3. fejezetet a tárgysor beállításáról), akkor a csatolt állomány neve bármi lehet, de a kiterjesztésének a programozási nyelvhez illőnek kell lennie (például ha a kódot C++ nyelven írták, akkor a kiterjesztésnek `cxx`-nek vagy `cpp`-nek kell lennie. Ha nem használunk tárgysort vagy gyenge tárgysort használunk, akkor a csatolt állományt a 5.4.3. fejezetben leírtak szerint kell elnevezni.

A forrásállományt mindig csatolva kell beküldeni, nem szabad beágyazni a levél törzsébe. Ha mégis oda kerülne, a PCRMPOP3 nem fogja felismerni. A csatolás érkezhet kódoltan is, a PCRMPOP3 képes a base64 kódolású csatolt állományokat is kezelni.

#### Parancssori opciók

Hasonlóan a PCRM programhoz, a PCRMPOP3 programnak szintén megadhatunk parancssori opciókat a tényleges parancs előtt.

A PCRMPOP3 futtatásához a parancssorba a következőket kell begépelni:

```
pcrmpop3 <options> <command>
```

Az opciók a következők:

`-w` Nem írja ki a logó szövegét.

`-l` Naplót (`pcrmpop3.log`) készít.

`-t` Időbélyeggel lát el minden bejegyzést a képernyőn vagy a naplóban.

Például a

```
pcrmpop3 -l
```

parancs futtatásával a program naplót fog készíteni.

Ezek a beállítások – a `-t` opció kivételével – nincsenek hatással a PCRM programra, amely a PCRMPOP3 programból kerül hívásra.

### Programparancsok

Az alábbiakban felsoroljuk azokat a parancsokat, amelyeket megadhatunk a PCRM-POP3 programnak a parancssorban. Amennyiben ciklikus paranccsal (`listen` vagy `autoall`) indítjuk a programot, akkor szabályosan csak a Ctrl-C billentyűkombináció használatával fejeztethetjük be a működését.

`check` Letölti az emaileket, tárolja a forráskódokat, de nem ellenőrzi őket a PCRM-mel.

`listen` Megadott időközönként letölti az emaileket, tárolja a forráskódokat, de nem ellenőrzi őket a PCRM-mel.

`auto1` Egyszerű automata zsűrizést hajt végre: letölti az emaileket, tárolja a forráskódokat, meghívja a PCRM programot a megoldások kiértékelésére, majd előállítja az aktuális HTML eredménylistát.

`autoall` Teljesen automatikus zsűrizést hajt végre. Megadott időközönként letölti az emaileket, tárolja a forráskódokat, és meghívja a PCRM programot a megoldások kiértékelésére, végül előállít egy kimenetet a PCRM `resgfx` parancsával (a kimenet generálásának a részleteit lásd az 5.3.2. fejezetben).

Az alábbiakban látható a PCRMPOP3 program futásának kimenete, ha a programot az `autoall` paranccsal futtatjuk, amely a hagyományos futtatási mód ACM-stílusú versenyek esetén:

```
acmjury@it:~$ ./pcrmpop3 autoall
PCRM (Programming Contest Result Manager) POP3 Client, version 1.9.1
Linux build, Mar 12 2004 14:41:57
Copyright (C) 2003-2004 by Lenard Gunda

Running in a loop - waiting time is 60 seconds

Email check! (connecting to server localhost)
Connected and logged in to POP3 Server
There is 1 message in mailbox (size: 514 bytes)
Disconnecting from POP3 Server
Result updated
Wait ...
Email check! (connecting to server localhost)
Connected and logged in to POP3 Server
There are 4 messages in mailbox (size: 9474 bytes)
New message detected (index:2 / uid:aaed734d9560913c89828c13db395885)
 + Attachment: p1c1.c
 + Storing attachment for: Problem 1 - Contestant 1
 + Writing file: "contestant/1/p1c1.c.1079488974.175127"
 + Updated "problem1/p1c1.c"
New message detected (index:3 / uid:8cec411abc2df1611e7bce2a3d3e27ab)
 + Attachment: p1c2.c
 + Storing attachment for: Problem 1 - Contestant 2
 + Writing file: "contestant/2/p1c2.c.1600234907.175140"
```

```
 + Updated "problem1/p1c2.c"
New message detected (index:4 / uid:5fde5a57d456b6bcd624f2769ff8c206)
 + Attachment: p1c3.c
 + Storing attachment for: Problem 1 - Contestant 3
 + Writing file: "contestant/3/p1c3.c.372261666.175156"
 + Updated "problem1/p1c3.c"
Disconnecting from POP3 Server
Checking received results with PCRM executable
======================================================================
Checking problem 1 for contestant #1 (Gunda Lenard)
 * Executing compile: gcc -lm -w -O2 -o p1c1 p1c1.c
 + Compiled successfully!
Contestant #1 / Problem 1 / Testcase 1
 * Executing: sudo /usr/sbin/chroot . ./p1c1
 - Failed: Runtime Error (RE)!
 - Skipping further tests! (break-break)
Checking problem 1 for contestant #2 (Panovics Janos)
 * Executing compile: gcc -lm -w -O2 -o p1c2 p1c2.c
 + Compiled successfully!
Contestant #2 / Problem 1 / Testcase 1
 * Executing: sudo /usr/sbin/chroot . ./p1c2
 - Failed: Wrong Answer (WA)!
 - Skipping further tests! (break-break)
Checking problem 1 for contestant #3 (Kosa Mark)
 * Executing compile: gcc -lm -w -O2 -o p1c3 p1c3.c
 + Compiled successfully!
Contestant #3 / Problem 1 / Testcase 1
 * Executing: sudo /usr/sbin/chroot . ./p1c3
 + Accepted! +(1)+
======================================================================
PCRM checks complete
Result updated
Wait ...
```

Amint az a kimenet végén látható, a program működése még nem fejeződött be ezen a ponton, hanem várja a további megoldásokat...

### Erős tárgysorok

Ha az erős tárgysorok használatát választjuk, minden beérkező emailnek tartalmaznia kell egy tárgysort, amelynek a következőket kell tartalmaznia:

```
PCRM:<contestantnumber>:<problemnumber>[:<passphrase>]
```

Az erős tárgysort a tárgysorra vonatkozó 1-es és 11-es beállítások esetén kell használni. A 11-es érték esetén meg kell adni a jelszót is, míg az 1-es értéknél ez nem szükséges. Ettől eltekintve a tárgysor felépítésének mindkét esetben a fent megadott formájúnak kell lennie. Az egyetlenegy csatolt állománynak a neve nem lényeges, mert

ebben az esetben a tárgysort használja a program a versenyző és a probléma azonosítására. A programozási nyelvet viszont a csatolt állomány kiterjesztése fogja jelezni a PCRM számára.

#### Gyenge tárgysorok

Gyenge tárgysort akkor használunk, amikor a tárgysor beállítása 12-es értékű. Ebben az esetben a tárgysornak csak a versenyző jelszavát kell tartalmaznia.

A versenyzők jelszavait a `pcrm.ini` állomány tartalmazza. A jelszavak nem tartalmazhatnak szóköz karaktert.

## 5.6. Egyéb tudnivalók

Ebben a fejezetben a PCRM programmal kapcsolatos egyéb, az eddigiekben el nem hangzott érdekes és fontos tudnivalókat gyűjtöttük össze.

### 5.6.1. A Windows-verzió

Amikor a program futási hibával terminál, a Windows egy párbeszédablakot dob fel. A programot ellenőrző személynek manuálisan kell bezárnia ezt az ablakot. Még mielőtt ezt megtenné, a PCRM időtúllépést állapíthat meg a feladatra, és ha az ablak nem záródik be, az megakadályozza további függvények megfelelő működését. Ez egy Windows-ra jellemző korlátozás. Ha tudjuk és szeretnénk, kapcsoljuk ki a futási hibákat jelző ablak felbukkanását.

A problémát az okozza, hogy amikor a PCRM elindít egy másik folyamatot, nem tudja detektálni, hogy történt-e hiba, amíg a párbeszédablak nincs bezárva. Így aztán időtúllépést detektál, miközben az ablak aktív, és így futási hiba helyett időtúllépést fog jelezni.

Linux operácis rendszer alatt ez a jelenség nem bukkan fel, ott a program rendben működik, a szegmentálási hibákat és az egyéb futási hibákat rendben felismeri.

### 5.6.2. A programba kódolt beállítások

A fordítási idő 30 másodpercre van korlátozva. Egyetlen program sem fordulhat 30 másodpercnél hosszabb ideig.

Ha több jó megoldás is létezik egy feladathoz, és saját eredményellenőrző programot használunk, az eredményellenőrző program nem futhat tovább 30 másodpercnél.

A fenti értékek `#define` utasítással definiált nevesített konstansok a forráskódban.

### 5.6.3. Az eredmény ellenőrzése

Miután a program lefutott, a PCRM összehasonlítja a program kimenetét az előre megadott kimeneti állománnyal. Az állományok összehasonlítása sorról sorra történik. A sorok végén található szóköz karaktereket a program figyelmen kívül hagyja. A versenyző kimeneti állománya a legeslegvégén tartalmazhat egy extra üres sort (az előre

megadott kimeneti állományhoz képest). Ezt leszámítva a két állomány tartalmának meg kell egyeznie.

A sorvégi `"\r\n"` és `"\n"` karaktersorozatokat a program egyformán dolgozza föl.

Az előzőekben végig állományokról beszéltünk, mivel a kimenet mindig egy állományba íródik, míg ha a programnak a standard outputra kell írnia, akkor a PCRM saját maga kezeli a standard handle átirányítását.

### 5.6.4. Webes felület

A PCRM program nem rendelkezik webes felülettel, nem kínál lehetőség arra, hogy a versenyzők bejelentkezzenek, és feltöltsék megoldásaikat HTTP protokollon keresztül egy böngésző ablakában. Mindazonáltal nem lenne nehéz egy ilyen webes felületet készíteni a jelenlegi verzió adta lehetőségekre alapozva.

A felületnek csak az állományok feltöltését kellene kezelnie, és aztán a feltöltött állományt egy mailcímre továbbítania, amelyet már a PCRMPOP3 figyel. A PHP például könnyen használható interfészeket nyújt emailek postázására, sőt PHP-ben akár egy olyan webes alkalmazás is készíthető, amely kapcsolatot kínál a PCRMPOP3 számára.

## 5.7. A PCRM az oktatásban

### 5.7.1. *Programozás 1* kurzus – házi feladatok kiértékelése

A Debreceni Egyetemen a 2003/2004-es tanév tavaszi félévében a gyakorlatban is alkalmaztuk a PCRM programot. Ez alkalommal a beérkezett megoldások kiértékelését és az eredmények közzétételét a beküldési határidők lejárta után, offline módon végeztük el. Informatika szakos hallgatóinknak 33 különböző programozási feladatot kellett megoldaniuk, hogy teljesítsék a *Programozás 1* kurzus gyakorlati követelményeit.

A feladatok témakörei a következők voltak:

1. *Egyszerű aritmetikai feladatok*: A programnak a bemeneten megadott számok függvényében kell előállítania a kimenetét.

2. *Sztringfeldolgozás*: A programnak a bemeneti karakterfolyamot kell feldolgoznia. A kimenetre minden bemeneti sor esetén pontosan egy sort kell írni.

3. *Különféle adatszerkezetek elemi algoritmusai*: Ezeknél a problémáknál a programnak nagy számú adatot kell feldolgoznia. A két legfontosabb megoldandó művelet a keresés és a rendezés.

4. *Saját függvények használata*: A programban előre specifikált függvényeket kell leprogramozni.

5. *Szöveges állomány feldolgozása*: Ennél a feladattípusnál a programnak a bemenetet szöveges állományból kell olvasnia és/vagy a kimenetet szöveges állományba kell írnia.

| *Feladatsor sorszáma* | *Témakör* | *Feladatok száma* | *Kiértékelések száma (elfogadott/összes)* |
|---|---|---|---|
| 1 | egyszerű aritmetikai feladatok | 7 | 1504 / 2937 |
| 2 | sztringfeldolgozás | 6 | 1486 / 2138 |
| 3 | elemi algoritmusok | 5 | 869 / 2855 |
| 4 | felhasználói függvények | 10 | 1807 / 2318 |
| 5 | állománykezelés | 5 | 339 / 3621 |

5.1. táblázat. Statisztika a *Programozás 1* kurzus házi feladatainak a kiértékeléséről

**Az eredmények elemzése**

A 5.1. táblázatban szereplő adatok elemzéséhez figyelembe kell vennünk a következőket:

- a feladatsorok kiírása (közzététele) a táblázatban előírt sorrendben, nagyjából 2–3 hetes időközönként történt;
- minden beküldött feladat értékelése aszerint történt, hogy az a helyes (elvárt) kimenetet állította-e elő, vagy sem;
- arról, hogy a beküldött program az elvárt kimenetet állította-e elő, a beküldő hallgatónak semmilyen információja nem volt;
- a hibás kimenetet előállító kódok helyett – az offline kiértékelésből adódóan – nem volt lehetőség új, javított kód beadására, bár egy-egy feladatra több kódot is be lehetett küldeni, melyek közül az utolsó eredményét vettük figyelembe;
- a feladatokra alkalmanként kb. 400 hallgató küldte be megoldásait;
- a kurzus hallgatói többségükben először hallgatták a tantárgyat, először vettek részt a gyakorlati kurzusokon, és először találkoztak a C programozási nyelvvel, amelyen a megoldásokat be kellett küldeniük.

A táblázatbeli adatokból – ismerve a kiértékelő program erényeit és gyengeségeit – az alábbi következtetéseket vonhatjuk le:

- a beküldött kódok 43%-át a PCRM program helyesnek fogadta el;
- az átlagos elfogadási aránynál jobb eredményeket adó három kategóriában (az egyszerű aritmetikai feladatoknál, a sztringfeldolgozásnál és a felhasználói függvények használatakor) a feladatok a hallgatóság felkészültségének, programramozási rutinjának megfelelő nehézségűek voltak;
- az elemi algoritmusok témakörében kiírt feladatok ellenőrzésénél a kiértékelő program „kegyetlen” volt: az időtúllépő, lassú algoritmusokat nem fogadta el;
- az állományok kezelését a PCRM nem tudta hatékonyan ellenőrizni, ezen feladattípusnál egyébként utóbb kézi ellenőrzésre is sort kellett keríteni.

### 5.7.2. *Mesterséges intelligencia 1* – programozó háziverseny

A PCRM programot a 2005/2006-os tanév tavaszi félévében a *Mesterséges intelligencia 1* tárgy gyakorlati kurzusain meghirdetett programozó háziverseny kiértékelésére használtuk. Ez a verseny egy nyílt verseny volt, egy előre megadott határidőig kellett két feladat megoldását beküldeni a zsűri címére.

A versenyre minden olyan hallgató kapott egy felhasználónevet és egy jelszót, aki abban a félévben felvette a tárgy gyakorlati kurzusainak valamelyikét. Ez kb. 160 hallgatót jelentett, ennyien próbálkozhattak a feladatok beküldésével. A feladatokat nagyjából másfél heti gondolkodási idő után 2006. május 3-án 9 órától 21 óráig lehetett beküldeni emailben. A verseny győztesének azt tekintettük, aki a két feladat közül bármelyikre minden más versenyzőnél hamarabb olyan kódot küld be, amely helyes eredményt szolgáltat, és mindezt a lehető leggyorsabban teszi.

A feladatok kiértékelése a határidő lejárta után kezdődött, beérkezési sorrendben. A rendszer 198 beküldést regisztrált, amit már menet közben is kevesellтünk, és ebből mindössze 9-et fogadott el helyesnek. A viszonylag alacsony beküldési számért egyrészt a kari levélszemét-szűrő szoftverek, másrészt maguk a levélküldő programok, harmadrészt pedig a PCRMPOP3 program voltak a felelősek. Ez utóbbi kettő szorosan összefüggött egymással: egyes levélküldő programok a levéltörzset reklámanyagokkal egészítették ki, amelyet a PCRMPOP3 program – helytelenül – a kód részének tekintett, és ezt továbbította a PCRM program felé. A fordítóprogramok persze ezeket a kódokat már lefordítani sem tudták. Ezzel szemben a kiértékelő szoftver nem hibázott, a kiértékelt kódok között valóban csak ennyi helyes megoldás volt.

A győztesek jutalma a tárgy gyakorlati aláírásának a megszerzése volt, minden korábbi és további számonkérés eredményétől függetlenül.

### 5.7.3. Versenyek lebonyolítása a PCRM-mel

A PCRM rendszer igazi terepe az online versenyek szervezése és lebonyolítása. Megszületésétől kezdve öt éven keresztül ennek a programnak a segítségével rendeztük meg és bonyolítottuk le a helyi (informatika intézeti, majd informatikai kari) programozó versenyeket, mind az őszi, mind a tavaszi félévekben. A rendszer arra is alkalmas volt, hogy egyszerre, egy időben rendezhettünk vele egyéni és csapatversenyt is. A programot – több sikeres verseny után – 2005. május 3-án tanszéki szemináriumon keretében is bemutattuk az Információ Technológia Tanszék munkatársainak és az érdeklődőknek.

Ellentétben a PCRM program oktatásban történő felhasználásával, versenykörülmények között a fő cél, hogy valós időben történjen meg a versenyzők által beküldött megoldások kiértékelése és az eredménylista előállítása. További különbség, hogy a versenyek feladatainak témakörei némiképp eltérnek a korábbiaktól. A leggyakrabban előforduló feladattípusok a következők:

- sztringfeldolgozás, mintaillesztés,
- keresések, rendezések,
- aritmetikai és algebrai feladatok,

- kombinatorikai feladatok,
- számelméleti feladatok,
- backtracking,
- gráfalgoritmusok,
- dinamikus programozás,
- geometriai feladatok.

A felsorolt feladattípusok lefedik a nemzetközi ACM programozó versenyeken általában megoldásra kijelölt feladatok típusait. Ezen feladatok bemeneti és kimeneti adatainak specifikációját ismerve könnyedén be lehet a PCRM rendszert úgy állítani, hogy alkalmas legyen akár önmagában, akár valamilyen külső ellenőrző program segítségével egy ilyen jellegű verseny teljes lebonyolítására.

A programot a következő intézeti és kari programozó versenyeken használtuk fel:

- 2002. április 14., a Matematikai és Informatikai Intézet programozó versenye (magyar)
- 2003. május 4., a Matematikai és Informatikai Intézet programozó versenye (magyar)
- 2003. május 16., Bláthy programozó verseny, Bláthy Ottó Villamosipari Szakközépiskola, Miskolc (magyar)
- 2004. április 18., az Informatikai Intézet programozó versenye (magyar)
- 2004. május 9., az Informatikai Intézet programozó versenye (magyar)
- 2005. május 3., az Információ Technológia Tanszék szemináriumának keretében rendezett bemutató
- 2006. április 2., az Informatikai Kar programozó versenye (magyar)
- 2006. május 3., a *Mesterséges intelligencia 1* tárgy keretein belül meghirdetett háziverseny
- 2006. május 7., az Informatikai Kar programozó versenye (magyar)
- 2006. október 8., az Informatikai Kar ACM programozó versenye (angol)
- 2007. április 15., az Informatikai Kar egyéni programozó versenye (magyar)
- 2007. április 15., az Informatikai Kar programozó csapatversenye (magyar)

Természetesen programozó versenyeket az ezt követő időszakban is szerveztünk, ám a versenyek lebonyolításához 2007 októberétől kezdve – kísérletképpen – más szoftvereket használtunk. Több versenyen is kipróbáltuk a California State University, Sacramento (CSUS) által 1989-től fejlesztett PC$^2$ szoftvert [3], de ennek 8-as verziói a kari számítógépes hálózat portjainak nehézkes beállításai miatt nem váltották be a hozzájuk fűzött reményeket.[5]

A mi programunk nagy előnye a PC$^2$-vel szemben az volt, hogy nem kellett hozzá speciális portokat megnyitni a hallgatói számítógépes hálózatot felügyelő tűzfalon, hiszen a PCRM a hagyományos és jól ismert levelező és internetes protokollokat használja a működése során. A telepítése is mindössze a konfigurációs állományának és a feladatokhoz tartozó könyvtárstruktúrának a helyes beállításaiból áll, ezek után azonnal működőképes.

[5] Igazság szerint visszacsöppentünk vele abba az időbe, amikor azért kellett izgulnunk egy-egy verseny közben, hogy valós időben eljuttathassuk a feladatok kiértékelésének eredményét a versenyzőkhöz.

6. FEJEZET

# Összefoglalás

Dolgozatomban bemutattam a Debreceni Egyetem informatikus hallgatóinak képzésében szereplő tárgy, *A mesterséges intelligencia alapjai* gyakorlati foglalkozásának keretein belül oktatott témaköröket, az állapottér-reprezentációs technikát, a reprezentációs gráfban megoldást kereső rendszereket és a teljes információjú, véges, determinisztikus, zérusösszegű kétszemélyes játékok reprezentálásának módszereit, valamint az ilyen típusú játékokhoz kapcsolódó lépésajánló algoritmusokat.

Mivel a tárgy több alapkurzus (*Az informatika logikai alapjai*, *Magas szintű programozási nyelvek*, *Adatszerkezetek és algoritmusok*) ismereteire épül, a dolgozatban az informatikai világban manapság széles körben elterjedt objektumorientált szemléletmód alkalmazásával egy olyan új megközelítését javasoltam a fent említett témaköröknek, amely egyfelől integrálja az alaptárgyak keretében elsajátított készségeket a mesterséges intelligencia algoritmusainak a helyes és alapos megértésében, másfelől kiindulópontja lehet az ezen algoritmusokkal megoldható feladatípusok további tanulmányozásának.

A dolgozat 2. fejezetében láthattuk, hogy milyen szerepet játszanak az informatika logikai alapjai az állapottér-reprezentációk elkészítésében, és megfigyelhettük az ugyanazon feladat különböző megközelítéseiből adódó reprezentációbeli különbségeket (különböző jellemző tulajdonságok és eltérő operátorok). Két példán keresztül mutattam meg a reprezentációk objektumorientált szemléletmódú modellezésének lehetőségeit, az ez alapján készült programkódokban pedig azt, hogy mennyire erős a kapcsolat a reprezentáció formulái és a belőlük készülő programkódok között. Könnyen beláthattuk, hogy a reprezentációk kódolása tulajdonképpen egyszerű programozási sablonok alapján történhet.

A 3. fejezetben a megoldást kereső rendszerek és az egyes keresőalgoritmusok világát tártuk fel részletesebben, szintén objektumorientált megközelítésben. Ennek eredményeképpen ismertettem két, egymással párhuzamosan felépülő és egymással szoros kapcsolatban álló osztályhierarchiát, a reprezentációs gráfok csúcsainak, valamint az őket használó keresőknek a hierarchiáját. Az egyes keresőosztályoknak számos paraméterezési opciót állítottunk be, lehetővé téve, hogy a probléma részletes analízise után a felhasználók (programozók) a lehető legjobb feltételekkel használhassák ki az algoritmusokban kódolt lehetőségeket, figyelembe véve a probléma reprezentációs gráfjának a tulajdonságait. A fejezet végén a korábbi példákon keresztül mutattam be a javasolt

programcsomag működését, majd a 3.3. alfejezetben ismertettem a programcsomaggal az oktatásban szerzett tapasztalatainkat.

A 4. fejezetben a gyakorlati foglalkozások harmadik nagy témaköréhez, a kétszemélyes játékok reprezentációjához javasoltam egy újabb objektumorientált modellt. Itt elsősorban a játékok absztrakt osztálya és a hozzá kapcsolódó lépésajánló algoritmusok alkották a modell fő vázát. A használatukat egy konkrét játék példáján keresztül mutattam be, amelyet az ACM nemzetközi programozó verseny 2002-es, Varsóban megrendezett közép-európai selejtezőjének feladatsorából választottam. Tettem mindezt azért, hogy összehasonlíthassuk az általam javasolt megoldás hatékonyságát a versenyfeladat megoldásához felhasználható egyéb módszerekkel (itt például a dinamikus programozással). A kapcsolat a két módszer között kölcsönös volt: a lépésajánló algoritmusok segítségével ötletet meríthettünk a feladat sikeres megoldásához, majd a sikeres megoldás algoritmusát beépíthettünk a lépésajánló algoritmusok hasznosságfüggvényeibe.

Az 5. fejezetben bemutattam az elmúlt években kifejlesztésre került, és a különböző helyi szervezésű programozó versenyeinken számtalanszor használt Programming Contest Result Manager (PCRM) szoftverrendszert. A programnak a versenyeken túl nagy hasznát vettük speciális, tantárgyakhoz köthető háziversenyeinken is, valamint egyes tárgyakból (*Magas szintű programozási nyelvek* a kiadott házi feladatok ellenőrzésénél is.

Összességében a dolgozatban javasolt objektumorientált szemléletmód bevezetése *A mesterséges intelligencia alapjai* című tárgy gyakorlatain több mint sikeresnek mondható. A hallgatói visszajelzések végig pozitívak voltak vele kapcsolatban, egyes képzési formákban pedig érezhetően jobb teljesítmény nyújtotak a hallgatók ezzel a módszerrel, mint korábban e nélkül. Az előtanulmányok során elsajátított ismeretek és a gyakorlati követelmények további finomítása azonban vélhetően még eredményesebbé teheti majd az új módszerekkel történő oktatást.

7. CHAPTER

# Summary

## 7.1 Introduction and Motivation

In the world of informatics, object-oriented approach has become very popular in the last decade. This method apparently must have come up also in the field of education. In my thesis, I presented the potential of teaching object-oriented aspect of programming at universities and in talent-care programs. During my researches, I was focusing on the practical courses of the subject titled *Introduction to Artificial Intelligence*. This subject is one of the compulsory subjects of the Software Engineering BSc level curriculum at the University of Debrecen.

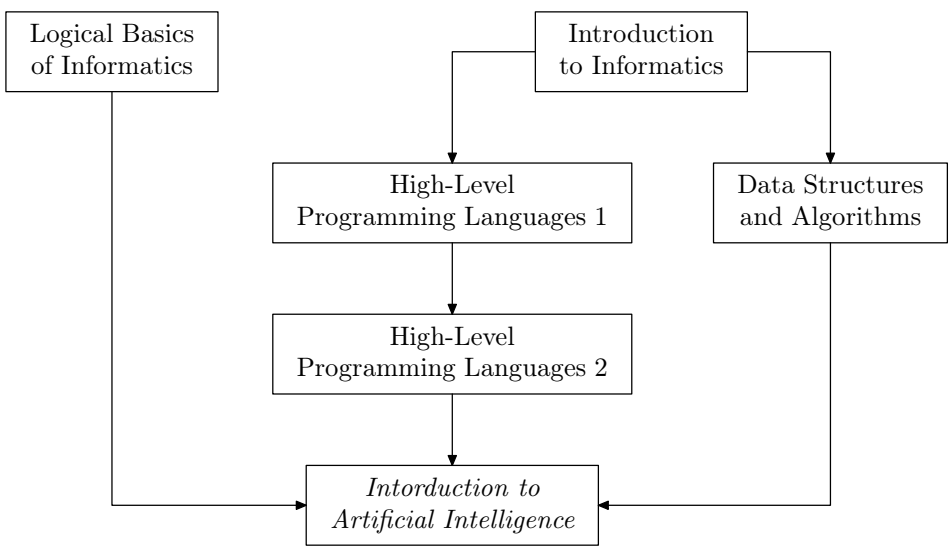



Figure 7.1. Prerequisite subjects of *Introduction to Artificial Intelligence*

The *Introduction to Artificial Intelligence* of the Software Engineering BSc level

curriculum at the University of Debrecen is based on the knowledge of four subjects which the students may have met during the preceding semesters. The relations of these subjects are shown in Figure 7.1.

The students may register to this subject in the fourth semester of the curriculum. Before this, they had fulfilled the two-semester programming course where they got acquainted with the C programming language and the basics of an object-oriented language (Java or C#), and they got over the course *Logical Basics of Informatics*. Apart from this, they could take part in lectures about data structures and algorithms. I would like to emphasize that students attending this course do not get in connection with the object-oriented world for the first time in this course, they already have experience with an object-oriented programming language.

The students learn about the classic research areas, basic methods and the most important results of AI in the lectures of this subject. Among the basic methods, they learn about the graph search algorithms in detail in the frame of this course.

For teaching graph search algorithms, first we have to represent the problem using state-space or problem-reduction representation. We give the appropriate graph representation in both cases. After analysing the structures of the search systems, we present the concrete procedures. We can classify the algorithms based on various aspects (non-modifiable—modifiable; non-informed—heuristic control strategies). We mainly deal with modifiable search algorithms (backtracking, graph search algorithms: breadth-first search, depth-first search, uniform cost search; heuristic graph search algorithms: best-first search, A algorithms). We also examine the algorithmic properties (completeness, soundness, complexity) of these procedures.

We talk about search algorithms using AND/OR graphs as generalizations of the previously mentioned algorithms used for problem-reduction representation. We give the concept of winning strategy of two-player zero-sum games of perfect information, and introduce methods to determine the probably best next move (minimax method with and without alpha-beta pruning).

The practical course of the subject *Introduction of Artificial Intelligence* covers two main topics: graph search algorithms and algorithms for two-player games. Both topics are very suitable for object-oriented approach and give numerous possibilities of optimization which are hidden by the generalizations of the object-oriented approach. This fact inspired me in case of certain problem classes to examine not only the object-oriented approaches which give us high-level abstraction but to find solution to the given problems using other methods, too.

Discovering the possibilities of optimization of the different problem classes is important also because the programmers can meet the algorithms learned in the frame of the subject at other areas of informatics (like web, mobile and database applications). Although the resources have escalated very rapidly nowadays, students can also learn to use them in a careful manner in these courses.

There are many ways to test the students' skills. For this purpose, we have organised programming contests a number of times in the previous years. The students can put their knowledge learned in the Artificial Intelligence courses to the test at contests in the frame of the subject on the one hand, and at contests organised by the Faculty

of Informatics[1] on the other hand. In the case of the latter, these are ACM style programming contests. Among the 7–10 problems to be solved during the five-hour programming time, the problem set often contains problems that can be solved using algorithms related to the topics of artificial intelligence.

Based on these facts, my aims were the following:

- to present the basic algorithms of artificial intelligence in the popular object-oriented approach;
- to integrate the state-space representation technique presented to the students into the object-oriented model;
- to discover the pros and cons of the object-oriented approach;
- to examine the applicability of "traditional" methods (the use of basic data structures, greedy algorithms, dynamic programming) in the case of special problem types;
- to analyse some problems of ACM international programming contests and to use them during the practical courses and for preparing students for programming contests;
- to develop an application which is capable of measuring and comparing the efficiency of the presented algorithms.

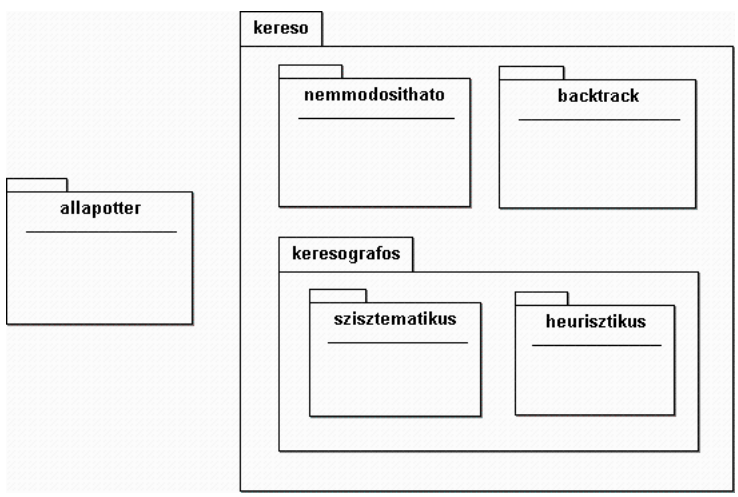



Figure 7.2. The package structure

[1] Formerly Institute of (Mathematics and) Informatics of the Faculty of Natural Sciences.

## 7.2 State-Space Representation and Search Algorithms Using Object-Oriented Approach

We have developed two Java packages for graph search algorithms which provide us with the possibility to solve any problem by using any search algorithm. As one of the requirements of the practical course of *Introduction to Artificial Intelligence*, the students have to write a program which may utilize these packages. One of the two Java packages contains classes regarding the state-space representation, the other of them contains classes regarding search algorithms.

In the package of the state-space representation two classes can be found (see Figure 7.3). The `Operator` class is the abstract superclass of the operators given in the various problems, while the `Allapot` class is the abstract superclass of the states of the problems. As the attributes of a state always depend on the actual problem, we define only one field in the latter class which is the collection of operators applicable to the states of the problem. Of course, independently of the problem, each state has to know if it is a goal state or not, if an operator is applicable to it or not, and the state which derives from it by applying an operator.

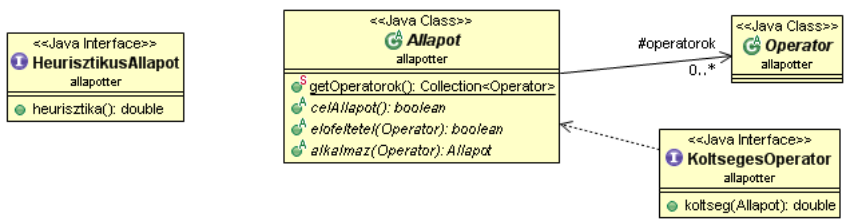



Figure 7.3. Classes and interfaces in the `allapotter` package

The heuristic search algorithms require states that have a further heuristic value in addition to the above mentioned attributes. The `HeurisztikusAllapot` interface helps us realize this because states implementing this interface will possess a method which provides a heuristic value. Similarly, the `KoltsegesOperator` interface will be implemented by operators having cost. These operators are utilized, for example, by the uniform-cost search algorithm or the A algorithm.

Classes in the package of the search algorithms are organised in two different hierarchies. One of these hierarchies serves for modeling the nodes of the representation graphs (see Figure 7.4), while the other is modeling the different search algorithms (see Figure 7.5).

Almost all of the information concerning the nodes can be found in the common superclass of the various node types, the `Csucs` class. This class stores the state represented by the node, the distance between the start node and the current node (the depth of the node) as well as the operator with which the state stored in the current node has been generated from the parent state. The derivatives of the `Csucs` class contain, besides these members, further fields required by the different search algorithms.

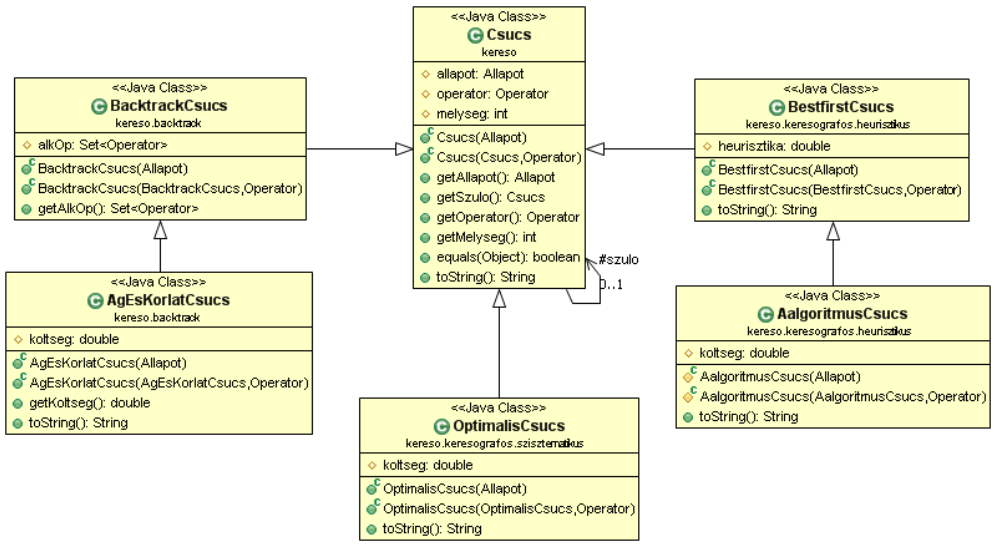



Figure 7.4. Class diagram of the representation graph nodes

The superclass of the search algorithms is the abstract `Kereso` class which does not implement its most important method, namely the one that controls the strategy of the search algorithm. This method will be implemented by the derived classes. At the same time, this class provides us with a means for storing the terminal nodes, printing the solutions, and setting the algorithm-independent properties of the search algorithms.

Among the new attributes of the `BacktrackKereso` class, we can mention the current path and options for cycle checking and path length bound. The `AgEsKorlatKereso` class differs from this only in that it uses cost bound instead of path length bound, and that this is no more an option but an integral part of the algorithm. The `KeresografosKereso` class uses two lists to register the nodes stored in the database. The derivatives of this class (the concrete graph-search algorithms) differ from one another in the control strategy and the classes of nodes stored in their databases.

## 7.3 Two-Player, Zero-Sum Games of Perfect Information Using Object-Oriented Approach

In the practical courses of *Introduction to Artificial Intelligence* we also talk about two-player, deterministic, finite, zero-sum games of perfect information. We have developed two Java packages which may be used to implement such a game. These packages contain classes regarding the state-space representation and algorithms for choosing the next move in a game (see Figure 7.6).

In the package of the state-space representation two classes can be found. The

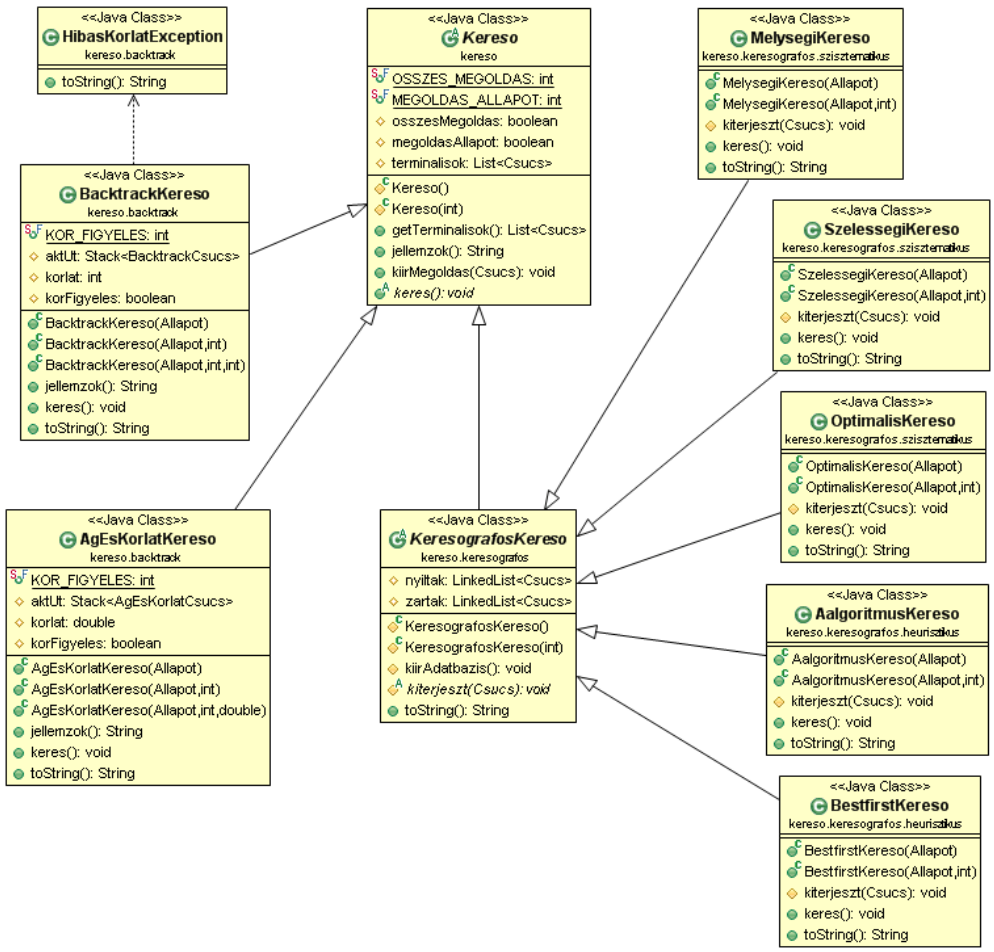



Figure 7.5. Class diagram of the search algorithms

`Operator` class is the abstract superclass of the operators given in the games, while the `Allapot` class is the abstract superclass of the states of the games. As the attributes of a state always depend on the actual game, we define only two fields in the latter class. One of these fields is the collection of the operators applicable to the states of the game, and the other one represents the player who is in turn in the current state of the game. Independently of the game, each state has to know if it is a final state, and if so what the result of the game is (which player wins if the game is not a tie). I can also examine if an operator is applicable to a state or not, and we can give the state which derives from the state by applying the operator. We can realize the human players' input by implementing the abstract `beker` method.

Classes implementing the algorithms related to games are placed in the `jatek`

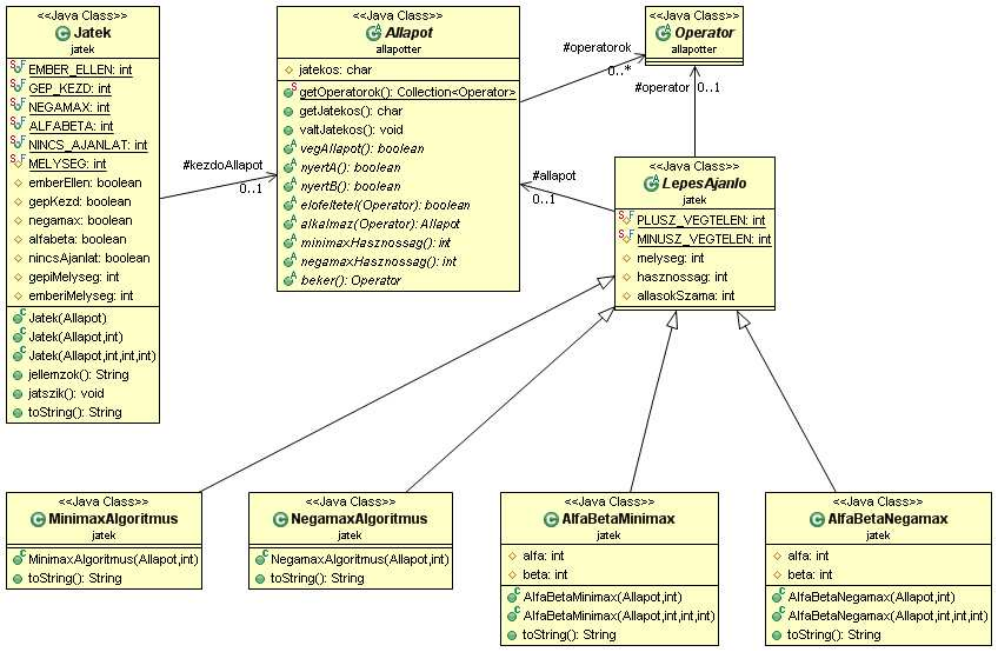



Figure 7.6. Class diagram of the algorithms for choosing the next move

package. The superclass of the algorithms choosing the next move is the abstract `LepesAjanlo` class. The fields of this class store the *state*, the *depth* of the search, the suggested *operator*, the *utility function's value* of the state reached by the suggested operator and the *number of states* evaluated during the search. These attributes will be computed by the concrete algorithms implemented in the derived classes. These algorithms can be compared to one another with the help of the number of evaluated states.

The concrete algorithms choosing the next move work recursively by instantiating objects of the contrete algorithm classes. Because of this, there was no need for other methods other than the constructors. Classes of algorithms using alpha-beta pruning have been extended by two fields which represent the *alpha* and *beta* values that belong to each node of the game tree.

The `jatszik` method defined in the `Jatek` class is responsible for controling the game. The body of this method contains a loop which runs until the current state instance is a final state. While in the loop, it displays the current state and determines which player is in turn. If we play interactively and the human player is in turn then it will read the player's move, otherwise it instantiates an algorithm object and determines the computer's move using this object. Afterwards it applies the operator representing this move in both cases and updates the current state.

We can customize the games and the algorithms choosing the next move at the time of the game's instantiation. We can choose whether a human player plays against

another human player or a computer player. In the latter case, we can give the player who will make the first move. We can set either minimax or negamax algorithm as the algorithm choosing the next move, and we can decide whether or not these algorithms should use alpha-beta pruning. We may ask for a suggestion for the human player. Similarly to the case of the computer's move, this suggestion will be computed by instatiating an algorithm object. The human player may either accept or reject this suggestion. Additionally, we can give different depths of search for computing the computer's move and the human's suggestion.

The strength of the suggestion is affected by two factors. One of these factors is the depth of the search and the other is the efficiency of the utility function applied to each state. Depending of the branching factor of the game tree, the time required to compute the next move may increase exponentially by increasing the depth of the search. That is why it is worth to write as efficient utility function as we can in the class of the actual game. If we can write a utility function that can determine for every state of the game how much that state is good for each of the players then there is no point in setting the depth of the search to a value greater than one. However, the coin has two sides: it may take a very long time to evaluate a state so what we really have to minimize is the product of the number of the evaluated states and the time spent for evaluating each state.

## 7.4 PCRM: An Evaluation Tool for Programming Contests

Precise checking of the solutions of the problems is of key importance for contests organised both in the frame of the subject *Introduction to Artificial Intelligence* and by the Faculty of Informatics. We developed a software called Programming Contest Result Manager (PCRM) to automatically evaluate the solutions. PCRM is a program that can be used to help judge a programming contest where the contestants must solve problems on a computer and then send the solutions to the online jury. It can also be used to judge an offline competition as well. The PCRM program can automatically check the sent solutions, i.e. compile them, run them, and check their output. It can work with programs that produce an output file and also with programs that read from the standard input and write to the standard output.

When the jury receives the source file from a contestant for a certain problem, it compiles it with the appropriate compiler and runs the program with test cases. For each problem, there is one or maybe more (but at least one) test case file, which the program must process, and generate a correct output. An output is correct, if it is the same as a pre-generated one, or if a program that verifies outputs, finds that the output is correct.

When using PCRM, there are test case files for each problem. However, each physical file can of course contain more test cases – this is the case in ACM mode, where all test cases must actually be in one file. This usually means that the input is built so that it can contain several test cases following one another.

PCRM helps you in automating the judging process by receiving the solutions

(automatically), compiling them, running them, evaluating the result, and keeping a track of events and generating (or showing) the complete result.

PCRM also includes a POP3 client program that can accept incoming solutions from the contestants via email and can fully automate the entire judging process.

The program can work in one of three modes: *problem based mode*, *score based mode* and *ACM mode*.

In *problem based mode*, the only measure is the number of problems solved. The individual test cases do not count, i.e. a problem is only solved if all test cases are solved correctly. You can have as many problems as you wish and each problem can have as many test cases as needed.

In *score based mode*, the winner is determined by the number of test case files for which the output of the program is correct. Every test case file that is evaluated is equal to a score of 1 (or more, if a solution checking program is used. In this case, the solution checking program determines the score for a test case file). That is, if we have 5 problems and 3 test case files for each, the maximum score is 15 and the minimum score is 0. You can have as many problems as you wish and each problem can have as many test cases as needed.

PCRM was originally developed for ACM-style competitions. In *ACM mode*, the contestant with the highest number of correctly solved problems wins. If there are two or more contestants with the same number of solutions, a score is calculated, and the contestant with the least score wins. The score is calculated as follows: For each problem, the contestant gets as many points as the number of minutes passed since the beginning of the competition. For every unsuccessful submission (compile error, runtime error, wrong answer, etc.) they also get penalty points. But penalty is only given if the problem is accepted in the end. So, if I sent in a solution to problem A after 80 minutes, got a wrong answer, sent it in after 84 minutes again (I am fast at finding the error), then again at 92 minutes, which is finally accepted, I will get $92 + 2 \times$ penalty points (assuming penalty is 20, that sums up to 132). If someone else sent in a correct solution after me with no prior incorrect submissions (say at 118 minutes in the competition), that contestant would still beat me. In ACM mode, there can only be one test case file for each problem. This does not mean one test case because these problems usually have a test case format which allows several test cases to be put into a single file. Even if you specify more test cases, PCRM will only work with one.

We used this software from 2002 during contests organised by the Faculty of Informatics and also for testing the homework of some subjects (*High-Level Programming Languages*, *Introduction to Artificial Intelligence*).

# Irodalomjegyzék


[1] The 2000's ACM-ICPC Live Archive.
URL http://acmicpc-live-archive.uva.es/nuevoportal/.

[2] Csákány Béla: *Diszkrét matematikai játékok*. Szeged, 1998, Polygon.

[3] CSUS Programming Contest Control ($PC^2$) Home Page.
URL http://www.ecs.csus.edu/pc2/.

[4] Fekete István – Gregorics Tibor – Nagy Sára: *Bevezetés a mesterséges intelligenciába*. 1990, LSI Oktatóközpont.

[5] Futó Iván: *Mesterséges intelligencia*. 1999, Aula.

[6] LightForce: FreeFall. URL http://lightforce.freestuff.gr/freefall.php. Online interactive puzzles, board and tile based games.

[7] Várterész Magda – Nagy Benedek – Kósa Márk – Pánovics János: A mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a debreceni egyetemen. 2002., *Informatika a Felsőoktatásban*, 1103–1109. p.

[8] Zbigniew Michalewicz – David B. Fogel: *How to Solve It: Modern Heuristics*. 2004, Springer.

[9] Kósa Márk: ACM versenyfeladatok programozóknak.
URL http://it.inf.unideb.hu/honlap/acm.

[10] Kósa Márk – Pánovics János – Gunda Lénárd: An evaluating tool for programming contests. 3. évf. (2005) 1. sz., *Teaching Mathematics and Computer Science*, 103–119. p.

[11] Bernd Owsnicki-Klewe: Search algorithms.
URL http://users.informatik.haw-hamburg.de/~owsnicki/search.html.

[12] Stuart J. Russell – Peter Norvig: *Mesterséges intelligencia modern megközelítésben*. 2005, Panem.

[13] Dennis E. Shasha: *Kiberrejtvények*. 2003, Typotex.

[14] Dennis E. Shasha: *Dr. Ecco Mathematical Detective*. 2004, Dover Publications.

[15] Dennis E. Shasha: *Puzzling Adventures*. 2005, W. W. Norton & Company.

[16] UVa Online Judge. URL http://uva.onlinejudge.org/.

# Tudományos közlemények

## Az értekezés témájához kapcsolódó referált közlemények

1. **Kósa Márk**, Pánovics János, Gunda Lénárd: An evaluating tool for programming contests, *Teaching Mathematics and Computer Science* (2005) **3** (1), p. 103–119.

   Ref. szám: Mathematics Didactics Database ME 2005f.02643.
   (http://www.zentralblatt-math.org/matheduc/)

2. **Kósa Márk**, Nagy Benedek: Logical puzzles (truth-tellers and liars), *Proceedings of the 5$^{th}$ International Conference on Applied Informatics* (2001) p. 105–112.

   Ref. szám: Zentralblatt MATH Zbl 1103.68826.
   (http://www.zentralblatt-math.org/zmath/)

## Az értekezés témájához kapcsolódó tankönyv

1. Juhász István, **Kósa Márk**, Pánovics János: *C példatár*, Panem, Budapest, 2005.

## Konferenciakiadványban megjelent dolgozatok

1. Várterész Magda, Nagy Benedek, **Kósa Márk**, Pánovics János: *A Mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a Debreceni Egyetemen*, Informatika a Felsőoktatásban 2002 Konferencia, Debrecen, 1103–1109. oldal.

2. **Kósa Márk**: *A kiszolgálási elvek hatása a Markov-vezérelt véges forrású sorbanállási rendszerek teljesítmény-mérőszámaira*, Informatika a Felsőoktatásban 2002 Konferencia, Debrecen, 1146–1153. oldal.

3. **Kósa Márk**, Pánovics János, Gunda Lénárd: *An evaluation tool for programming contests*, 6$^{\text{th}}$ International Conference on Applied Informatics, Eger, Vol. I., p. 163–172.

4. **Kósa Márk**: *Stochastic Simulation of Markov-Modulated Finite-Source Queues in Java Environment*, 6th International Conference on Applied Informatics, Eger, Vol. II., p. 369–377.

5. **Kósa Márk**, Nagy Benedek, Pánovics János: *Megoldáskereső algoritmusok hatékonyságának vizsgálata az állapottér-reprezentációk függvényében*, Számítástechnika az Oktatásban 2006, XVI. nemzetközi konferencia, Szováta, Románia, 76–81. oldal.

6. **Kósa Márk**, Pánovics János: *Keresőalgoritmusok objektumorientált megközelítése a* Mesterséges intelligencia *tárgy bevezető kurzusán*, Számítástechnika az Oktatásban 2007, XVII. nemzetközi konferencia, Nagyvárad, Románia, 94–97. oldal.

7. **Kósa Márk**, Pánovics János: *Szoftverfejlesztés a Qt keretrendszer használatával*, Számítástechnika az Oktatásban 2008, XVIII. nemzetközi konferencia, Csíksomlyó, Románia, 179–184. oldal.

## Az értekezés témájához kapcsolódó konferencia előadások

1. Nagy Benedek, **Kósa Márk**: *Igazmondó-hazug fejtörők gráfelméleti megközelítésben*, XXV. Magyar Operációkutatási konferencia, Debrecen, 2001. október 17–20.

2. Várterész Magda, Nagy Benedek, **Kósa Márk**, Pánovics János: *A Mesterséges intelligencia tárgy bevezető kurzusának gyakorlatai a Debreceni Egyetemen*, Informatika a Felsőoktatásban 2002 Konferencia, Debrecen, 2002. augusztus 28–30.

3. **Kósa Márk**, Pánovics János, Gunda Lénárd: *An evaluation tool for programming contests*, 6th International Conference on Applied Informatics, Eger, 27–31 January 2004.

4. **Kósa Márk**, Nagy Benedek, Pánovics János: *Megoldáskereső algoritmusok hatékonyságának vizsgálata az állapottér-reprezentációk függvényében*, Számítástechnika az oktatásban 2006, XVI. nemzetközi konferencia, Szováta, Románia, 2006. május 25–28.

5. **Kósa Márk**, Nagy Benedek, Pánovics János: *Performance analysis of search algorithms depending on the state space representation*, 6th Joint Conference on Mathematics and Computer Science, Pécs, 12–15 July 2006.

6. **Kósa Márk**, Pánovics János: *Search algorithms at ACM contests*, 7th International Conference on Applied Informatics, Eger, 28–31 January 2007.

## További konferencia előadások

1. Almási Béla, **Kósa Márk**, Sztrik János: *Nemmegbízható terminálrendszerek optimalizálási problémái a MOSEL segítségével*, XXV. Magyar Operációkutatási Konferencia, Debrecen, 2001. október 17–20.

2. **Kósa Márk**: *Markov-vezérelt véges forrású sorbanállási rendszerek szimulációja*, XXV. Magyar Operációkutatási konferencia, Debrecen, 2001. október 17–20.

3. Sztrik János, Oliver Möller, **Kósa Márk**: *The effects of service disciplines on the performance measures of Markov modulated finite-source queuing systems*, XII Seminar on Stability Problems of Stochastic Models, Varna, Bulgaria, 25–31 May 2002.

4. **Kósa Márk**: *A kiszolgálási elvek hatása a Markov-vezérelt véges forrású sorbanállási rendszerek teljesítmény-mérőszámaira*, Informatika a Felsőoktatásban 2002 Konferencia, Debrecen, 2002. augusztus 28–30.

5. **Kósa Márk**: *Stochastic Simulation of Markov-Modulated Finite-Source Queues in Java Environment*, 6$^{\text{th}}$ International Conference on Applied Informatics, Eger, 27–31 January 2004.

## További tudományos közlemények

1. **Kósa Márk**, Pánovics János: *Programming Contest Result Manager*, Technical reports, 2009/9, Preprints No. 369, Faculty of Informatics, University of Debrecen.

## Szoftvertermék

1. Programming Contest Result Manager, 2003.